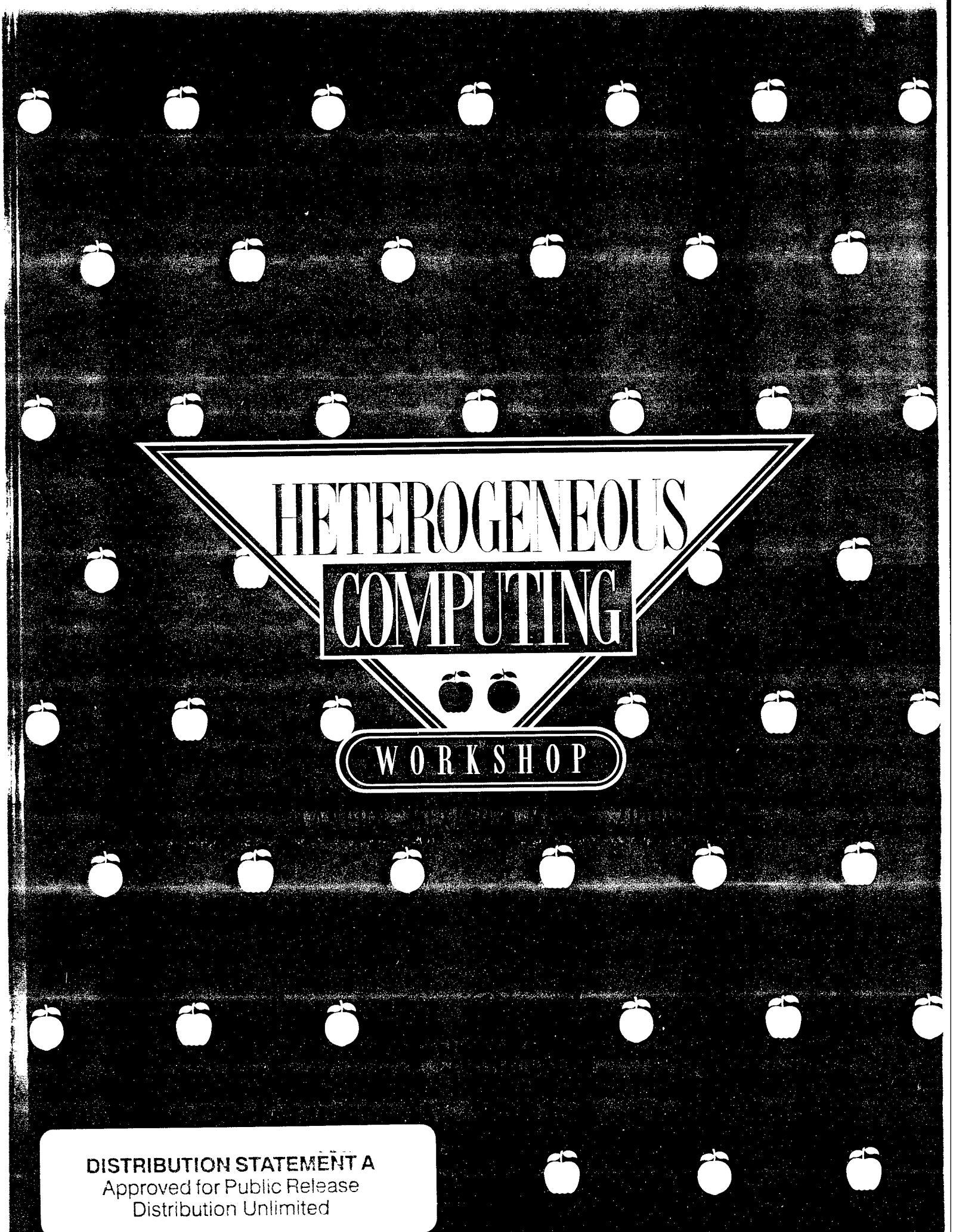


DISTRIBUTION STATEMENT A
Approved for Public Release
Distribution Unlimited

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.

1. AGENCY USE ONLY *(Leave Blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	Dec. 1, 2000	Final, Nov. 24, 1999 to Sept. 30, 2000

4. TITLE AND SUBTITLE
The Ninth Workshop on Heterogeneous Computing: HCW 2000

5. FUNDING NUMBERS
N00014-00-1-0189

6. AUTHOR(S)
H. J. Siegel

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
School of Electrical and Computer Engineering
Purdue University
West Lafayette, IN 47907-1285

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Dr. Andre M. van Tilborg, Director
Math, Computer & Information Sciences Division
Office of Naval Research
Arlington, VA 22217-5660

10. SPONSORING / MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION / AVAILABILITY STATEMENT
Unlimited

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

This grant funded the proceedings of the 9th Heterogeneous Computing Workshop (HCW 2000), which was held in May 2000. HCW 2000 was part of the 14th International Parallel and Distributed Processing Symposium, which was sponsored by the IEEE Computer Society Technical Committee on Parallel Processing and help in cooperation withACM SIGARCH. Heterogeneous computing systems range from diverse elements within a single computer to coordinated, geographically distributed maches with different architectures. A heterogeneous computing system provides a variety of capabilities that can be orchestrated to execute multiple tasks with varied computational requirements. Applications in these environments achieve preformance by exploiting the affinity of different tasks to different computational platforms or paradigms, while considering the overhead of inter-task communication and the coordination of distinct data sources and/or administrative domains. Topics representative of those inlevant to heterogeneous computing include: network profiling, configuration tools, scheduling tools, analytic benchmarking, programming paradigms, problem mapping, processor assignment and scheduling, fault tolerance, programming tools, processor selection criteria, and compiler assistance.

14. SUBJECT TERMS
heterogeneous computing, distributed computing, high-performance computing

15. NUMBER OF PAGES
1

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UNLIMITED

PROCEEDINGS

9TH HETEROGENEOUS COMPUTING WORKSHOP (HCW 2000)

20001212 017

PROCEEDINGS

9TH HETEROGENEOUS COMPUTING WORKSHOP (HCW 2000)

MAY 1, 2000
CANCUN, MEXICO

Edited by
Cauligi Raghavendra, The Aerospace Corporation

Cosponsored by
IEEE Computer Society Technical Committee on Parallel Processing
U.S. Office of Naval Research

Industrial Affiliate
NOEMIX

Los Alamitos, California

Washington • Brussels • Tokyo

Copyright © 2000 by The Institute of Electrical and Electronics Engineers, Inc.
All rights reserved

Copyright and Reprint Permissions: Abstracting is permitted with credit to the source. Libraries may photocopy beyond the limits of US copyright law, for private use of patrons, those articles in this volume that carry a code at the bottom of the first page, provided that the per-copy fee indicated in the code is paid through the Copyright Clearance Center, 222 Rosewood Drive, Danvers, MA 01923.

Other copying, reprint, or republication requests should be addressed to: IEEE Copyrights Manager, IEEE Service Center, 445 Hoes Lane, P.O. Box 133, Piscataway, NJ 08855-1331.

The papers in this book comprise the proceedings of the meeting mentioned on the cover and title page. They reflect the authors' opinions and, in the interests of timely dissemination, are published as presented and without change. Their inclusion in this publication does not necessarily constitute endorsement by the editors, the IEEE Computer Society, or the Institute of Electrical and Electronics Engineers, Inc.

IEEE Computer Society Order Number PR00556
IEEE Order Plan Catalog Number PR00556
ISBN 0-7695-0556-2
ISSN 1097-5209
Library of Congress Number 00-100757

Additional copies may be ordered from:

IEEE Computer Society
Customer Service Center
10662 Los Vaqueros Circle
P.O. Box 3014
Los Alamitos, CA 90720-1314
Tel: + 1-714-821-8380
Fax: + 1-714-821-4641
E-mail: cs.books@computer.org

IEEE Service Center
445 Hoes Lane
P.O. Box 1331
Piscataway, NJ 08855-1331
Tel: + 1-908-981-1393
Fax: + 1-908-981-9667
mis.custserv@computer.org

IEEE Computer Society
Asia/Pacific Office
Watanabe Bldg., 1-4-2
Minami-Aoyama
Minato-kuTokyo 107-0062
JAPAN
Tel: + 81-3-3408-3118
Fax: + 81-3-3408-3553
tokyo.ofc@computer.org

Editorial production by A. Denise Williams

Cover art production by Alex Torres

Printed in the United States of America by Technical Communication Services

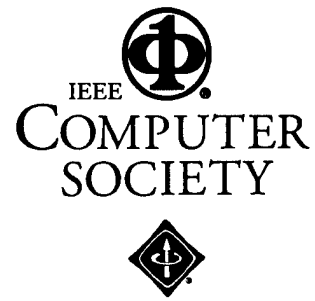


TABLE OF CONTENTS

9TH HETEROGENEOUS COMPUTING WORKSHOP

MESSAGE FROM THE GENERAL CHAIR

Welcome to the 9th Heterogeneous Computing Workshop (HCW 2000). The field of *heterogeneous computing* continues to mature, with several focused themes evolving over the past few years. These include *cluster computing, grid computing,* and *metacomputing,* among others. The Heterogeneous Computing Workshop series offers an international forum for researchers in all of these overlapping areas to present their research findings and interact with their peers.

I would like to thank H.J. Siegel, the Steering Committee Chair, for inviting me to be the General Chair. Throughout the past year, he provided me with invaluable inputs in resolving meeting-related issues. I also would like to thank C.S. Raghavendra, the Program Chair. He did an outstanding job in putting together an excellent technical program that addresses diverse aspects of heterogeneous computing. In addition, he assisted in resolving meeting-related issues including planning and publicity. It was a pleasure working with H.J. and Raghu.

This year, the response to the call for papers was overwhelming. For the first time, we had to arrange parallel sessions to accommodate so many excellent papers that were submitted! I would like to thank Susamma Barua, IPDPS 2000 Local Arrangements Chair, for her assistance in arranging the meeting space for us.

The workshop is cosponsored by the US Office of Naval Research and the IEEE Computer Society Technical Committee on Parallel Processing. I would like to thank Richard Freund of NOEMIX, Inc, for his continued support and guidance of the meeting series.

Muthucumaru Maheswaran acted as the Publicity Chair. I would like to thank him for the excellent job he did in maintaining our website as well as publicizing the meeting. Denise Williams of the IEEE Computer Society Press deserves special mention for her efforts in putting together the proceedings. Finally, I would like to thank my assistant Henryk Chrostek for coordinating meeting related interactions over the past year. It was a pleasure to work with all of them.

Viktor K. Prasanna
University of Southern California

MESSAGE FROM THE PROGRAM CHAIR

It has been a pleasure to organize the 9th Heterogeneous Computing Workshop (HCW 2000). This workshop is a forum to discuss the latest findings in heterogeneous computing and promising work-in-progress. Heterogeneous computing systems range from diverse elements within a single computer, to coordinated, geographically-distributed machines with different architectures. A heterogeneous computing system provides a variety of capabilities that can be orchestrated to execute multiple tasks with varied computational requirements. Applications in these environments achieve performance by exploiting the affinity of different tasks to different computational platforms or paradigms, while considering the overhead of inter-task communication and the coordination of distinct data sources and administrative domains. Such computing systems support information infrastructure and other terms, including *Cluster Computing* and *Grid Computing*, are also used to describe heterogeneous computing.

In prior years, the HCW program had a single track of paper presentations with an invited paper session. This year we received many quality papers, and with the surveyed opinions of authors, we decided to have two parallel tracks. The technical program includes 32 papers arranged in 10 sessions along the two parallel tracks. Each of the submitted papers was reviewed by two program committee members and external reviewers. These presentations cover a range of heterogeneous computing topics, including *grid computing and applications, scheduling algorithms, theory and modeling,* and *resource management.* I would like to thank the members of the Technical Program Committee for their valuable and timely review work and their help in putting together the program.

The success of a workshop such as this depends on the contributions of many individuals. First, I would like to thank Viktor Prasanna, the General Chair, for inviting me to be the Program Chair and for providing me with a number of pointers on organizational matters. Next, I would like to thank H.J. Siegel, the Steering Committee Chair, for his continued support on resolving meeting-related issues and for his help in getting the financial support for publishing the workshop proceedings. I also would like to thank Muthucumaru Maheswaran for publicizing this workshop through various on-line mailing lists and postings on the Web. I am thankful to Henryk Chrostek and Ammar Alhusaini for their help in handling submitted papers and in the organization of the Program Committee meeting. Finally, on behalf of the Program Committee, I would like to extend my gratitude to the authors, session chairpersons, and the reviewers who contributed to making the 9th Heterogeneous Computing Workshop a success.

Cauligi S. Raghavendra
The Aerospace Corporation

MESSAGE FROM THE STEERING COMMITTEE CHAIR

These are the proceedings of the 9th Heterogeneous Computing Workshop, also known as HCW 2000. *Heterogeneous computing* is a very important research area with great practical impact. The topic of *heterogeneous systems* covers many types of systems. A heterogeneous system may be set of machines interconnected by a wide-area network and used to support the execution of jobs submitted by a large variety of users to process data that is distributed throughout the system. A heterogeneous system may be a suite of high-performance machines tightly interconnected by a fast dedicated local-area network and used to process a set of production tasks, where the subtasks of each task may execute on different machines in the suite. A heterogeneous system may also be a special-purpose embedded system, such as a set of different types of processors used for automatic target recognition. In the extreme, a heterogeneous system may consist of a single machine that can reconfigure itself to operate in different ways (e.g., in different modes of parallelism). All of these types of heterogeneous systems (as well as others) are appropriate topics for this workshop series. I hope you find the contents of these proceedings informative and interesting. I encourage you to look also at the proceedings of past and future Heterogeneous Computing Workshops.

Many people have worked very hard to make this workshop happen. Cauligi "Raghu" Raghavendra, of the University of Southern California and The Aerospace Corporation, was this year's Program Committee Chair, and he assembled the excellent program and collection of papers in these proceedings. Raghu did this with the assistance of his Program Committee, which is listed in these proceedings. Viktor Prasanna, of the University of Southern California, was the General Chair, and he was responsible for the overall organization and administration of this year's workshop, and he did a fine job. I thank Richard F. Freund, of NOEMIX, for founding this workshop series, and for asking me to succeed him as Chair of the Steering Committee.

Due to the increasing importance of this research area and the efforts of the workshop organizing committee, we received so many excellent submissions this year that we had to go to parallel sessions for the first time. While we realize this splits the audience, we did not want to turn away good papers simply because this is a one-day workshop (and we did not want to extend the workshop another day, conflicting with our host symposium's sessions).

This year IEEE Computer Society and the Office of Naval Research (ONR) cosponsored the workshop, with additional support given from our industrial affiliate, NOEMIX. I thank Andre M. van Tilborg, the Director of the Math, Computer, & Information Sciences Division of the Office of Naval Research, for arranging funding for the publication of the workshop proceedings (under grant number N00014-00-1-0189). We greatly appreciate their continued support of our proceedings. I thank Richard F. Freund, of NOEMIX, for again providing the plaque given to the Program Chair in recognition of his efforts.

This workshop is held in conjunction with the International Parallel and Distributed Processing Symposium (IPDPS), which is a merger of symposia formerly known as the International Parallel Processing Symposium (IPPS) and the Symposium on Parallel and Distributed Processing (SPDP). The Heterogeneous Computing Workshop series is very grateful for the constant cooperation and assistance we have received from the IPDPS/IPPS/SPDP organizers.

H. J. Siegel
Purdue University

ORGANIZING COMMITTEE

General Chair

Viktor K. Prasanna

University of Southern California

Program Chair

C.S. Raghavendra

The Aerospace Corporation

Steering Committee Chair

H. J. Siegel

Purdue University

Steering Committee

Francine Berman, *UCSD*

Jack Dongarra, *Univ. of Tennessee and Oak Ridge Nat'l. Lab*

Richard F. Freund, *NOEMIX*

Debra Hensgen, *Naval Postgraduate School*

Paul Messina, *Caltech*

Jerry Potter, *Kent State University*

Viktor K. Prasanna, *USC*

Vaidy Sunderam, *Emory University*

Publicity Chair

Muthucumaru Maheswaran

University of Manitoba

PROGRAM COMMITTEE

Ishfaq Ahmad, *Hong Kong Univ. of Sci. and Tech.*
John Antonio, *University of Oklahoma*
Mark Baker, *University of Portsmouth, UK*
Rajkumar Buyya, *Monash University, Australia*
Steve Chapin, *Syracuse University*
Alok Choudhary, *Northwestern University*
Frederica Darema, *NSF/CISE*
Rudolf Eigenmann, *Purdue University*
Mary Eshaghian, *California State Univ., Long Beach*
Salim Hariri, *University of Arizona*
Debra Hensgen, *Naval Postgraduate School*
Kai Hwang, *University of Southern California*
Zvi Kedem, *New York University*
Vipin Kumar, *University of Minnesota*
Craig Lee, *The Aerospace Corporation*
Jorg Liebeherr, *University of Virginia*
Muthucumaru Maheswaran, *University of Manitoba*
Dhabaleshwar Panda, *Ohio State University*
Ira Pramanick, *Sun Microsystems*
Karsten Schwan, *Georgia Institute of Technology*
Behrooz Shirazi, *University of Texas at Arlington*
Arun Somani, *Iowa State University*
Mitchell Theys, *University of Illinois at Chicago*
Jon B. Weissman, *University of Minnesota*

REFEREES

Ishfaq Ahmad
Shahriar M. Akramullah
Ammar Alhusaini
Shoukat Ali
John Antonio
Mark Baker
Jorge Barbosa
Noah Beck
Fran Berman
Machael D. Beynon
Prashant Bhat
Tracy Braun
Rajkumar Buyya
Henri Casanova
Charles Cavanaugh
Fangzhe Chang
Steve Chapin
Surjamukhi Chatterjea
Andrew Chien
Alok Choudhary
Holly Dail
Lynn K. Daley
Frederica Darema
Elise deDoncker
Rudolf Eigenmann
Mary Eshaghian
Rasit Eskicioglu
Silvia Figueira
Jose Fortes
Joern Gehring
Claudia Gold
Peter Graham
Andrew Grimshaw
Salim Hariri
Christopher Hogue
Robert Hood
Hai Jin
Mahesh V. Joshi
Abhay Kanhere
Nirav H. Kapadia
Zvi Kedem
Carl Kesselman
Yoonhee Kim
Stephen D. Kleban
Eileen Kraemer
Vipin Kumar
Ricky Kwok
Domenico Laforenza
Craig Lee
William J Leinberger
Tim Levin
Wei-Keng Liao
Jorg Liebeherr
Hwa-Chun Lin
Muthucumaru Maheswaran
Dan Marinescu
Graziano Obertelli
Zoran Obradovic
Paolo Palazzari
Dhabaleshwar Panda
Ira Pramanick
Viktor Prasanna
Xiao Qin
Andrei Radulescu
C.S. Raghavendra
Albert I. Reuther
Paul Roe
Arnold Rosenberg
Ellard Roush
Charles Salisbury
Ahmed S. Sameh
Karsten Schwan
Gary Shao
Behrooz Shirazi
H. J. Siegel
Fabricio Silva
Shava Smallen
Arun Somani
Paul Stelling
Vaidy Sunderam
Rajeev Thakur
Mitchell Theys
Raju Venkataramana
Manish Verma
Jon Weissman
Vladimir Yarmolenko

SESSION 1-A
GRID ENVIRONMENT

Chair: K. Hwang, *University of Southern California, USA*

Master/Slave Computing on the Grid

Gary Shao *
Department of Computer Science
and Engineering
University of California, San Diego
San Diego, CA 92093-0114
gshao@cs.ucsd.edu

Francine Berman †
Department of Computer Science
and Engineering
University of California, San Diego
San Diego, CA 92093-0114
berman@cs.ucsd.edu

Rich Wolski †
Department of Computer Science
107 Ayres Hall
University of Tennessee
Knoxville, TN 37996-1301
rich@cs.utk.edu

Abstract

Resource selection is fundamental to the performance of master/slave applications. In this paper, we address the problem of promoting performance for distributed master/slave applications targeted to distributed, heterogeneous "Grid" resources. We present a work-rate-based model of master/slave application performance which utilizes both system and application characteristics to select potentially performance-efficient hosts for both the master and slave processes. Using a Grid allocation strategy based on this performance model, we demonstrate a performance improvement over other selection options for a representative set of Master/Slave applications in both simulated and actual Grid environments.

1. Introduction

The **master/slave** paradigm is a fundamental and commonly used approach for parallel and distributed applications. In master/slave applications, a single *master* process controls the distribution of work to a set of identically operating *slave* processes. The master/slave paradigm has been used successfully for a wide class of parallel applications [12][6][14], and is well suited as a programming model for applications targeted to distributed, heterogeneous "Grid" resources[1].

Methods which can improve the performance of master/slave applications are of considerable interest to many people. Researchers and application developers have previously experimented with tuning the granularity of master and slave processes to balance computation and communication, varying parameters such as the number and complexity of tasks assigned to slaves, and varying the number of slave processes used [3] [8][16]. Note that in a homogeneous environment, any processor can reasonably be chosen as a master or a slave, as all resources are typically considered to be equivalent. However, in a heterogeneous Grid environment, non-uniformity in both the peak and deliverable capacities of computational and communication resources can produce very different application execution times depending on which processor is chosen for the master and which processors are chosen for the slaves.

In this paper, we address the problem of how to determine a performance-efficient placement of master and slave processes running in shared, distributed and heterogeneous environments. In a heterogeneous environment, the choice of processor for the master can have a significant effect on total available work rate, directly impacting application performance. Our strategy for selecting a location for the master process involves identifying the host processor which allows for the largest aggregated system work rate, which we will define in the next section. Our strategy for selecting slaves utilizes the performance capacity of the available computation and communication resources to determine a

*Supported in part by NSF grant #ASC-9701333, DARPA/ITO contract #N66001-97-C-8531, NPACI award #ASC9619020

†Supported in part by NSF grant #ASC-9701333, DARPA/ITO contract #N66001-97-C-8531, NPACI award #ASC9619020

0-7695-0556-2/00 $10.00 © 2000 IEEE

performance-efficient collection of workers.

This paper is organized as follows: Section 2 provides a performance model for distributed master/slave applications. Section 3 describes how we obtain and use input parameters for calculating resource work capacity values in our performance models. Section 4 describes our algorithm for selecting the resources to use for the master and slave processes. Section 5 gives a representative set of performance results from our experiments, Section 6 includes a short discussion of some related work, and Section 7 provides a summary of our work.

2. A master/slave performance model

We consider a model of master/slave applications in which the primary function of the master process m is to pass out and collect work from a set of slave processes $s \in S$.[1] We assume that communication patterns are simple and well-defined, requiring communication only between the master process and individual slave processes. We will define the application's *work* as a divisible set of *tasks*; where each task may require some input data and produces some output data.

Tasks are completed in an application by progressing through four stages in the master/slave computation:

Stage 1 is the transmission of a command to initiate a task on one of the slave processes, including any data needed by the slave to perform the computation.

Stage 2 is the execution of the task by the designated slave.

Stage 3 is the transmission of results from the slave back to the master.

Stage 4 is any immediate processing of task results from the slave that must be done by the master.

While passing through each stage in the computation, a particular system resource must be employed by a task for some period of time, after which the task can move on to the the next stage. As an example, we can consider the simple network topology shown in Figure 1. If processor A in Figure 1 is designated as the master process, a task intended for slave processor B during Stage 1 will employ the use of network Net1 to transfer required data from processor A to processor B. During Stage 2 the task will utilize processor time on B to run task computations. During Stage 3 the task will again utilize network Net1 to transfer result data from B to A. Finally, during Stage 4 the task will utilize processor time on A to process the incoming results and to prepare for initiating additional task transfers to B.

[1] It would be straightforward to extend this work to the case in which the master may also perform some work as a slave.

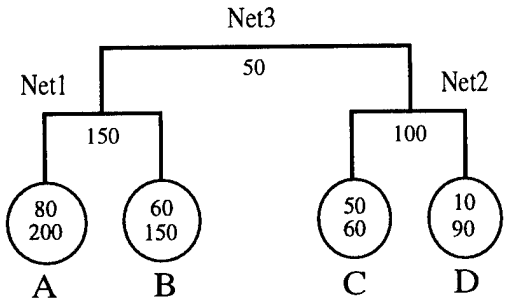


Figure 1. Example network configuration.

In constructing a performance model for master/slave applications, we look at the rate that applications process tasks. *The rate at which an application cycles through tasks can be used as a measure of application performance, as faster overall cycle rates will correspond directly to reduced application execution times.*

If we consider the flow of tasks between a master process m and a slave process s, we can make the definition:

$SlaveRate(m, s)$ is the task completion rate occurring between master m and slave s, in units of tasks per unit of time.

For master/slave computations where there is no communication between different slave processes, the total rate of task completions for an application will be the sum of the rates arising from task completions by individual slaves. We define $AppRate(m, S)$ in Equation (1) to be the rate of task completions by an application with master process m and a set of slave processes S.

$$AppRate(m, S) = \sum_{s \in S} SlaveRate(m, s) \qquad (1)$$

We can then define execution time, $ExecTime(m, S)$, for an application with a master process m and the set of slave processes S, and where $Tasks$ is the total number of tasks in the application.

$$ExecTime(m, S) = Tasks/AppRate(m, S), \qquad (2)$$

Application performance can thus be derived from values for $SlaveRate(m, s)$. One way to solve for these $SlaveRate$ values is to consider the system resource constraints which bound achievable application performance. To illustrate the concept, we go back to our simple example system in Figure 1, and observe that each processor and network has been labeled with one or more numerical values. We define the numbers in the diagram to represent resource work capacities in terms of tasks per unit of time. The values next to network links represent network work capacity for that network link, the upper number within each circle

represents slave work capacity for that processor, and the lower number within each circle represents master work capacity for that processor.

Consider an application which uses processor C to host the master process. To solve for application performance, we would like to determine values for $SlaveRate(C, A)$, $SlaveRate(C, B)$ and $SlaveRate(C, D)$. The fundamental constraint condition to meet is that total task flow rates through any resource cannot exceed the capacity value of that resource. This means, since task flow from both processor A and processor B passes through network Net3 in our example, that the sum of $SlaveRate(C, A)$ and $SlaveRate(C, B)$ can be at most 50, the capacity of Net3.

In general, we can define the following resource work capacity terms for processor and network resources. All terms are rates with units of tasks per unit of time.

$W_{MasterCPU}(i)$ = the maximum master work rate supported by a processor i. This is determined by processor i's capacity to perform Stage 4 computations for a specified application.

$W_{SlaveCPU}(i)$ = the maximum slave work rate supported by a processor i. This is determined by processor i's capacity to perform Stage 2 computations for a specified application.

$W_{Net}(n)$ = the maximum communication rate supported by a network n. This is determined by network n's capacity to perform the Stage 1 and Stage 3 communication of a specified application.

Assuming we have a graph G representing network connectivity (such as the diagram in Figure 1) that allows us to identify which network resources are shared between different task flows, and resource work capacity rates for each of the resources in our system, we can form a set of upper bounds on possible $SlaveRate(m, s)$ values. The process by which the network connectivity graph G and the work capacity rate terms can be derived for resources in a Grid environment will be discussed later in section 3.

First, to aid us in defining our upper bound constraints, we define a helper set constructor function:

$ShareNet(G, S, m, n)$ takes as input a network connectivity graph G, a set of slaves processes S, a master process m, and a network resource n, and returns the set of slave processes from S which share the use of network resource n when communicating with m.

For master/slave applications, $ShareNet(G, S, m, n)$ can be easily determined for a network graph G, master process m, set of slaves S, and network resource n by following the single path in the graph G from each slave process $s \in S$ to the master process m, recording each path passing through the resource n.

Now we can give bounds which form constraints on application performance, as shown below[2].

$$\sum_{i \in S} SlaveRate(m, i) \leq W_{MasterCPU}(m) \quad (3)$$

$$SlaveRate(m, i) \leq W_{SlaveCPU}(i) \quad (4)$$

$$\sum_{i \in ShareNet(G,S,m,n)} SlaveRate(m, i) \leq W_{Net}(n) \quad (5)$$

Our goal is to find the values of $SlaveRate(m, s)$ which meet the constraints given above and which yield the largest value of $AppRate(m, S)$. The solution will correspond to a configuration which delivers the best achievable application performance.

We can frame the problem of determining values for $SlaveRate(m, s)$ which yield the largest $AppRate(m, S)$ value as a **flow-rate problem** where: (1) the $SlaveRate(m, s)$ values are the flows we wish to solve for, (2) m is the sink for all flows, (3) the set S of slave processes are the sources for flows, and (4) the flow constraints correspond to the $W_{MasterCPU}(i)$, $W_{SlaveCPU}(i)$, and $W_{Net}(n)$ work capacities in our target environment.

Because the work flows in a master/slave computation form a tree rooted at the master, and because we have limited our investigation to considering no more than one process hosted on each processor, efficient algorithms like the *Maximum-Flow algorithm* [5] exist for solving this problem. This approach can be used to solve the flow-rate problem for several candidate processes m, finding the one which is expected to deliver the maximum work flow, and hence the best expected application performance. Section 4 describes the implementation of one such maximum-flow algorithm that can be used to find the largest possible work flow.

3. Modeling work capacity rates in a Grid environment

In order to apply our work flow performance model to real applications running in a Grid environment, we must derive a network connectivity graph G and appropriate values for the work capacity rate terms $W_{MasterCPU}(i)$, $W_{SlaveCPU}(i)$, and $W_{Net}(n)$.

The flow-rate algorithm for determining application performance requires a graph G which represents the network connectivity between processor resources. For wide-area

[2]Since we consider here only cases where processors can host at most one process from the same application, we allow the process identifier to be the identifier of the processor hosting it in our inequality expressions.

Input Data	Description	Used In	How Acquired	When Acquired
$Graph_{Net}$	Network connectivity	G	ENV	periodically
$T_{SlaveCPU}$	CPU slave task time	$W_{SlaveCPU}$	benchmark	install
$T_{MasterCPU}$	CPU master task time	$W_{MasterCPU}$	benchmark	install
$Avail_{CPU}$	CPU availability	$W_{SlaveCPU}$, $W_{MasterCPU}$	NWS	run-time
$Size_{TaskXfer}$	Task data transfer size	W_{Net}	analysis, logging	application
BW_{Net}	Network bandwidth	W_{Net}	NWS	run-time

Table 1. Inputs for constructing the performance model.

Grid environments, it might be very difficult to get complete physical network configuration data about every platform in the system. It is reasonable, however, to represent the target computational resources and their interconnection by a *logical view* which captures those areas where network constraints present potential bottlenecks to application performance. We derive a logical view of resource interconnection using a logical network configuration discovery tool called Effective Network Views (ENV) [13]. (Other systems for discovery of effective system topology such as [9] might also be used.) The output of the ENV tool is a network graph representation where every processor belongs to a cluster of one or more machines. Machines in a cluster are connected together through a local network, where the capacity of the local network represents the limiting capacity of a network resource shared by each machine in the cluster. Clusters of local networks are connected together in our logical representation through a single layer of non-local network links. This representation is suitable for use in graph-based analysis techniques like our maximum flow-rate problem, and directly translates to the network graph G in our flow-rate solution.

The processor work capacity rates $W_{SlaveCPU}(i)$ and $W_{MasterCPU}(i)$ in our model are determined with two components: an application-specific component representing the maximum performance delivered by a processor resource in its unloaded state, and a dynamic component that is determined at run-time to adjust capacity rates to account for current loading conditions. The application-specific component is obtained by running a benchmark of the target application code on an unloaded processor, and measuring the times $T_{SlaveCPU}(i)$ and $T_{MasterCPU}(i)$ that are required to compute a single task on processor type i by the slave and master processes respectively. If the task computation time is variable over time, perhaps because of data dependencies in the application, we take an average value for all task times in one run of the application benchmark. This value could be scaled for particular classes of data sets at run-time if the variation in average task run times is large when different data sets are used. The benchmark times only have to be measured once for each platform type on which the application is built to run, so obtaining these values is computationally efficient.

The dynamic component of the work capacity terms for processor resources is calculated with the help of real-time monitoring and forecasting services such as the Network Weather Service [19] (NWS). The NWS provides real-time predictions of dynamic processor availability $Avail_{CPU}(i)$ (the percentage of CPU time a process can expect to get on processor i). $Avail_{CPU}(i)$ describes the predicted availability status of a processor resource, and can be generated independently from any particular application. This enables a single NWS system to provide simultaneous service to many applications requiring real-time information about resource behavior.

The processor work capacity rates can be calculated using the application-specific and dynamic components as shown below. The input parameters for these functions are summarized in Table 1.

$$W_{SlaveCPU}(i) = Avail_{CPU}(i)/T_{SlaveCPU}(i) \quad (6)$$

$$W_{MasterCPU}(i) = Avail_{CPU}(i)/T_{MasterCPU}(i) \quad (7)$$

The network work capacity rate $W_{Net}(n)$ in our model is also calculated using two components. One component is the application-specific term $Size_{TaskXfer}$, which represents the amount of data transferred between a master process and a slave process for each task in an application.

If the task data transfer sizes are a variable quantity over time, perhaps due to data dependencies in the application, we must calculate an average data transfer value that represents expected steady-state communication behavior over the time of an entire application run.

The second component used in calculating network work capacities on a network resource n is a dynamic prediction of expected available network bandwidth $BW_{Net}(n)$, which we obtain from the NWS. The network work capacity rates can be calculated using application-specific and dynamic components as shown below. The input parameters are again summarized in Table 1.

$$W_{Net}(n) = BW_{Net}(n)/Size_{TaskXfer} \quad (8)$$

Having constructed a set of resource constraint values to help model the performance of a Grid environment, we should also discuss an obvious limitation of our approach. For each of the terms derived in this section, we have generated an average-value expression for use in our steady-state application performance model. Yet each of the properties being modeled might in reality exhibit considerable variability over time, either due to time-varying load conditions or data dependent behavior of the application being run. Our experience has been that despite the limitations of converting many variable terms to average steady-state values, our approach still yields a performance model which can do a good job at estimating application performance, and which provides an effective tool for helping to solve the resource selection problem, which we discuss next.

4. Selecting a master and the slaves

Given the work-rate performance model described in Section 2 and a logical representation of the work capacities of Grid resources, we can now consider strategies for selecting processors to host the master and slave processes. These are important issues for master/slave applications running in Grid environments, where users may be able to choose from among many different types of resources, and where availability of these resources may change over time.

Selection of the right processor to host the master process can significantly impact the overall application performance, as the following section will show. Knowing which master placement produces the best application performance might also influence other important decisions, such as where to efficiently position input and output files for the application. Selection of the right set of processor resources to host the slave processes has two goals: (1) selecting enough resources from the available set to produce the best achievable application performance,and (2) limiting the selection to resources that will actually benefit application performance. The second goal is important for Grid environments where resources can be shared by many users, and resources can be owned and managed by many different organizations. In these environments, it is desirable that applications use only those resources they really need; thereby allowing limited pools of shared resources to satisfy the largest number of users. We will first consider the issue of selecting the right host for the master process.

4.1. Master selection example

In a heterogeneous system, selection of a location for the master process very strongly depends on the deliverable work capacity of candidate resources. Consider the logical Grid configuration shown back in Figure 1, where four processors are connected by a system of three networks. We have labeled the network resources with values representing the W_{Net} capacity terms. The processor resources, shown as circles in the diagram, have been labeled with two values: a $W_{SlaveCPU}$ capacity term on top, and a $W_{MasterCPU}$ capacity term on the bottom. All capacity terms are in units of tasks per second.

For this simple example system, we can determine the assignment of the master process to a processor that gives us the greatest achievable work flow. If processor A is selected to host the master process, processor B is able to provide 60 tasks/sec work rate as a slave. In addition, a maximum of 50 tasks/sec worth of data can be transferred over network Net3, a work rate which can be supplied by processor C. The total expected application work rate with processor A hosting the master is therefore 110 tasks/sec. If we consider selecting processor C to host the master process, we observe that processor D can deliver a work rate of 10 tasks/sec working as a slave. In addition, we can transfer a maximum of 50 tasks/sec worth of data over network Net3, which can be supplied by processor A. It becomes apparent that processor C is constrained from achieving any higher application work rate by the limitations on the Net3 capacity, as well as the capacity of processor C to serve as the master host, to no more than 60 tasks/sec. We could proceed in a similar manner for all the processors, and derive expected application work rates for each candidate. Table 2 shows one set of possible outcomes for this process. It is apparent from the last column in Table 2 that processor B is the best choice, yielding a potential application work rate of 130 tasks/sec.

4.2. Selecting the master

More generally, we have developed a basic algorithm for finding the best performing host for the master process. It is based on the well-known maximum-flow algorithm by Ford and Fulkerson [7]. In this algorithm, we keep augmenting the estimated flow rate for each master host by adding the

Master Location m	$W_{MasterCPU}$ (m)	$SlaveRate$ (m, A)	$SlaveRate$ (m, B)	$SlaveRate$ (m, C)	$SlaveRate$ (m, D)	$AppRate(m)$
A	200	0	60	50	0	110
B	150	80	0	50	0	130
C	60	50	0	0	10	60
D	90	40	0	50	0	90

Table 2. Work rates resulting from master placement decision.

contributions of slave processors. Additional contributing slaves are selected first from those on the same local network as the master. This continues until either all of the slaves have been included, or no further slave work rates can be incorporated because of either capacity limitations on network resources, or capacity limitations of the master processor itself. If further capacity is available from processors on non-local networks, they are added one by one to the accumulated master total until no further additions are possible without exceeding one of the resource capacities. Figure 2 illustrates our basic algorithm for finding the best performing master host. Upon termination of the algorithm, the processor with the highest calculated work rate is selected as the master.

4.3. Complexity

In deriving the complexity of our algorithm, we note that our simplified logical representation of network configuration reduces the entire system to sets of processors connected by local networks. Each of these local networks is then connected to other local networks by at most one level of remote networking. With this logical topology, data transfers between slaves on the same local network pass through only one level of networking, and encounter only one network resource constraint. Data transfers between slaves located on different local networks will pass through at most three levels of networking, and must satisfy at most three networking constraints. All slave work rates must meet the resource constraints of the master processor. With this arrangement, there are at most four tests of constraints in our algorithm that have to be checked for each master and slave pairing.

If we have n processors in our system, then each master candidate can have at most $n-1$ slaves, and each individual master work rate calculation takes $O(n)$ time to calculate. Calculating maximum work rates for all n possible master candidates thus takes $O(n^2)$ time. Since our algorithm requires only simple compare and accumulation operations for each resource constraint test, the entire algorithm is efficient for the numbers of processors and networks we currently find in Grid environments available to a typical user.

4.4. Selecting the slaves

After selecting the master processor, we turn to selection of the slave processors. The issue is to select a set of processors for hosting slave processes that will deliver good aggregate performance. One approach is to start with the set of slave processors found in our master selection algorithm that yielded the highest expected application performance. Our algorithm keeps track of this set in the $Found(m)$ list, a list containing slaves used by the algorithm to calculate the maximum work rate for an application with processor m as the master host. Our master selection algorithm ensures that this set of processors results in work flows that fall within the constraints imposed by resource capacity limitations.

In numerous experimental trials using the set of processors from $Found(m)$ as slave hosts, we observed that the slave processors were often not delivering the maximum work rate values we expected in our algorithm. Observations of selected slaves showed the reduction in slave performance was due to the presence of unaccounted idle time, periods of time when slave processors were not doing useful work. An explanation for the observed idle times comes from observing the manner in which tasks are distributed to slave processors from the master. Each master/slave application we tested maintained a queue of available tasks on the master process, and distributed new tasks to individual slave processes upon request (a very commonly used technique). Because of contention for shared resources, such as networks and the master processor, delays sometimes occurred between the time a slave processor finished one task and the time at which the next task appeared for processing. These delays appeared as idle time in our observations of the slaves. With a minimum set of slaves selected to achieve the desired work rate, the unexpected idle time in the slaves resulted in a reduction of the actual total work rate achieved.

The work flow-rate performance model correctly determines possible application performance based on resource capacity limits. Our master selection algorithm uses this performance model, and in the process identifies a set of slaves which delivers this performance, assuming that each slave delivers its maximum work rate. Experimentation has shown that sometimes these slaves actually deliver less than their predicted maximum work rates, resulting in less per-

For all networks k
    Calculate maximum network capacity $W_{Net}(k)$
For all processors j
    Calculate maximum master processor capacity $W_{MasterCPU}(j)$
    Calculate maximum slave processor capacity $W_{SlaveCPU}(j)$
For each candidate master processor p on local network n
    Set sum for candidate slave work rates $CandRate(p) = 0$
    Set found set $Found(p)$ to empty
    For all networks k
        Set network utilization sum $NetUtil(k) = 0$
    Get maximum capacity $W_{Net}(n)$ of local network n
    Get maximum master processor capacity $W_{MasterCPU}(p)$
    While $CandRate(p) < W_{Net}(n)$ and $CandRate(p) < W_{MasterCPU}(p)$
        Select new processor s from same local network as p with
        the largest available $W_{SlaveCPU}(s)$ value
        Get slave processor capacity $W_{SlaveCPU}(s)$
        Get fraction F of $W_{SlaveCPU}(s)$ that will not cause
        utilization $NetUtil(n)$ to exceed $W_{Net}(n)$
            Add F to $CandRate(p)$
            Add F to $NetUtil(n)$
            Add processor s to found set $Found(p)$
    Total candidate work rate $CandRate(p)$ = min($CandRate(p)$, $W_{MasterCPU}(p)$)
    Total local network utilization $NetUtil(n) = CandRate(p)$
    While $CandRate(p) < W_{Net}(n)$ and $CandRate(p) < W_{MasterCPU}(p)$
        Select new processor q from outside local network with
        the largest available $W_{SlaveCPU}(q)$ value
        Get slave processor capacity $W_{SlaveCPU}(q)$
        Get fraction F of $W_{SlaveCPU}(q)$ that will not cause
        utilization $NetUtil(i)$ to exceed $W_{Net}(i)$ for any network i
            Add F to $CandRate(p)$
            Add F to $NetUtil(n)$
            Add F to other $NetUtil(k)$ where network k is involved in
            communications between processors p and q
            Add processor q to found set $Found(p)$
Select processor p with largest $CandRate(p)$ as master

Figure 2. Algorithm for finding best processor for the master.

formance than resource capacity constraints would allow. One way to get application performance back up to predicted levels is to add additional slave processors to the originally selected mix, thereby raising the effective slave work rates achieved up to expected values. Our goal is to compensate for lost performance due to idle time on the individual slave processors, while keeping the number of additional processors down to the minimum needed to accomplish this goal.

Our steady-state flow-rate performance model was not useful in helping to decide how many slaves to add to increase effective performance because it could not account for idle times caused by slaves waiting for new tasks to arrive. To address this shortcoming and others in our steady-state approaches to performance analysis, we developed a master/slave application performance simulator to provide significant new capabilities. We discuss this simulator and how it can be used to help solve the slave selection problem in the following subsections.

4.5. An application performance simulator

We originally developed a master/slave application performance simulator to provide detailed predictions of performance and resource behavior for applications running in Grid environments. One effective use we have found for this simulator is to help determine how many additional slave processors might be added to a predicted group of master and slave processors to make up for performance losses due to slave idle time.

At its core, our simulator is a set of routines which model the behavior of tasks as they pass through a system comprised of two kinds of resources: processors and networks. The resources are modeled as single servers with first-in-first-out input queues. Service times for the processor resources determine how long a task has control of the processor before relinquishing the resource to the next task in the input queue, and are dependent on the same processor availability parameters $Avail_{CPU}(i)$ and estimated task execution times $T_{SlaveCPU}(i)$ and $T_{MasterCPU}(i)$ developed earlier for our flow-rate model. Service times for the network resources determine how long a network resource is committed to servicing data transfers for each task, and are dependent on the same network bandwidth parameters $BW_{Net}(n)$ and size of the data transfers values $Size_{TaskXfer}$ developed for the flow-rate model presented earlier. In addition, all of the parameters can be adjusted to use either static steady-state values like those in the flow-rate performance model, or more dynamic data inputs such as statistical distributions or actual measured trace values from application runs. Network connectivity is represented using the same graph G, an output of the ENV tool, used in the work flow-rate performance model.

The simulator is written in highly portable C-language code, with the help of a simulation library package called Sim++ [4]. This simulator can be easily embedded into other programs, such as an application scheduler, to provide detailed predictions of application performance and resource utilization levels. It is particularly useful for observing the performance impact of changing application or resource parameters.

4.6. Using simulation to enhance slave selection

Our algorithm for finding the correct set of slave processors starts with the master processor m and the $Found(m)$ set of slaves from the master selection algorithm. The simulator is run with these machines as the target environment, using the same values for resource capacities as were used in the master selection algorithm. Results from the simulation are checked to see if any idle time on the simulated slaves results in a significant decrease in overall application performance. If a substantial performance decrease is found, resource utilization figures from the simulation are checked to see where additional processors might be added without exceeding existing resource constraints. If more slave processors are available to be added that will not violate any known resource constraints, they are added to the set of found slaves. A new system configuration with the additional processors added in is constructed and simulated once again. The process of slave additions and testing by simulation repeats until either there are no further performance gains realized by adding more slave processors, or no more processors can be found and placed without exceeding one of the known resource capacity constraints. Figure 3 illustrates our algorithm for finding the set of slave processors.

The algorithm given above makes good use of simulator results which calculate predicted resource utilization values for every resource in the system. These values allow us to quickly identify where in the system, if anywhere, slave processors might be added to improve application performance. In practice, the number of times the simulation cycle needs to be run is small as the process quickly converges to a situation where either additional performance gains are insignificant, or no further additions can be made without exceeding a resource constraint.

5. Experimental results

In this section, we describe experiments whose goal it is to test the usefulness and accuracy of our work-rate performance model and application performance simulator, as well as the performance of our algorithms for selecting master and slave processors.

Run master selection algorithm to get master processor m, set of slaves $Found(m)$, and predicted application work rate R

Run application performance simulator using m and $Found(m)$ to get simulated work rate S and slave utilization values $U(s)$

While S less than R
    Using $U(s)$, check which slaves s in $Found(m)$ have large simulated idle times

    Find additional processors A' that make up for idle time without exceeding any $W_{Net}(n)$ or $W_{MasterCPU}(m)$ constraints

    Add processors A' to $Found(m)$ to form $Found'(m)$

    Run simulator using m and $Found'(m)$ processors to get new simulated work rate S' and slave utilization values $U'(s)$

    If $S = S'$ or $Found(m) = Found'(m)$
        Return $Found(m)$ as slave solution

    Set S equal to S', all $U(s)$ equal to $U'(s)$
    Set $Found(m)$ equal to $Found'(m)$

Return $Found(m)$ as slave solution

Figure 3. Algorithm for finding best processors for slaves.

We use as an application test suite three applications chosen to represent a spectrum of potential master/slave distributed applications. The applications were selected and implemented to test the sensitivity of our approach to computation and communication granularity. Our master/slave implementation of the Mandelbrot image application is expected to display a relatively high sensitivity to communication constraints, as the amount of image data transferred during execution is large compared to the overall computation time. At the other extreme is the NAS Parallel Benchmarks' EP [18] application, which performs relatively little data transfer compared to the time spent computing. The Povray [11] ray-tracing application falls somewhere in the middle, with the transfer of one fourth the amount of image data as the Mandelbrot application which was spread out over a longer computation time.. Each of the applications was initially benchmarked on all target processor types to produce the application-specific parameters needed for our performance analysis tools. The applications are summarized in Table 3.

5.1. Experimental design

In the experiments, we compared *predicted execution time* (resulting from our performance model), *simulated execution time* (using the application simulator), and *actual execution time* (determined from experimental runs). All comparisons were made in a non-dedicated environment where the load traces used for the predicted and simulated execution times were determined from the NWS load trace of the actual execution time runs. We used identical parameter inputs for network configuration, resource constraints, and application characteristics in both work-flow analysis and performance simulation tools. In this way, we attempted to compare each set of execution times under the same environmental conditions.

The target experimental platform was a heterogeneous mix of Intel processor-based machines running Linux, and Sun SPARC machines running Solaris located in the Parallel Computation Laboratory in the Department of Computer Science and Engineering at the University of California, San Diego. The experiments were run with all machines in non-dedicated mode, but outside loading from compet-

Name	Description	Emphasis
Mandelbrot	parallel fractal image generator	communication
Povray	parallel implementation of popular ray-tracer	both
NBP EP	NAS Parallel Benchmark EP variant	computation

Table 3. List of applications used in experiments.

ing jobs was observed to be relatively light for most of the machines during the course of experimentation.

5.2. Results

In the first set of experiments, we ran the test suite of applications on a set of nine workstations shown in Table 4. For the three applications, trials were run with each of the nine processors being selected to run as the master while the other eight were included to run as slaves. In all cases, the work flow-rate problem was solved for each configuration of master and slaves to give the expected application execution time, shown as the light bars in Figures 4-6. The application performance simulator was run for all cases to give a predicted application execution time, shown by the middle bars in the graphs. And finally, the real applications were run on each configuration and their execution times recorded to appear as the dark bars on the graphs.. Figure 4 shows the results while running the relatively communication-heavy Mandelbrot application. Figure 5 shows the same set of execution times for the more balanced Povray application, while Figure 6 shows execution times for the computation-intensive NAS Parallel Benchmarks' EP application.

In these experiments, the work-rate performance model would have done a good job of identifying the correct master host to produce the fastest application execution times. In the Mandelbrot series of experiments, the machine *thing1* was calculated to yield the lowest execution time, which was confirmed in the actual application run. For this application the highest execution time, achieved with the machine named *lorax*, took 170% longer to finish than the best choice. For the other two applications, the work-rate performance model estimates of execution time again showed results which correlated closely with actual application run times. For these applications, which exhibited lower dependence on network constraints, the differences between the worst and best performers was smaller: about 25% for Povray and 10% for NAS EP. The work-rate based performance model correctly ordered master performance for both communication and computation constrained applications. The results also show that the application performance simulator did a good job of tracking the actual application execution times as well.

The experimental results show a small number of cases where the execution time was significantly underestimated for the Mandelbrot application. Analysis of experimental results leads us to believe the discrepancy in predicted and actual performance on the communication-heavy application was due to inadequate benchmarking of the $W_{MasterCPU}$ constraint terms. Actual application performance is worse than that predicted by both the work-flow model and the simulator because both tools overestimated the capacity of the single master process to process incoming data and respond to new task requests. When the real master process fails to keep up with projected work rates, the overall application work rate is reduced and execution time becomes relatively larger. Improved methods for benchmarking master processor performance are currently being developed to overcome this shortcoming.

In the second set of experiments, we look at two of our applications: Mandelbrot and Povray. In these trials we pick a specific host for the master process, then run our application for different numbers of slave processes. We show measured execution times and simulated execution times for our two applications as we increase the number of slave processors.

Figure 7 shows results with our communication-intensive Mandelbrot application for two different choices of the master host. These results show that the number of slaves which can beneficially be employed varies under different conditions, and is heavily constrained by the network speed of the master process host. Figure 8 shows results with the Povray application, whose performance is less dominated by communication costs. In our test environment, this application shows more scalable performance than Mandelbrot, but eventually also reaches a point where additional processors do not significantly decrease execution time. Results are shown for only one master case because data for other cases produces almost identical graphs. Results for our third application, NPB EP, are not shown here, but they are very similar to those for povray, with simulation predicted run times and actual application run times very close for all numbers of processors. These results indicate that for our representative examples, the performance simulator can be a useful tool to help predict the points at which either additional slaves should be added to a computation to increase performance, or when additional slaves cease to have any useful effect.

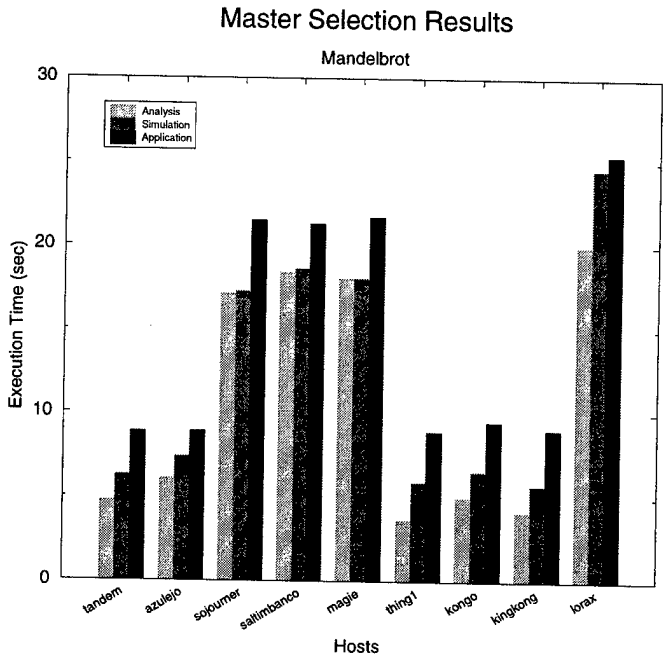


Figure 4. Execution time of communication-intensive application while varying master host.

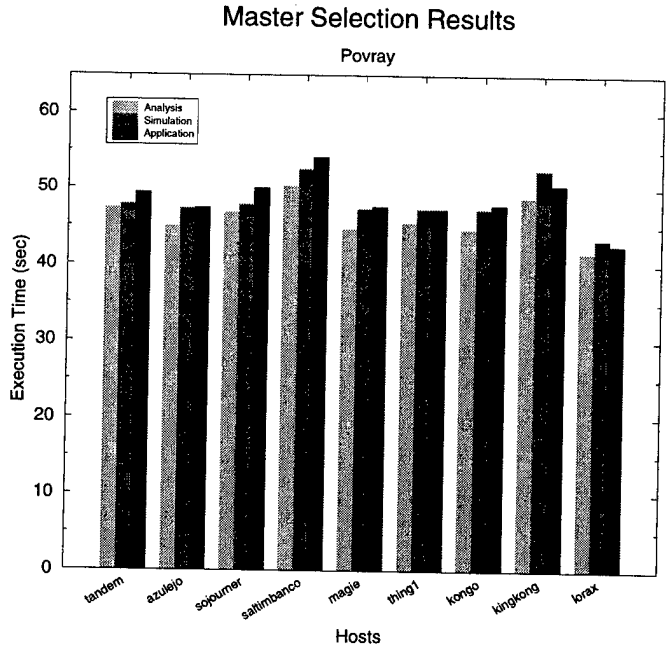


Figure 5. Execution time of application while varying master host.

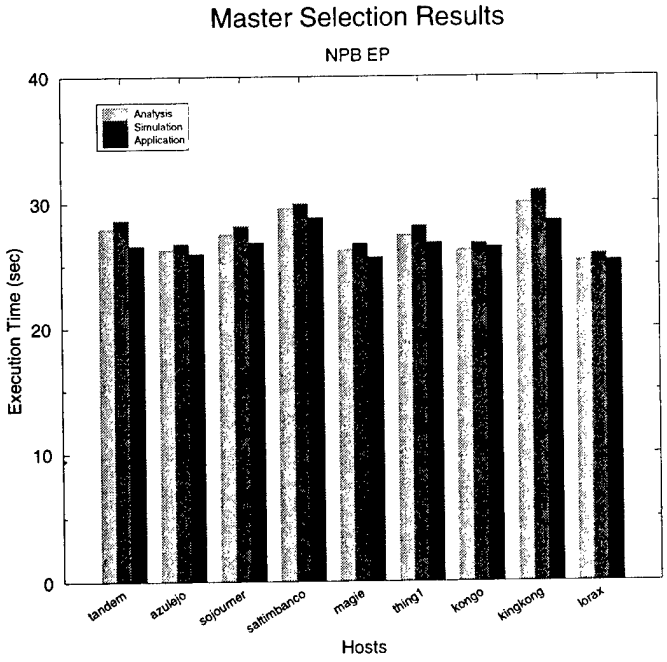


Figure 6. Execution time of computation-intensive application while varying master host.

Name	Processor	Network	OS
azulejo	Intel Pentium Pro 200	100 Mbit/s ethernet	Linux 2.0.36
kingkong	Sun UltraSPARC-IIi 333MHz	100 Mbit/s ethernet	Solaris 2.6
kongo	Sun UltraSPARC 166MHz	100 Mbit/s ethernet	Solaris 2.6
lorax	Sun microSPARC II 85MHz	100 Mbit/s ethernet	Solaris 2.6
magie	Intel Pentium Pro 200	10 Mbit/s ethernet	Linux 2.1.125
saltimbanco	Intel Pentium II-400	10 Mbit/s ethernet	Linux 2.1.125
sojourner	Intel Pentium II-266	10 Mbit/s ethernet	Linux 2.2.9
tandem	Intel Pentium II-300	100 Mbit/s ethernet	Linux 2.0.36
thing1	Sun UltraSPARC 200MHz	100 Mbit/s ethernet	Solaris 2.6

Table 4. Partial list of heterogeneous mix of machines used in experiments.

6. Related Work

Many different approaches to predicting the performance of parallel applications on distributed-memory machines have appeared in the literature. A partial summary of some earlier efforts can be found in [10]. Unfortunately, these approaches often suffered from either limited accuracy under real-world conditions (caused by making many simplifying assumptions), or from excessive complexity when either constructing or using the models. Our approach to performance prediction focuses on achieving useful levels of prediction accuracy while limiting model complexity and allowing efficient measurement and quantification of important model parameters.

The application of performance prediction to the problem of resource selection has also been addressed recently by Weissman and Zhao [17]. In their work, Weissman and Zhao use heuristics to select a number of candidate configurations, then employ cost functions to derive computation and communication times for each configuration. They then select the configuration yielding the lowest total cost. Our approach to resource selection efficiently evaluates application performance for different configurations using only simple constraint calculations.

Subhlok, Lieu and Lowekamp [15] have looked at automatically selecting processor nodes for applications running on high-speed networks. For their results, Subhlok, Lieu and Lowekamp present algorithms which allow them to automatically select nodes with three different goals: maximizing computation capacity, maximizing communi-

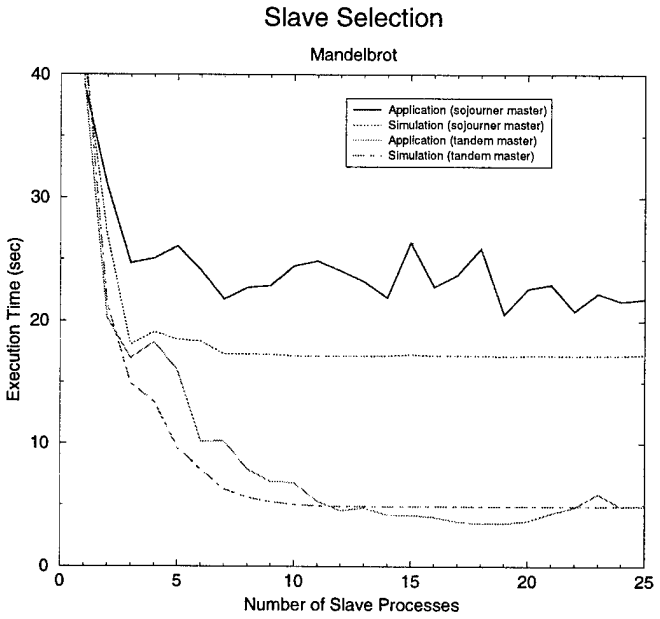


Figure 7. Application performance with varying numbers of slaves.

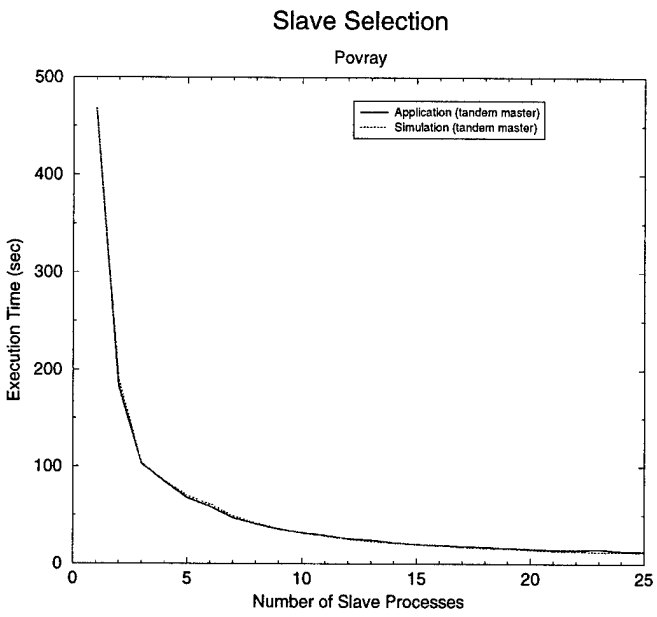


Figure 8. Application performance with varying numbers of slaves.

cation capacity, or balancing computation and communication. Their paper does not explain how the correct goal is selected to match specific application characteristics in order to give optimum performance. Our approach automatically determines performance bottlenecks based on both computation and communication constraints, and finds the best performing configuration for all cases.

7. Summary

In this paper, we have described a rate-based performance model for master/slave applications running on distributed heterogeneous processors and networks. By parameterizing this steady-state performance model with some dynamic run-time information, we are able to accurately predict maximum achievable application performance rates – even in the cases where application characteristics and resource behavior are not steady over time.

We have also described an application performance simulator which accurately simulates the dynamic interaction of a master/slave application with a defined configuration of performance constrained resources. This simulator allows for a detailed analysis of where performance bottlenecks due to resource limitations may occur in an application. This kind of detailed information about how applications interact with resources in a Grid environment can be very valuable for resource selection at application runtime, advanced application and platform planning, and program development activities. The key to our success with our performance prediction tools has been the identification of a common set of application and resource parameters which could be quantified and measured, and which captured both the static and dynamic aspects of application performance in Grid environments.

Based on the effectiveness of our performance prediction tools, we have developed algorithms for master and slave resource selection on Grid platforms. These algorithms enable the selection of a master processor and a set of slave processors which allow maximum application performance to occur. Actually achieving the maximum application performance in dynamic Grid environments may also require the use of other run-time techniques to handle issues like load balancing and fault tolerance. These are issues we are actively researching, and will be the subject of future publications.

Some brief experimental data was presented to verify that both our performance prediction tools and our strategies for selecting master and slave resources were sound. We are currently integrating the performance tools and resource selection strategies into an AppLeS [2] Grid application scheduler with the goal of providing an automatic mechanism for high-quality distributed master/slave scheduling in heterogeneous and dynamic Grid environments.

In the future, we would like to extend the work-rate-based performance model to other common classes of paral-

lel computing in Grid environments. We would also like to study whether other physical resource characteristics, such as available memory, might be beneficial to include in our constraint analyses. Our experience has shown that the idea of estimating application performance by accounting for application/resource constraints appears promising as a tool for enabling more effective application scheduling.

References

[1] F. Berman. High-performance schedulers. In I. Foster and C. Kesselman, editors, *The Grid: Blueprint for a New Computing Infrastructure*, chapter 12. Morgan Kaufmann Publishers, July 1998.

[2] F. Berman and R. Wolski. Scheduling from the perspective of the application. In *Proceedings of the Fifth IEEE Symposium on High Performance Distributed Computing*, pages 100–111, Aug. 1996.

[3] A. Clematis and A. Corana. Performance analysis of task-based algorithms on heterogeneous systems with message passing. In *Proceedings Recent Advances in Parallel Virtual Machine and Message Passing Interface, 5th European PVM/MPI Users' Group Meeting*, Sept. 1998.

[4] R. M. Cubert and P. Fishwick. *Sim++, Version 1.0.* Department of Computer and Information Science and Engineering, University of Florida, Gainesville, FL, 1995.

[5] J. R. Evans and E. Minieka. *Optimization Algorithms for Networks and Graphs*, chapter 5, pages 178–233. Marcel Dekker, Inc., second edition, 1992.

[6] K. Everaars and B. Koren. Using coordination to parallelize sparse-grid methods for 3-d cfd problems. *Parallel Computing*, 24(7):1081–1106, 1998.

[7] L. R. Ford and D. R. Fulkerson. *Flows in Networks.* Princeton University Press, Princeton, New Jersey, 1962.

[8] D. Gelernter, M. R. Jourdenais, and D. Kaminsky. Piranha scheduling: Strategies and their implementation. *International Journal of Parallel Programming*, 23(1):5–33, Feb. 1995.

[9] B. Lowekamp, N. Miller, D. Sutherland, T. Gross, P. Steenkiste, and J. Subhlok. A resource query interface for network-aware applications. In *Proceedings of Seventh International Symposium on High Performance Distributed Computing*, July 1998.

[10] W. Meira. Modeling performance of parallel programs. Technical Report 589, Computer Science Department, University of Rochester, Rochester, NY, June 1995.

[11] Persistence of vision raytracer. Persistence of Vision Development Team, 1999. http://www.povray.org/.

[12] J. Pruyne and M. Livny. Interfacing condor and PVM to harness the cycles of workstation clusters. *Future Generation Computer Systems*, 12(1):67–85, 1996.

[13] G. Shao, F. Berman, and R. Wolski. Using effective network views to promote distributed application performance. In *Proceedings of the 1999 International Conference on Parallel and Distributed Processing Techniques and Applications*, June 1999.

[14] L. M. Silva, V. Batista, P. Martins, and G. Soares. Using mobile agents for parallel processing. In *Proceedings of the International Symposium on Distributed Objects and Applications*, Sept. 1999.

[15] J. Subhlok, P. Lieu, and B. Lowekamp. Automatic node selection for high performance applications on networks. In *to appear in Proceedings of the Seventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, May 1999.

[16] A. S. Wagner, H. V. Sreekantaswamy, and S. T. Chanson. Performance models for the processor farm paradigm. *IEEE Transactions on Parallel and Distributed Systems*, 8(5):475–489, May 1997.

[17] J. B. Weissman and X. Zhao. Scheduling parallel applications in distributed networks. *Journal of Cluster Computing*, 1(1), 1998.

[18] S. M. White, A. Alund, and V. S. Sunderam. Nas parallel benchmark kernels for pvm 3. http://www.nas.nasa.gov/NAS/NPB/, Oct. 1993.

[19] R. Wolski. Dynamically forecasting network performance using the network weather service. In *Proceedings of the 6th High-Performance Distributed Computing Conference*, pages 316–325, Aug. 1997.

Gary Shao is a graduate student in the Department of Computer Science and Engineering at the University of California, San Diego. His research interests include parallel and distributed computing, adaptive scheduling, and application development environments. He received his B.S from the University of Missouri, Columbia and his M.S. from Washington University in St. Louis, Missouri.

Francine Berman is a Professor of Computer Science and Engineering at the University of California, San Diego. She is also a Senior Fellow at the San Diego Supercomputer Center, Fellow of the ACM, and founder of the Parallel Computation Laboratory at UCSD. Her research interests over the last two decades have focused on parallel and distributed computation, and in particular the areas of programming environments, tools, and models that support high-performance computing. She received her B.A. from the University of California, Los Angeles, her M.S. and Ph.D. from the University of Washington.

Rich Wolski is an Assistant Professor in the Department of Computer Science at the University of Tennessee and a partner in the National Partnership for Advanced Computational Infrastructure. His research interests include parallel and distributed computing, on-line performance analysis techniques and software, compiler runtime system, and dynamic scheduling. He received his B.S. from the California Polytechnic University, San Luis Obispo and his M.S. and Ph.D. from the University of California at Davis/Livermore Campus.

Heterogeneity as Key Feature of High Performance Computing: the PQE1 Prototype ♦

P.Palazzari[1], L.Arcipiani[1], M.Celino[1], R. Guadagni[2], A.Marongiu[1], A.Mathis[1], P.Novelli[1], V.Rosato[1]
ENEA. Casaccia Research Center – Rome
[1]HPCN Project , [2]Funzione Centrale Informatica
palazzari@casaccia.enea.it

Abstract

In this work we present the results of a project aimed at assembling an hybrid massively parallel machine, the PQE1 prototype, devoted to the simulation of complex physical models. The analysis of some of the existing parallel architectures has revealed that general-purpose machines are largely over-dimensioned and often perform inefficiently in grand-challenge scientific applications. We have thus developed an heterogeneous parallel system which matches task-heterogeneity with architecture-heterogeneity: in fact special-purpose massively parallel architectures, when coupled to general-purpose machines, are able to efficiently satisfy the requirements of complex scientific computing. We present the HW structure and the SW tools developed for the PQE1 prototype. Starting from the concept of machine-granularity and task-granularity, we show the necessity to exploit both high granularity and low granularity parallelism to efficiently use the PQE1 system. Some examples describing application fields in which the PQE1 prototype has been successfully used are briefly described.

1. Introduction

Technical applications (image processing, real-time control,...) and simulation of complex models used in scientific applications (quantum chemistry, weather forecast, electromagnetic compatibility...) require sustained computational powers of the order of tens (or hundreds) of Gflops (1Gflops = 10^9 floating point operations per second). Massively parallel processing seems to be the only practical way to reach these figures. To date, commodity off-the-shelf processors are able to provide peak performance in the range of 1÷2 Gflops (for example, the 667 MHz Alpha 21264 chip has a peak performance of 1.3 Gflops [1]): hundreds of those processors can be coupled, for instance, up to reaching the desired sustained performances.

The Accelerated Strategic Computing Initiative (ASCI, [2]) and the Path Forward project, finalized to build very powerful parallel machines to implement extremely complex simulations, have produced, to date, the installation of several general purpose platforms:

1. ASCI Red: composed by 9,216 Pentium Pro processors, has 584.5 Gbytes of RAM, bi-directional cross-section bandwidth of 51.6 Gbyte/sec and peak performance of 1.8 Tflops;
2. ASCI Blue Mountain: assembled with 48 Silicon Graphics/Cray Origin2000 servers (each is configured with 128 SMP processors) containing a total of 6,144 processors, with projected peak performance of 3 Tflops;
3. ASCI Blue Pacific: has 1,344 PowerPC 604e processors (332 MHz), 504 Gbytes of RAM, nodes connected through an Omega Network with a node-to-node bandwidth of 150MB/sec and offers a peak performance of 0.89 Tflops.

Also these platforms, like the most widespread commercial parallel systems, are based on commercial general-purpose computing devices which allow to sustain very irregular programming models. If, on one side, this property makes these systems well suited for most of the computational tasks related to complex scientific applications, on the other side this can also be considered as their main limitation. In fact, the need of being general-purpose implies that these systems are designed to support multitasking/multiuser operative environments, so that most of their silicon, instead of

♦ This work has been performed in the framework of an industrial collaboration between ENEA (The Italian Agency for New Technologies, Energy and the Environment) and QSW (Quadrics Supercomputing World Ltd., a Finmeccanica group company).

0-7695-0556-2/00 $10.00 © 2000 IEEE

being devoted to implement computing devices, is used to build cache memory and control HW to manage complex memory hierarchies, out of order execution of instructions, processor scheduling and multiprocess environment. This fact, while enhancing system operability, largely decreases the efficiency per silicon area in floating point dominated applications, being a large part of the electronic devices not operative for most of the time [3].

A completely opposite approach to high performance scientific computing can be found in the physicists community, where small research groups are used to design by themselves dedicated machines which can efficiently solve their computational problems. An example of this approach is given by the GRAPE (GRAvitational PipE) system [4]: GRAPE-4, the system version now available, is a special purpose computer for astrophysical simulations (N-body gravitational problems requiring $O(N^2)$ computations) with peak speed exceeding 1 Tflops [5]. GRAPE system is a completely not programmable machine, allowing only to load/read initial/final data into/from the machine. Extreme specialization is the key to achieve very high efficiency in the use of the silicon area: in such platforms only the required functions are implemented, thus maximizing the performances per unit of volume of electronics. The GRAPE project is going to release a 200 TFlops computer [6], yielding a computational speed from 10 to 100 times larger than that achievable, on the same problem, in the platforms developed in the framework of ASCI project.

A further example of the advantages which can be achieved through HW specialization is given by the APE project [7] launched by Italian physicists, aimed to build a massively parallel system to be used in Lattice Quantum Chromo Dynamics (LQCD). These platforms, the APE series (APE100 is the old system [8], APEmille is the new prototype which will be soon launched [9],[10]) are SIMD programmable systems equipped with up to 2048 Very Long Instruction Word (VLIW) custom processors and offering peak performances of 100Gflops (APE100 series) and 1 Tflops (the new APEmille system). In both cases, the machines in the largest configurations are easily contained in few rack-mounted containers.

In scientific computations, most of the time is usually spent in the execution of quite regular codes which iterate (e.g. in time, frequency, space) several transformations on large domains of data. In such a computational scenario, heterogeneous computing is a very promising way to achieve high performances: the key idea is to connect a (small) general-purpose parallel machine to several, very powerful, specialized parallel systems. The less flexible, specialized machines are dedicated to provide most of the computational power required by the numerical programs, while the general-purpose machine is used to give the necessary flexibility to the whole system, coordinating tasks and pre/post processing data produced by the specialized systems.

Heterogeneous computing has been used to achieve the very high performances required when dealing with challenging problems: machine heterogeneity is exploited to match task heterogeneity, using massively parallel systems as dedicated, high-efficiency boosters attached to a single user general-purpose parallel machine.

In this work we present the outcome of a scientific program aimed at developing a massively parallel hybrid machine. In the first part of the paper a theoretical framework to describe heterogeneous tasks and heterogeneous systems is presented. Task and machine granularity are introduced and their influence on the efficient implementation of heterogeneous tasks onto heterogeneous systems is discussed. Then we describe the PQE1 prototype, the massively parallel hybrid system which has been developed in our research center. Along with the description of the HW and the SW of the system, we discuss the rationale of such architecture and we sketch the results obtained in two different, successful applications of the PQE1 platform. Finally the next version of this hybrid prototype, now in its final design phase, is presented.

2. Hierarchical Modeling of Heterogeneous Tasks and Systems

An algorithm to be implemented on a parallel system can be represented as a labeled Control Data Flow Graph (CDFG) G(N,E,C_N,C_E), being

1. $N=\{n_i \mid i=1,2,\ldots,N\}$
   the set of functionality necessary to implement the algorithm,
2. $E\subseteq N\times N=\{e_{ij}=(n_i,n_j) \mid n_i \text{ sends data to } n_j\}$
   the internode communication set,
3. $C\_N=\{c\_n_i \mid c\_n_i \in \aleph,\ n_i \in N,\ i=1,2,\ldots,N\}$
   the node labeling set, containing the integer value which is a measure of the complexity of the functionality corresponding to n_i (e.g., the number of operations needed to implement n_i) and
4. $C\_E=\{c\_e_{ij} \mid c\_e_{ij} \in \aleph,\ e_{ij} \in E\}$
   the channel labeling set, containing the integer value which is a measure of the complexity of the communication corresponding to e_{ij} (e.g., the number

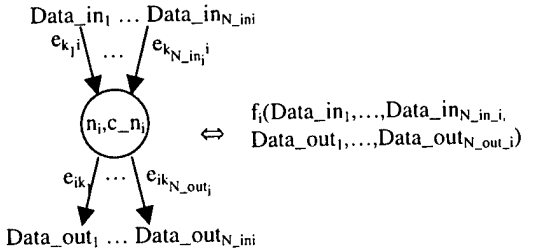


Figure 1: graph node, with input and output edges, representing the computation n_i.

of byte sent through e_{ij}).

Each node n_i in the computation is associated to a functionality which transforms N_in_i input data (with their corresponding associated data type) into N_out_i output data (with their data type). N_in_i and N_out_i are, respectively, the input and the output degrees of n_i. The correspondence between node n_i and function f_i is depicted in figure 1.

In a completely similar way, a parallel system can be represented through a labeled graph PS(R,IN,M_R,B_IN), being

1. $R=\{r_i \mid i=1,2,\ldots,r\}$
   the resource set (processing elements with their local memory, shared memory banks, I/O devices) which can be decomposed into basic sets, i.e.
   $R=\{\cup_{i=1,\ldots,k} p_i\}\cup\{\cup_{i=1,\ldots,m} M_i\}\cup\{\cup_{i=1,\ldots,t} I/O_i\}$.
   So the parallel system resource set is constituted by
   - k sets of processing elements p_i, each set p_i being characterized by the number and by the type of homogeneous computing devices contained in it;
   - m sets of memory banks M_i, each set of memory banks being characterized by the number of memory banks, by their access time and by the size of each bank given (in byte) by sizeof(m_j), $m_j \in M_i$;
   - t sets of I/O channels I/O_i, each set being characterized by the directionality, the bandwidth and the number of channels contained in it.
2. $IN \subseteq Pow(R)\times Pow(R)=\{c_{ij}=(\{r_{i1},\ldots,r_{ih}\},\{r_{j1},\ldots,r_{jn}\}) \mid \{r_{i1},\ldots,r_{ih}\}$ is connected to $\{r_{j1},\ldots,r_{jn}\}\}$
   the interconnection network set, where *Pow*(R) denotes the power set of R. *Pow*(R) is used to model shared interconnections: a set of homogeneous processors $p_i=\{p,p,\ldots,p\}$ sharing a memory bank $m \in M_i$ are represented through the couple (p_i,m); a shared bus connecting the processors of p_i is represented by the couple (p_i,p_i); a point to point connection with one dedicated channel between two not homogeneous processors is represented by the couple $(p_a \in p_i, p_b \in p_j)$;
3. $M\_R=\{m\_r_i \mid m\_r_i \in \aleph\ r_i \in R,\ i=1,2,\ldots,r\}$
   the resource labeling set, which associates to each resource a number measuring its performances (e.g. the number of flops executed per clock cycle by a general purpose processor $p \in p_i$, the number of clock cycles necessary to compute functionality *f* in the computing devices dedicated to its HW implementation, the access time for shared memory banks $m \in M_i$, the bandwidth for I/O channels $c\_i/o \in I/O_i$);
4. $B\_IN=\{b\_c_{ij} \mid b\_c_{ij} \in \aleph,\ c_{ij} \in IN\}$
   the labeling set which associates the bandwidth to each channel $c_{ij} \in IN$.

It is fundamental to underline that, in the cases of both task and parallel system graphs, each node can be modeled through another task or parallel system graph: such a hierarchical description of a graph allows to put in evidence only the degree of parallelism (and of detail) which the user wants to consider. All the lower level details are hidden at this stage of abstraction. For instance, a complex program can be represented through a CDFG in which nodes are very complex routines; after a refinement step, each routine can be detailed through several simpler routines (for instance, an iterative solver can be expressed by means of Basic Linear Algebra Subroutines (BLAS)); going on with the zooming of details, each BLAS routine can be decomposed into (dependent, i.e. interconnected) elementary operations expressed in a standard imperative language (e.g., C or Fortran). As example of hierarchical representation of a parallel system, we can think to a system graph whose nodes are large systems (Vector Computers, Distributed Memory SIMD and MIMD systems, Shared Memory Multiprocessors, DSP and specialized computing devices) connected through some kind of (eventually not homogeneous) IN. Each node can be detailed through several lower level nodes (processors of the system and their IN) which can still be detailed through a lower level representation (interconnected functional units within a processor). A sketch of this hierarchical description is

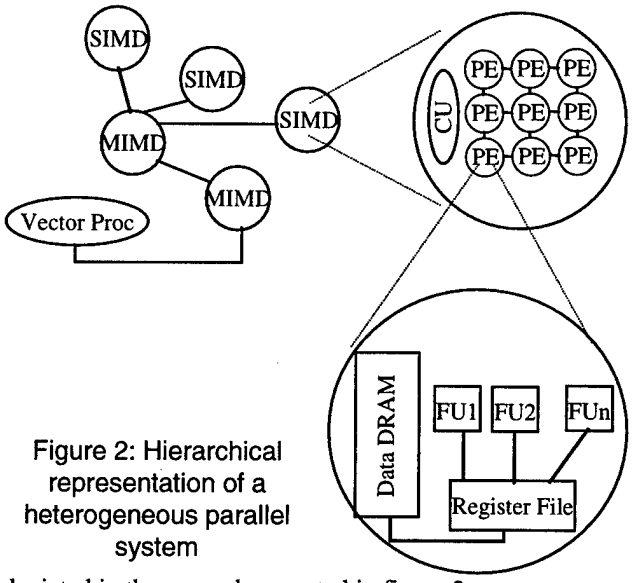


Figure 2: Hierarchical representation of a heterogeneous parallel system

depicted in the example reported in figure 2.

3. Task and Machine Granularity: Formal Definition of Heterogeneous Systems and Heterogeneous Tasks

Once introduced the formal hierarchical definitions to model computations and parallel systems, we try to give a (not exhaustive) definition of task and system heterogeneity. We need first to introduce the fundamental concepts of task and machine granularity.

The granularity of a task is usually referred to as being proportional to the ratio between the computation and the communication times involved in the execution of the task [23]. This definition of granularity is, indeed, machine dependent, as both communication and computation times may vary when the task is executed on different architectures. Being interested to a heterogeneous environment, we prefer to introduce the concepts of machine granularity (g_m) and task granularity (g_t). g_m is a measure of the balance between computational and communication speed of a system and is defined as

$$g_m = \frac{\text{node peak computation speed}}{\text{I/O bandwidth}} = \frac{PCS}{BW} \quad (1)$$

where the node peak computation speed (PCS) is the maximal number of operations per second executed by the node (usually PCS are expressed in terms of flops in the context of numerical computations). Previous definition can be applied to nodes at different hierarchical levels. Referring to figure 2, for instance, we can define the granularity of vector nodes, SIMD nodes and MIMD nodes; at this level (the system level) nodes usually have very high granularity g_m, ranging typical computation speeds from few Gflops to several tens of Gflops and typical I/O bandwidth from tens of Mbyte/sec up to few Gbyte/sec (for massively system with parallel fast I/O); a typical value for a medium-large system can be $g_m = \frac{50\times10^9}{500\times10^6} = 100$. When moving to a lower level of detail (the sub-system level), granularity of a node diminishes, as typical computation speeds of today's processing elements are in the range of few hundreds of Mflops up to 1-2 Gflops and communication bandwidths range from tens up to few hundreds of Mbyte/sec; typical value for a high-end processing element (like the Alpha EV6.7) equipped with a 64-bit PCI connection is $g_m = \frac{1.3\times10^9}{200\times10^6} = 6.5$. Moving into a lower level of detail (the processor level, inside the processing element), granularity assumes a smaller value, as communication speed is always in the range of few hundreds of Mflops up to few Gflops, while communication bandwidth (processor⇔memory) ranges from few hundreds of Mbyte/sec up to few Gbyte/sec; for instance, a processing element with an EV6.7 processor (peak speed 1.3 Gflops) and a fast chipset for the memory control (e.g the Tsunami chipset, allowing an internal memory bandwidth of 2.6Gbyte/sec) is characterized by a granularity $g_m = \frac{1.3\times10^9}{2.6\times10^9} = 0.5$.

We are now able to give the following

definition of heterogeneous system: a parallel system is heterogeneous when

1. it is composed by more than one computing element and
2. its computing elements are based on different architectural paradigms (Vector systems, Distributed Memory/Shared Memory MIMD systems, SIMD systems, etc..) and/or
3. it can be described through a hierarchical classification evidencing different node granularities throughout the hierarchical levels.

g_m is a measure of the ratio between system computation speed and system communication bandwidth. Following a similar reasoning, task granularity g_t is defined as a measure of the balancing of computational and communication requirements of a task and is defined as

$$g_t = \frac{\text{number of computing op.}}{\text{number of bytes of I/O data}} = \frac{n\_op}{n\_I/O\_byte} \quad (2)$$

The hierarchical classification approach, used to model heterogeneous tasks, can be applied also in the case of CDFGs. Given a CDFG with k different nodes, $g_t(n_i)$ is the granularity of each node ($i=1,2,..,k$; $n_i \in N$) and the granularity of the whole CDFG is the maximal value of the granularity of its composing tasks, i.e.

$$g_t(CDFG)=\max_{i=1,k}(g_t(n_i)) \quad (3)$$

Granularity of a set of nodes is defined as the largest granularity in the set because it seems to be reasonable to represent computation/communication demands of a complex task through its largest component; in fact, given for instance a CDFG with 9 different nodes with the same (small) granularity 1 and one node with (large) granularity 100, computation/communication demands are well represented by the value 100 (worst case). If we use an average value to represent the global task granularity, in previous example we would obtain $g_t=10.9$, which clearly underestimates the influence of the 'large' task, probably yielding, as we will discuss later, an inefficient implementation of the task on the parallel system.

When a hierarchical representation of a CDFG is used, the change from the procedural level (i.e. the level in which nodes represent routines) to the instruction level of detail (nodes represent elementary instruction, e.g. basic C statements) determines a decrease in the node granularity. In fact, if we indicate with n the size (in byte) of the input/output parameters, the number of operations N_OP(n) executed by the routine has, in most cases, a dependency law larger than O(n), i.e. $N\_OP(n) \geq O(n)$. $N\_OP(n)=O(n)$ is a lower bound, being O(n) the number of elementary operations necessary to read/write input data (with the obvious exception of data structures already stored in memory and communicated through a pointer; however, also in this case the following inequality (4) is satisfied). As a consequence, the law connecting the

granularity of a task to the size n of input/output data is given by

$$g_t(n) = \frac{N\_OP(n)}{O(n)} \geq O(1) \quad (4)$$

At the instruction level the granularity is O(1), i.e. the number of bytes used to encode input and output parameters of one operation is a constant number (with very few, and particular, exceptions involving data movement), because elementary operations manipulate one or two scalar values and return another scalar value. As a consequence, when moving from the procedural to the instruction level, task granularity does not increase (typically diminishes).

Other parameters characterizing nodes of CDFG, at the procedural level, are

- the type T∈{'control-dominated', 'computation-dominated'}; a node is control dominated when has small granularity and contains a number of decision operations (i.e. conditional jumps) significantly larger than the computing operations; on the contrary, a node is computation dominated when has large granularity,
- computational paradigm; each node of the graph, when expressed at a lower level of detail, can be represented by means of the 'data-parallel', the 'pipeline', the 'farm', the 'loop', the 'unrestricted' structuring constructs; for the description of the structuring constructs, except the 'unrestricted', see [29]; the 'unrestricted' paradigm refers to a generic computation represented by means of an irregular CDFG.

We are now able to give the following

definition of heterogeneous task: a CDFG represents an heterogeneous task when

1. it is composed by more than one node at the 'procedural' level and
2. its nodes are based on different computational paradigms or have different types T and/or
3. its nodes have different granularities

4. Matching Task and System Heterogeneity to Maximize System Performances

Once fixed the meaning of heterogeneity for tasks and systems, it is important to evaluate their mutual relation and to describe the associated heterogeneity parameters (granularity, computational/architectural paradigms, node type T). In this framework, it is worth investigating the connections among system/task granularity, heterogeneity and global performances.

The granularity G, in its classical form, is defined as the ratio between Run time (R) and Communication time (C) of a given task [28], i.e.

$$G = \frac{\text{task execution time}}{\text{task communication time}} = \frac{R}{C} \quad (5)$$

In order to avoid a too formal explanation, far beyond the scope of this paper, we do not go into the details necessary to define task execution and communication times; intuitively, we consider as execution (communication) time the summation of all the time intervals in which at least one computational unit (I/O channel) is computing (communicating).

Previous definition of granularity is machine dependent, being execution and communication times connected to processor speeds and I/O bandwidths. The previously introduced definitions of g_m and g_t can be used to make explicit this dependence; in fact G can be expressed as

$$G = \eta \frac{g_t}{g_m} \quad (6)$$

being $\eta = \frac{\eta_{proc}}{\eta_{comm}}$ an efficiency figure which takes into account the partial utilization of the processor speed (η_{proc}) and of the bandwidth (η_{comm}). In order to verify the validity of (6), it is sufficient to substitute in it the expressions of g_t and g_m and, with few algebraic operations, we obtain

$$G = \frac{\eta_{proc} g_t}{\eta_{comm} g_m} = \frac{\eta_{proc} \frac{n\_op}{n\_I/O\_byte}}{\eta_{comm} \frac{PCS}{BW}} =$$

$$= \eta_{proc} \frac{n\_op}{PCS} \frac{1}{\eta_{comm} \frac{n\_I/O\_byte}{BW}} = \frac{R}{C}$$

The actual value of η depends on the characteristics of tasks and on their implementation on the physical system. A reasonable estimation, to be confirmed through some experiences on a given system, is η=0.1÷0.5. For instance, when dealing with tasks with large I/O packets (small granularity), usually communication startup time is negligible and $\eta_{comm} \cong 1$; in such a case $\eta = \eta_{proc}$ and processor utilization in the range from 20% up to 60% is a realistic figure.

The expression of G as ratio between task and machine granularity underlines how the relative values of task and machine granularity are relevant to achieve high performances when implementing the task on an actual (heterogeneous) parallel system. Efficient task implementation requires to match two conflicting behaviors: that of a task with maximum parallelism (to minimize execution time) with the constraint of minimizing communication costs (overheads). The only information on the granularity G is not sufficient to determine if the implementation of the task on a machine

is efficient: G gives just a measure of the relative influence of communication overhead on system performances. Given an implementation of a task on a parallel system, efficiency reaches its maximum when, for a fixed degree of parallelism, communication overhead is minimum. In fact, indicating with R the time spent executing computations and with C the time spent in communications, and defining as efficiency the ratio

$$\text{Eff} = \frac{\text{R}}{\text{Actual Computing Time}} \quad (7)$$

where the Actual Computing Time is the elapsed time from the start of the parallel program till its end, the following inequalities hold:

$$\text{Eff}_{\text{Min}} = \frac{\text{R}}{\text{R}+\text{C}} \le \text{Eff} \le \frac{\text{R}}{\text{MAX}(\text{R},\text{C})} = \text{Eff}_{\text{Max}} \quad (8)$$

which can be rewritten, introducing the granularity G, as

$$\text{Eff}_{\text{Min}} = \frac{1}{1+\frac{1}{\text{G}}} \le \text{Eff} \le \frac{1}{\text{MAX}(1,\frac{1}{\text{G}})} = \text{Eff}_{\text{Max}} \quad (9)$$

It is worthwhile to note that the fraction of unused computational resources is given by (1-Eff). The lowest value for the efficiency (Eff_{Min}) corresponds to the complete absence of overlapping between computation and communications; the highest value for the efficiency (Eff_{Max}) corresponds to a complete overlapping between computations and communications. In figure 3 the values Eff_{Min} anf Eff_{Max} are sketched as function of the granularity G.

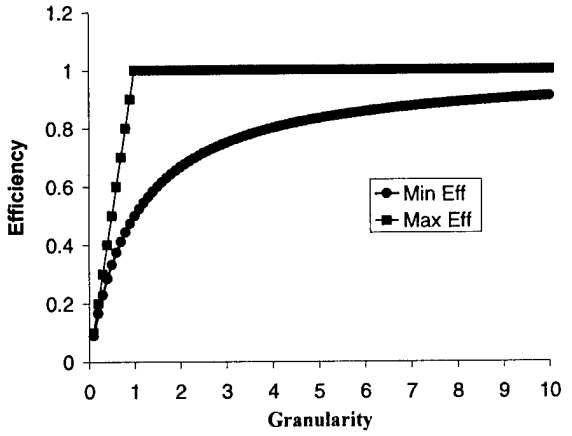


Figure 3: Minimum and maximum efficiency values vs Granularity

Actual efficiency values lie within the two plots, being closer to the lower or to the higher depending on the algorithm structure and the HW support for computation and communication overlapping (number of DMA channels, routing processors).

G=1 indicates equality between computation and communication times. The larger is G, the more negligible is the communication time with respect to computing time. Values of G<1 originate I/O bound problems.

In order to avoid situations with processors stalling due to I/O operations, with a consequent strong decreasing of efficiency, granularity of the task should be greater of a certain value G_0 so that efficiency results greater than the minimum acceptable threshold Eff_T. If the not overlapping model is assumed, the granularity must respect the following inequality in order to have $\text{Eff}>\text{Eff}_T$,

$$G > \frac{\text{Eff}_T}{1-\text{Eff}_T} \quad (10)$$

and, evidencing dependence of G on g_t and g_m, we obtain

$$\frac{g_t}{g_m} > \frac{\text{Eff}_T}{1-\text{Eff}_T}\cdot\frac{1}{\eta} \quad (11)$$

From previous inequality, setting $k = \frac{\text{Eff}_T}{1-\text{Eff}_T}\cdot\frac{1}{\eta}$, we obtain a fundamental relation between task and machine granularity:

$$\boxed{g_t > k \cdot g_m} \quad (12)$$

In the case of perfect overlapping between computation and communications, it is easy to verify that expression (12), from the position G>1, becomes

$$\boxed{g_t > \frac{1}{\eta}\cdot g_m} \quad (13)$$

Previous expressions ensure a correct implementation of tasks on heterogeneous systems. The value k has to be estimated on the basis of 'reasonable' assumptions about the degree of overlapping between computations and communications (expression of k has been determined assuming the worst case, with no overlapping) and about the efficiency η which can be achieved when implementing the task on the system.

The scheme to allocate a heterogeneous task onto a heterogeneous system is the following:

- Consider the highest levels of detail both for the system and the task graphs;
- Ordinate the node tasks in descending order of computational complexity;
- For each node in the task graph, chosen according to previous decreasing ordering, select the system nodes which match, with their architectural paradigms, the node computational paradigm;
- Among all the candidate system nodes, choose the one which has the highest computational power and respects the relation $g_t > k \cdot g_m$; as the choice of the system depends on k, i.e. on the efficiency of the implementation of the task node on the system node, the process can be iterated at a lower level of detail

(i.e. the node is expanded (if possible) into a smaller granularity CDFG and also the system node is considered at a lower level of detail) until a reasonable estimation for k is achieved.
- Assign the task node to the system node found in previous point (the choice of the system with the highest computational power allows to satisfy the tasks with highest computing requirements).

As the previous 'recipe' does not consider the load balancing, some policy must be chosen to avoid the overloading of the most powerful systems; a method could be based on a cyclic allocation policy or on some dynamic updating of system performances (as a system node becomes more loaded, its computational speed appear smaller to the other task nodes that must still be allocated). In order to take into account precedence relations among nodes in the CDFG, techniques discussed in [23], [35] can be used.

5. The Heterogeneous PQE1 System

The previous discussion is aimed at stressing that a heterogeneous system is not a mere collection of several platforms used, sometimes, as a parallel system, but it is an integrated system that must be designed from scratch to behave as a heterogeneous parallel system. In fact, heterogeneity is a property of the problems to be solved. A 'well balanced' heterogeneous system will thus provide the best way to solve complex 'real' problems. Heterogeneity moreover, avoids to over-dimension a parallel system, as the computational power is 'dedicated' (according to several computational paradigms), allowing very high efficiency. The idea is to avoid, as much as possible, the use of general purpose systems just because they perform 'quite well' in all the problems but not 'very well' for any problem. On the contrary, heterogeneous systems could contain different 'dedicated' parallel systems, some of which very well suited for a certain class of problems, others for others different classes. In this way, in principle, it would be possible to have a system which often behaves 'very well' on a lot of problems, because different parts of a complex application could be efficiently implemented onto architecturally different parts of the system. Furthermore, on the basis of the previous analysis, specialized architectures are, often, less costly (in terms of silicon area, power consumption, volume) than general purpose systems.

5.1 Rationale for the PQE1 prototype

General-purpose parallel machines support the Single Program Multiple Data (SPMD) asynchronous programming paradigm. Their HW structure is inherently asynchronous and some silicon area, other than some time, must be wasted to manage process synchronization and asynchronous communications. Such a wasting of resources can be avoided by using synchronous machines to which could be efficiently allotted computational tasks requiring synchronous algorithms.

On the basis of the experience gained using SIMD systems in several fields of technical-scientific computing (material science [14][15], astrophysics [40], atmospheric modeling [16], image processing and compression [17][18], computational electromagnetic [19][20], linear algebra [21], neural networks [22]) we are convinced that SIMD architectural paradigm can efficiently express programs solving problems related to such fields. Moreover it is also preferable to the MIMD paradigm because many algorithms

1. are synchronous;
2. often require that all the processors execute the same instructions on different domains;
3. need interprocessor communications executed in a synchronous way;
4. do not need deep memory hierarchies thanks to the regular patterns of memory accesses.

Point 1 and 3 show that, for such classes of algorithms, the time spent in synchronization phases, required by MIMD systems, is a completely unnecessary overhead introduced by the asynchronous HW structure of the machine. This overhead is not required by SIMD machines with synchronous communications. Point 2 shows that all the HW dedicated to manage different program flows in the processors is unnecessary, being sufficient one centralized controller of program flow. Point 4 means that cache memory and the related management policies are not needed in most scientific applications, being the 'locality' of the problem [30] easily controlled by the programmer through instructions of vector movements between main memory and an internal register file. Although cache memory results to be particularly useful in multi-programmed environments, where several processes are running and the fast memory is not large enough to keep the whole image of all the running processes, in most cases of scientific computing only one process is running and its locality is easily captured by the programmer through instructions which allow burst memory transfers, through DMA channels, between the slow external RAM and a fast internal register file (or a multi-port/multi-bank internal static memory). A further discussion on SIMD vs MIMD architectures, along with a description of SIMD/MIMD mixed mode systems, is reported in [27].

5.2 HW description of the PQE1 prototype

The PQE1 is an 'hybrid' MIMD-SIMD platform where the flexibility and operability of a MIMD (distributed memory) architecture (the eight node Meiko/QSW CS-2) are coupled to the power and efficiency of SIMD

machines (7 APE100/Quadrics systems) which enable to efficiently perform in small granularity tasks.

If we take into account the 4 points listed above and we assume that most algorithms arising in scientific applications can be expressed through synchronous programs with synchronous communications, executing the same instruction on a set of different data which can be easily mapped onto a data parallel structure with regular patterns of memory access, it results very reasonable to allot those parts of the computation to the SIMD machine APE100/Quadrics, leaving the remaining tasks of the computation to be executed on the MIMD part.

We used 7 APE100/Quadrics machines, built in 1994: two with 512 processors arranged as an (8x8x8) 3D torus and 5 with 128 processors arranged as an (8x4x4) 3D torus. Each computing node is based on a custom VLIW processor, has clock frequency f_{ck}=25 MHz and is able to terminate a 'normal operation' AxB+C every clock cycle, so each processor executes two floating point operations in one clock cycle (when the pipeline is full) and has a peak speed of 50 Mflops; floating point are represented according to the IEEE 754 standard (single precision). Each node is connected to a data memory of 4Mbytes and has an internal register file (RF) with 128 registers; each clock cycle the processor is able to read two operands from RF and write one result to RF. Communications with other adjacent nodes, connected in the north, south, east, west, up and down directions are synchronous and memory mapped; interprocessor communication bandwidth is 12.5 Mbyte/sec, so the 512 (128) processor configuration has an aggregate bandwidth of 6.4 (1.6) Gbyte/sec and a peak speed of 25.6 (6.4) Gflops.

The connection of the APE100/Quadrics machines to a MIMD system, the Meiko/QSW CS-2 [24], has been performed to give more flexibility to the SIMD machines. Each node of the MIMD platform is based on two Ultra Sparc processors, connected in the SMP configuration, it offers a peak speed of 180 Mflops and has 128 Mbytes of RAM. The connection between CS-2 nodes and APE100/Quadrics systems is implemented through an HiPPI (High Performance Parallel Interface) channel, which provides a bandwidth of 20 Mbyte/sec. The connection among the nodes of the CS-2 machine takes place via the Meiko/QSW proprietary network based on the ASIC circuits Elan/Elite and implementing a multistage interconnection network with Fat Tree topology and point-to-point bandwidth of 100 Mbyte/sec. The scheme of the complete prototype is shown in Fig.4.

The PQE1 hybrid systems is thus composed by 7 SIMD machines which allow to obtain an aggregate computational speed of 83.2 Gflops, 20.8 Gbyte/sec of bandwidth and 6.5 Gbytes of RAM. These parallel systems communicate through 7 HiPPI channels with a CS-2 machines, so the communication bandwidth between the two systems is 140 Mbytes/sec. The CS-2 MIMD parallel system has 8 twin nodes, offers a peak speed of 1Gflops, has 1 Gbyte of RAM and has an aggregate bandwidth of 800 Mbyte/sec.

Looking at previous data, it is clear that the machine is strongly unbalanced, having the most of computational and communication speed in the SIMD part. If we analyze the sub-unit composed by a CS-2 node and the attached SIMD machine, seen as co-processing system, the

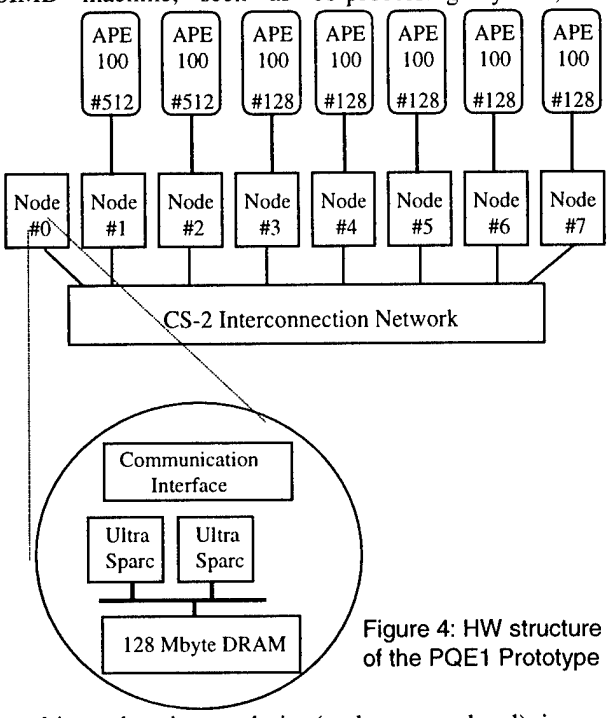


Figure 4: HW structure of the PQE1 Prototype

resulting sub-unit granularity (at the system level) is

$$g_m(APE100\text{–}\#512) = \frac{25.6 \cdot 10^9}{20 \cdot 10^6} = 1280 \quad (14)$$

and

$$g_m(APE100\text{–}\#128) = 320 \quad (15)$$

The PQE1 system can be considered, at a first level, as a parallel machine with small parallelism (parallelism degree is 7). In order to avoid wasting of performances, at this level of parallelization we have to consider only very large granularity tasks. If we consider, for instance, the product of two (n x n) single precision matrices, the task granularity is given by $g_t = \frac{2n^3}{12n^2} = \frac{n}{6}$ ($2n^3$ is the number of operations required, while $12n^2$ is the number of byte to transfer, being necessary reading the two input matrices and writing the result matrix). In order to avoid I/O bound behavior, $\eta g_t > g_m$ must result; in the case of a 128 processor machine, supposing η=0.5 (reasonable value for this type of computations, using sequences of not independent operations of the type AxB+C), this corresponds to the condition n>3840.

The second level of parallelism can be exploited within the single task. The SIMD machine has granularity (at the sub-system level) is

$$g_m = \frac{25.6}{6.4} = \frac{6.4}{1.6} = 4 \quad (16)$$

In this case we have a lot of parallelism available (parallelism degree is 128 or 512) and we can deal with small granularity tasks.

As stated above, the rationale for such a strong machine imbalance is that SIMD systems are very well suited to implement numerical computations, allowing to reach very high sustained performances. The MIMD nodes are not devoted to solve the 'number crunching' part of the problem, but to perform data pre/post processing and to allow communications among different algorithms implemented on the SIMD systems. We underline that typical sustained performances obtained on the APE100/Quadrics machines range from 30% to 70% of the peak performances, i.e. they vary from 7.7 to 18 Gflops on the 512 node machines.

5.3 SW description of the PQE1 prototype

The basic modality to program PQE1 system is the using of a message passing paradigm (the MPI library) to manage the high granularity tasks allocated into the MIMD part. In order to allow a low-level interaction between CS-2 nodes and SIMD machines, a communication library has been devised and implemented. This library contains a set of commands to load/run programs into the SIMD machines, to synchronize the execution between the program running on the MIMD node and the program running on the connected SIMD system, to communicate data to/from the SIMD system. Due to the large granularity of the programs running on the SIMD nodes, no particular effort has been spent to reduce start-up times which, for all the operations, are in the order of 10 ms.

As the MIMD system is devoted to manage the whole hybrid system and to increase the flexibility of the PQE1 platform, a library implementing the functionality of a Distributed Virtual Shared Memory (DVSM) was developed [25]. This library allows to declare physically distributed memory areas as 'shared', thus allowing the user to operate on such areas with the usual operations of locking/unlocking and implementing atomic instructions to perform blocking/non-blocking read/write operations with synchronized/unsynchronized access. Typical times for locking (unlocking) an area are 60 (45) μs; the time necessary to access in writing (reading) a page is 19 (50) μs. Previous times do not depend on the size of the memory area.

A further tool, called SkIE-CL [26], has been devised and implemented to improve the programmability of PQE1 system, is a skeleton based coordination language which allows to express task/data parallelism through some predefined schemes (pipeline, farm, map, loop). Once the program has been written through the available parallel constructs, SkIE-CL is able to generate MPI code to program the MIMD part of the machine, performing a (near)-optimal mapping of tasks on the MIMD part of the system, by using some analytical model of the constructs; furthermore SkIE-CL allows to control the SIMD systems by means of the communication library described above.

Previous tools (the DVSM and the SkIE-CL) were jointly developed by QSW and the Information Science Department of University of Pisa.

Two interesting applications using PQE1 prototype features, i.e. overlapping computations between the SIMD and the MIMD parts of the system can be found in [21] and [16]. The first refers to the implementation of Basic Linear Algebra Subroutines-3 on the SIMD part of the system. The MIMD connections are used to perform a block-based partitioned matrix-matrix product, being the sub-blocks products distributed among several SIMD machines. The second work is related to the implementation of a high resolution meteorological limited area model coupled with an ocean model for the prediction of the state of the Mediterranean Sea and of high water events in the Venice Lagoon. The code was parallelized by allotting the computation of the most time consuming models (the High Resolution and the Very High Resolution Limited Area Models) to the SIMD part and the resolution of the less intensive computing spectral wave model (WAM) to the MIMD nodes. To these nodes is also demanded the computation of the two dimensional model (POM) for the prevision of the Adriatic Sea circulation and, ultimately, the finite elements shallow water model of the Venice Lagoon.

In the following two paragraphs we give some details on the implementations and the results achieved when using the PQE1 system to perform n-body gravitational computations and electromagnetic simulations.

5.4 n-body computations

The PQE1 architecture has been recently used for performing n-body ($O(N^2)$) calculations to study the dynamic behavior of a galactic globular cluster hosting a massive object (black hole) in its center [40]. Calculations have been carried out by exploiting a double level of parallelism which can be attained with the machine: the first, related to the SIMD parallelization of the $O(N^2)$ loop, was obtained by partitioning the stellar positions among the different nodes and by allotting the force calculations on the given partition to the single SIMD node. The hypersystolic loop ([36],[37)] has been successfully used to reduce communication times within the force loop calculation. The second level of parallelism has been exploited by using the MIMD resources to

evaluate the black hole-stars interactions (O(N) loop) during the time spent by the SIMD part to evaluate the interstellar interactions. The concurrent use of both the MIMD and the SIMD parts allowed to perform the integration of one reference time (crossing time) of a system of N=128000 stars in a CPU time of the order of t=72500 sec (with the SIMD part constituted by a platform with 512 nodes).

5.5 Electromagnetic simulation

We investigated the simulation of dynamic evolution of electromagnetic fields through the integration of Maxwell equations by means of the Finite Difference in the Time Domain (FDTD) scheme. A domain with $(n \times n \times n)$ cells was considered. Simulating one period of the input signal requires N_s time steps. At the end of each period of the simulation (i.e. at simulation time $n+N_s$, $n=0,1,\ldots$) in each cell the value

$$E_{max}(i,j,k) = \max_{t=1,\ldots,N_s} \left(E_{i,j,k}^{n+t}\right)$$

is computed. These maximal values are then sub-sampled with step s and communicated to the host to be post-processed (for example reordered, normalized and stored). In order to simulate an EM phenomenon with frequency f=1.9 GHz on a domain with $(n \times n \times n)$ cells, we have chosen spatial discretization Δ=1.5 cm and temporal discretization $\Delta t=2.88\times10^{-11}$ [sec] to avoid numerical and modal dispersion, so N_s=19. The number of computations executed in one period is

$$Nflops = N_s \times 36 \times n^3 = 684n^3 \quad (17)$$

Setting the sub-sampling step s=5 (i.e. two samples for wavelength are saved), at the end of each period the number of bytes to be sent is given by

$$Nbytes = 4\times\left(\frac{n}{s}\right)^3 = \frac{4}{125}n^3 \quad (18)$$

According to (2), task granularity is given by

$$g_t = \frac{684n^3}{\frac{4}{125}n^3} = 21675 \quad (19)$$

From (6), (14) and (19), assuming an efficiency in the implementation η=0.2, we obtain the granularity value for the EM simulation executed on the 512 processor APE100 system

$$G(APE\text{-}\#512) = \frac{\eta \cdot g_t}{g_m} = \frac{0.2\times21675}{1280} \cong 3.4$$

and, from (6), (15) and (19) the granularity value for the execution on the 128 processor APE100 system

$$G(APE\text{-}\#128) = \frac{\eta \cdot g_t}{g_m} = \frac{0.2\times21675}{320} \cong 13.5$$

Resulting G>1 in both previous cases, the simulation of one period of the EM phenomenon and the communication of sub-sampled results does not originate an I/O bound problem.

The second level of parallelism can be exploited within the single FDTD task. In this case we have a lot of parallelism available (parallelism degree is 128 or 512) and we can deal with small granularity tasks. For example, going inside the structure of the parallel FDTD simulation (described in [20]), $36(n_c)^3$ is the number of floating point operations executed in one time step within a processor where $(n_c \times n_c \times n_c)$ cells have been allocated and $2\times6\times4(n_c)^2$ is the number of bytes to communicate at each time step (3 faces with two of the E_x, E_y, E_z components (depending on the face) and 3 faces with two of the H_x, H_y, H_z, components must be communicated); in such a case task granularity is given by

$$g_t = \frac{36(n_c)^3}{48(n_c)^2} = \frac{3}{4}n_c \quad (20)$$

In order to avoid an I/O bound problem, being in the case in which the overlapping between communications and computations is allowed, $g_t > \frac{1}{\eta}\cdot g_m$ must result (eq(13)); from(13), (16) and (20) we derive the condition $\frac{3}{4}n_c > \frac{1}{0.2}4 \Rightarrow n_c \geq \left\lceil\frac{80}{3}\right\rceil \cong 27$ which gives the linear dimensions of the sub-domain in which the global simulation domain is partitioned.

Performances achieved in EM simulations were close to the value η=0.1, which corresponds to sustained performances of 2.5 Gflops when using the PQE1 system with one 512 node SIMD machine. This quite low figure is due to the Absorbing Boundary Conditions (ABC), not discussed above, which present a very low degree of small granularity parallelism, thus diminishing the global performances of the system.

6. Next generation of the PQE1 hybrid prototype

The very interesting results obtained with the hybrid PQE1 prototype confirmed the validity of the approach of coupling specialized massively parallel systems to general purpose parallel machines. The PQE1 prototype, presented in this work, is based on HW of a previous technological generation: we are thus planning to design a new system with up-to-date components. A next system is planned and will be based on several images of the new APEmille SIMD parallel machine. The MIMD part will be constituted, according with recent trends in parallel computing with large granularity systems, by a Linux cluster connected through a proprietary fast

interconnection network. Furthermore, the new prototype will allow the insertion of ad hoc designed specialized systems, based on programmable HW (e.g. FPGA).

One of the main novelty of the next generation prototype, along with its technological improvements which put it in the very high-end section of today supercomputers, relies on the possibility to apply and test methodologies derived from the HW/SW co-design field. In fact, the capability to implement on programmable HW some specific classes of algorithms will allow, at compile time, on the basis of some cost criteria, the choice between SW or HW implementation of some nodes in the CDFG specifying the application behavior.

6.1 The APEmille system

APEmille, being the evolution of the Quadrics/APE100 system, is a SIMD machine. The first prototypes have been built in 1999. Similarly to APE100, APEmille has a 3D toroidal topology and uses custom VLIW processors. Each processor, working at a clock frequency of 66 MHz, at every clock cycle is able to terminate a 'normal operation' A×B+C on complex numbers. As executing 8 floating point operations per cycle, the peak performance of an APEmille processor is equal to 528 Mflops.

Each node has an internal register file with 512 locations at 32 bits and is equipped with 32 Mbytes of Synchronous DRAM which can be accessed with a bandwidth of 528 Mbyte/sec, thus resulting in a node granularity, at the processor level, $g_m=1$.

Each node can access memory of its neighbors in the 3 spatial directions with a bandwidth of 66Mbyte/sec, so the granularity of APEmille machine with p processors, at the sub-system level, is $g_m = \frac{p \cdot 528\times10^6}{p \cdot 66\times10^6} = 8$.

The I/O is based on the use of one PCI channel for each cluster of 32 computing nodes, thus resulting in a granularity, at the system level, $g_m = \frac{32 \cdot 528\times10^6}{100\times10^6} \cong 170$, being 100 MByte/sec the actual bandwidth measured on the PCI channel.

The largest configuration of the APEmille is constituted by 2048 nodes, yielding a peak speed exceeding 1 Tflops.

Other than the improvements in processor and memory access speeds, APEmille differs from APE100 because double precision and integer operations are provided in the computing nodes.

6.2 The MIMD system

Following the evolution of high-end commodity processors, as computing core the ALPHA EV6.7 has been chosen because of its high computational performances (1.3 Gflops).

The MIMD system will be based on 16 nodes, each equipped with 1 Gbyte of DRAM. The nodes are constituted by two EV6.7 processors connected in SMP configuration. Internal memory bandwidth is 2.6 Gbyte/sec, so the granularity at the node level, is $g_m = \frac{2 \cdot 1.3\times10^9}{2.6\times10^9} = 1$.

Interconnection network uses the QsNet, based on the Elan III network adapter and the Elite III switch. QsNet [31] has a fat-tree topology, as shown in figure 5 for a 128 node system, and offers a remote access latency of 2.5 μsec and a bandwidth of 210 Mbyte/sec. For a system with p nodes, granularity at the interval level is $g_m = \frac{p \cdot 2.6\times10^9}{p \cdot 210\times10^6} \cong 12.4$. Granularity at the system level has the same value, because both interprocessor communications and I/O operations are limited by the PCI speed.

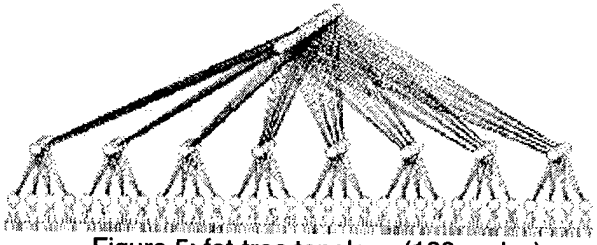


Figure 5: fat-tree topology (128 nodes)

An interesting comparison enlightening the better performances of QsNet with respect to the Gigabit Ethernet and Myrinet networks are reported in [41], where the MPI measured latency and bandwidth are given. In Table 1 we summarize such values.

Table 1: Network Comparisons

Network	Latency (μs)	Bandwidth (MB/s)
Fast Ethernet	50	12.5
Gigabit Ethernet	15	125
Myrinet	20	62
QsNet	5	200

6.3 Specialized system design

In order to design specialized HW systems, we have developed a High Level Synthesis (HLS) methodology which, starting from a high level description of an affine iterative algorithm, allows its automatic hardware synthesis; theoretical basis of this approach can be found in [38],[39]. The HLS methodology is based on a sequence of steps which transform the high level description into

several lower level representations, until reaching the hardware implementation (described through a Hardware Description Language). Each transformation step is correct-by-construction, i.e. it preserve application semantics allowing the automatic implementation of the HLS methodology. In order to ensure the generation of correct-by-construction transformation steps, the algorithm high level description is given through a mathematical model of computation. In such a way each transformation step is mathematically proved to be correct.

The chosen model of computation is the System of Affine Recurrence Equations (SARE) ([32] [33] [34]) which is one of the most promising model of computation in such fields arising in signal and image processing, linear algebra, scientific computing. SARE computational model allows the specification of an algorithm by means of recurrence equations.

7. Conclusions

In this work a brief review of the supercomputer scenario has been presented, discussing advantages of custom vs commodity system implementation.

Some theoretical aspects involved in heterogeneous system design and management have been introduced. Particular emphasis has been devoted to definition and discussions of task and system granularity. After underlying impact of a correct matching between task/system granularity, they were presented some results obtained in a scientific project aimed to exploit the advantages connected both to heterogeneity and to the use of custom parallel architectures. The outcome of this project was the PQE1 hybrid parallel system. After a brief description of its HW and SW environment, some examples of its use in several application domains have been reported (simulation of the sea level in the Venice lagoon, of the dynamic of galactic globular cluster, of electromagnetic field evolution).

Finally, on the basis of the experience gained while developing this project, the HW/SW architecture of a next hybrid parallel prototype has been shortly presented.

Acknowledgments

The authors acknowledge the fundamental role played by the Italian Project PQE2000 (which groups INFN, ENEA, CNR and QSW) for having triggered the idea of the PQE1 platform and for the collaboration during the course of the project. It should be emphasized the preminent role of M. Vanneschi, F. Baiardi, D. Guerri, M. Danelutto and S. Pelagatti (University of Pisa) in the realization of most of the SW structures (DVSM, SkIE-CL) which are supported by the PQE1 architecture and constitute its relevant assets. The role played by the QSW staff (R. Marega, B. Bacci, R. Castino, L. Di Iulio, S. Pratesi, D. Rowet, A. Scippa, R. Simonazzi) in the platform realization is also acknowledged. The authors are also indebted to the staff of ENEA Funzione Centrale Informatica (INFO) for its constant technical support throughout the course of the project.

8. References

[1] ALPHA 21264 Microprocessor Hardware Reference Manual

[2] Accelerated Strategic Computing Initiative – ASCI – URL: http://www.sandia.gov/ASCI/

[3] Mathis, A.: Technologies for Teracomputing: a European Option. Para98 -Workshop On Applied Parallel Computing In Large Scale Scientific and Industrial Problems - Umea, Sweden June, 14-16, 1998

[4] The GRAPE project. URL: http://grape.c.u-tokyo.ac.jp/grape/

[5] Makino, J., Taiji, M.: Astrophysical N-body simulation on GRAPE-4 special purpose computer. Proceedings of Supercomputing 1995

[6] Makino, J.: Stellar dynamics on 200 Tflops special purpose computers. Proceedings of the International Symposium on Supercomputing (1997)

[7] APE: The Italian SIMD supercomputer in the teraflop range. URL http://chimera.roma1.infn.it/ape.html

[8] Battista, C. et al.: The APE100 Computer: (I) the Architecture. Int. Journal of High Speed Computing n. 5 –1993

[9] Bartoloni, A. et al.: The new wave of the APE Project: APEmille. Nucl. Phys. B, n. 42 – 1995

[10] Tripiccione, R. APEmille. Parallel Computing, vol. 25, n. 10-11, Oct. 1999, Special Issue: High performance computing in lattice QCD.

[11] PQE1 prototype description - URL: http://www.pqe2000.enea.it/home/pqe1/PQE1_a.html

[12] Vanneschi, M.: The PQE2000 Project on General Purpose Massively Parallel Systems". Alta Frequenza, IEEE. November 1996.

[13] Vanneschi, M.: PQE2000:HPC tools for industrial applications IEEE Concurrency, Vol 6, n.4, Oct-dec. 1998

[14] Pucello, N., Rosati, M., Celino, M., D'Agostino, G., Pisacane, F., Rosato,V.: Search of molecular ground state via genetic algorithm; implementation on a hybrid SIMD-MIMD platform, Lecture Notes in Computer Science; A.Bode, J.Dongarra, T.Ludwig, V.Sunderam Eds. (Springer) 1996

[15] Pucello, N., Celino, M., Rosato,V.: SuperComputing Application to Materials Science Engineering Proc. SIMAI 98 Conference, Giardini Naxos; Messina, Italy (1-5 June 1998)

[16] Nicastro, S., Valentinotti, F.: An Atmosphere-Ocean Forecast System on a Hybrid Architecture. Proceedings of the Euromicro International Workshop on Parallel and Distributed Computing PDP 99. Madeira (Spain), 1999.

[17] Valentinotti, F., Taraglio, S.: Phase Difference Stereo Disparity Computation on a SIMD Parallel Machine. Lecture Notes in Computer Science N.1225. Proceedings of High Performance Computing and Networking – Europe, HPCN'97, Vienna, Austria, April 1997.

[18] Palazzari, P., Coli, M., Lulli, G.: Massively parallel processing approach to fractal image compression with near-optimal coefficient quantization. Journal of Systems Architecture vol 45, n. 10, April 1999 pp. 765-779

[19] Palazzari, P., D'Atanasio, P., Ragusini, F.: Simulation of Patch Array Antennas by Parallel Finite-Difference Time-Domain Algorithm. Proceedings of the High Performance Computing and Networking – Europe (HPCN Europe 98), April 21st-23rd, 1998, Amsterdam (Nederland).
[20] Palazzari, P., Adda, S., D'Atanasio, P.: A Tool for the Simulation of Electromagnetic Field Dynamic in Complex Environments through Massively Parallel Systems. 13th European Simulation Multiconference (ESM'99), Warsaw, Poland, June 1-4, 1999
[21] Coletta, M., Lippert, T., Palazzari, P.: Hyper-Systolic Implementation of BLAS-3 Routines in the APE100/Quadrics Machine. Para98 -Workshop On Applied Parallel Computing In Large Scale Scientific and Industrial Problems - Umea, Sweden June, 14-16, 1998
[22] Taraglio, S., Massaioli, F.: An efficient implementation of a Backpropagation learning algorithm on a Quadrics parallel supercomputer", Proceedings of the High Performance Computing and Networking – Europe (HPCN-Europe 1995), April 1995.
[23] Gerasoulis,A., Yang,T.: On the granularity and clustering of directed acyclic task graphs. IEEE Transactions on Parallel and Distributed Systems, vol. 4, N. 6, Jun. 1993.
[24] Meiko Web Page:
www.meiko.com/info/TechnicalDescription/TechnicalDescription.html
[25] Baiardi,F., Bernasconi,C., Guerri,D., Ricci,L., Vaglini,L.: Implementing concurrent paradigms through virtual shared areas. Technical report from Informatics Department, University of Pisa, November 1998.
[26] Bacci,B., Cantalupo,B., Danelutto,M., Orlando,S., Pasetto,D., Pelagatti,S., Vanneschi,M. : An environment for structured parallel programming. In Advances in High Performance Computing, NATO ASI series e, Vol. 30, 1997.
[27] Siegel, H. J., Maheswaran, M., Watson, D.W., Antonio, J.K., Atallah, M.J.: Mixed Mode System Heterogeneous Computing. Chapter 2 in the book 'Heterogeneous Computing', Ed. Eshigian,M.M., Arctech House, Norwood, MA, 1996.
[28] Stone, H.S.: Multiprocessors. In High-Performance Computer Architecture, First Edition, Addison Wesley, 1987.
[29] Bacci,B., Danelutto,M., Pelagatti,S., Vanneschi,M.: SkIE: A heterogeneous environment for HPC applications. Parallel Computing, Vol. 25, n.13-14, Dec. 1999.
[30] Silbershatz, A., Galvin, P.B.: Virtual Memory. Chapter in Operating System Concepts, Addison Wesley, 1994
[31] www.quadrics.com/web/public/fliers/qsnet.html
[32] Mongenet C.: Data Compiling for System of Affine Recurrence Equations. IEEE International Conference on Application - Specific Array Processors, ASAP, Aug. 1994.
[33] Mongenet C., Clauss P., Perrin G.R.: Geometrical Tools to Map System of Affine Recurrence Equations on Regular Arrays. Acta Informatica, Vol. 31, No. 2, 1994.
[34] Loechner V., Mongenet C.: Solutions to the Communication Minimization Problem for Affine Recurrence Equations. Proceedings of the EUROPAR '97, Passau, Germany, Vol. LNCS 1300, August 1997.
[35] Sarkar, V. : Partitioning and scheduling parallel programs for multiprocessors. The MIT Press, 1989
[36] Lippert, T., Seyfried, A., Bode, A., Schilling, K.: Hyper-Systolic Parallel Computing. IEEE Trans. on Parallel and Distributed Systems, n. 1, 1998.
[37] Lippert, T., Glaessner, U. , Hoeber, H., Schilling, K. and Seyfried, A.: Hyper-Systolic Processing on APE100/Quadrics, in n^2-loop computations'. Int. Jour. Mod. Phys. C, 7, 1996.
[38] Marongiu, A., Palazzari, P.: A new memory-saving technique to map System of Affine Recurrence Equations (SARE) onto Distributed Memory Parallel Systems. International Parallel Processing Symposium IPPS99, April 12-16, 1999, San Juan, Puerto Rico.
[39] Marongiu, A., Palazzari, P.: Automatic Mapping of System of N-dimensional Affine Recurrence Equations (SARE) onto Distributed Memory Parallel Systems. Accepted to appear in the special issue of IEEE Trans. on Software Engineering on Architecture-Independent Languages and Software Tools for Parallel Processing
[40] Saraceni, F., Coletta, M., Pucello, N., Rosato, V., Capuzzo-dolcetta,R.: On the use of a hybrid MIMD-SIMD platform to simulate the dynamics of globular clusters with an internal massive object. In preparation.
[41] Allan, R.J., Guest, M.E.: Distributed Computing Programme.HPC Profile – The national publication for High – Performance computing applications and techniques, 24, pp12-15, Dec. 1999

Paolo Palazzari, graduated in Electronic Engineering '89, Ph.D. '94. From '89 to '95 he was scientific assistant at the Electronic Engineering Department of University 'La Sapienza' in Rome. From '96 he is staff scientist at the Italian National Agency for New Technologies, Energy and the Environment (ENEA) where works on the High Performance Computing and Networking project. Main fields of his research are automatic parallelism detection for High Level HW/SW synthesis, routing, allocation and scheduling of parallel programs, image processing, neural networks and non linear optimization techniques.

Lidia Arcipiani graduated in Mathematics and received a post-graduate "diploma" in Statistical Methodology. After ten years at the Electronic Division of the Italian Company Olivetti, she moved at the Italian Commission for Nuclear Energy (CNEN, now ENEA) as director of Casaccia Computing Centre. She has been acting as coordinator of computing, information and networking programs for the Department of Energy. At present she is staff scientist and R&D expert on the High Performance Computing and Networking Project of ENEA. She has been member of the National Committee for the Mathematical Science of the Consiglio Nazionale delle Ricerche (CNR) and vice-president of the National Committee for the Information Technologies of the CNR. She was professor of Computer Science at Catania University.

Massimo Celino, graduated in Physics at the University "La Sapienza" of Rome (1992), has been working as Research Associate at the Department of Physics in the field of Statistical Physics. From 1994 he is working at the Italian National Agency for New Technologies, Energy and the Environment (ENEA). His work is mainly

focused on Computational Physics for applications in Materials Science and Quantum Chemistry. Other interests are in the field of computer programming and parallel algorithms for high performance computing platforms.

Roberto Guadagni, graduated in Electronic Engineering at University "La Sapienza" of Rome in 1983. He is working at the Italian National Agency for New Technologies, Energy and the Environment (ENEA) from 1986. Now he is responsible of the supercomputing facilities in the Casaccia Research Center.

Alessandro Marongiu, graduated in Electronic Engineering '95, Ph.D. 2000. Main fields of his research are automatic parallelism detection for High Level HW/SW synthesis, allocation and scheduling of parallel programs, cellular neural network design.

Agostino Mathis is Director of the Interdepartmental Project "High Performance Computing and Networking" (HPCN) at the Italian National Agency for New Technology, Energy and the Environment (ENEA). He graduated in Electrical Engineering, and received a post-graduate "diploma" in Nuclear Engineering, at the Turin Polytechnic School of Engineering. He holds the "Libera Docenza" in Control Engineering at the Rome "La Sapienza" University. He is Professor of High Performance Computing at the Rome "Tor Vergata" University. He has published numerous papers on mathematical modeling of industrial processes, techniques for computer simulation, information systems for organizational management and high performance computing.

Paolo Novelli graduated in Physics at the University of Rome (Italy) in 1969. From 1971 to 1975, he was an assistant professor of electronics at the Physics Department of the same University. At the Italian National Agency for New Technologies, Energy and the Environment (ENEA) since the beginning of his professional activity, he was involved in the following fields: design of electronic devices for nuclear plants and analog computers, development of numerical models of uranium enrichment cascades, computational gas-dynamics of ultracentrifuges. From 1982, he was involved in the corporate planning activities, and in 1987 he became the Head of the Organizational Development unit. In 1994, he came back to his past research interests and joined the High Performance Computing and Networking unit (HPCN).

Vittorio Rosato, graduated in Physics at the University of Pisa (1979), received the Ph.D. at the University of Nancy (F) on 1986. He has been working as Research Associate at the University College of Wales in Aberystwyth (UK), at the Centre d'Etudes Nucleaires de Saclay (F), at the CECAM (Centre Europeen de Calcul Atomique et Moleculaire) in Orsay (F). From 1990 he is a Staff Scientist at the Italian National Agency for New Technologies, Energy and the Environment (ENEA). His research field is Computational Materials Science; in this area, he has been working on the development of interatomic potentials for metals and intermetallics and in their use to study thermodynamic and structural properties of complex materials. He is now active in the field of atomic-scale simulations of new C-based and Si-based materials. He is involved in design and realization of new tools (HW and SW) for HPC for scientific applications.

The NRW-Metacomputer

Building Blocks for A Worldwide Computational Grid *

Claus Bitten
Regional Computing Centre, University Cologne, 50923 Cologne, Germany
Jörn Gehring
Paderborn Center for Parallel Computing, University Paderborn, 33095 Paderborn, Germany
Uwe Schwiegelshohn
Ramin Yahyapour
Computer Engineering Institute, University Dortmund, 44221 Dortmund, Germany

Abstract

In this paper, we present the results of the NRW-Metacomputing task force, which has been working on the development of a country-wide metacomputer since 1996. The resulting installation is among the very few that are already operational, have full support for heterogeneous resources, contain a decent security model, and feature an advanced scheduling sub-system for the metacomputing environment. The NRW-Metacomputer has been implemented using a modular software architecture. Hence, concepts and components of it can be re-used by others without the need of having to obtain the metacomputing-software as a whole. Furthermore, the NRW-Metacomputer already provides well defined interfaces for linking the system with other metacomputing environments to form a truly global computational grid. Distinctive features of this system are its highly scalable and fault tolerant software architecture, its advanced resource planning mechanisms as well as an integration into a DCE/DFS environment.

1 Metacomputing and the Computational Power Grid

The term *metacomputer* was initially coined by Larry Smarr around 1987 [13]. According to his definition, a metacomputer is a network of globally distributed machines that are linked together by a complex software system which enables them to act like a single very large supercomputer. The advantages of the concepts are obvious. First of all, a metacomputer can theoretically provide more computing power than any existing single machine. Furthermore, it offers its users free choice, which machine(s) to use for a specific job and it helps the participating computing centers to distribute the load more evenly.

Building a metacomputer however is a complex and difficult task. Since the concept was invented, many researchers have been working on the subject and nowadays there are several software systems, which all cover different aspects of Smarr's and Catlett's vision. Among them are advanced cluster management systems like CODINE [8], LSF [15], or CONDOR [5], which focus mainly on connecting Unix workstations. Furthermore, there are a couple of projects which take a more general approach and work on the integration of fully heterogeneous resources like e.g. supercomputers or remote data sources (e.g. satellites, weather radar, unique scientific instruments). The *NRW-Metacomputer* initiative, which is presented in this paper, is one of these projects. Other are e.g. GLOBUS [6], LEGION [10], or MSHN [11].

Over the time it became clear that a global metacomputer which meets the definition of Smarr and Catlett and is *as easy to use as a single workstation*, is unlikely to be established by any single research group. The large variety of problems that have to be solved (resource management, administration, accounting, security, scheduling, etc.) requires close cooperation between researchers from many different fields. This is why in the recent past a couple of open forums have been established, which focus on the installation of large *computational power grids*[1] by putting together the pieces that have been invented during seven years of metacomputing research all over the world [1, 2].

In the following, we present the outcomes of the NRW-

*Supported by a grant from the NRW Metacomputing project

[1] a synonym for *metacomputer* which emphasizes the analogy to a nationwide power grid

0-7695-0556-2/00 $10.00 © 2000 IEEE

Metacomputing task force [3], which has been working on the development of a country-wide metacomputer since 1996. The resulting installation is among the very few that are already operational, have full support for heterogeneous resources, contain a decent security model, and features an advanced scheduling sub-system for the metacomputing environment. The NRW-Metacomputer has been implemented using a modular software architecture. Hence, both concepts or components of it can be re-used by others without the need of having to obtain the metacomputing-software as a whole. Furthermore, the NRW-Metacomputer already provides well defined interfaces for linking the system with other metacomputing environments to form a truly global computational grid.

2 Architecture of the NRW-Metacomputer

The backbone of the NRW-Metacomputer was built during the HPCM (*high performance computer management*) project, which provides basic infrastructure for metacomputing management. It features a multi-tier architecture in which a server layer receives user requests from access modules (see Sec. 5) and forwards them to the attached resources like for example supercomputers (Fig. 1). These resources are encapsulated by so called *coupling modules* that implement an abstract service layer on top of the heterogeneous hardware pool. Besides abstracting the available information and access methods from the management, the coupling modules interact with the locally installed management system of the HPC component. Although the coupling modules have to be adapted to every new kind of hardware, each implementation is based on a generic frame that already covers a significant amount of the required functionality. So far, there exist implementations of coupling modules for UNIX, CODINE [8], NQE [4], CCS [12] and LoadLeveler. Additional modules, as e.g. for LSF, can easily be derived from the existing implementations.

One of the major goals of the initiative was to develop a metacomputer that maintains maximum autonomy for the participating service centers. Hence, each institute is free to tailor the behavior of its coupling modules to comply with the local policies. For example, if a certain service shall be made available to the metacomputer only when the local machines are lightly loaded or the remote user is willing to pay an extra fee, the behavior of the coupling module can be adjusted accordingly.

It is furthermore possible to attach completely different types of services to the metacomputer by using the same *abstract service interface* that defines the behavior of the coupling modules. For example, the scheduling services described in Sec. 3 integrate itself into the metacomputer through this interface. A similar approach could be used to establish a link between the NRW-Metacomputer and e.g. a metacomputing system in another country or even on a different continent.

Communication links between the different components of the metacomputer are established by the so called *communication layer*. This is a separate module that provides secured communication services for both Java- and C++/C-based components. Currently the communication layer uses TCP sockets for message passing and the GSS API [9] for security. However, these can be easily exchanged or even mixed with other paradigms, if necessary.

Another important issue for any metacomputer installation that needs to be brought to practical use is administration and authentication of its users. Typically, service centers already have set high standards for the management of their local users and are not willing to compromise this by installing the metacomputer software. Hence, we decided to rely on the services provided by the *Distributed Computing Environment* (DCE) [7], since this is a vendor supported product and already accepted and used by many computing centers (see Sec. 4).

Much effort has been spent on designing the NRW-Metacomputer as a highly reliable and fault tolerant system. Its architecture does not contain any *single point of failure*. This could be achieved by having the distributed environment being managed by the coordinated effort of the management daemons. These daemons are all alike and none of them performs any specific tasks that are not directly related to the corresponding computing center. Hence, a failure at one node can at most render that center unavailable where the problem occurred. As long as there is one management daemon alive, the NRW-Metacomputer remains available.

It should be noted that all management daemons actively contribute to the operation of the whole system. There are no shadow daemons that monitor the system operations silently until a component fails and take the place of that component. As a consequence, the principle of cooperative management not only increases fault tolerance but also helps improving the overall system performance. Clearly, this system architecture required some extra effort in specifying transaction protocols and during the implementation. The key concept here was to employ the technique of *virtual shared memory* for the metacomputer (Fig. 2).

All information that refers to the global state of the metacomputer (such as the lists of active jobs or available resources) are stored in so called *shared objects*. These are object oriented encapsulations of virtual shared memory segments. Whenever the content of a shared object changes, these modifications are transparently propagated to all other remote instances of the same object. Internal methods of the shared object class ensure that the data is kept in a consistent state, even if parts of the metacomputer should fail while remote instances of shared objects were updated.

The use of shared objects enables each management

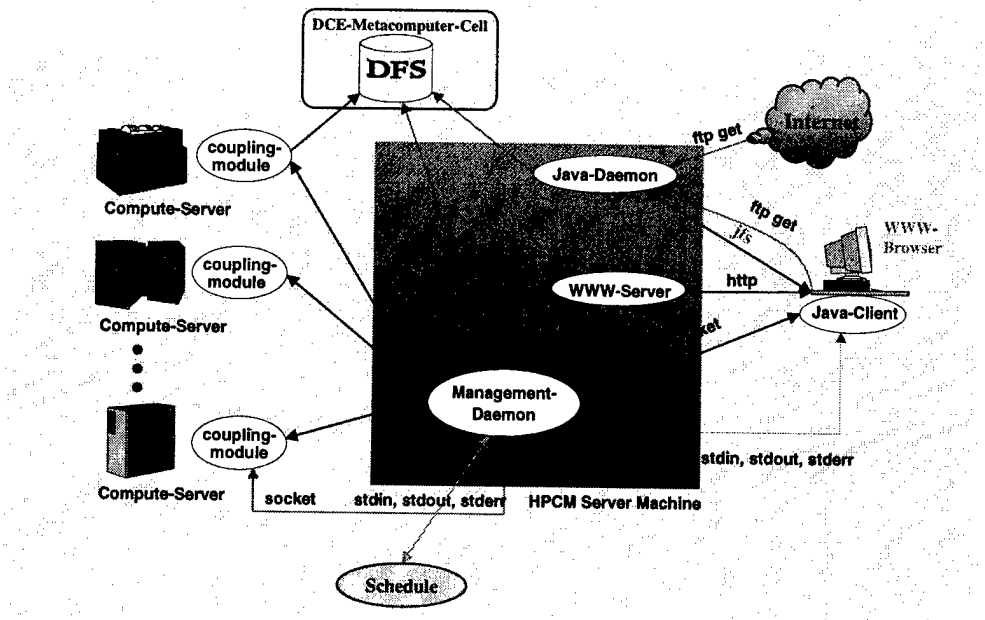


Figure 1. Layout of the management infrastructure

daemon to accept incoming requests and co-ordinate their fulfillment by underlying metacomputing services. If one management daemon should fail, the remaining ones take over its tasks. It only becomes visible to the users that the state of their jobs is reverted to the last completed transaction. In most cases, this equals the current state of a job.

3 Integrated Metacomputer–Scheduling

Job scheduling and resource allocation are one of the core problems in the metacomputing architecture. The owners of HPC installations are only willing to include their resources into a metacomputer if the performance of their components will not degrade. Similarly, users expect a better performance for their jobs. Note that the expression *performance* has not been defined as different people may attach a different meaning to it. Therefore, the scheduler must provide a high efficiency for the metacomputer while also taking additional requirements into account.

3.1 Scheduling Considerations

The paradigms for scheduling on a metacomputer differs significantly from job scheduling on a parallel computer. Therefore, we give in the following some properties of meta-scheduling that must be considered for building a computational power grid:

Variable Scheduling Objectives: In common job scheduling there usually exists a single scheduling objective or performance metric that is fixed for a parallel computer and all its jobs. For example, this can be the minimization of the average response or turnaround time [?]. The objective is typically determined by the local management system or by the administrator. In metacomputing this objective is variable. As we assume a distributed system that is not controlled by a single instance, the objective should further be adaptable for each resource in the metasystem. While the schedule target for some machines may for instance be the maximization of the throughput, others have the objective to minimize the response time. Besides the objectives for the resources, we must also take the needs of the user into account. Some user may favor the availability of specific resource properties like the size of the main memory while other may have additional constraints about the execution of a job. A typical example would be a deadline for a job that must be met while it is of no particular interest if the job is completed as fast as possible. For this user the minimization of the response time would not reflect her demands. In metacomputing it is necessary that user objectives are considered. For instance, the user may only be interested in resources that fit her needs better than any local resources.

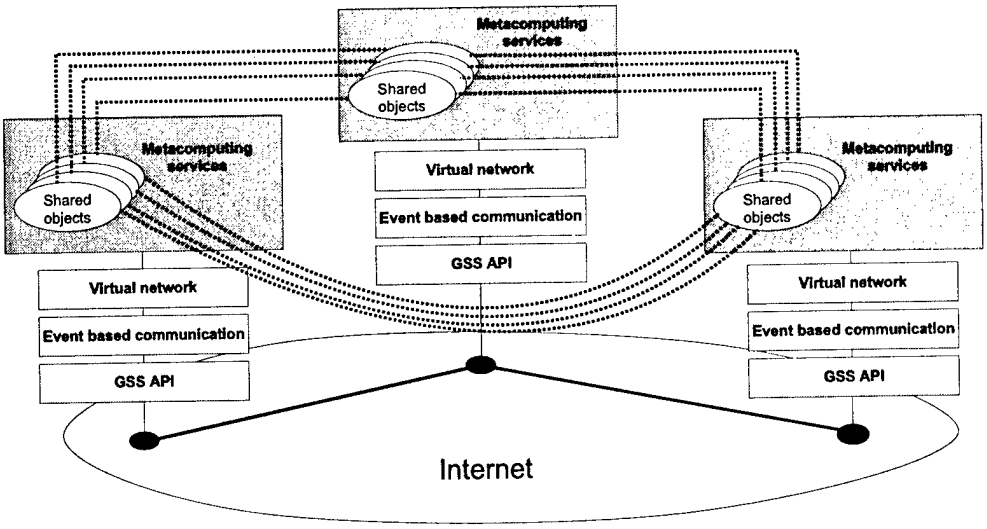


Figure 2. The NRW-Metacomputer uses virtual shared memory for its distributed management

Therefore, the scheduling must be adaptable to generate the most appropriate result.

Independent Schedulers: Usually a scheduler in metacomputing cannot demand exclusive control over all resources. For scheduling in metasystems, we have to cope with the situation that jobs may not only be submitted via the metacomputing interface. Hence, any limitations of the local management must also be considered. For instance, one problem is the list scheduling of most management systems which do not provide any information about the expected completion time of a job. Unfortunately, availability of this kind of information is important in distributed metasystems to allow future allocation planning. Of course, if the local management provides additional information, the metacomputing scheduler should be able to utilize it. In this case the metacomputing management does not perform any local scheduling but relies on the existing system scheduler. The resulting schedule efficiency highly depends on the features of the lower-level scheduler. If a resource does not provide the requested features like a guaranteed completion time, it cannot be considered suitable for some job requests. This limits the usability of this resource for the metasystem. Nevertheless, the metacomputing scheduling should support all kind of local management systems.

Arbitrary Resource Requests: As job requirements and resources in a metasystem may vary according to type and application, there is a need for the description of complex requests. For instance, assume two different users: The first user does not provide a very detailed request as she wants to get as many computing resources as possible. More restrictive requirements would only reduce the possible resource set for her job. The other user is looking for very specific resources. He may have access to an alternative set of local resources for the execution of his job and is therefore only looking for a better resource allocation. Consequently, he formulates special requirements and preferences. The meta-scheduler must support both approaches. The individual user should be able to influence the resource selection and the scheduling so that she gets the best suited set of resources. The attributes of a resource and therefore the available fields in a request should not be considered invariant. Different resources may have different attributes and features that may not be known to the scheduler at the time of implementation. But the system should still be able to handle them.

Resource Reservation: This feature is necessary for some applications as well as for the consideration of resource maintenance. For instance, demonstrations may require the reservation of a resource allocation for a dedicated time span. It is also advantageous for the schedule to consider system downtime or restricted usage that is known in advance. Reservations are further needed for multi-site applications. As there is no global scheduler instance, it must be possible for the local scheduler to reserve resources for a specific time span in order to guarantee the concurrent availability of resources at different locations.

Job Execution Guarantees: In metacomputing it would be inefficient to schedule jobs on an ad-hoc strategy as it is difficult to respect several objectives by not assuming a central scheduler. For example, if a job does not need to be executed as soon as possible, this flexibility can be used to improve the schedule. Assume again the mentioned case of a job with an execution deadline. Typically, the user needs immediate feedback whether his requirements can be met. It is therefore necessary for the user to receive in advance guarantees about the schedule of his job so that he can react

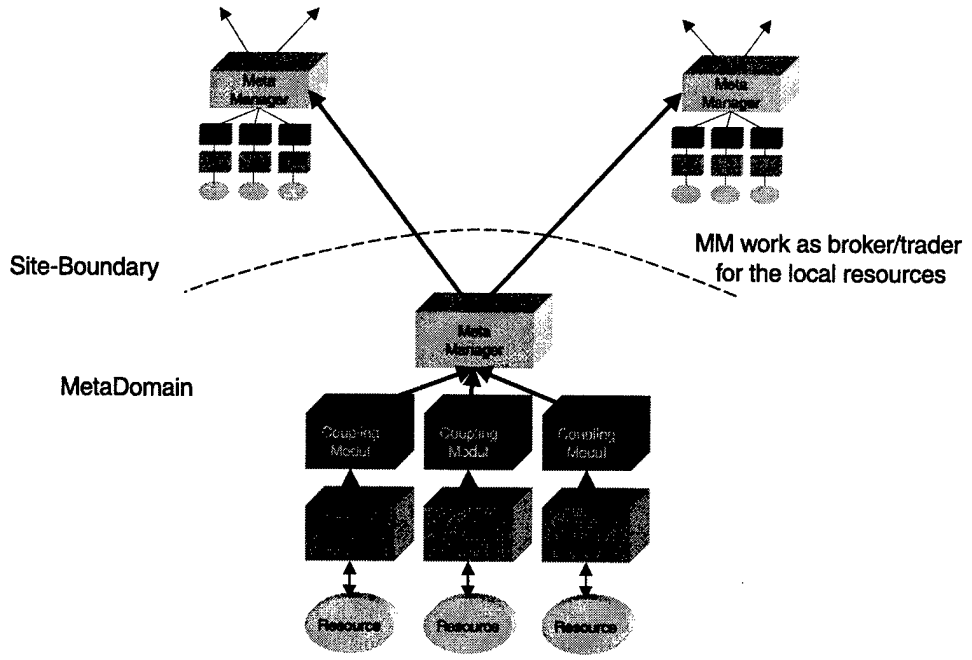


Figure 3. Logical Scheduling Infrastructure

accordingly. The scheduler need not always provide such guarantees, but it should be capable of giving them if they are required. Those guarantees are additional constraints in the scheduling of a job.

3.2 Scheduling Architecture

To avoid the bottleneck of a central scheduler and to increase flexibility a distributed approach is employed. To implement a distributed metacomputing scheduler we use an architecture which is based upon so called *MetaDomains*. All MetaDomains of a metacomputer form a redundant network. Typically, a MetaDomain is associated with local HPC resources and is controlled by a *MetaManager*, that is all HPC resources at one site are connected to a single MetaDomain. We use these autonomous scheduling domains to manage the local resources and offer them to other domains. This concept supports the idea of a *computational power grid*, where several independent sites can join a larger network and share jobs and computing power while not loosing control over the local resources. The logical structure of such a scheduler is described in Fig. 3. This network can be dynamically extended or altered. The presented architecture guarantees a high degree of flexibility by allowing different implementations.

The MetaDomains communicate among each other by transmitting or requesting information about resources and jobs. To this end a MetaManager inquires local schedulers about system load and job status. A MetaManager can also allocate local HPC resources to requests. The distributed scheduling itself is based upon a brokerage and trading concept which is executed between the MetaManagers.

In detail, a MetaDomain tries to

- satisfy local demand if possible,
- ask other MetaDomains for resources, if the local demand cannot be satisfied,
- offer local HPC resources to other MetaDomains for suitable remote jobs, and
- act as an intermediary for remote requests.

Once a suitable allocation of HPC resources (including network resources) to a job has been found, the actual submission is independent of the scheduler. In our architecture the scheduling objectives are not specified. As already mentioned there may not only be a single scheduling objective in a metacomputer. Each HPC component can define its specific objectives. Similarly, each user may associate specific constraints with his job like a deadline or a cost limit. It is the task of the trading system to find matches between requests and offers. This way not all users and all components are forced to fit into a single framework as is usually done in conventional scheduling. Now, it is their responsibility to define their own objectives. The implementation of the metacomputing scheduler only provides the framework for such a definition and it must be able to compare any request with any offer to find a match.

The selection of the best suited allocation is based on a comparison of the provided objective functions. The objective function of a request is applied to an allocation in order to generate a value for the utility from the user's point of view. Similarly, the offers also provide an objective function or a value for its utility to represent the resource's point of view. Now, the responsible *MetaManager* combines the objectives and determines an allocation that maximizes the overall objective with respect to its full schedule. It is also possible to provide the user with a front-end that allows interactive selection of allocations. Such a front-end can also be used to obtain status information about the metasystem with the help of the request mechanism. This information may help a user to generate a request which results in the best suited set of resources depending on the current condition in the metasystem.

Note that our method is not an auction system as we do not provide a market where several jobs compete for a resource. Instead our schedulers select allocations that fit a request best at a particular time instance. However, the selected allocation need not necessarily be executed. The *MetaManager* maintains a schedule with all current allocations in its domain. Its scheduler is free to modify the current schedule at any time. However, changes in the current scheduling are only allowed as long as they do not violate any guarantees that have been given for a job. Requested guarantees are additional constraints that limit future requests for rescheduling. This procedure is used to improve the current schedule and to cope with resource failures or cancellations of jobs. The rescheduling requires new requests for offers if other allocations are not active anymore. Note that a valid schedule exists at every moment. Also, there is a tentative schedule for a job after each request and a following allocation.

Any improvement of the schedule is measured by combining all objective values. To this end, the scheduler attempts to maximize the overall objective value of the schedule. As an objective function is received for every request and for every offer, there is a combined objective function or value for each allocation. The objective functions of all allocations together define the optimization problem. An improvement can be achieved for instance by moving existing allocations while all constraints are observed. Alternatively, the scheduler can look for new allocations. The metascheduling concept further supports multi-site scheduling and co-allocation. However, this requires the inclusion of network management as just another high performance computing resource to provide guaranteed communication bandwidth between participating resources. In addition the local resource managers must provide offers with schedule guarantees which must be exactly met. Note that this scheduling strategy does not guarantee an optimal schedule in general, but it meets all requirements of Sec. 3.1 as separate objectives are allowed for each resource in the metasystem.

In our metacomputer scheduling concept only the local HPC scheduler is responsible for the load distribution on the corresponding HPC resource. Therefore, it can also accept jobs from sources other than the metacomputer. The metacomputer scheduler only addresses the load imbalance between different HPC resources. However, to execute multi site applications, the concurrent availability of different HPC resources and sufficient network bandwidth between them becomes necessary. For reasons of efficiency this requires resource reservation for future time frames and the concept of guaranteed availability. Although most HPC schedulers do not presently support such an approach it can be implemented by using preemption (a *checkpoint-restart* facility) while still maintaining a high system load.

In the project *SCHEDULE* [14] of the initiative a metacomputer scheduler was designed using *CORBA* to allow transparent and language independent access to distributed management instances. For the evaluation of different scheduling methods a simulation framework has further been implemented. It is used to compare different scheduling algorithms regarding their applicability for a metacomputing network. The benefit of possible technology enhancements, like for example preemption, to the quality of the schedule is also determined with the help of the simulator. As already mentioned, communication between resources during a multi site job execution must be taken into account as well. To this end the available network must be considered as a limited resource that is managed by the schedulers in the MetaDomains. The inclusion of this objective into the scheduler is part of the future work.

4 Data Distribution, Security and Administration

Like every metacomputing system that is brought to practical use, the NRW-Metacomputer has to cope with a variety of problems. Though it runs on a large number of different hardware architectures, it must still provide a standard API for the HPC components and other related software. Furthermore, it needs a secure mean of communication across the public and inherently insecure Internet. It must also provide scalability for its servers and the necessary means of authentication. And finally, it must not impose additional overhead on its users and the administrators of the attached HPC-nodes.

Hence, it was decided to use the standardized Distributed Common Environment (DCE) as an existing and reliable software solution. DCE/DFS is a middleware, created by the OSF and available in license for commercial usage. Currently we are using the Versions 1.0 and 1.1 of DCE/DFS on Solaris, AIX and NT. Versions for IRIX and UP-UX were

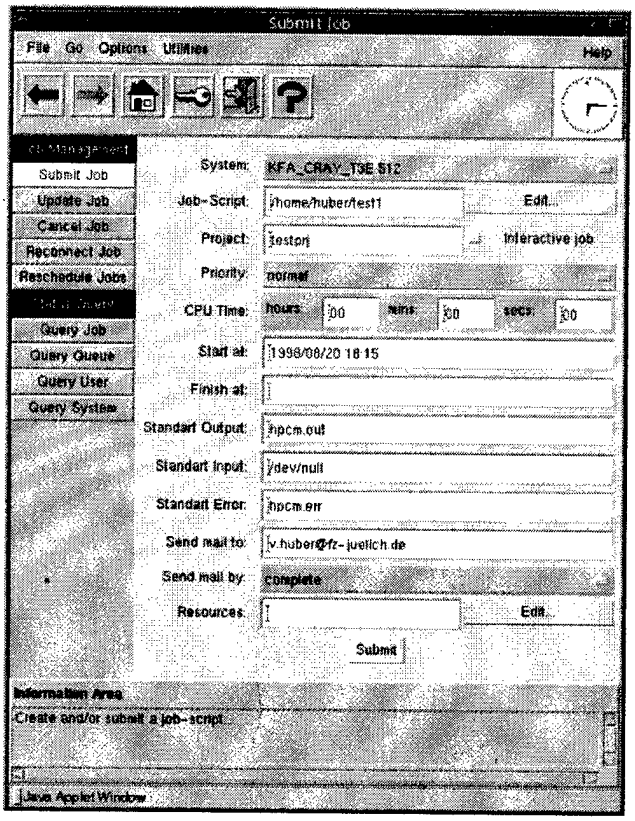


Figure 4. Screenshot of the Java User Interface

tested. This approach provides us with the possibility to include any DCE/DFS-capable system into the metacomputer. There is also a project to create a free DCE/DFS port for Linux, but in its early state and without the DFS it is not usable for the NRW-Metacomputer.

An administrative domain built on DCE is called a *cell*. DCE provides a namespace for every cell that allows easy access to all its resources. Cells can be connected using cross cell authentication to provide a secure way for sharing resources and to minimize administrative overhead. Thus, users can authenticate to any client within the DCE cell, submit jobs or use the provided distributed file system DFS.

Since all computers within the metacomputer are part of a DCE cell, the administrative overhead is minimized and can be distributed throughout the metacomputer. This is possible because DCE uses *access control lists* (ACL) for administrative commands and functions, which allow privileges to be selectively granted to either individuals or groups. With these ACLs it is for example possible to create administrative accounts for special tasks, such as creating new users or incorporating new clients, without the need to have full root access to the entire cell.

Communication within the DCE cell is secured by using encrypted RPC calls. Because of the restrictions to export encryption algorithms from the USA, the international version of DCE/DFS does not have this security. Hence, we decided to use GSS API for the HPC components to protect communication across the public internet.

The use of DFS on top of the DCE environment offers several advantages for the NRW-Metacomputer. All users have a home directory, which they can access independently from their physical location, the compute-servers can instantly access the required input and output files, and installed software packages are available to the whole metacomputer. DFS uses the DCE ACLs to offer a high level of security and flexibility for file or data access. It is possible to create groups who have the same set of permissions, which lowers the administrative workload. The actual files within the DFS are stored in *filesets*, which can be compared to Unix filesystems or DOS partitions. These filesets provide key features for a distributed environment. Several servers can host a specific fileset to provide scalability and stability. The DFS-Server keeps these so called *replicas* synchronized with the original fileset. This allows fast and up to date access to HPC Software from all clients within the NRW-Metacomputer. Filesets can be backed up during

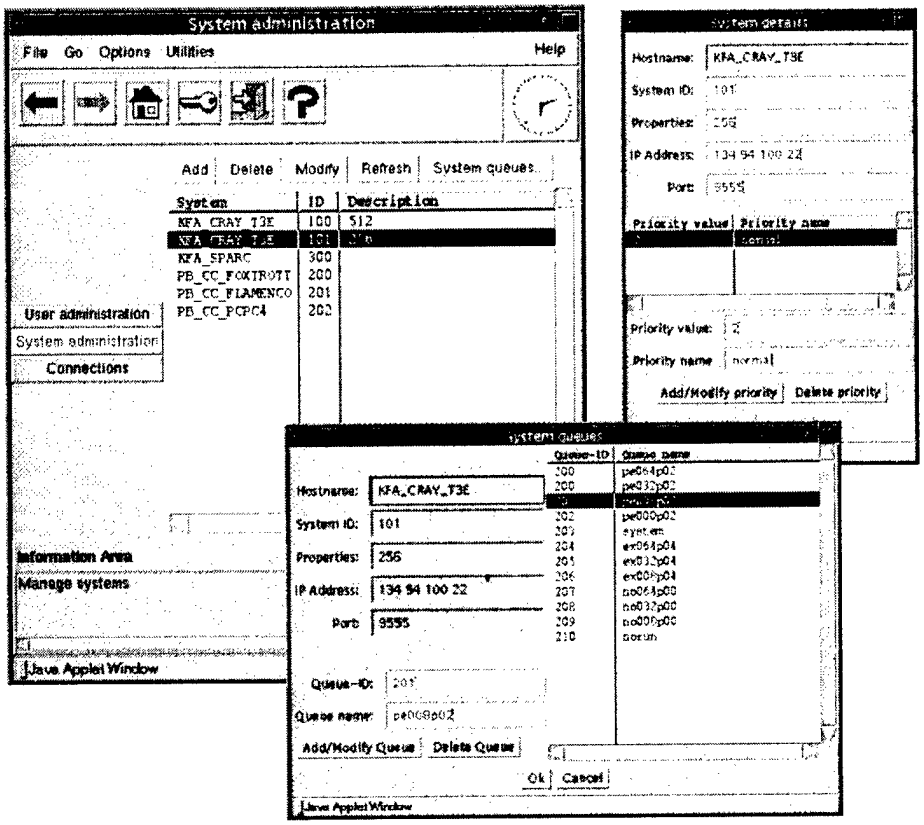


Figure 5. Screenshot of the Java Administration Interface

normal system operations, they can be enlarged or even relocated from one DFS-Server to another without the need to stop the working software, which is importand for long running jobs. Given these facts, a centralized backup for all data within the cell is possible and is used for the NRW-Metacomputer.

For the special needs of the NRW-Metacomputer the DCE internal database (*registry*) is used to store additional information like e.g. user accounts. It is planned to expand this, so that the registry will contain all information needed to run the metacomputer. For instance, this may be information about special needs or restrictions of connected compute-servers or software. Using the registry offers several advantages, because all its data is accessible within the whole cell via the DCE API or online commands, and the database itself is scalable and fault tolerant.

Since DCE provides an API, software can be adapted to use special features, such as sendmail, the apache webserver or samba. Furthermore, it is possible to incorporate DCE into any other proprietary software. DCE has a standard command line interface, which is called *dcecp*, the dce controll programm. By using this interface in a batch mode, it is possible to write different kinds of administrative software, such as a frontend using HTML and CGI-scripts for webservers, or a X11-Interface using Tcl/Tk without the need to directly use the API or other low-level functions of DCE. The dcecp itself is written using Tcl, which allows new modules to be developed for it.

5 Accessing the Metacomputer via the World Wide Web

An important aspect of the idea of a computational power grid is the freedom for its users to access the system from wherever they want, ideally even from a hotel room in a remote corner of the world. Hence, we decided to use Java and the WWW-technology for implementing the user interface of the NRW-Metacomputer (Fig. 4). In order to provide the same comfort to the administrators of the metacomputer, the administration interface has been implemented in the same way thereby enabling both use and administration of the metacomputer from whereever there is a connection to the Internet (Fig. 5). Furthermore, both interfaces are capable of storing personal customization data within the metacomputer. Thus, users always find their own personal access interface, no matter from where the system

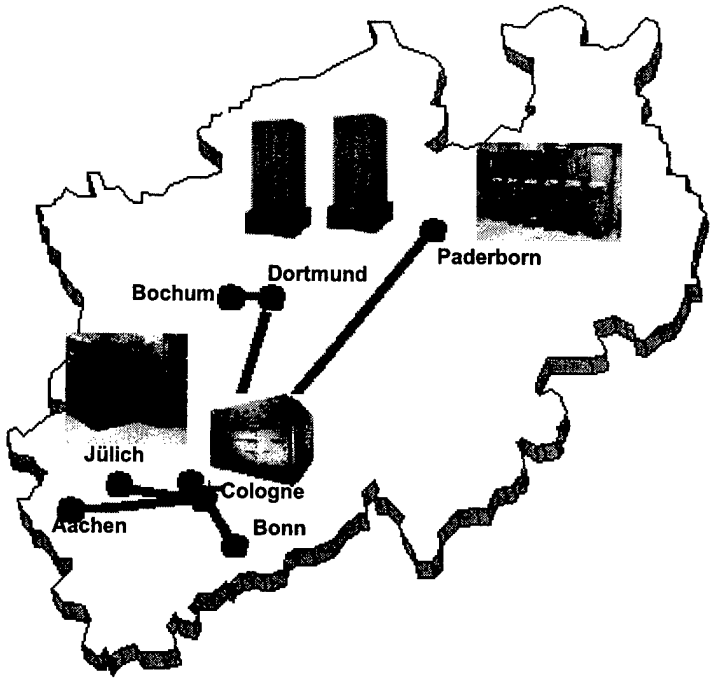


Figure 6. Current size of the NRW-Metacomputer

is being accessed.

The metacomputer performs its own user management, authentication and authorization based on the DCE infrastructure. Thus, users will only have to login once for each metacomputing session. If the machine used for accessing the system runs a DCE-client, users can even work with their private files independent from their current location. This is therefore the recommended way of working with the NRW-Metacomputer. However, if the DCE services are not available, user files can be transferred to and from the metacomputer by a set of ftp-based services, which we have added to the user interface (Fig. 1).

6 Conclusions

We have presented the NRW-Metacomputer as a working connection of four computing centers spread all over Nortrhine-Westphalia, which contains heterogeneous HPC resources like Cray T3E, IBM SP2, Sun Enterprise, or Siemens hpcLine (Fig. 6). Among the important aspects of this system are the modular, multi-tier architecture as well as its powerful scheduling component and the integration into a DCE-based environment. Knowing that there exist several other metacomputing environments with similar capabilities, we described how these projects can benefit from the results of the NRW-Metacomputer initiative. Possible aspects are the re-use of concepts or modules and the creation of a large metacomputing environment by using the interfaces of the NRW-Metacomputer.

Besides the integration of more sites into the metacomputer, future work will also focus on the development and evaluation of improved scheduling strategies on top of the now existing infrastructure. Furthermore, we will employ the service interface of the NRW-Metacomputer to add new kinds of services like for example streaming video or other real-time data sources.

7 About The Authors

Claus Bitten has studied computer science at the University of Bonn and is currently studying Philosophy at the University of Cologne. He is working as a member of Regional Computing Centre of the University of Cologne and as an IT Administrator for the Mindfact Interaktive Medien AG, a member of the Swedish Framfab group. His research interest focuses on distributed and secure computing enviroments and their implementation on various systems. Besides this, his interests include Linux, e-commerce, and the design of computer games.

Jörn Gehring studied computer science at the University of Paderborn and received a Diploma with distinction in 1994. Currently he is a staff member at the Paderborn Center for Parallel Computing. His research interest focuses in the field of parallel computing, in particular high-performance- and metacomputing. He works in the design

of distributed resource management software and runtime environments for metacomputers. Besides metacomputing, his research interests include distributed genetic algorithms and sorting on massively parallel machines.

Uwe Schwiegelshohn is professor at the Computer Engineering Institute of the University Dortmund since 1994. His main research interests include scheduling problems, metacomputing and embedded systems. He received a diploma and a doctor degree in electrical engineering from the Technical University Munich in 1984 and 1988, respectively. From 1988 to 1994 he was with the IBM T.J. Watson Research Center in Yorktown Heights.

Ramin Yahyapour studied electrical engineering at the University of Dortmund and received his Diploma in 1996. Currently he is a staff member at the Computer Engineering Institute in Dortmund. His research interest focuses in the field of parallel computing, in particular high-performance- and metacomputing. He works in the design of distributed resource management software and runtime environments for metacomputers. Besides metacomputing, his research intersts include network management.

References

[1] EGRID: The european grid forum. `http://www.egrid.org/`.

[2] The grid forum. `http://www.gridforum.org/`.

[3] Nrw-metacomputing task force. `http://www.upb.de/pc2/nrw-mc/`.

[4] *Introducing NQE*. Cray Research Publication, Silion Graphics, Inc., 1998.

[5] D. H. J. Epema, M. Livny, R. van Dantzig, X. Evers, and J. Pruyne. A worldwide flock of condors: load sharing among workstation clusters. Technical Report DUT-TWI-95-130, Delft University of Technology, Department of Technical Mathematics and Informatics, Delft, The Netherlands, 1995.

[6] I. Foster and C. Kesselman. Globus: A metacomputing infrastructure toolkit. *International Journal of Supercomputer Applications*, 1998.

[7] Frederic Gittler and Anne C. Hopkins. The DCE security service. *Hewlett-Packard Journal: technical information from the laboratories of Hewlett-Packard Company*, 46(6):41–48, December 1995.

[8] GENIAS Software GmbH. CODINE: Computing in distributed networked environments. `http://www.genias.de`, sep 1995.

[9] D. Gollmann. Cryptographic APIs. *Lecture Notes in Computer Science*, 1029:290–??, 1996.

[10] A.S. Grimschaw and W.A. Wulf. The legion vision of a worldwide virtual computer. *Coomunications of the ACM*, 40(1), jan 1997.

[11] D. Hensgen, T. Kidd, D. St. John, M. C. Schnaidt, H. J. Siegel, T. Braun, J.-K. Kim, S. Ali, C. Irvine, T. Levin, V. Prasanna, P. Bhat, R. Freund, and M. Gherrity. An overview of the management system for heterogeneous networks (MSHN). In *8th Workshop on Heterogeneous Computing Systems (HCW '99)*, San Juan, Puerto Rico, 1999.

[12] F. Ramme, T. Römke, and K. Kremer. A distributed computing center software for efficient use of parallel computer systems. In *High-Performance Computing and Networking*, volume 2 of *Lecture Notes in Computer Science*, pages 129–136. Springer, 1994.

[13] Larry Smarr and Charles E. Catlett. Metacomputing. *Communications of the ACM*, 35(6):44–52, June 1992.

[14] R.Yahyapour U. Schwiegelshohn. Resource allocation and scheduling in metasystems. (1593):851–860, april 1999.

[15] Songnian Zhou. LSF: load sharing in large-scale heterogeneous distributed systems. In *Proceedings of the Workshop on Cluster Computing*, Tallahassee, FL, December 1992. Supercomputing Computations Research Institute, Florida State University.

SESSION 1-B
RESOURCE DISCOVERY AND MANAGEMENT

Chair: F. Darema, *NSF, USA*

Agent-Based Resource Discovery

Kyungkoo Jun, Ladislau Bölöni, Krzysztof Palacz, and Dan C. Marinescu
Computer Sciences Department,
Purdue University
West Lafayette, IN 47907
email: {junkk, boloni, palacz, dcm}@cs.purdue.edu

Abstract

In this paper we present a distributed discovery method allowing individual nodes to gather information about resources in a wide-area distributed system made up of autonomous systems linked together by a network technology substrate. We introduce an algorithm and a model for distributed awareness and a framework for dynamic assembly of agents monitoring network resources. Whenever an agent needs detailed information about individual components of another system it uses the information gathered by the distributed awareness mechanism to identify the target system, then creates a description of a monitoring agent capable of providing the information about remote resources, and sends this description to the remote site. There an agent factory assembles dynamically the monitoring agent. This solution is scalable and suitable for heterogeneous environments where the architecture and the hardware resources of individual nodes differ, the services provided by the system are diverse, the bandwidth and the latency of the communication links cover a broad range.

1. Introduction

In this paper we address the problem of resource discovery in a wide-area distributed system made up of autonomous systems linked together by a network technology substrate. The system is heterogeneous, the architecture and the hardware resources of individual nodes differ, the services provided by the system are diverse, the bandwidth and the latency of the communication links cover a broad range.

Individual nodes in such a distributed system may cooperate to accomplish tasks that require resources above and beyond those available in any single node, clients and servers may need to negotiate the quality of service, system administrators may wish to gather synthetic data regarding resource utilization to identify bottlenecks. A data intensive problem may generate a request to assemble dynamically a cluster of workstations with a compound CPU rate, memory, and secondary storage space determined by the problem size. A system administrator may wish to determine the overall secondary storage utilization in a virtual Intranet.

Resource management in a distributed system can be delegated to a subset of nodes providing site-coordination, negotiation, resource monitoring, and other services. For example the Open Data Network, ODN, model [13] is based upon an hourglass architecture with four layers: applications, middleware services, transport services, and bearer services provided by LANs, wireless networks, ATMs, satellite networks and so on. The architecture is conceived to support services ranging from teleconferencing to financial services, from remote login to interactive education. In turn middleware services cover security, name services, multi-site coordination, file systems and so on, and use transport services for video, audio, text, fax, and other types of data. The diversity of the networking substrate, the heterogeneity and autonomy of the nodes, the variety of services provided by the system make all aspects of resource management in this model rather challenging and motivate the desire to search for solutions that are more scalable and able to accommodate rapidly changing heterogeneous environments.

Distributed algorithms for resource management have been known for some time. The flooding algorithm is used by routers in the Internet, broadcasting by local queries, known as "gossiping" [11], [15] have been used to maintain consistency in replicated databases [3] and to gather information about system failures [4].

Autonomous and mobile software agents are widely regarded as necessary components of large-scale distributed systems. Agents can facilitate access to existing services to thin clients, support nomadic computing, perform functions related to resource management, support negotiations among several parties involved in a transaction, reconfigure servers, and so on. For example mobile agents to map network topology were proposed in [14].

Autonomy implies that the agents are active objects with

0-7695-0556-2/00 $10.00 © 2000 IEEE

their own tread of control, they can exhibit intelligent behavior. Mobility ensures that the agents can operate in rapidly changing heterogeneous environments. Yet, ensuring code mobility in a heterogeneous environment when the architecture of the nodes is different and we have several operating systems installed is a non-trivial endeavor.

The implicit assumption of agent-based solution for resource discovery in a wide-area system is the existence of a framework for the interoperabilty of different agent families, like the one proposed in [1]. Throughout this paper we assume that a system like the one described in Section 3.1 is installed in every node and the system has an agent factory, an object able to respond to external requests and assemble agents based upon a description of an agent provided by the entity that initiated the request.

In this paper we introduce an agent-based model for resource discovery. Agents running at individual nodes learn about the existence of each other using a mechanism called *distributed awareness*. Each agent maintains information about the other agents it has communicated with over a period of time and exchange periodically this information among themselves. Whenever an agent needs detailed information about individual components of the system we use the information gathered by the distributed awareness mechanism and then assemble dynamically agents capable of reporting the state of remote resources and to negotiate the use of these resources. The remote agent creation and surgery techniques discussed in Section 3.3 are general and allow us to alter drastically the behavior of an agent. For example we can add additional planes for resource negotiations with clients and with the local resource manager, planes to reconfigure a local server and so on.

The contributions of this paper are an algorithm and a model for the distributed awareness and a framework for dynamic assembly of agents capable of providing detailed information about network resources.

The rest of this paper is structured as follows. Section 2 reviews some of the existing algorithms for resource discovery, presents their basic assumptions and relevant performance measures. Then it presents our distributed awareness algorithm and two models for its behavior. Section 3 introduces the agent-based resource discovery architecture and describes an implementation based upon *Bond* [6], a component-based agent framework.

2. Algorithms and Models for Distributed Awareness

A first step in all applications that require some knowledge about the other nodes of a network is to learn about the existence of each other. We call this process "distributed awareness", while other authors [10] refer to it as resource discovery. We believe that in a heterogeneous environment learning about the existence of other nodes is only the first step in a complex process and that resource discovery requires a set of progressively more intricate interactions with the newly discovered object.

2.1. Related work

We review briefly some of the algorithms presented in the literature, their basic assumptions, and the proposed performance measures to evaluate an algorithm. Virtually all algorithms model the distributed system as a directed graph, in which each machine is a node and edges represent the relation "machine A knows about machine B". The network is assumed to be weakly connected and communication occurs in synchronous parallel rounds.

One performance measure is the *running time of the algorithm*, namely the number of rounds required until every machine learns about every other machine. The amount of communication required by the algorithm is measured by: (a) the *pointer communication complexity* defined as the number of pointers exchanged during the course of the algorithm, and (b) the *connection communication complexity* defined by the total number of connections between pairs of entities.

The *flooding* algorithm assumes that each node v only communicates over edges connecting it with a set of initial neighbors, $\Gamma(v)$. In every round node v contacts all its initial neighbors and transmits to them updates, $\Gamma(v)^{updates}$ and then updates its own set of neighbors by merging $\Gamma(v)$ with the set $\{\Gamma(u)^{updates}\}$, with $u \in \Gamma(v)$. The number of rounds required by the flooding algorithm is equal with the diameter of the graph.

The *swamping* algorithm allows a machine to open connections with all their current neighbors not only with the set of initial neighbors. The graph of the network known to one machine converges to a complete graph on $O(log(n))$ steps but the communication complexity increases. Here n is the number of nodes in the network.

In the *random pointer jump* algorithm each node v connects a random neighbor, $u \in \Gamma(v)$ who sends $\Gamma(u)$ to v who in turn merges $\Gamma(v)$ with $\Gamma(u)$. A version of the algorithm called *the random pointer jump with back edge* requires u to send back to v a pointer to all its neighbors. There are even strongly connected graphs that require with high probability $\Omega(n)$ time to converge to a complete graph in the random pointer jump algorithm.

The *Name-Dropper* algorithm is proposed in [10]. During each round each machine v transmits $\Gamma(v)$ to one randomly chosen neighbor. A machine u that receives $\Gamma(v)$ merges $\Gamma(v)$ with $\Gamma(u)$. In this algorithm after $O(log^2 n)$ rounds the graph evolves into a complete graph with probability greater than $1 - 1/(n^{O(1)})$.

2.2. Distributed Awareness; Algorithm and Models

2.2.1 A Distributed Awareness Algorithm

Distributed awareness is a mechanism for the nodes of a message–passing distributed system to learn about the existence of each other. Each node maintains an *awareness table* and exchanges the information in this table with other nodes. An entry in the awareness table contains: (1) `Node location`, the IP address of a node, (2) `lastHeardFrom`, the time when we last heard from the node, and (3) `lastSync` the time when the awareness information was last sent to the node. The awareness information is piggybacked onto regular messages exchanged between two nodes.

Incoming/outgoing message handling and table merging are discussed now. The algorithm to add new or update existing items is:

```
for every incoming message
   find sender, S
   if the local awareness table has an item I with
the same node location as S
      set lastHeardFrom of I as current time
   else
      add a new item initialized with S and last-
HeardFrom set as current time
if the incoming message has piggybacked awareness
information
   execute table merging algorithm
```

The table merging algorithm is:

```
for each awareness item, I, of the piggybacked
awareness table
   if the local table has item I_local with the same
node location of I
      set lastHeardFrom of I_local with more recent
time stamp between those of I_local and I
   else
      add I to the local table with lastSync set
zero
```

The outgoing message handling algorithm appends the local awareness table to the outgoing message:

```
for an outgoing message M_outgoing destined to a
node N
look up an item I with node location N in the
local table
if lastSync of I reached a specified age,
   add the local table to M_outgoing
   set lastSync of I as current time
send out M_outgoing
```

Notice that `lastSync` is checked to control the interval between sending awareness information and that the awareness table is periodically purged based upon `lastHeardFrom` field.

2.2.2 Deterministic and Non-deterministic Models

Modeling and analysis of the distributed awareness algorithm is rather difficult. The problem is unstructured, in the general case we do not know either the network topology or the communication patterns among nodes thus it is rather difficult to simplify assumptions leading to a tractable analysis. Yet we need to get a rough idea of the overhead incurred by this method and the asymptotic properties of the algorithm. Intuitively we expect that after some time all agents will learn about the existence of all other agents.

To model the distributed awareness we propose to use models similar to the ones for the spread of a contagious disease. An epidemic develops in a population of fixed size consisting of two groups the infected individuals and the uninfected ones. The progress of the epidemic is determined by the interactions between these two groups.

We introduce first a deterministic model. Given a group of n nodes this simple model is based upon the assumption that the rate of change in agent's awareness list, is proportional with the size of the group the agent is already aware of, y, and also with the size of the group the agent is unaware of, $n - y$. If k is a constant we can express this relation as follows:

$$y(t)' = k \times y(t) \times (n - y(t))$$

The solution of this differential equation with the initial condition $y(0) = 0$ is:

$$y(t) = \frac{n}{1 + (n-1)e^{-knt}}$$

This function is plotted in Figure 1 and shows that after time τ a node becomes aware of all the other nodes in the network. The parameter k as well as the value τ can be determined through simulation.

Call η the ratio of the awareness information exchanges to the total number of instances an agent communicates with other agents. A typical value for this parameter is $\eta = 0.001$. If the amount of awareness information is only a fraction b, say $b = 0.1$ of the payload carried out during communication between two agents, it follows that the additional load due to the distributed awareness is only a small fraction, in our example only $\eta \times b = 0.01\%$ of the total network traffic.

This deterministic model allows only a qualitative analysis. Rather than the smooth transition from 0 to n we should expect a series of transitions each one corresponding to a batch of newly discovered agents. Yet this simple model provides some insight into the overhead incurred during the learning phase of the awareness mechanism we propose.

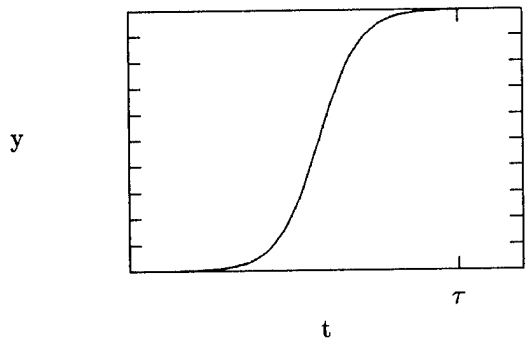


Figure 1. The number of agents known to a given agent, function of time, using a deterministic distributed awareness model. After time τ, each agent becomes aware of all the other agents in the network

A non-deterministic model is sketched below. New acquaintances occur in batches at time intervals determined by the overall rate of information exchange among nodes and by η. Call p the probability of contact between two agents such that as a result of the contact the awareness list are modified, and let $q = 1 - p$. Assume that the contacts between agents are stochastically independent and observe that the probability that among the i entries in the list supplied to an agent, $k, \leq i$ entries are not already in its list is

$$\binom{i}{k} \times p^k \times q^{i-k}$$

Call $Y(s)$ the random variable denoting the number of entries in the list of the "typical" agent at discrete time $s = 1, 2, \ldots$. Then

$$P(Y(s+1) = j | Y(s) = i) = \binom{i}{k} \times p^{i-j} \times q^j$$

if $i \geq j$ and zero otherwise.

The probability distribution of $Y(s+1)$ is independent of the values assumed by the random variables $Y(r)$, $r \leq s$. Therefore $(Y(s))_{s \geq 0}$ is a Markov chain with states $0, 1, \ldots..n$ and the transition matrix is:

$$\begin{array}{c} \\ 0 \\ 1 \\ 2 \\ . \\ . \\ . \\ r \end{array} \begin{array}{c} \begin{array}{ccccc} 0 & 1 & 2 & \ldots & r \end{array} \\ \begin{bmatrix} 1 & 0 & 0 & \ldots & 0 \\ p & q & 0 & \ldots & 0 \\ p^2 & 2pq & q^2 & \ldots & 0 \\ . & . & . & \ldots & . \\ . & . & . & \ldots & . \\ . & . & . & \ldots & . \\ p^r & rp^{r-1}q & \binom{r}{2}p^{r-2}q^2 & \ldots & q^r \end{bmatrix} \end{array}$$

Once a large system is operational we can attempt to determine the parameters of the model, including the transition probabilities, and then validate the model. The large number of parameters make this model very cumbersome for analysis of a realistic system, with a large number of nodes. The model is useful for theoretical studies, assuming different communication patterns, but this is beyond the scope of this paper.

3. Monitoring Agents and Resource Discovery

Information about the resources and the state of the nodes of a wide area distributed system is sometimes needed to coordinate the activity of a group of nodes, to provide synthetic information about resource utilization, or for other needs. A common approach taken by commercial as well as research systems is to install on each node a monitor to gather local resource information. The local monitors may update periodically a centrally stored database or provide the information on demand. Sometimes the information may be stored in servers hierarchically arranged.

Several metacomputing projects [9], [7] rely on a group of central entities to maintain the resource information reported by local entities. Globus [9] provides a *Metacomputing Directory Service* where network resource information is stored in a tree-like structure and it is accessible using the Lightweight Directory Access Protocol [16]. Local monitors residing on each node report the structure and state of resources. Monitors have to be installed and configured for each site. Legion [7] uses *collections* as repositories for information describing the state of the resources comprising the system. The collection is a database of static information reported by remote monitors. Resource management software provided by several companies including Tivoli [2] follow the same paradigm.

The information provided by a local monitor is determined at the time the monitoring program is installed. To provide additional information the program must be modified and reinstalled, and also it must be non-intrusive. Often the information obtained from static databases is obsolete. These considerations justify the need to investigate alternative means for gathering resource information.

Using software agents for resource discovery and monitoring has several advantages over the traditional approaches outlined above. Monitoring agents have an autonomous behavior and evolve based upon the characteristics of the local system and the requirements. Agents can engage in a gradual discovery process and respond to a changing set of requirements. They are able to adapt to the architecture and the operating environment of local nodes. An agent may decide its behavior based upon the results of an inference process thus the tasks assigned can be rather complex.

Now we describe an agent-based, distributed resource discovery architecture where agents are created at remote locations and modified as needed, to gather the information for resource management.

3.1. Bond; a Distributed Object System

Bond is a Java-based distributed object system and agent framework, with an emphasis on flexibility and performance. It is composed of (a) a core containing an object model and message oriented middleware, (b) a service layer containing distributed services like directory and persistent storage services, and (c) the agent framework providing basic tools for creating autonomous agents and a set of reusable components, called *strategies*, from which developers assemble agents with no or minimal amount of programming.

Bond Core. At the heart of the Bond system there is a Java Bean-compatible component architecture. Bond objects extend Java Beans by allowing users to attach new properties to the object during runtime, and offer a uniform API for accessing regular fields, dynamic properties and Java Bean style `setField/getField`-defined virtual fields. This allows programmers the same flexibility like languages like Lisp or Scheme, while maintaining the familiar Java programming syntax.

Bond objects are network objects; they can be both senders and receivers of messages. No post-processing of the object code as in RMI or CORBA-like stub generation, is needed. Bond uses *message passing* while RMI or CORBA-based component architectures use *remote method invocation*.

The system is largely independent from the message transport mechanism thus several communication engines can be used interchangeably. We currently provide TCP–based, UDP–based, Infospheres–based, and, separately, a multicast engine. Other communication engines will be implemented as needed. The API of the communication engine allows Bond objects to use any communication engines without changing or recompiling codes. On the other hand, the properties of the communication engine are reflected in applications as a whole. For example the UDP–based engine offers higher performance but does not guarantee reliable delivery.

All Bond objects communicate by an agent communication language, KQML [8]. Recently XML–based inter–agent communication was provided as an alternative to KQML. Bond defines the concept of *subprotocols*, highly specialized, closed set of commands. Subprotocols generally contain the messages to perform a specific task. Examples of generic Bond subprotocols are *property access* subprotocol, *agent control* subprotocol or *security* subprotocol.

Subprotocols group the same functionality of messages which in a remote method invocation system would be grouped in *interface*. But the larger flexibility of the messaging system allows for several new techniques which are difficult to implement in the remote method call case:

- The subprotocols implemented by an object are properties of the object, thus two objects can use the property access subprotocol, which is implemented by every Bond object, to find the common set of subprotocols between them.
- An object is able to control the internal path of a message delivery and to delegate the processing of the message to subcomponents called *regular probes*. Regular probes can be attached dynamically to an object as needed.
- Messages can be intercepted before they are delivered to the object, thus providing a convenient way for fire wall, accounting, logging, monitoring, filtering or preprocessing. These operations are performed by subcomponents called *preemptive probes*.
- Subprotocols, like interfaces, group some functionality of the object, which may or may not be used during its lifetime. A subcomponent called *autoprobe* allows the object to instantiate a new probe, to handle an incoming message which could not be understood by existing probes.
- Objects can be addressed by their unique identifier, or by their *alias*. Aliases specify the services provided by the object or its probes. An object can have multiple aliases and multiple objects can be registered under the same alias. The latter enables the architecture to support *load balancing* services.

These techniques can be implemented through different means in languages which treat methods as messages, e.g. Smalltalk. In Java and C++ they can be implemented at compile time, not at runtime, e.g. using the delegation design pattern. Techniques from the recent CORBA specifications e.g. the simultaneous use of DII, POA, trading service and others, also allow to implement a similar functionality, but with a larger overhead, and significantly more complex code.

Bond Services. Bond provides a number of services commonly found in distributed object systems, like directory, persistent storage, monitoring and security. Event, notification, and messaging services, which provide message passing services in remote method invocation based systems are not needed in Bond, due to the message-oriented architecture of the system.

Some of Bond services perform differently than their counterparts in other middleware systems, like CORBA. For example, Bond never requires explicit registration of a new object with a service. Finding out the properties of a remote object, i.e. the set of subprotocols implemented by the object, is achieved by direct negotiation among the objects. The directory service in Bond combines the functionality of the naming and trading services of other systems and it is implemented in a distributed fashion. Objects are located by a search process which propagates from local directory to local directory. The directories are linked into a virtual network by a transparent *distributed awareness* mechanism, which transfers directory information by piggybacking on messages as discussed in the previous Section.

Compared with the naming service implementations in systems like CORBA or RMI, which are based on the existence of a name server, this approach has the advantage that there is no single point of failure, and the distributed awareness mechanism reconstitutes the network of directories even after catastrophic failures. However, a distributed search can be slower than lookup on a server, especially for large networks. For these cases, Bond objects can be registered to external directories, either to a CORBA naming service through a gateway object, or to external directory services based on LDAP.

3.2. Bond Agents

The *Bond agent framework* is an application of the facilities provided by the Bond core layer to implement collaborative network agents. Agents are assembled dynamically from components in a structure described by a multi-plane state machine [5]. This structure is described by a specialized language called `blueprint`. Bond also supports agent description in XML. The components (*strategies*) are loaded locally or remotely, or can be specified in interpretive programming languages embedded in the blueprint script. The state information and knowledge base of the agents are collected in a single object called *model of the world* which allows for easy checkpointing and migration of agents. The multiplane state machine describing the behavior of agents can be modified dynamically by *agent surgery*, which will be discussed shortly.

The *behavior* of the agent is described by the *actions* the agent is performing. The actions are performed by the strategies either as reactions to external events, or autonomously in order to pursue the *agenda* of the agent. The current state of the multiplane state machine (described by a *state vector*) is specifying the strategies active at a certain moment. The multiple planes are a way of expressing parallelism in Bond agents. A good technique is to use them to express the various facets of the agents behavior: sensing, reasoning, communication/negotiation, acting upon the environment and so on. The *transitions* in the agent are modifying the behavior of the agent by changing the current set of active strategies. The transitions can be triggered by internal events or from external messages - these external messages form the *control subprotocol* of the agent.

Strategies are reusable by having interface requirements. The Bond agent framework provides a strategy database, for the most commonly used tasks, like starting and controlling external agents or legacy applications. A number of base strategies for common tasks like dialog boxes or message handlers are also provided, which can be sub-classed by developers to implement specific functionality. External algorithms, especially if written in Java are usually easily portable to the strategy interface.

3.3. Remote Creation and Surgery of Monitoring Agents

In this section, we discuss the remote creation of an agent and its surgery. To illustrate the concepts outlined in Section 3.2 we present the creation and modification of a monitoring agent. Several entities are involved in this process: a *beneficiary agent* at the site where the resource information is needed, an *agent factory* at the target site, and possibly a *blueprint repository*. The target site is identified by the distributed awareness or by a name or directory service. To install a monitoring agent on the target site, the beneficiary agent needs to obtains a blueprint of a monitoring agent. The blueprint can be retrieved from the blueprint repository or created dynamically by an inferencing agent given a set of rules and facts. After that, a message containing the blueprint or the location of the repository and the blueprint name is sent to the agent factory. Figure 2 illustrates this process. A *Bond Resident* is a container object including directory, communicator, and all other objects. In this example the message sent by the beneficiary agent contains the blueprint:

```
(achieve :content assemble-agent
 :blueprint-program [agent blueprint])
```

The beneficiary agent in this example decides to create a single plane monitoring agent with the blueprint shown in Figure 3. Figure 4 shows the monitoring agent with one plane designed to gather information about the primary storage, e.g. the amount of physical memory available in the node, the amount of used and free storage, a list of the top users of memory, and so on. Notice that each plane describes a state machine.

The agent factory receives the message, interprets the blueprint, and creates a monitoring agent with one plane called `PrimaryStorage` using one strategy included in the blueprint as `JPython` program [12], associated with `MemoryCheck` state. The complete JPython strategy is

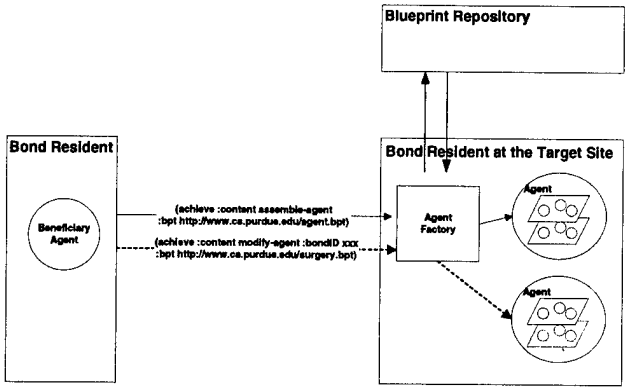


Figure 2. The communication between Beneficiary Agent, the Agent Factory and the Blueprint Repository. Messages instructing the agent factory to create a monitoring agent (solid line) and to perform surgery (dotted line) are shown.

shown in the Appendix. After creating the agent, the agent factory sends back an acknowledgment to the beneficiary agent.

Once started, the agent performs a transition to the `MemoryCheck` state. The Jpython strategy identifies the operating system running on that node and invokes the system calls, e.g. `vmstat` in Unix, necessary to gather the information about the primary storage. If successful, the state machines performs a transition to the `MemoryReport` state with strategy `ReportPS`, and sends back the information to the beneficiary agent named in the `BeneficiaryAddress` of the blueprint and finishes its execution by transition to the `Done` state with the `End` strategy.

The primary storage map changes in real time, thus it might be desirable to have an agent capable of reporting the information periodically. In addition, it may be necessary to gather information about secondary storage, e.g. the total amount of disk space available, the amount in use, the free disk space, the number of file systems, etc.

To obtain the periodic memory report and the secondary storage information, the agent can be modified through surgery as shown in Figure 5. In our example we (a) add another plane, called `SecondaryStorage`, to report the amount of free secondary storage space, and (b) modify the memory plane by adding transition from `MemoryReport` state to `MemoryCheck` state while deleting `Done` state and `gotoEnd` transition. As a result, the agent reports periodically the state of the primary storage. The reporting interval is specified in the blueprint as `Interval`, in this case, 5000 msec.

To perform the surgery, we send the agent factory at the

```
create agent MonitoringAgent
plane PrimaryStroage
   add state Init with strategy InitCheck;
   add state MemoryCheck with strategy language
   python embedded {:
      def getcmdresults(cmd):
      ''''''Run a command and return its output
            as a string and exit value
      ''''''
      .
      .
      def vmstat():
      ''''''Return the statistics
      ''''''from vmstat output
      ''''''in form of a hashtable
      [list, exitcode] =
         getcmdresults('vmstat 1 2')
      .
      .
      def save(map, prefix = ''):
      ''''''Save a hashtable into model
      .
      .
      save(vmstat(), 'discover.')
      self.fsm.transition(''gotoReport'');
   :}
   add state MemoryReport with strategy ReportPS;
   add state Done with strategy End;

   internal transitions {
      from InitCheck to MemoryCheck on gotoCheck;
      from MemoryCheck to MemoryReport
         on gotoReport;
      from MemoryReport to Done on gotoDone;
   }
   model {
      BeneficiaryAddress =
         ''ResourceAgent@peter.cs.purdue.edu:2000''
   }
end plane;
end create.
```

Figure 3. The blueprint of a monitoring agent designed to gather information about available physical memory, the amount of used and free storage, and a list of top memory users

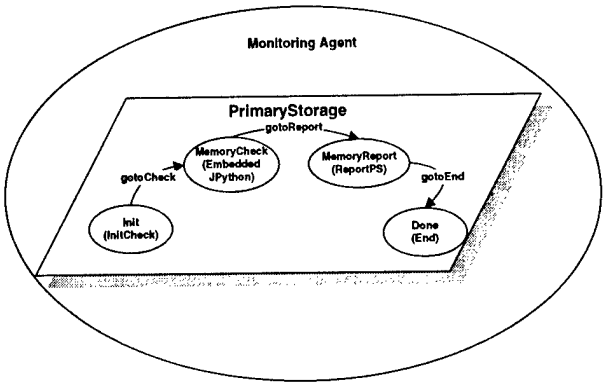


Figure 4. The monitoring agent built using the blueprint in Figure 3. The strategies associated with every state are shown in parenthesis.

```
modify agent Probing
plane SecondaryStorage
add state Init with strategy InitSS;
add state StorageCheck
   with strategy MeasureSS;
add state StorageReport
   with strategy ReportSS;
internal transitions {
   from InitSS to StorageCheck on gotoCheck;
   from StorageCheck to StorageReport
      on gotoReport;
   from StorageReport to StorageCheck
      on gotoCheck;
}
end plane;
plane PrimaryStorage
delete state Done;
internal transitions {
   delete from MemoryReport to Done
      on gotoEnd;
   from MemoryReport to MeamoryCheck
      on gotoCheck;
}
model {
   Interval = 5000;
}
end plane;
```

Figure 5. The agent surgery script. S second plane, `SecondaryStorage` is added and state machine of the first plane, `PrimaryStorage` is modifyed.

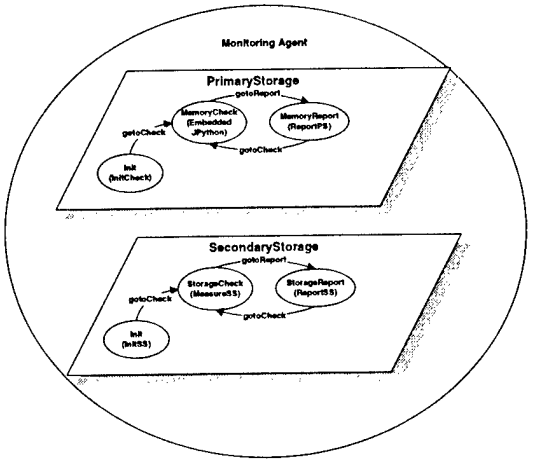


Figure 6. The agent after the surgery

target site the following message:

```
(achieve :content modify-agent :bondID [agent ID]
:blueprint-program [agent surgery script])
```

The message contains the unique Bond ID of the agent. This allows the agent factory to identify the target of the surgery request. Figure 6 shows the monitoring agent after the surgery of Figure 5. Agent surgery involves the modification of the data structure used to control the scheduling of various strategies in the planes of the agent. The surgery can be performed while the agent is running and the blueprint of the modified agent can be generated.

4. Conclusions

Information about the topology, resources and the state of the nodes of a wide area distributed system is sometimes needed to coordinate the activity of a group of nodes, to provide synthetic information about resource utilization, or for other needs. A common approach taken by commercial as well as research systems is to install on each node a monitor to gather local resource information. The local monitors may update periodically a centrally stored database or provide the information on demand.

Using software agents for resource discovery and monitoring has several advantages over the more traditional approach outlined above. Monitoring agents have an autonomous behavior and evolve based upon the characteristics of the local system and the requirements of the beneficiary agent. Agents can engage in a gradual discovery process and respond to a changing set of requirements. They are able to adapt to the architecture and the operating environment of the local node. An agent may change its behavior based upon the results of an inference process and

the tasks assigned to an agent can be rather complex. On the other hand, the amount of resources used by the agency may be larger than resources required by a custom-made monitoring system.

In this paper we introduce an agent-based model for resource discovery. Agents running at individual nodes learn about the existence of each other using a mechanism called *distributed awareness*. Each agent maintains information about the other agents it has communicated with over a period of time and exchange periodically this information among themselves. Whenever an agent needs detailed information about individual components of the system we use the information gathered by the distributed awareness mechanism and then assemble dynamically agents capable of reporting the state of remote resources and to negotiate the use of these resources. The remote agent creation and surgery techniques are general and allow us to alter drastically the behavior of an agent.

We present two models for distributed awareness, a deterministic model that supports a qualitative analysis and a more intricate, quantitative model. We introduce the Bond system and discuss the assembly and surgery of a monitoring agent capable to report the use of primary and secondary storage.

The Bond systems is available under an open source license from `http://bond.cs.purdue.edu`.

5. Acknowledgments

The work reported in this paper was partially supported by a grant from the National Science Foundation, MCB-9527131, by the Scalable I/O Initiative, and by a grant from the Intel Corporation.

6. Appendix: a JPython Strategy to Gather Memory Information

```
add state MemoryCheck with strategy language
python embedded
{:

from java.lang import Runtime, StringBuffer
from java.io import InputStream, StringWriter

import string

def getcmdresults(cmd):
    """Run a command and return its output
       as a string also return the exit
       value as the tuple's second arg
       Runtime.exec() writes some nonsense
       on standard output at least in Linux
    """
    p = Runtime.getRuntime().exec_(cmd)
    p.waitFor()
    output = p.getInputStream()
    buf = StringWriter()
    c = output.read()
    while c != -1:
       buf.write(c)
       c = output.read()
    return (buf.getBuffer().toString(), p.exitValue())

def accustat(param):
    """Accumulate information about users
       and return a hashtable requires System V
       ps (Solaris 2.x, newest Linux) see man page
       for parameter names, try e.g. pmem
    """
    [list, exitcode]
       = getcmdresults('ps -eo user,'+ param)
    if exitcode > 0:
       return None
    broken =  string.split(list, '\n')
    map = {}
    for line in broken[1:]:
       spl =  string.split(line)
       if len(spl) != 2:
        continue
        [user, param] = spl
        if map.has_key(user):
           map[user] = map[user] + string.atof(param)
    else:
           map[user] = string.atof(param)
    return map

def vmstat():
    """ Return the statistics from the vmstat
        output in form of a hashtable. See manual
        page for the meanings of the keys (system
        dependent although some are common).
    """
    [list, exitcode]
        = getcmdresults('vmstat 1 2')
    if exitcode > 0:
       return None
    broken =  string.split(list, '\n')
    names = string.split(broken[1])
    values = string.split(broken[3])
    map = {}
    i = 0
    for name in names:
       map[name] = string.atoi(values[i])
       i = i + 1
    return map

def save(map, prefix = ''):
    """ save a hashtable into the model with
        optional prefix (should include the dot)
    """
    for name in map.keys():
        model.set(prefix + name, map[name])
save(vmstat(), 'discover.')
self.fsm.transition("gotoReport")
:};
```

References

[1] MASIF - The CORBA Mobile Agent Specification. URL `http://www.omg.org/cgi-bin/doc?orbos/98-03-09`.

[2] Tivoli Enterprise Solutions. URL http://www.tivoli.com/products/solutions.

[3] D. Agrawal, A. Abbadi, and R. Steinke. Epidemic Algorithms in Replicated Databases. In *Proceedings of the 16th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 161–172, May 1997.

[4] S. Assmann and D. Kleitman. The number of rounds needed to exchange information within a graph. *SIAM Discrete Applied Math*, (6):117–125, 1983.

[5] L. Bölöni and D. C. Marinescu. A Multi – plane State Machine Agent Model. Technical Report CS TR 99-027, Computer Sciences Department, Purdue University, 9 1999.

[6] L. Bölöni and D. C. Marinescu. An Object-Oriented Framework for Building Collaborative Network Agents. In A. Kandel, K. Hoffmann, D. Mlynek, and N. Teodorescu, editors, *Intelligent Systems and Interfaces*. Kluwer Publising House, 1999.

[7] S. Chapin, D. Katramatos, J. Karpovich, and A. Grimshaw. Resource Management in Legion. In *Proceedings of the 5th Workshop on Job Scheduling Strategies for Parallel Processing in conjunction with the International Parallel and Distributed Processing Symposium*, April 1999.

[8] T. Finin et al. Specification of the KQML Agent-Communication Language – plus example agent policies and architectures, 1993.

[9] S. Fitzgerald, I. Foster, C. Kesselman, G. Laszewski, W. Smith, and S. Tuecke. A directory service for configuring high-performance distributed computations. In *Proceedings of the 6th IEEE Symp. on High-Performance Distributed Computing*, pages 365–375, 1997.

[10] M. Harchol-Balter, T. Leighton, and D. Lewin. Resource discovery in distributed networks. In *Proceedings of PODC'99*, pages 229–237, Atlanta, Georgia, 1999.

[11] S. Hedetniemi, S. Hedetniemi, and A. Liestman. A Survey of Gossiping and Broadcasting in Communication Networks. *Networks*, 1988.

[12] J. Hugunin. Python and java: The best of both worlds. In *Proceedings of the 6th International Python Conference*, San Jose, California, October 1997.

[13] L. Kleinrock and J. Major. *Defining A Decade*, chapter Computing and Communications Unchained: The Virtual World, pages 36–46. National Research Council, National Academy Press, 1999.

[14] N. Minar, K. H. Kramer, and P. Maes. *Cooperating Mobile Agents for Dynamic Network Routing*, chapter 12. Springer-Verlag, Berlin Germany, 1999.

[15] A. Pelc. Fault-tolerant Broadcasting and Gossiping in Communication. *Networks*, 1996.

[16] W. Yeong, T. Howes, and S. Kille. RFC 1777: Lightweight Directory Access Protocol, Mar. 1995. Obsoletes RFC1487. Status: DRAFT STANDARD.

Kyungkoo Jun is a PhD student and Research Assistant in the Computer Sciences Department, Purdue University, W. Lafayette, Indiana, USA, working with professor Dan C. Marinescu. He received an MS in Computer Science from Purdue University in 1998 and a BS with honors in Computer Science from Sogang University, Korea, in 1996. He is a member of the Upsilon Pi Epsilon, a computing honorary society. His research interests include resource management, Quality of Service in multimedia and autonomous agents.

Ladislau L Bölöni is a PhD student and Research Assistant in the Computer Sciences Department at Purdue University. He received a Master of Science degree from the Computer Sciences department of Purdue University in 1999 and Diploma Engineer degree in Computer Engineering with Honors from the Technical University of Cluj-Napoca, Romania in 1993. He received a fellowship from the Hungarian Academy of Sciences for the 1994-95 academic year. He is a member of ACM and the Upsilon Pi Epsilon honorary society. His research interests include distributed object systems, autonomous agents and parallel computing.

Krzysztof Palacz is a PhD student and Research Assistant in the Computer Sciences Department at Purdue University. He received a Master of Science degree in Computer Science in 1997 and in Physics in 1995 from Adam Mickiewicz University, Poznań, Poland. He is a member of ACM and the Upsilon Pi Epsilon honorary society. His research interests include distributed computing, workflow management systems and autonomous agents.

Dan C. Marinescu is Professor of Computer Sciences at Purdue University in West Lafayette, Indiana. He is conducting research in parallel and distributed computing, scientific computing, Petri Nets, and software agents. He has co-authored more than 110 papers published in refereed journals and conference proceedings. He was the chief architect of a distributed data acquisition and analysis system used in nuclear physics, now he is the PI of an NSF funded Grand Challenge project to solve large structural biology problems using parallel and distributed systems and the Director of the Bond project.

Evaluation of PAMS' Adaptive Management Services

Yoonhee Kim,
Department of Electrical Engineering and Computer Science,
Syracuse University
Syracuse, NY 13244
yhkim@ecs.syr.edu

Salim Hariri, and Muhamad Djunaedi
Department of Electrical and Computer Engineering, University of Arizona
Tucson, AZ 85721
{hariri, djunaedi}@ece.arizona.edu

ABSTRACT

Management of large-scale parallel and distributed applications is an extremely complex task due .to factors such as centralized management architectures, lack of coordination and compatibility among heterogeneous network management systems, and dynamic characteristics of networks and application bandwidth requirements. The development of an integrated network management framework that is proactive, scalable and robust is a challenging research problem. In this paper, we present our approach to implement a Proactive Application Management System (PAMS). PAMS architecture consists of two main modules: Application Centric Management (ACM) and Management Computing System (MCS). The ACM module provides the application developers with all the tools required to specify the appropriate management schemes to manage any quality of service requirement or application attribute/functionality (e.g., performance, fault, security, etc.). The MCS provides the core management services to enable the efficient proactive management of a wide range of network applications. The services offered by the MCS are implemented using mobile agents. Furthermore, each MCS service can be implemented using several techniques that can be selected dynamically by invoking the corresponding mobile agent template for the service implementation. In this paper, we present our preliminary results of evaluating PAMS management services to manage the performance and fault tolerance execution of three applications of different sizes (small, medium and large). The experimental results demonstrate that our agent-based approach can lead to significant gains in the performance and low overhead fault management of parallel/distributed. For example, the overhead incurred in the application fault management to tolerate one task failure, two task failures, and three task failures in a medium to large size application is less than 0.02%.

1. Introduction

The emerging high speed networks and the advances in computing technology are important driving forces to merge the communications and computing technologies that will result in an explosive growth in network complexity, size and networked applications. Furthermore, we are observing an explosive growth in network applications that use computing, networking and storage resources that can be accessed from global national and/or international networks. The management of such networks and their distributed applications has become increasingly complex, and unmanageable. Unfortunately, the current network management technologies focus on collecting management information and manually manage the network using platform-specific products. There has been little research toward the development of intelligent, efficient, proactive end-to-end management of large networks and their applications.

The increased importance of network management for large-scale networks has stimulated research on novel approaches to reduce the management complexity and cope with dynamic management change. Instead of a centralized manager, multi-managers and their communication protocols are proposed such as Management by Delegation (MbD)[4] and Code Mobility[5]. Another approach replaces the manger-agent relationship among managers and agents with peer-to-peer relationship using the Common Object Request Broker Architecture (CORBA) has been studied in the area of Telecommunications Information Networking Architecture (TINA) framework [2]. A few web-based approaches to network management have emerged recently (JMAPI, WEBEM). [3].

However, distributed network management of applications over heterogeneous has not fully studied and is becoming increasingly important. Recently, Application Management MIB [7] and MIB for Application [6] have been proposed to collect and store common application management information in

0-7695-0556-2/00 $10.00 © 2000 IEEE

IETF. Common Information Model (CIM) by DMTF is proposed a similar process information definition for WBEM [Patrck98]. Still, there has been little work done to achieve programmable application management schemes and is not well understood.

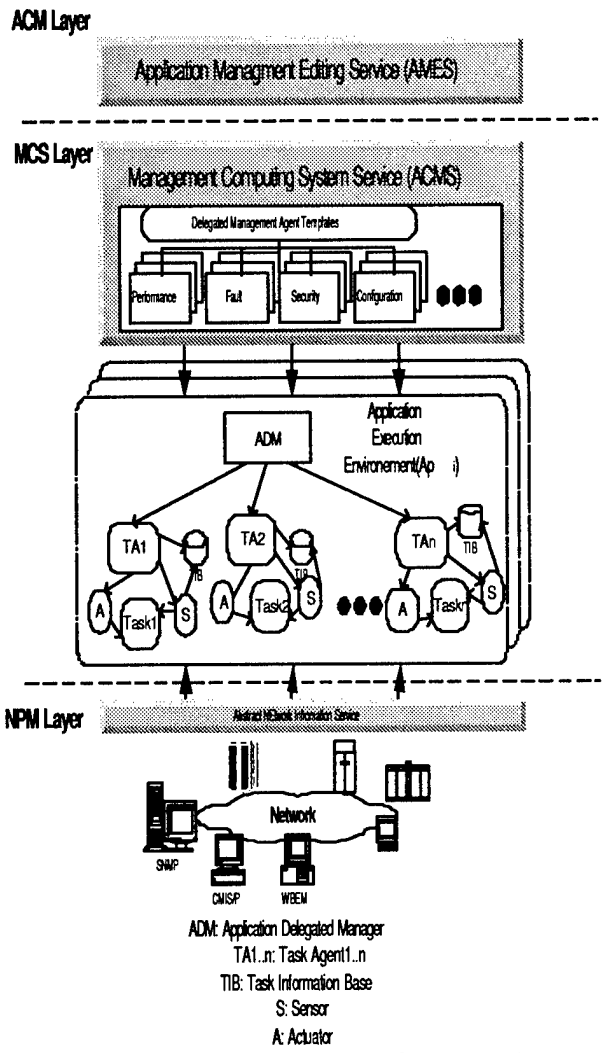


Figure 1. The Runtime Architecture of the Proactive Application Management System.

In this paper, we present the design and evaluation of a Proactive Application Management System (PAMS) prototype being developed at the University of Arizona. PAMS provides adaptive application management services to dynamically manage the performance and fault of parallel/distributed applications in an unreliable and heterogeneous computing environment. PAMS implementation is based on using mobile agents that can be programmed to maintain the quality of service requirements of distributed applications. We have evaluated three adaptive techniques to manage the performance and fault tolerance of distributed applications. The first approach is based on using active redundancy to improve performance and tolerate faults. The second approach is based on passive redundancy in which a set of machines is designated as backup machines to be used to replace any of the machines assigned to the application tasks in order to improve performance or to tolerate software/hardware failures. The third approach does not introduce redundancy in the system and it requires task migration to another machine in order to improve performance or to tolerate software/hardware failures. The preliminary results of applying these techniques demonstrate that our agent-based approach can lead to significant gains in the performance and low overhead fault management of parallel/distributed application. The organization of the paper is as follows. In Section 2, we give a brief overview of the PAMS prototype. In Section 3, we discuss our approach to benchmark and evaluate the adaptive performance management services offered by PAMS. In Section 4, we benchmark and evaluate the adaptive fault management service.

2. Architecture of the Proactive Application Management System (PAMS)

The architecture of PAMS is shown in Figure 1. The ACM layer provides application developers with the tools required to specify and characterize the application requirements in terms of performance, fault, security, and also specify the appropriate management scheme to maintain the application requirements. Once the application management requirements are defined using the ACM tools, the next step is to utilize the management services provided by the Management Computing System (MCS) to build the appropriate application execution environment that can dynamically control the allocated resources to maintain the application requirements during the application execution. The MCS assigns one Application Delegated Manager (ADM) to manage one or more application attributes (performance, fault, security, etc.). For each task in the application, the ADM launches an appropriate Task Agent (TA) to monitor and manage the task execution. The TA monitors the task execution using appropriate task sensors and intervenes whenever the task execution on the assigned machine can not meet its requirements using the task actuators that can suspend, save task execution state, or migrate the task execution to another remote machine. Our approach supports

several strategies to maintain each task attribute. For example, to manage the task performance, ADM could use active redundancy, passive redundancy, or by migrating the task execution to a faster machine when the assigned machine becomes heavily loaded. The appropriate management scheme can be selected at runtime depending on the system state and the current available resources as will be discussed in further detail later.

The main management activities of TA can be abstracted into three procedures or functions: Change_Detection, Analsis_Verification, and Adaptation_Plan. The Change_Detection procedure is responsible for detecting the conditions in which the monitored tasks deviates from the acceptable behavior or operation (e.g., the task performance degrades severely due to bursty traffic conditions, or due to software or hardware failures). The Analysis_Verification algorithm is invoked whenever a change is detected and to make sure that the change is real and not due to false alarms. Once the change event is verified and its type is identified, the Adaptation Plan procedure is invoked to execute the appropriate adaptation scheme.

```
Proactive_Application_Management Algorithm
1 For each Ap Api∈ ACM(Api),
2       Assign Application Delegated Manager ADM
(Api)
3       Lunch ADM (Api)
4       While (AEE(Api) is running) do
5               For each Service Si∈ APi
6                       Si ∈ {Sft, Sperf, Ssecurity, Sconfig}
7                       Start Service Si(Api),
8                       Monitor Si(Api)
9               EndFor
10      EndWhile
End Proactive_Application_Management_Algorithm
```

Figure 2 Proactive Application Management Algorithm

Figure 2 shows the general Proactive Application Management Algorithm for the PAMS prototype. The application Execution Environment (AEE(Api)) refers to all the resources allocated to run a give application ApI . While the application is running (step 4 in the Proactive Application Management Algorithm of Figure 2), the ADM starts all the task agents required to manage the application requirements (performance, security, fault, etc.) (Step 7,8 in the algorithm of Figure 2) and then monitor the execution of that application to detect any changes or deterioration while it is running. In what follows, we discuss PAMS approach to use mobile agents to manage the performance and fault tolerance of parallel/distributed applications.

3. Adaptive Performance Application Management

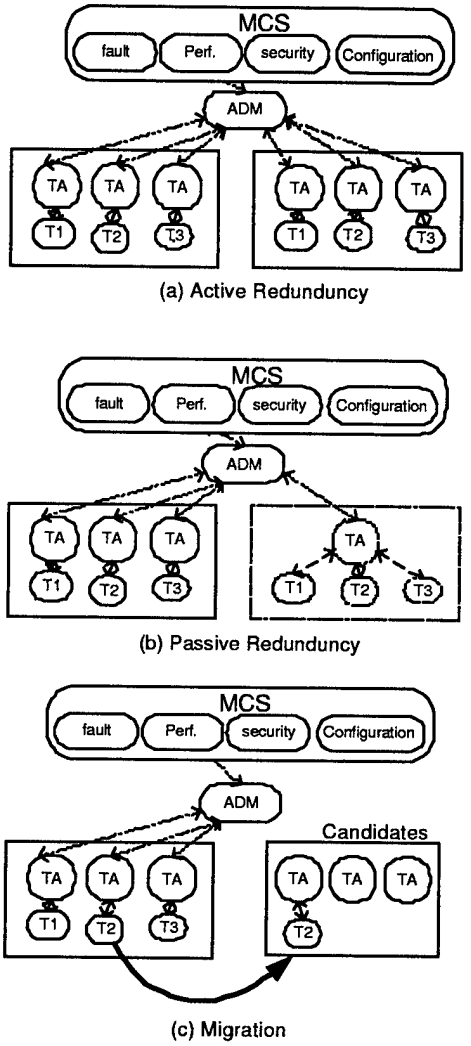


Figure 3 Controlling Techniques of Performance Management

Performance management for distributed systems is complex due to the existence of many components that need to be monitored and controlled. Performance management techniques can be broadly characterized into two schemes: monitoring and controlling. Monitoring is the function that tracks the performance activities of the resources, networks and their applications. The controlling function enables performance management to make adjustments to

improve performance. We need algorithms and techniques to derive appropriate performance metrics [9][10], and resource indicators for different levels of performance. Adjusting threshold schemes [13] and polling intervals [14] are the main issues in implementing the performance monitoring function. Performance statistics can be used to recognize potential bottlenecks or failures before they cause problems. Five major prediction models for performance predictions for parallel or distributed applications are discussed in [10]. With performance prediction, performance management schemes can proactively manage large and complex systems. Dynamic load-balancing [12] and process migration [11] have also been studied to provide appropriate performance management.

In our application performance management, we monitor the execution times of an application as well as the resource and network utilization. In addition, we use redundancy techniques and task migration to implement the control functions required to dynamically manage the application performance. In this paper, we evaluate three techniques to manage the application performance: active redundancy, passive redundancy and migration. Each technique is implemented as an agent template as shown in Figure 3.

The active redundancy scheme duplicates the execution of the application on two machines (see Figure 3 (a)). In this scheme, the task agent will pick up the results from the first machine that completes the task execution. This approach has several advantages. First, lead to better performance because we always pick up the results from the faster machine. Second, it simplifies the performance management since no need to perform task migration or load balancing in the system due to load changes or bursty traffic conditions.

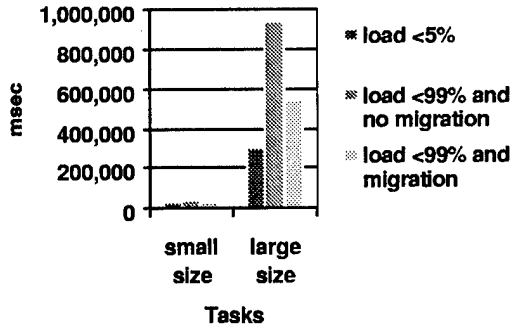


Figure 4 Application Execution with migration scheme

The passive redundancy assigns each task to a primary machine that will run the task and another machine to be used as a backup whenever the task performance deteriorates on the assigned machine (see Figure 3 (b)). The backup machine is kept-up-to-date in order to be ready to resume the task execution from the last updated checkpoint. The main advantage of this approach is that it needs less resources than the active redundancy approach. In this scheme, one backup machine can be used as a backup machine to several tasks.

The third approach does not introduce redundancy and improves the performance by task migration (see Figure 3 (c)). However, the overhead of task migration is high and it should be used only for large task granularities where the migration overhead is relatively small when compared to the task execution time.

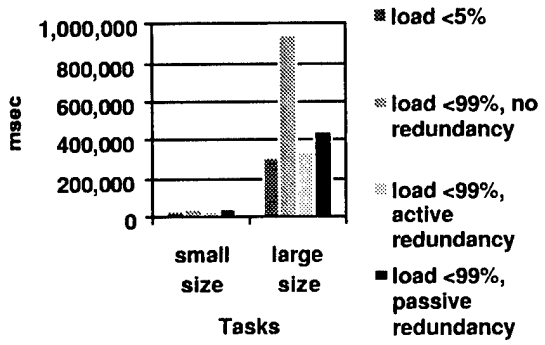


Figure 5 Application Execution with Redundancy policies

We benchmarked the overhead associated with implementing PAMS performance management service for two application types: a small application with an average execution time of 30 seconds and a large application with an average execution time of 450 seconds. We evaluated the use migration, active redundancy and passive redundancy techniques to dynamically mange the performance of these two applications. If, during the application execution, the load on a machine suddenly increased to 99% CPU utilization, the migration approach was able to improve the performance by 25% for the small size application (approximately 40 seconds) and by 75% for the large application (approximately 308 seconds) as shown in Figure 4. The active redundancy technique achieved a 31% performance gain for the small application and 174% for the large application as shown in Figure 5. Similar results were achieved in the passive redundancy approach, where a 22% performance gain was achieved for the small

application and a 114% performance gain for the large application.

4. Adaptive Fault Tolerance

The main goal of the application fault management is to efficiently recover from hardware/software failures of the system resources. Redundancy is an important technique to detect and recover from component failures in the system. The redundancy can be in the form of hardware, software, or time [15]. As the system increases its complexity, more sophisticated techniques are needed to manage those redundancies. In addition, the fault management scheme must be flexible and adaptive. In SCOP [17], a design methodology is proposed to introduce support techniques to reduce the resource cost of fault-tolerant software, both in space and time, by providing designers with a flexible redundancy architecture in which dependability and efficiency can be adjusted dynamically at run time. In another work [18], the use of mobile agents to support adaptive fault tolerance is implemented. In our adaptive application fault-tolerance approach, we use mobile agents to efficiently manage the redundancy. We evaluate two redundancy techniques: Passive and Active redundancy.

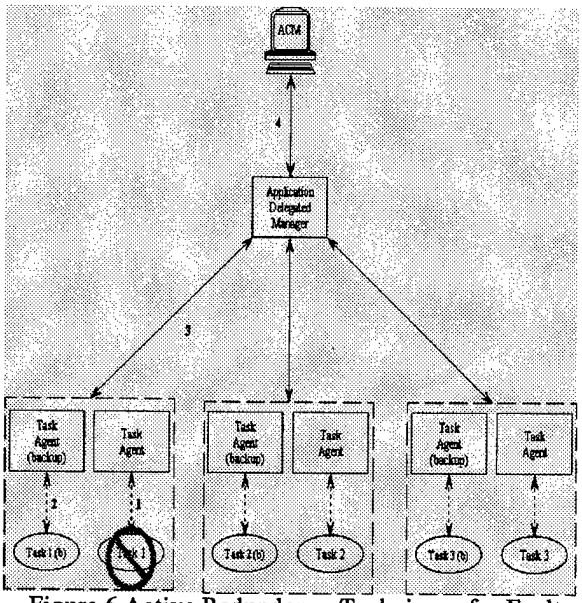


Figure 6 Active Redundancy Techniques for Fault Management

In the active redundancy technique shown in Figure 6, we assign two identical tasks to two machines that are managed by two Task Agents (TAs); one task is designated as the primary task while the second one is referred to as the secondary task. In this scenario, the ADM doesn't need to determine the adaptation plan when a fault occurs. If the fault occurs in the primary task, the results can be picked up without any delay from the secondary task that becomes the new primary task once its task agent detects the failure in the primary task due to software or hardware failures. In addition to reducing the time for fault detection, active redundancy technique simplifies the communication between task agents. Figure 8 shows the overhead incurred by applying this redundancy scheme to adaptively manage the faults of three applications with three tasks each. In the small application case (execution time is around 60s), the overhead incurred in using our scheme to detect and recover from one task failure, two task failures, and three task failures are 0.10%, 0.18%, and 0.22%, respectively (see Figure 7). For medium and large applications, the overhead in managing one, two or three task failures is very small (less than 0.02%).

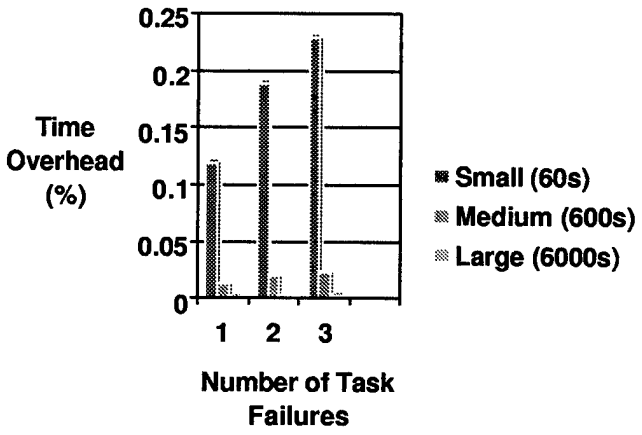


Figure 7 The overhead of Active Redundancy Technique

The second approach is based on using passive redundancy in managing the application faults (see Figure 8). In this scenario, we assign the task to two machines: one is designated as the primary machine while the second machine is designated as the backup machine. The backup machine does not run the task as is done in the active redundancy case, but it is kept up-to-date about the task execution periodically so it can resume the task execution from the last checkpoint (update) if a fault occurred in the primary task. Furthermore, the backup machine could be assigned as

a backup machine for more than one task. This improves the utilization of the system resources. Figure 9 shows the overhead incurred in applying this redundancy technique to manage the faults of three applications. For a small application with three tasks, the overhead incurred to manage one task failure, two task failures, and three task failures are 0.18%, 0.26%, and 0.42%. For a medium to large size application, the overhead to manage one, two or three task failures is very small (less than 0.02%).

It is clear from the experimental results that our approach is very efficient, especially, for large parallel/distributed applications. Furthermore, the use of mobile agents and agent templates, we can dynamically select the appropriate redundancy technique at runtime depending on the system load and number of available resources.

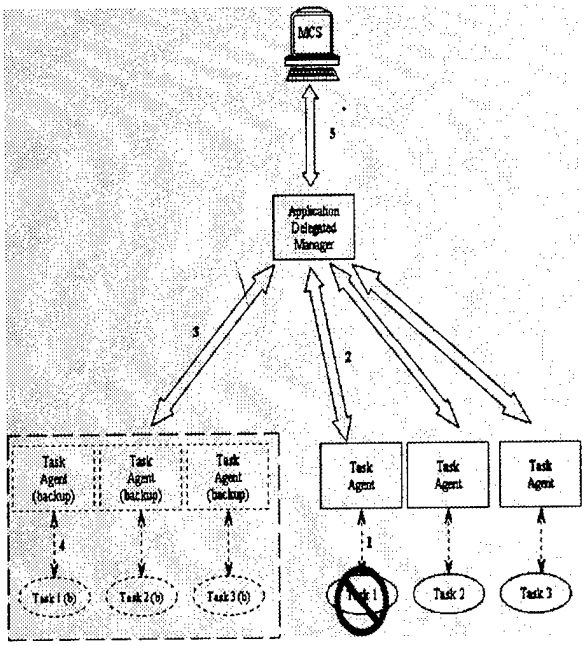


Figure 8 Passive Redundancy Techniques for Fault Management

5. Conclusion

In this paper, we presented our approach to implement a Proactive Application Management System (PAMS). The PAMS architecture is based on integrated management framework being developed at the University of Arizona [8]. The experimental results of the PAMS management services to manage the performance and fault tolerance execution of three applications of different sizes (small, medium and large demonstrate that our agent-based approach can lead to significant gains in performance and low overhead in fault management. We are currently implementing additional services to balance the load across the network resources and maintain the system and application security requirements.

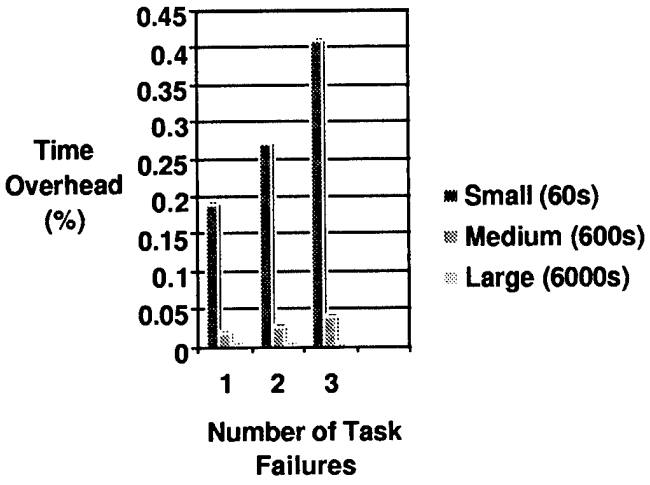


Figure 9 The overhead of Passive Redundancy Technique

6. Reference

[1] S. Waldbusser, Remote Network Monitoring Management Information Base RFC1757, Feb. 1995.

[2] J. Pavon and J. Tomas, CORBA for Network and Service Management in the TINA Framework, IEEE Communication Magazine, March 1998.

[3] J. P. Thompson, Web-Based Enterprise Management Architecture, IEEE Communication Magazine, March 1998

[4] G. Goldszmidt and Y. Yemini, Distributed Management by Delegation, in 15th international Conference on Distbuted Computing, June 1995.

[5] M. Baldi, S. Gai and G. Picco, Exploiting Code Mobility in Decentalized and Flexible Network Management, In First International Workshop, MA97, Berlin, Germany, April 97.

[6] C. Krupczak and J. Saperia, Definition of System-Level Managed Objects for Applications, RFC2287, Feb 1998.

[7] C. Kalbfleisch, C. Krupczak, R. Presuhn, and J. Saperia, Application Management MIB, Internet-draft, Nov. 98.

[8] S. Hariri, Y. Kim, P.Varshney, R.Kamiski. D haugue, C Maciag, The End-to-End Proactive Management. IEEE/IFIP 1998 Network Operations and management Symposium, Feb. 1998
[9] Michael Katchabaw, Stephen Howard, Andrew Marshall, Michael Bauer, "Evaluating the Cost of Management: A Distributed Application Management Testbed," Proceeding of the 1996 CAS conference(CASCON'96) Toronto, Canada, Nov.12-14 pp29-41.
[10] Tomas Fahringer, "Automatic Performance Prediction of Parallel Programs" Kluwer Academic Publishers, 1996
[11] Michael Litzkow, "Supporting Checkpointing and Process Migration outside the Unix Kernel," Usenix Winter Conference, San Francisco, California, 1992
[12] Mohammed Zaki, Wei Li, Srinivasan Parthasarathy "Customized Dynamic Load Balancing of a Network of Workstations,"Technical Report 602, Dec. 1995
[13] Marina Thottan, Chuanyi Ji, "Adaptive Thresholding for Proactive Network Problem Detection, Procedding of the 1998 international workshop for Systems Management, Newport, April, 1998.
[14] P Dini, G. Bochmann,T. Koch, B. Kramer, "Agent based Management of Distributed Systems with Variable Polling Frequency Policies,"
[15] A. Avizienis. "Fault-Tolerant Systems." IEEE Transactions on Computers, C-25(12):1304-1312, December 1976.
[16] P. Jalote. "Fault Tolerance in Distributed Systems." Prentice Hall, 1994
[17] J. Xu, A. Bondavalli, F. D. Giandomenico. "Dynamic Adjustment of Dependability and Efficiency in Fault-Tolerant Software", in "Predictably Dependable Computing Systems", B. Randell, J. C. Laprie, H. Kopetz and B. Littlewood Ed., Springer-Verlag, 1995, pp.155-172.
[18] S. Bagchi, K. Whisnant, Z. Kalbarcyzk, R.K. Iyer. "Chameleon: Adaptive Fault Tolerance Using Reliable, Mobile Agents", The 27th Fault Tolerance Computer Symposium, Munich, Germany, June 23-25 1998

Yoonhee Kim is currently a Ph.D. candidate in the department of Electrical Engineering and Computer Science at Syracuse University and work in a research engineer position at the University of Arizona. She received her M.S. degree in Computer Information Science from Syracuse University, New York at 1996. Her research interests include system, network and application management, distributed and parallel computing systems, and software architecture. Email: yhkim@ece.arizona.edu

Dr. Salim Hariri is currently an Associate Professor in the Department of Electrical and Computer Engineering at The University of Arizona. Dr. Hariri received his Ph.D. in computer engineering from University of Southern California in 1986, and a M.Sc. degree from The Ohio State University. He is the Director of the Center for Advanced TeleSysMatics (CAT): Next-Generation Network-Centric Systems. His current research focuses on high performance distributed computing, agent-based proactive and intelligent network management systems, design and analysis of high speed networks, benchmarking and evaluating parallel and distributed systems, and developing software design tools for high performance computing and communication systems and applications. Dr. Hariri is the co-Editor-In-Chief for the *Cluster Computing*. Dr. Hariri served as the General Chair of the IEEE International Symposium on High Performance Distributed Computing (HPDC).

Muhamad Djunaedi received the B.S. degree in computer and electrical engineering from Purdue University in 1995. Since 1998, he has been studying for M.S. degree in electrical and computer engineering department at University of Arizona. His research interests include mobile agent, fault tolerance, distributed system and management of information system. Email: djunaedi@ece.arizona.edu

Load Balancing Across Near-Homogeneous Multi-Resource Servers *

William Leinberger, George Karypis, Vipin Kumar
Army High Performance Computing and Research Center
Department of Computer Science and Engineering, University of Minnesota
{leinberg, karypis, kumar}@cs.umn.edu

Rupak Biswas
MRJ Technology Solutions, Numerical Aerospace Simulation Division
NASA Ames Research Center, Moffett Field, CA 94035
rbiswas@nas.nasa.gov

Abstract

An emerging model for computational grids interconnects similar multi-resource servers from distributed sites. A job submitted to the grid can be executed by any of the servers; however, resource size or balance may be different across servers. One approach to resource management for this grid is to layer a global load distribution system on top of the local job management systems at each site. Unfortunately, classical load distribution policies fail on two aspects when applied to a multi-resource server grid. First, simple load indices *may not recognize that a resource imbalance exists at a server. Second, classical* job selection policies *do not actively correct such a resource imbalanced state. We show through simulation that new policies based on* resource balancing *perform consistently better than the classical load distribution strategies.*

1. Introduction

An emerging model in high performance supercomputing is to interconnect similar computing systems from geographically remote sites, creating a *near-homogeneous* computational grid system. Computing systems, or servers, are homogeneous in that any job submitted to the grid may be sent to any server for execution. However, the servers may be heterogeneous with respect to their exact resource configurations. For example, the first phase of the NASA Metacenter linked a 42-node IBM SP2 at Langley and a 144-node SP2 at Ames [7]. The two servers were homogeneous in that they were both IBM SP2s, with identical or *synchronized* software configurations. However, they were heterogeneous on two counts: the number of nodes in each server, and the fact that the Langley machine consisted of thin nodes while the Ames machine had wide nodes. A job could be executed by either server without modifications, provided a sufficient number of nodes were available on that server.

The resource manager for the near-homogeneous grid system is responsible for scheduling submitted jobs to available resources such that some global objective is satisfied, subject to the constraints imposed by the local policies at each site. One approach to resource management for near-homogeneous computational grids is to provide a *global load distribution system* (LDS) layered on top of the *local job management system* (JMS) at each site. This architecture is depicted in Figure 1. The compute server at each site is managed by a local JMS. Users submit jobs directly to their local JMS which places the jobs in wait queues until sufficient resources are available on the local compute server. The global LDS monitors the load at each site. In the event that some sites become heavily loaded while other sites are lightly loaded, the LDS attempts to equalize the load across all serves by moving jobs among the sites. The JMS at each site is then responsible for the detailed allocation and scheduling of local resources to jobs submitted directly to it, as well as to jobs which are assigned to it by the global LDS. The local JMS also provides load status to the LDS to support load distribution decisions, as well as a scheduling Applications Programming Interface (API) to implement these decisions. For example, in the NASA Metacenter, a *peer-aware* receiver-initiated load balancing algorithm was used to move work from one IBM SP2 to the other. When the workload on one SP2 dropped below

*This work was supported by NASA grant NCC2-5268 and contract NAS2-14303, and by Army High Performance Computing Research Center (AHPCRC) cooperative agreement DAAH04-95-2-0003 and contract DAAH04-95-C-0008. Access to computing facilities was provided by AHPCRC, Minnesota Supercomputer Institute.

0-7695-0556-2/00 $10.00 © 2000 IEEE

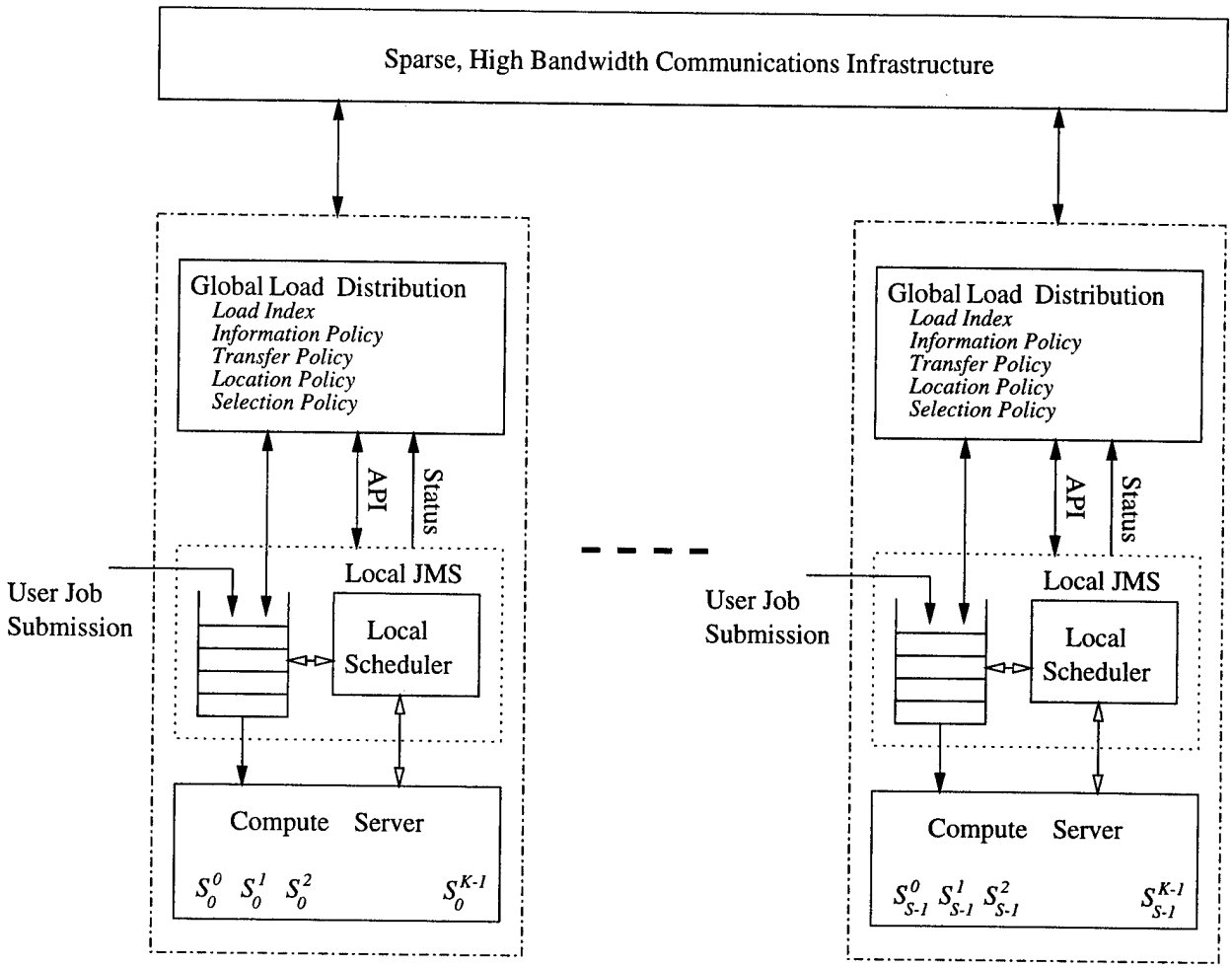


Figure 1. Near-Homogeneous Metacomputing Resource Management Architecture

a specified threshold, the peer-aware load balancing mechanism would query the other SP2 to see if it had any work which could be transferred for execution.

The architecture depicted in Figure 1 is conceptually identical to classical load balancing in a parallel or distributed computer with two notable exceptions. First, the compute server at each site may be a complex combination of multiple types of resources (CPUS, memory, disks, switches, and so on). Similarly, the applications submitted by the users are described by multiple resource requirements. We generalize these notions and define a K-resource server and corresponding K-requirement job. Each server S_i has K resources, $S_i^0, S_i^1, \ldots, S_i^{K-1}$. Each job J_j is described by its requirements for each resource type, $J_j^0, J_j^0, \ldots, J_j^{K-1}$. Note that the servers are still considered homogeneous from the jobs' perspective, as any job may be sent to any server for execution.

The second exception is that the physical configurations of the K resources for each server may be heterogeneous. This heterogeneity can be manifested in two ways. The *amount* of a given resource at one server site may be quite different than the configuration of a server at another site. For example, server S_i may have more memory than server S_j. Additionally, servers may have a different *balance* of each resource. For example, one server may have a (relatively) large memory with respect to its number of CPUs while another server may have a large number of CPUs with less memory.

Classical load balancing attempts to maximize system throughput by keeping all processors busy. We extend this notional goal to fully utilizing all K resources at each site. One heuristic for achieving this objective is to *match* the job mix at each server with the capabilities of that server, in addition to balancing the load across servers. For example, if a server has a large shared memory, then the job mix in the local wait queue should be adjusted by the global LDS to contain jobs which are generally memory intensive. Compute intensive jobs should be moved to a server which has

a relatively large number of CPUs with respect to its available memory. The goal of the LDS is to therefore balance the total resource demand among all sites, *for each type of resource*.

This work investigates the use of load balancing techniques to solve the global load distribution problem for computational grids consisting of near-homogeneous multi-resource servers. The complexity of multi-resource compute servers along with the multi-resource requirements of the jobs cause the methods developed in past load balancing research to fail in at least two areas. First, the definition of the *load* at a given server is not easily described by a single load index. Specifically, a *resource imbalance*, in which the local job mix does not match the capabilities of the local server, is not directly detectable. This impacts the ability of the global LDS to match the workload at a site to the capabilities of the site. We propose a simple extension to a classical load index measure based on a *resource balancing heuristic* to provide this additional level of descriptive detail. Second, once a resource imbalance is detected, existing approaches to selecting which jobs to move between servers fail to actively correct the problem. We provide an analogous job selection policy, also based on resource balancing, which heuristically corrects the resource imbalance. The combination of these two extensions provides the framework for a global LDS which consistently outperforms existing approaches over a wide range of compute server characteristics.

The remainder of this paper is organized as follows. Section 2 provides an overview of relevant past research, concluding with variants of a baseline load balancing algorithm drawn from the literature. Section 3 investigates the limitations of the baseline algorithms, and provides extensions based on the resource balancing heuristic. A description of our simulation environment is given in Section 4. The performance results of our new load balancing methods as compared to the baseline algorithms is also summarized in Section 4. Finally, Section 5 provides conclusions and a brief overview of our current work in progress.

2. Preliminaries

Research related to this effort is drawn from single server scheduling in the presence of multiple resource requirements and general load balancing methods for homogeneous parallel processing systems.

Recent research in job scheduling for a single server has demonstrated the benefits of including information about the memory requirements of a job in addition to its CPU requirements [13, 14]. The generalized K-resource single server scheduling problem was studied in [10], where it was shown that simple backfill algorithms based on multi-dimensional packing heuristics consistently outperform single-resource algorithms, with increasing K. These efforts all suggest that the local JMS at each site should be multi-resource *aware* in making its scheduling decisions. This induces requirements on the global LDS to provide a job mix to a local server which maximizes the success rate of the local server.

The general goal of a workload distribution system is to have sufficient work available to every computational node to enable the efficient utilization of that node. A centralized work queue provides every node equal access to all available work, and is generally regarded as being efficient in achieving this goal. Unfortunately, the centralized work queue is generally not scalable as contention for the single queue structure increases with the number of nodes. In massively parallel processing systems where the number of nodes was expected to reach into the thousands, this was a key concern. In distributed systems, the latency for querying the central queue potentially increases as the number of nodes is increased. Load balancing algorithms attempt to emulate a central work queue by maintaining a representative workload across a set of distributed queues, one per compute node. In this paper, we investigate only the performance of load balancing across distributed queues.

Classical load balancing algorithms are typically based on a *load index* which provides a measure of the workload at a node relative to some global average, and four *policies* which govern the actions taken once a load imbalance is detected [15]. The load index is used to detect a load imbalance state. Qualitatively, a load imbalance occurs when the load index at one node is much higher (or lower) than the load index on the other nodes. The length of the CPU queue has been shown to provide a good load index on time-shared workstations when the performance measure of interest is the average response time [2, 11]. In the case of multiple resources (disk, memory, etc.), a linear combination of the length of all the resource queues provided an improved measure, as job execution time may be driven by more than CPU cycles [5].

The four policies that govern the action of a load balancing algorithm when a load imbalance is detected deal with information, transfer, location, and selection. The *information* policy is responsible for keeping up-to-date load information about each node in the system. A global information policy provides access to the load index of every node, at the cost of additional communication for maintaining accurate information [1].

The *transfer* policy deals with the dynamic aspects of a system. It uses the nodes' load information to decide when a node becomes eligible to act as a sender (transfer a job to another node) or as a receiver (retrieve a job from another node). Transfer policies are typically threshold based. Thus, if the load at a node increases beyond a threshold T_s, the node becomes an eligible sender. Likewise, if the load

at a node drops below a threshold T_r, the node becomes an eligible receiver. Load balancing algorithms which focus on the transfer policy are described in [2, 15, 16].

The *location* policy selects a partner node for a job transfer transaction. If the node is an eligible sender, the location policy seeks out a receiver node to receive the job selected by the selection policy (described below). If the node is an eligible receiver, the location policy looks for an eligible sender node. Load balancing approaches which focus on the use of the location policy are described in [8, 9].

Once a node becomes an eligible sender, a *selection* policy is used to pick which of the queued jobs is to be transferred to the receiver node. The selection policy uses several criteria to evaluate the queued jobs. Its goal is to select a job that reduces the local load, incurs as little cost as possible in the transfer, and has good affinity to the node to which it is transferred. A common selection policy is *latest-job-arrived* which selects the job which is currently in last place in the work queue.

The primary difference between existing load balancing algorithms and our global load distribution requirements is that our node is actually a *multi-resource server*. With this extension in mind, we define the following baseline load balancing algorithm:

- Load Index. The load index is based on the average resource requirements of the jobs waiting in the queue at a given server. This index is termed the resource average (RA) index. For our multi-resource server formulation, each resource requirement for a job in the queue represents a *percentage* of the server resource that it requires, normalized to unity. Therefore, the RA index is a relative index which can be used to compare the loads on different servers.

- Information Policy. As the information policy is not the subject of this study, we choose to use a policy which provides perfect information about the state of the global system. We assume a global information policy with instantaneous update.

- Transfer Policy. The transfer policy is threshold based, since it has been shown to provide robust performance across a range of load conditions. A server becomes a *sender* when its load index grows above the global load average by a threshold, T_s. Conversely, a server becomes a *receiver* when its load index falls below the global average by a threshold T_r.

- Location Policy. The location policy is also not the subject of this study. Therefore, we use a simple location policy which heuristically results in fast convergence to a balanced load state. In the event that the transfer policy indicates that a server becomes a sender, the location policy selects the server which currently has the least load to be the receiver. However, the selected server must also be an eligible receiver, meaning that it currently has a load which is T_r below the global average. Conversely, if the server is a receiver, the location policy selects the server which currently has the highest load that is T_s above the global average. If no eligible partner is found, the load balancing action is terminated.

- Selection Policy. A latest-job-arrived selection policy (LSP) is used to select a job from the sending server to be transferred to the receiving server. This selection policy generally performs well with respect to achieving a good average response time, but suffers from some jobs being moved excessively. Therefore, each job keeps a *job transfer count* which records the number of times it has been moved. When this count reaches a threshold T_c, the job is no longer eligible to be selected for a transfer. Jobs which are already executing are excluded from being transferred.

The sender initiated (SI), receiver initiated (RI), and symmetrically initiated (SY) algorithm variants are generated using a transfer policy which triggers a load balancing action on T_s, T_r, or both, respectively. All baseline variants use the RA load index and the LSP job selection policy.

3. Multi-Resource Aware Load Balancing Policies

In this section, we first discuss the limitations of the resource average load index, RA, and the latest-job-arrived selection policy, LSP, of the baseline load balancing algorithms for the heterogeneous multi-resource servers problem. We provide an example which illustrates where these naive strategies can fail to match the workload to the servers, resulting in local workloads which exhibit a resource imbalance. We then provide extensions to the load index and the job selection policy which strive to balance the resource usage at each server.

3.1. Limitations of RA and LSP

The resource average load index, RA, and the latest-job-arrived job selection policy, LSP, in the baseline algorithm fail in the multi-resource server load balancing context. The following discussion gives an example of these failures and provides some insight into possible new methods. Our new methods will be further discussed in Section 3.2.

In past research, the index used to measure the load on a server with respect to multiple resources consisted of a

linear combination or an average of the resource requirements for the actively running jobs in a time-shared system. A corresponding index which may be applied to batch queued space-shared systems is to use the average of the total resource requirements of the jobs waiting in the wait queue. However, this may not always indicate a system state where there exists a resource imbalance, that is, the total job requirements for one resource exceeds the requirements for the other resources. Essentially, a server with a mismatched work mix will be forced to leave some resources idle while other resources are fully utilized, resulting in an inefficient use of the system as a whole.

Figure 2(a) depicts the state of the job ready queues, RQ_0 and RQ_1 for a two-server system, S_0 and S_1. Assume that each server has three resources, S_i^0, S_i^1, and S_i^2, and that the configuration for the two servers is identical, $S_0^0 = S_1^0, S_0^1 = S_1^1$, and $S_0^2 = S_1^2$. Each of the two ready queues currently has two jobs. The job which arrived latest at each server is on the top of the ready queue for that server. For example, the latest arriving job , J_L, in RQ_0 has the resource requirements $J_L^0 = 2, J_L^1 = 3$, and $J_L^2 = 2$. Note that the resource requirements for a job are given as a *percentage* of the total available in the server. The total workload for each resource, k, in a given server, S_i, is denoted as

$$W_i^k = \sum_{J_j \in RQ_i} (J_j^k), \quad 0 \leq i < S, \quad 0 \leq k < K.$$

The resource average load index for a given server, S_i, is then given by

$$RA_i = Avg(W_i^k), \quad 0 \leq k < K.$$

In this example, $K = 3$ and $RA_0 = RA_1 = 4$.

The third queue in Figure 2(a), RQ_{Avg}, represents the global average workload for each resource in RQ_0 and RQ_1. The global average workload for resource k, is then given by

$$W_{Avg}^k = Avg(W_i^k), \quad 0 \leq i < S.$$

Here, $S = 2$ and $W_{Avg}^0 = W_{Avg}^1 = W_{Avg}^2 = 4$, meaning that on average, each RQ_i has a total requirement of 4 percent for each resource. The global resource average load index is simply

$$RA = Avg(W_{Avg}^k), \quad 0 \leq k < K,$$

which in this example is $RA = 4$. Server S_i is defined to be in a load balanced state as long as $RA * (1 - T_x) < RA_i < RA * (1 + T_x)$, where T_x is the transfer policy threshold, as defined in Section 2. Since $RA_0 = RA_1 = RA$, the system is believed to be in a load balanced state.

Even though the RA index indicates a balanced load, it is clear from Figure 2(a) that the job mix in RQ_0 has a higher requirement for resource S_0^1 than for resources S_0^0 and S_0^2. The result is that S_0 will probably be unable to fully utilize resources S_0^0 and S_0^2 as resource S_0^1 becomes the bottleneck. Conversely, the job mix in RQ_1 has a higher requirement for resources S_1^0 and S_1^2 than for S_1^1, resulting in an inefficient use of resource S_1^1. Therefore, the workload at each server suffers from a resource imbalance.

In order to detect this problem, we define a second load index, called resource balance (RB), which measures the resource imbalance at a given server or globally across the system. Namely, for server $S_i, 0 \leq i < S$,

$$RB_i = \frac{Max(W_i^k)}{Avg(W_i^k)} = \frac{Max(W_i^k)}{RA_i}, \quad 0 \leq k < K.$$

Similarly,

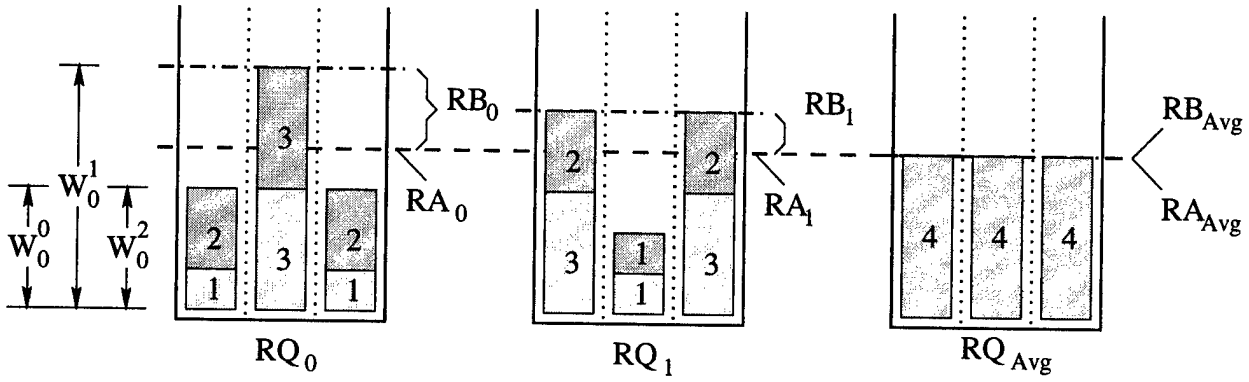
$$RB = \frac{Max(W_{Avg}^k)}{Avg(W_{Avg}^k)} = \frac{Max(W_{Avg}^k)}{RA_{Avg}}, \quad 0 \leq k < K.$$

Heuristically, the RB index of a server measures how balanced the job mix is with respect to their different resource requirements. If the total resource requirements are all the same, then the local RB measure is unity, since $Max(W_i^k) = Avg(W_i^k)$. This corresponds to the case where the workload is *matched* to the server. The global RB is a measure of how well the total work in the system matches the capabilities of all the servers in the system. The goal of the load balancing algorithm is to move each server towards this global balanced resource level. In Figure 2(a), $RB_0 = 6/4$ or 1.5, while $RB_1 = 5/4$ or 1.25. Since $RB = 4/4$ or 1.0, the two servers recognize the existence of a resource imbalanced state.

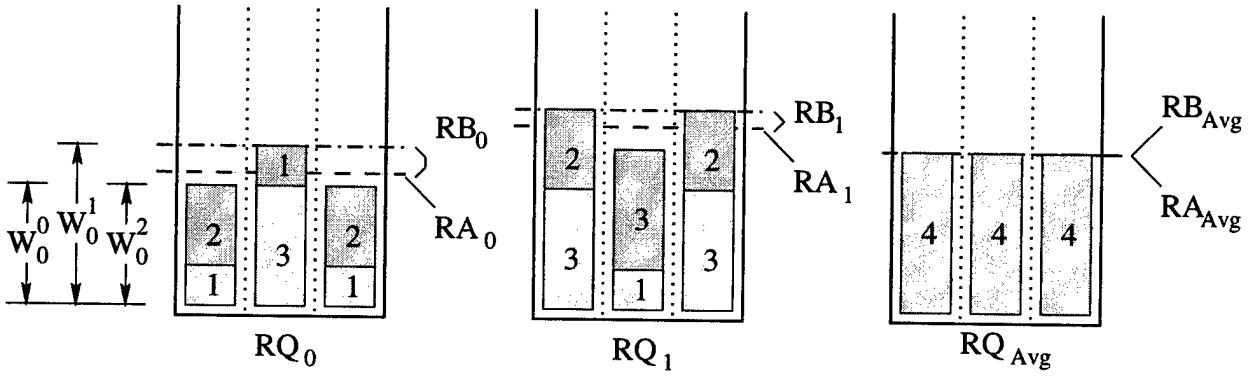
Once a resource imbalance is detected, the load balancing policies must actively correct the imbalance. Figure 2(b) shows the result of using the LSP policy to adjust the resource imbalance. Server S_0 sends its latest job to S_1, while S_1 sends its latest job to S_0. Note that the resource balance index improves on both servers, with $RB_0 = 4/3.33$ or 1.2, while $RB_1 = 5/4.66$ or 1.07. However, the resource balance could have been improved even further, as shown in Figure 2(c), by transferring the jobs which best balance the workload at both servers. We refer to this heuristic policy as the balanced job selection policy or BSP.

3.2. Resource Balancing Algorithms

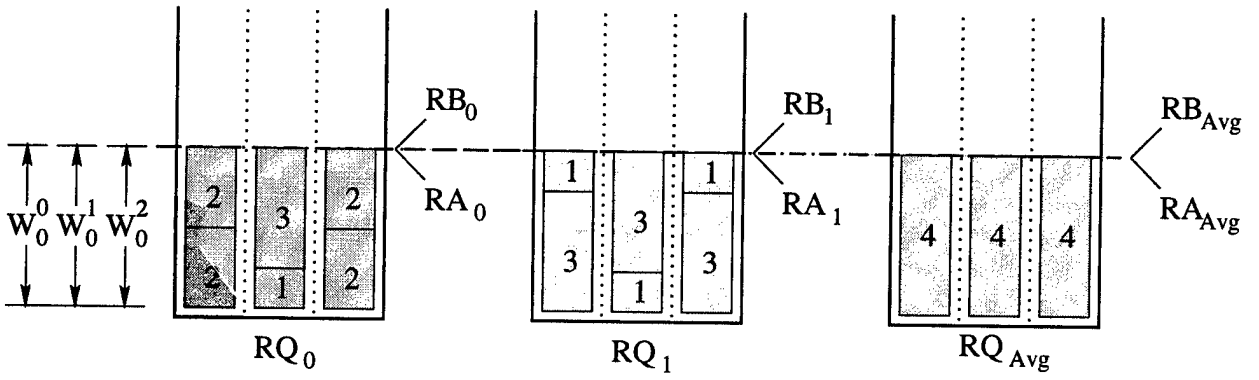
In the following discussion, we extend the baseline load balancing algorithm with the heuristic RB load index and the BSP job selection policy. In general, the goal of these extensions is to move the system to a state where the load is balanced across the servers *and* the job mix at each server matches the resource capabilities provided by that server. These extensions are described below.



(a) Comparison of RA and RB Load Index Measures



(b) Result of Latest Job Selection Policy (LSP)



(c) Result of Balance Job Selection Policy (BSP)

Figure 2. Limitations of RA and LSP

Sender Initiated, Balanced Selection Policy: SI_BSP. The baseline sender initiated algorithm, SI, is extended to SI_BSP by modifying the selection policy as follows. The fact that the load balancing action was triggered by the condition that the load index, RA, of a given server was above the global average implies that it has more work than at least one other server. Thus, this heavily loaded server needs to transfer work to another server. The BSP policy selects the job for transfer (out) which results in the best resource balance of the local queue. Note that transferring a job may actually *worsen* the resource imbalance, but we proceed nonetheless so that the overall excess workload can be reduced. Also, the resource balance at the receiving server may worsen as well. However, the receiving server currently has a workload shortage, so it may be executing less efficiently anyway.

Sender Initiated, RB Index, Balanced Selection Policy: SI_RB_BSP. The SI_RB_BSP algorithm extends the SI_BSP algorithm by including the RB load index, and modifying the transfer and selection policies as follows. First, the transfer policy triggers a load balancing action based on RA *or* RB. If the action is based on RA, SI_RB_BSP proceeds as SI_BSP. However, if the action is based only on RB, the selection policy is further modified over that used for SI_BSP. The job which *positively* improves the resource balance of the local queue the most is selected for transfer (out). If no such job is found, no action occurs.

Receiver Initiated, Balanced Selection Policy: RI_BSP. The baseline receiver initiated algorithm, RI, is extended to RI_BSP in a fashion complementary to SI_BSP.

Receiver Initiated, RB Index, Balanced Selection Policy: RI_RB_BSP. The RI_RB_BSP algorithm extends the RI_BSP algorithm in a fashion complementary to SI_RB_BSP.

Symmetrically Initiated, Balanced Selection Policy: SY_BSP. The baseline symmetrically initiated algorithm, SY, is extended to SY_BSP as follows. If the transfer policy triggers a send action, SY_BSP proceeds as SI_BSP. Alternatively, if the transfer policy triggers a receive action, SY_BSP proceeds as RI_BSP.

Symmetrically Initiated, RB Index, Balanced Selection Policy: SY_RB_BSP. The SY_RB_BSP algorithm extends the SY_BSP algorithm as follows. If the action is based on RA, SY_RB_BSP proceeds as SY_BSP. However, if the action is based only on RB, then SY_RB_BSP performs both send *and* receive actions using methods identical to SI_RB_BSP and RI_RB_BSP. Heuristically, this maintains the RA index but improves the RB index.

4. Experimental Results

The baseline and extended load balancing algorithms were implemented on a simulated system that is described in Section 4.1. The experimental results are summarized in Section 4.2.

4.1. System Model

The simulation system follows the general form of Figure 1. The server model, workload model, and performance metrics are discussed below.

Server Model. A system with 16 servers was used for the current set of experiments. A server model is specified by the amount of each of the K resource types it contains and the choice of the local scheduler. For all simulations, the local scheduler uses a backfill algorithm with a resource balancing job selection criteria [10]. To our knowledge, this is the best performing local scheduling algorithm for the multi-resource single server problem. At this point, we assume that the jobs are *rigid*, meaning that they must receive the required resources before they can execute. We also assume that the execution time of a job is the same on any server. Simulation results are reported for a value of $K = 8$.

Two independent parameters were used to specify the degree of heterogeneity across the servers in the simulated system. First, within a single server, the *server resource correlation*, S_{rc}, parameter specifies how the resources of a given server are balanced. This represents the *intra-server* resource heterogeneity measure. For example, assume each server has two resources, CPUs and memory. If a correlation value of about one were specified, then a server with a large memory would also have a large number of CPUs. Conversely, if a correlation value of about negative one were used, then a server with a large memory would probably have a low number of CPUs. Finally, a correlation value near zero implies that the resource sizes within a given server are unrelated. The value of the resource correlation ranged from 0.15 to 0.85 in the simulations (our simulator is capable of generating S_{rc} values in the range $-1.0/(K-1) < S_{rc} \leq 1.0$).

The second parameter is the *server resource variance*, S_{rv}, which is used to describe range of sizes for a single resource which may be found across all of the servers. This represents the *inter-server* heterogeneity measure. Again, a resource variance about one implies that the number of CPUs found in server S_i will be approximately the same as the number of CPUs found in server S_j for $0 \leq i, j < S$. In general, a resource variance of $S_{rv} = V$ implies that

the server S_i with the largest amount of a resource k has V times as much of that resource as the server S_j which has the smallest amount of that resource. All other servers have some amount of resource k between S_i^k and S_j^k. The value of the resource variance ranged from 1.2 to 8.0 for our experiments.

Workload Model. The two main aspects of the simulated workload are the generation of multi-resource jobs and the job arrival rate. Recent studies on workload models have focused primarily on a single resource — the number of CPUs that a job requires. Two general results from these studies show that the distribution of CPU requirements is generally hyperexponential, but with strong discrete components at powers of two and squares of integers [3]. An additional study investigated the distribution of memory requirements on the 1024 processor CM-5 at Los Alamos National Laboratory. The conclusion was that memory requirements are also hyperexponentially distributed with strong discrete components. Additionally, there was a weak correlation between the CPU and memory requirements for the job stream studied [4].

We generalize these results to a K-resource workload as follows. The multiple resource requirements for a job in the job stream are described by two parameters. The kth resource requirement for job j, J_j^k, is drawn from a hyperexponential distribution with mean $\overline{X}_k$. Additionally, the correlation between resource requirements within a single job, J_{rc} is also specified. A single set of workload parameters was used for all of the initial simulations reported here, in which $\overline{X}_k = 0.15, 0 \leq k < K$, and the resource correlation $J_{rc} = 0.25$. Essentially, the average job requires 15% of each resource in an average server, and its relative resource requirements are near random.

Figure 3(a) shows the single resource probability distribution used for the workload. Note that the probability for small resource requirements is reduced over a strictly exponential distribution. We justify this modification by noting that small jobs are probably not good candidates for load balancing activity as they do not impact the local job scheduler efficiency significantly (except to improve it). Figure 3(b) shows the joint probability distribution for a dual resource ($K = 2$) system. In general, the joint probability distribution shown in Figure 3(b) is nearly identical for all pairs $(i, j), 0 \leq i, j < K$, of resources in the job stream. This workload model has also been used to study multi-resource scheduling on a single server [10].

The job arrival rate generally affects the total load on the system. A high arrival rate results in a large number of jobs being queued at each server, while a low arrival rate reduces the number of queued jobs. For our initial simulations, we selected an arrival rate that resulted in an average of 32 jobs per server in the system. As each job arrives, it is sent to a server selected randomly from a uniform distribution ranging from 0 to $S - 1$. A final assumption is that the nature of the workload model impacts only the absolute values of the system performance, not the relative performance of the algorithms under study.

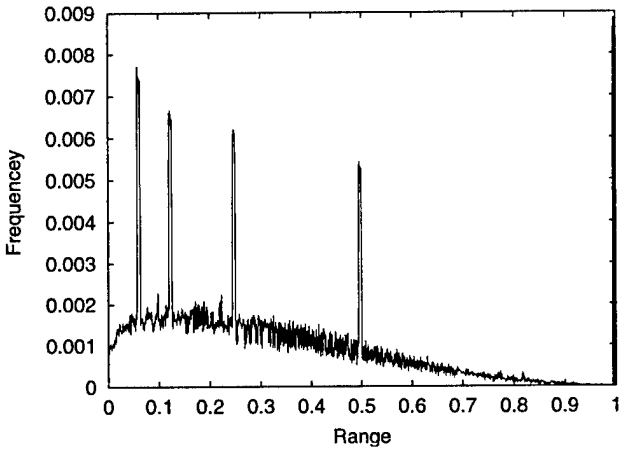
Performance Metrics. A single performance metric, *throughput*, is our current method for evaluating these algorithms. Throughput is measured as the total elapsed time from when the first job arrives to when the last job departs.

4.2. Simulation Results

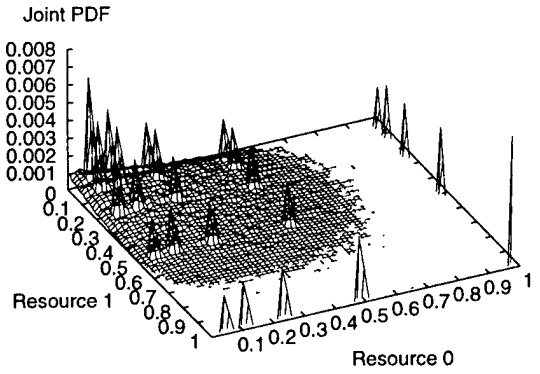
Our initial simulation results are depicted in Figures 4(a)–(f). Recall that load balancing algorithms essentially try to mimic a central work queue from which any server can select jobs as its resources become available. Therefore, the performance results for the load balancing algorithms are normalized against the results of a system with a central work queue. For each graph in the figure, the x axis represents the server resource variance parameter, S_{rv}, as described previously, while the y axis represents the throughput of the algorithms, normalized to the throughput of the central queue algorithm. The following paragraphs summarize these results.

Impact of the Resource Balancing Policies. Figures 4(a)–(c) depict the performance of the sender initiated, receiver initiated, and symmetrically initiated baseline and extended algorithms, normalized to the performance of the central queue algorithm. For these experiments, $K = 8$ and $S_{rc} = 0.50$ (resources within a server are very weakly correlated). In comparing the performance of the baseline and the extended algorithms, we see that the extended variants consistently out-perform the baseline algorithm from which they were derived. The addition of the intelligent job selection policy, BSP, provides a 5–10% gain in the SI_BSP, RI_BSP, and SY_BSP algorithms over the SI, RI, and SY algorithms, respectively. Moreover, the addition of the RB load index and associated transfer policy further increases these gains for SI_RB_BSP, RI_RB_BSP, and SY_RB_BSP.

Effects of Server Resource Correlation, S_{rc}. The jobs which arrive at each server may or may not have a natural affinity for that server. For example, if a server has a large memory and a few CPUs, a job which is memory intensive has a high affinity for that server. However, a job which is CPU intensive has a low affinity to that server. For a job stream with a fixed intra-job resource correlation, J_{rc}, the probability that an arriving job has good affinity to a server increases as S_{rc} increases. A larger natural affinity increases the packing efficiency of the local schedulers, improving the throughput. Figures 4(d)–(f) compare



(a) Single Resource Probability Distribution



(b) Dual Resource Joint Probability Distribution, Correlation=0.25

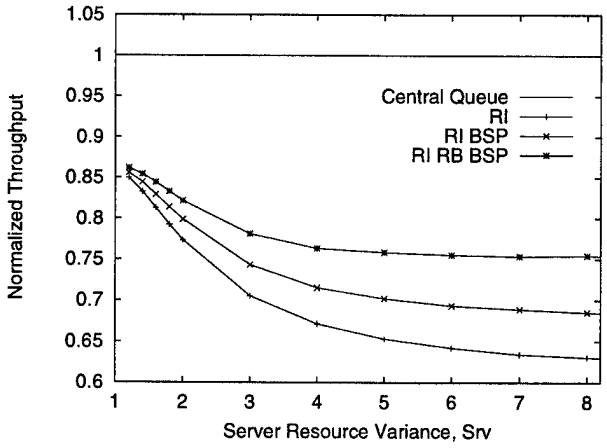
Figure 3. Multi-Resource Workload Model

the performance of the RI_RB_BSP, SI_RB_BSP, and the SY_RB_BSP algorithms, over the range of server resource correlation values, $S_{rc} = \{0.15, 0.50, 0.70\}$. Generally, as the value of S_{rc} increases, the performance of the load balancing algorithms also improve, due to an increased probability of natural affinity.

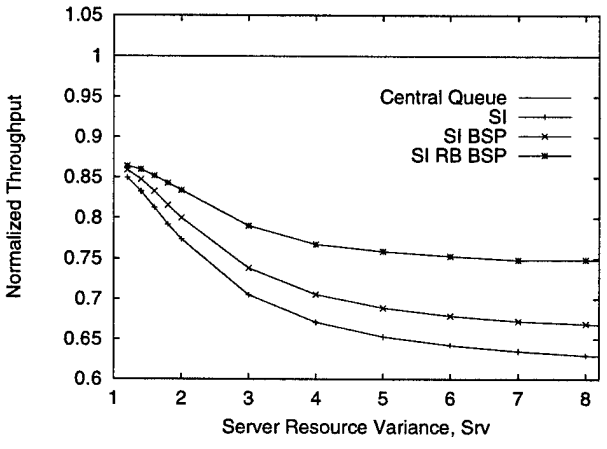
The SI_RB_BSP algorithm performs slightly better than RI_RB_BSP at low values of S_{rc} as seen in Figure 4(d). However, RI_RB_BSP begins to outperform SI_RB_BSP at higher values of S_{rc}, as seen in Figures 4(e) and 4(f). At low values of S_{rc}, the SI variant can actively transfer out jobs with low affinity, which occur with high probability, while the RI variant can only try to correct the affinity of their total workload. Higher values of S_{rv} magnify this problem. Therefore, the performance advantage goes to the SI variant. For higher values of S_{rc}, the probability of good job-server affinity is also higher. When accompanied by higher S_{rv}, the system may be thought of as having some larger servers and some smaller servers, with good job affinity to any server. In this case, the throughput of the system is driven by the efficiency of the larger servers. In the SI variant, the smaller servers will tend to initiate load balancing actions, by sending work to the larger servers. So while the smaller servers may execute efficiently, the larger servers may not. However, in the RI variant, the larger servers will tend to initiate load balancing, and intelligently select which work to receive from the smaller servers. Now, the larger servers will tend to execute more efficiently. For this reason, the performance advantage goes to the RI variant.

Impact of Server Resource Variation, S_{rv}. As the resource variation, S_{rv}, increases in the graphs of Figure 4, the throughput of the load balancing algorithms drops relative to the central queue algorithm. This is due to the fact that the average job size (size of the jobs resource requirements) is not taken into account when selecting jobs for transfer. At higher server resource variances, some servers have a very small amount of one or more resources. However, the average job size ending up on the servers with small resource capacities is the same as those ending up on the larger servers. The small size of the resources in these servers, relative to the average resource requirement of the arriving jobs, causes packing inefficiencies by the local scheduler, due to job size *granularity*. In the case of a centralized queue, the servers with small resource capacities are more likely to find jobs with smaller resource requirements. In short, simply balancing the workload resource characteristics is not sufficient. Other workload characteristics must also be emulated in the local queues, such as the average job requirements relative to the server resource capacities. This is a topic in our current work in progress and is briefly discussed in Section 5.

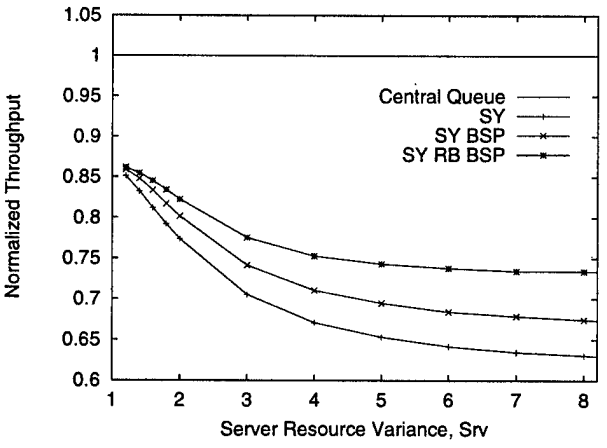
Central Queue vs. Load Balancing. A final observation may be drawn from the graphs in Figure 4. Even when the servers are all similarly configured (e.g. $S_{rv} \sim 1$ and $S_{rc} \sim 1$), there is a consistent performance gap of 15% for all baseline and extended load balancing algorithms with respect to the central queue algorithm. This is due to the fact that even if the load balancing algorithms are successful in balancing the load, the local scheduler at each server may not be able to find a job in its *local* queue to fill idle resources, even if such a job exists in the queue of a different server. Closing this gap is the subject of our current work and is briefly discussed in Section 5.



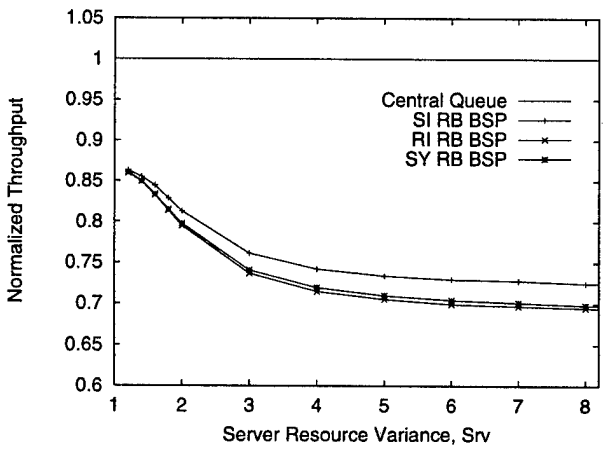
(a) Receiver Initiated Variants



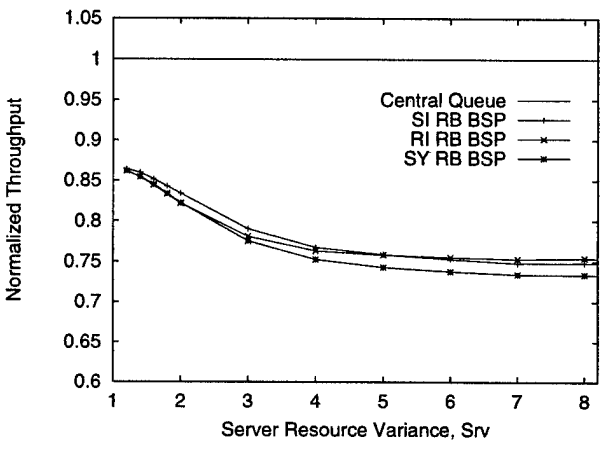
(b) Sender Initiated Variants



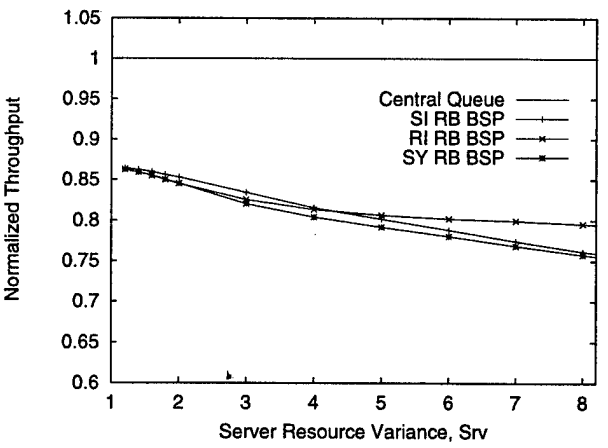
(c) Symmetrically Initiated Variants



(d) Server Resource Correlation: Src=0.15



(e) Server Resource Correlation: Src=0.50



(f) Server Resource Correlation: Src=0.70

Figure 4. Baseline and Extended Algorithm Performance Comparison

5. Summary and Work in Progress

In this paper, we defined a workload distribution problem for a computational grid with near-homogeneous multi-resource servers. First, servers in the grid have multiple resource capacities, and jobs submitted to the grid have requirements for each of those resources. Additionally, the servers are homogeneous in that any job submitted to the grid can be executed by any of the servers, but heterogeneous in their various resource configurations. We next investigated a load balancing approach to workload distribution for this grid. We showed how previous baseline load balancing policies for single resource systems failed to maintain a workload at each server which had a good affinity towards that server. We then proposed two orthogonal extensions based on the concept of resource balancing. The basic idea of resource balancing is that the local scheduler is more effective in utilizing the resources of the local server, if the total relative resource requirements of all jobs in a local work queue match the relative capacities of the server. Our simulation results show that our policy extensions provided a consistent 5–15% increase in system throughput performance over the baseline load balancing algorithms.

However, there is still significant room for improvement in the workload distribution approach. First, as the resource variance between servers grows, additional workload characteristics, beyond the total resource balance, must be taken into account when evaluating the workload for a given server. Specifically, we noted that the granularity of jobs in a local queue impacts the performance of the smaller servers. Intuitively, small jobs should be sent to small servers, and large jobs should be sent to large servers. Here, a large job is one that generally has large resource requirements, and a large server is one that generally has large resource capacities. Note that the size of a job is relative to the size of the server to which it is being compared. Our current work in progress is investigating refinements to the load balancing policies which improve the affinity of the local workload to the local server. Note that these investigations apply to single resource servers as well.

Second, there is a persistent performance gap between a central queue approach to workload distribution and our load balancing algorithms. Our conjecture is that even if the load is perfectly balanced, restricting a server, S_i, to execute jobs only from its local queue will increase the percentage of time that some of S_i's resources remain idle, when there may be a job in the queue of a different server, S_j, which would fit in to the idle resources of server S_i. These effects were noted in our simulations in that even when the servers were all nearly identical, and an equal load was being delivered to each server, the system throughput was still significantly below the performance of the central queue algorithm. Load balancing schemes were limited to about 85% of the throughput of the central queue scheme at all tested values of S_{rv} and S_{rc}, as seen in Figures 4(a)–(f).

We are further motivated to look at a more centralized approach by real-world computational grids, such as NASA's Information Power Grid (IPG) [6]. The current implementation of the IPG uses services from the Globus toolkit to submit jobs, query their status, and query the state of the grid resources. Globus uses a centralized directory structure, the Metacomputing Directory Service (MDS) to store information about the status of the metacomputing environment and all jobs submitted to the grid. Information in the MDS is used to assist in the placement of new jobs onto servers with appropriate resources within the grid. While this approach is currently being used in the IPG, there are questions about the scalability of such a centralized structure. For example, can a central structure handle hundreds of sites and thousands of jobs? How about fault tolerance? Our current work in progress is therefore investigating compromises between a single central queue and completely distributed queues. The general concept is to keep work close to the servers where it will most likely execute, and move work to a specific server when needed. Recent research in dynamic matching and scheduling for heterogeneous computing systems use similar approaches, along with heuristics for matching a job to idle server resources [12]. Our work in progress attempts to combine the centralized nature of current mapping approaches with our resource-balanced workload affinity approach.

6. Author Biographies

William (Bill) Leinberger is a Ph.D. student and Research Fellow in the Department of Computer Science and Engineering at the University of Minnesota. He received a BS in Computer and Electrical Engineering from Purdue University in 1984. His thesis covers topics in scheduling in the presence of multiple resource requirements. His other research interests include resource management for computational grids, and general topics in the area of high-performance computing architectures. Bill is currently on an educational leave from General Dynamics Information Systems, Bloomington, MN, where he has held positions as a hardware engineer, systems engineer, and systems architect in the area of special-purpose processing systems.

George Karypis is an assistant professor at the department of Computer Science and Engineering at the University of Minnesota. His research interests spans the areas of parallel algorithm design, data mining, applications of parallel processing in scientific computing and optimization, sparse matrix computations, parallel preconditioners, and parallel programming languages and libraries. His recent work has been in the areas of data mining, serial and parallel

graph partitioning algorithms, parallel sparse solvers, and parallel matrix ordering algorithms. His research has resulted in the development of software libraries for serial and parallel graph partitioning (METIS and ParMETIS), hypergraph partitioning (hMEITS), and for parallel Cholesky factorization (PSPASES). He has coauthored several journal articles and conference papers on these topics and a book title "Introduction to Parallel Computing" (Publ. Benjamin Cummings/Addison Wesley, 1994). He is a member of ACM, and IEEE.

Vipin Kumar is the Director of Army High Performance Computing Research Center and Professor of Computer Science at the University of Minnesota. His current research interests include high performance computing, parallel algorithms for scientific computing problems, and data mining. His research has resulted in the development of the concept of isoefficiency metric for evaluating the scalability of parallel algorithms, as well as highly efficient parallel algorithms and software for sparse matrix factorization (PSPACES), graph partitioning (METIS, ParMETIS), VLSI circuit partitioning (hMETIS), and dense hierarchical solvers. He has authored over 100 research articles, and coedited or coauthored 5 books including the widely used text book "Introduction to Parallel Computing" (Publ. Benjamin Cummings/Addison Wesley, 1994). Kumar has given numerous invited talks at various conferences, workshops, national laboratories, and has served as chair/co-chair for many conferences/workshops in the area of parallel computing and high performance data mining. Kumar serves on the editorial boards of IEEE Concurrency, Parallel Computing, the Journal of Parallel and Distributed Computing, and served on the editorial board of IEEE Transactions of Data and Knowledge Engineering during 1993-97. He is a Fellow of IEEE, a member of SIAM, and ACM, and a Fellow of the Minnesota Supercomputer Institute.

Rupak Biswas is a Senior Research Scientist with MRJ Technology Solutions at NASA Ames Research Center. He is the Task Leader of the Algorithms, Architectures, and Applications (AAA) Group that performs research into technology for high-performance scientific computing. The AAA Group is part of the Numerical Aerospace Simulation (NAS) Division of NASA Ames. Biswas has published over 70 technical papers in major journals and international conferences in the areas of finite element methods, dynamic mesh adaptation, load balancing, and helicopter aerodynamics and acoustics. His current research interests are in dynamic load balancing for NUMA and multithreaded architectures, scheduling strategies for heterogeneous distributed resources in the IPG, mesh adaptation for mixed-element unstructured grids, resource management for mobile computing, and the scalability and latency analysis of key NASA algorithms and applications. He is a member of ACM and the IEEE Computer Society.

References

[1] D. L. Eager, E. D. Lazowska, and J. Zahorjan. Adaptive load sharing in homogeneous distributed systems. *IEEE Trans. on Software Engineering*, SE-12(5):340–353, May 1986.

[2] D. L. Eager, E. D. Lazowska, and J. Zahorjan. A comparison of receiver-initiated and sender-initiated adaptive load sharing. *Performance Evaluation*, 6:53–68, 1986.

[3] D. G. Feitelson. Packing schemes for gang scheduling. In D. Feitelson and L. Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, volume 1162, pages 65–88. Springer-Verlag, New York, 1996. LNCS.

[4] D. G. Feitelson. Memory usage in the lanl cm-5 workload. In D. Feitelson and L. Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, volume 1291, pages 78–94. Springer-Verlag, New York, 1997. LNCS.

[5] D. Ferrari and S. Zhou. An empirical investigation of load indicies for load balancing applications. In *Proc. 12th Intl. Symposium on Computer Performance Modeling, Measurement, and Evaluation*, pages 515–528. North-Holland, Amsterdam, 1987.

[6] I. Foster and C. Kesselman, editors. *The GRID: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, 1998.

[7] J. P. Jones. Implementation of the NASA Metacenter: Phase 1 report. Technical report, NASA Ames Research Center, October 1997. Technical Report NAS-97-027.

[8] L. V. Kale. Comparing the performance of two dynamic load distribution methods. In *Proc. Intl. Conference on Parallel Processing*, pages 77–80, August 1988.

[9] V. Kumar, A. Gramma, and V. Rao. Scalable load balancing techniques for parallel computers. *Journal of Parallel and Distributed Computing*, 22(1):60–79, July 1994.

[10] W. Leinberger, G. Karypis, and V. Kumar. Job scheduling in the presence of multiple resource requirements. In *Supercomputing '99*, November 1999.

[11] M. Livny and M. Melman. Load balancing in homogeneous broadcast distributed systems. In *Proc. ACM Computer Network Performance Symposium*, pages 47–55, April 1982.

[12] M. Maheswaran, S. Ali, H. J. Siegel, D. Hensgen, and R. F. Freund. Dynamic matching and scheduling of a class of independent tasks onto heterogeneous computing systems. In *8th IEEE Heterogeneous Computing Workshop (HCW'99)*, April 1999.

[13] C. McCann and J. Zahorjan. Scheduling memory constrained jobs on distributed memory computers. In *Proc. ACM SIGMETRICS Joint Intl. Conference on Measurement and Modeling of Computer Systems*, pages 208–219, 1996.

[14] E. W. Parsons and K. C. Sevcik. Coordinated allocation of memory and processors in multiprocessors. Technical report, Computer Systems Research Institute, University of Toronto, October 1995.

[15] N. G. Shivaratri, P. Krueger, and M. Singhal. Load distributing for locally distributed systems. *IEEE Computer*, 25(12):33–44, December 1992.

[16] M. Y. Wu. Symmetrical hopping: A scalable scheduling algorithm for irregular problems. *Concurrency: Practice and Experience*, 7(7):689–706, October 1995.

SESSION 2-A
COMMUNICATION AND DATA MANAGEMENT

Chair: D. Panda, *Ohio State University, USA*

Evaluation of Expanded Heuristics in a Heterogeneous Distributed Data Staging Network

Mitchell D. Theys
University of Illinois at Chicago
Elec. Engr. and Comp. Sci. Dept. (MC 154)
851 S. Morgan St. RM 1120
Chicago, IL 60607-7053, USA
mtheys@eecs.uic.edu

Noah B. Beck and Howard Jay Siegel
School of Elec. and Comp. Engr.
Purdue University
West Lafayette, IN 47907-1285 USA
{noah, hj}@purdue.edu

Michael Jurczyk
Dept. of Comp. Engr. and Comp. Sci.
University of Missouri - Columbia
Columbia, MO 65211 USA
mjurczyk@cecs.missouri.edu

Abstract

Providing up-to-date input to users' applications is an important data management problem for a heterogeneous distributed computing environment, where each data storage location and intermediate node may have different data available, storage limitations, and communication links available. Sites in the heterogeneous network request data items and each request has an associated deadline and priority. In a military situation, the data staging problem involves positioning data for facilitating a faster access time when it is needed by programs that will aid in decision making. This work concentrates on solving a basic version of the data staging problem in which all parameter values for the communication system and the data request information represent the best known information collected so far and stay fixed throughout the scheduling process. The heterogeneous network is assumed to be oversubscribed and not all requests for data items can be satisfied. Three multiple-source shortest-path algorithm based procedures for finding a near-optimal schedule of the communication steps for staging the data are described. Each procedure can be used with each of three cost criteria developed here (based on results from earlier experiments). A subset of the possible procedure/cost criterion combinations are evaluated in simulation studies considering different priority weighting schemes, different average number of links used to satisfy each data request, and different network loadings. The proposed heuristics are shown to perform well with respect to upper and lower bounds.

1. Introduction

The DARPA Battlefield Awareness and Data Dissemination (BADD) [15] and the Agile Information Control Environment (AICE) [2] programs include designing an information system for forwarding (staging) data to proxy servers prior to their usage as inputs to a local application in a heterogeneous distributed computing environment, using satellite and other communication links. The focus is on providing the ability to operate in a distributed server-server-client environment to optimize information currency for many critical classes of information.

Data staging is an important data management problem that needs to be addressed by the BADD and AICE programs. A simplified informal description of an example of a data staging problem in a military application is as follows. A warfighter is in a remote location with a portable computer and needs data as input for a program that plans troop movements. The data can include detailed terrain maps, enemy locations, troop movements, and current weather predictions. The data will be available from Washington D.C., foreign military bases, and other data storage locations. Each location may have specific data available, storage limitations, and communication links. Also, each data request is associated with a specific deadline and priority. Each priority level then has a corresponding weight, so that

This research was supported by the DARPA/ISO BADD and ONR under ONR grant number N00014-970100804, and by the DARPA/ITO AICE program under contract numbers DABT63-99-C-0010 and 0012. Some of the equipment used was donated by Microsoft and Intel. In addition, M. Theys was funded in part by an AFCEA Fellowship.

0-7695-0556-2/00 $10.00 © 2000 IEEE

two levels can be compared analytically. Depending on the particular environment, there may be hundreds of warfighters, all making multiple requests. It is assumed that not all requests can be satisfied by their deadline. In a military situation, the data staging problem involves positioning data for facilitating a faster access time when it is needed by programs that will aid in decision making.

Positioning the data before it is needed can be complicated by: the dynamic nature of data requests and network congestion; the limited storage space at certain sites; the limited bandwidth of links; the changing availability of links and data; the time constraints of the needed data; the priority of the needed data; and the determination of where to stage the data [16]. Also, the associated garbage collection problem (i.e., determining which data will be deleted or reverse deployed to rear-sites from the forward-deployed units) arises when existing storage limitations become critical [15, 16]. The multiple copies provide an increased level of fault tolerance, in cases of links or storage locations going off-line, and allow the scheduler to select from among different sources to satisfy a data request [18].

The simplified data staging problem addressed here requires a schedule for transmitting data between pairs of nodes in the corresponding communication system for satisfying as large a sum of weighted priorities as possible. Each node in the system can be: (a) a source machine of initial data items; (b) an intermediate machine for storing data temporarily; and/or (c) a final destination machine that requests a specific data item.

It is also assumed in this simplified model of the data staging problem that all parameter values for the communication system and the data request information (e.g., network configuration and requesting machines) represent the best known information collected so far and stay fixed throughout the scheduling process. It is assumed that not all of the requests can be satisfied by their deadlines due to storage capacity and communication constraints. The model is designed to create a schedule for movement of data from the source of the data to a "staged" location for the data. It is assumed that a user's application can easily retrieve the data from this location.

Three multiple-source shortest-path algorithm based procedures for finding a near-optimal schedule of the communication steps for staging the data are described [20]. Each procedure can be used with each of seven cost criteria developed. A subset of fourteen of the possible 21 resulting heuristics that are expected to perform well (based on experiments in [20]) are examined in simulation studies considering different priority weighting schemes, different average number of links used to satisfy each data request, and different network loadings. The rationale for considering each of these procedures and costs is provided. The proposed heuristics are shown to perform well with respect to upper and lower bounds. Furthermore, the heuristics using a complex cost criterion are shown to allow more highest priority messages to be received than a simple-cost-based heuristic that schedules all highest priority messages first. Finally, an approach considering data items with "more desirable" and "less desirable" available versions is evaluated using a variable time, variable accuracy algorithm, and simulation results are presented. This research serves as a necessary step toward solving the more realistic and complicated version of the data staging problem involving fault tolerance, dynamic changes to the network configuration, ad hoc data requests, sensor-triggered data transfers, etc.

The material in this paper extends the earlier work presented in [19] by introducing three new cost criteria and two new bounds. This work also varies additional simulation parameters, including eight network loadings, three average numbers of links used to get from a source machine to a destination machine, and five priority weighting schemes. This paper also introduces a variable time, variable accuracy approach for using data items with "more desirable" and "less desirable" versions.

Section 2 provides an overview of work that is related to the data staging problem. In Section 3, a mathematical model for a basic data staging problem is reviewed. Section 4 provides a description of Dijkstra's algorithm used to find paths of links for transferring data items within the presented network model. Section 5 presents seven cost criteria for use in conjunction with different resource allocation procedures. Three multiple-source shortest-path algorithm based procedures for finding a near-optimal schedule of the communication steps for data staging are described in Section 6. These heuristics adopt the simplified view of the data staging problem described by the mathematical model. Three upper bounds and three lower bounds used to evaluate the performance of these heuristics are presented in Section 7. The set of simulation studies given in Section 8 were created after studying the results of [19]. These new simulation studies examine the effects of (1) having six priority levels with five different weighting schemes, (2) varying the average number of links required for a data item to reach a destination from its source, and (3) varying the total number of requests that must be scheduled in a given network. In Section 9, an approach considering data items with "more desirable" and "less desirable" available versions is evaluated using a variable time, variable accuracy algorithm, and simulation results are presented.

2. Related Work

To the best of the authors' knowledge, there is currently no other work presented in the open literature that addresses the data staging problem, designs a mathematical model to quantify it, or presents a heuristic for solving it. Due to space constraints, the reader is referred to [6] for a more thorough discussion of the related work. A problem that is,

at a high level, remotely similar to data staging is the facility location problem in management science and operations research (e.g., [13]). Data management problems similar to data staging for the BADD/AICE program are studied for other communication systems [1, 3, 5, 11]. Other areas that are somewhat related include modifying routing schemes [4], mapping tasks onto a suite of distributed heterogeneous machines (e.g., [8, 9, 21]), and earliest deadline first [7, 17] scheduling for real-time systems. Lastly, other research exploring heuristics for use in the BADD/AICE environment have been performed [14]. All of this research is related, but does not develop a mathematical model like the one researched here nor do they examine a network similar to BADD-like network being used in this research.

3. Mathematical Data Staging Model

3.1. Model Definition

Some of this background material is based on [20], and is included here for completeness. It has been expanded to include all of the concepts needed for the new experiments and results presented here.

Consider a network topology graph G_{nt} composed of a set of vertices that represent the set of machines M in the network and a set of directed edges that represent the set of communication links L. There are m machines in M, identified as $\{M[0], M[1], \ldots, M[m-1]\}$, and each can be a source, destination, and intermediate location for data items in the network. Source machines for data items are the machines where data items are initially located within the network; these data items may eventually be transferred by the network to destination machines, possibly stored at intermediate machines along the way. Each machine $M[i]$ (where $0 \leq i < m$) also has an associated constant unused storage capacity during the time interval $[t_j, t_{j+1})$, $Cap[i](t_j)$. Note that the times t_j and t_{j+1} may not differ by exactly one time unit.

Each Communication link in this system is represented as one or more virtual links. A virtual link corresponds to a period of constant, continuous, available bandwidth from one machine to one other machine. Bidirectional communication links are therefore represented as two virtual links—one for each direction. $Nl[i,j]$ is the number of virtual links from machine $M[i]$ to $M[j]$ (where $i \neq j$ and $0 \leq i, j < m$). The kth virtual link from machine $M[i]$ to $M[j]$ is identified as $L[i,j][k]$ (where $0 \leq k < Nl[i,j]$). The virtual link $L[i,j][k]$ also has an associated link starting time $Lst[i,j][k]$, denoting the time when it becomes available, as well as a link ending time $Let[i,j][k]$, which specifies the time when the link is no longer available.

Data items are blocks of information that can be transmitted from one machine to another. The set of data items with unique names or identifiers that are available on the machines in M is called Δ. Names or identifiers assigned to data items must be different if the contents of the data items are different in any way, including details such as differing timestamps on weather maps of the same region. The number of distinctive data items in Δ is n, and individual unique data items are identified as $\{\delta[0], \delta[1], \ldots, \delta[n-1]\}$. For a data item $\delta[l]$ (where $0 \leq l < n$), the size of the data item is represented as $|\delta[l]|$. The time duration required to transfer data item $\delta[l]$ from machine $M[i]$ to machine $M[j]$ (where $i \neq j$ and $0 \leq i, j < m$) via the virtual link $L[i,j][k]$ (where $0 \leq k < Nl[i,j]$) during the time interval $[Lst[i,j][k], Let[i,j][k]]$ is $D[i,j][k](|\delta[l]|)$. Machine $M[i]$ may be a source of $\delta[l]$, or an intermediate storage location or destination that already holds a copy of $\delta[l]$. Machine $M[j]$ may be an intermediate storage location or a destination.

Let $N\delta[l]$ (where $0 \leq l < n$) represent the number of source machines holding a copy of $\delta[l]$, and $M[Source[l,j]]$ represent the jth source machine for data item $\delta[l]$ (where $0 \leq j < N\delta[l]$ and $0 \leq Source[l,j] < m$). The starting time $\delta st[l,j]$ refers to the time data item $\delta[l]$ becomes available at its jth source machine. The removal time $\delta rt[l,i]$ (where $0 \leq i < m$) refers to the time data item $\delta[l]$ can be removed from machine $M[i]$, if a copy of $\delta[l]$ is being stored at $M[i]$. This allows the value of $Cap[i](\delta rt[l,i])$ to be increased by $|\delta[l]|$. Intermediate machines, for example, could set $\delta rt[l,i]$ to be some small time period γ after the last deadline at any machine for data item $\delta[l]$. This would allow the storage space to be reclaimed at intermediate machines after the usefulness of the data item has expired. The scheduling heuristics do not remove a data item from any of its sources or destinations because this is considered outside the scope of responsibility of the scheduler.

Consider now a data item such as an image showing a map of a planned battle area. It may be possible to have available a higher quality version of the image that shows a higher level of detail, as well as a lower quality version showing less detail. An application requesting this data item would prefer to receive the higher quality image, but it may be that there are not enough resources (e.g., network bandwidth) available to fulfill this data request. In this event, however, there may be enough resources available to send the lower quality image, which would be better than sending nothing at all.

The set Rq (where $Rq \subseteq \Delta$) contains unique data items requested by destination machines in M. The number of unique data items in Rq is 2ρ. The higher quality data items are identified as $\{Rq[0], Rq[1], \ldots, Rq[\rho-1]\}$, and the lower quality data items are identified as $\{Rq[\rho], Rq[\rho+1], \ldots, Rq[2\rho-1]\}$. Here, each requested higher quality data item $Rq[i]$ (where $0 \leq i < \rho$) has a corresponding lower quality data item $Rq[i+\rho]$ that may be sent in place of $Rq[i]$ if system resources become limited. Note that for

every i there must exist exactly one j and exactly one k such that $Rq[i] = \delta[j]$ and $Rq[i+\rho] = \delta[k]$. These data items $\delta[j]$ and $\delta[k]$ are assumed for simplicity to be present at the same source machines, and to have the same associated starting times and removal times. This model also assumes for simplicity that $|Rq[i+\rho]| = \frac{1}{2}|Rq[i]|$.

The number of destination machines that request $Rq[i]$ (where $0 \leq i < \rho$) is denoted with $Nrq[i]$. If $0 \leq k < Nrq[i]$, then $M[Request[i,k]]$ refers to the kth machine that requested $Rq[i]$ (where $0 \leq Request[i,k] < m$). Each of these machines also implicitly requests $Rq[i+\rho]$ in the event that $Rq[i]$ cannot be sent, so that $Nrq[i+\rho] = Nrq[i]$, and $Request[i+\rho,k] = Request[i,k]$ for all values of k. The finishing time $Rft[i,k]$ (and equivalent $Rft[i+\rho,k]$) refers to a deadline time, after which data item $Rq[i]$ (and $Rq[i+\rho]$) is no longer useful to machine $M[Request[i,k]]$. The requesting machine $M[Request[i,k]]$ also associates the data item $Rq[i]$ with a numbered priority class $Priority[i,k]$ (equal to $Priority[i+\rho,k]$). The highest, or most important priority class is P, and the lowest, or least important priority class is 0, so that $0 \leq Priority[i,k] \leq P$. In actual systems, the deadline and priority for a data request would be set by some combination of the user, application, system administrator, and commander.

Define a schedule as a series of communication steps, among the machines of M using the communication links in L, that transfer some or all of the data items in the set Rq from their respective source machines to some or all of their respective destination machines, possibly being stored at intermediate machines along the way. Suppose that there are σ possible distinct schedules, enumerated $\{S_0, S_1, \ldots, S_{\sigma-1}\}$. The kth (where $0 \leq k < Nrq[j]$) request for a data item $Rq[j]$ (where $0 \leq j < 2\rho$) is considered satisfiable with respect to a specific schedule S_h (where $0 \leq h < \sigma$) if and only if the data item $Rq[j]$ is available at machine $M[Request[j,k]]$ at or before the deadline time $Rft[j,k]$. The set $Srq[S_h]$ then denotes the set of two-tuples (j,k) such that the kth request for the data item $Rq[j]$ is satisfiable with respect to the schedule S_h.

There must be a way to represent the relative importance of a priority class α (where $0 \leq \alpha \leq P$) compared to another priority class β (where $0 \leq \beta \leq P$ and $\alpha \neq \beta$). The relative weight of any priority class α is denoted by $W[\alpha]$. This means that if priority class α is ten times as important as priority class β, then $W[\alpha] = 10 * W[\beta]$. In an actual system, these weights would be set by the system administrator and commander, and would be a function of the current operating situation (e.g., peace or war).

Let $Worth[j,k]$ (where $0 \leq j < 2\rho$ and $0 \leq k < Nrq[j]$) denote a percentage of value to a user of data item $Rq[j]$ sent to satisfy a request at machine $M[Request[j,k]]$. that if $Rq[i]$ for $0 \leq i < \rho$ is sent to $M[Request[i,k]]$ by its deadline, then $Worth[i,k] = 1$ (meaning 100% for the preferred data version), and $Worth[i+\rho,k] = 0$ (meaning no additional worth for the second data version). If $Rq[i]$ is not sent to $M[Request[i,k]]$ by its deadline, and $Rq[i+\rho]$ is sent to $M[Request[i+\rho,k]]$ by its deadline, then $Worth[i+\rho,k] = 0.25$ (meaning 25% for the lower quality version), and $Worth[i,k] = 0$. Now, the effect of the schedule S_h (where $0 \leq h < \sigma$) can be defined as $E[S_h] = -\left(\sum_{(j,k) \in Srq[S_h]} W[Priority[j,k]] * Worth[j,k]\right)$ (where $0 \leq j < 2\rho$ and $0 \leq k < Nrq[j]$). The global optimization criterion, and hence, the objective of all of the heuristics presented later, is to find the schedule with the minimum effect, defined as $\min_{0 \leq h < \sigma} E[S_h]$.This performance criterion is related to the one described in [12]. Another way to view this minimization is to think of it as trying to find the schedule of data transfers that produces the maximum sum of satisfied requests' priority weights.

3.2. Heuristic Solution Approach

The heuristic approach used here to create the schedule S_h with minimum effect $E[S_h]$ utilizes Dijkstra's shortest path algorithm. This algorithm, presented in Section 4, calculates arrival times for data items and establishes paths of virtual links to get data items from source machines to destination machines. The paths calculated by this algorithm give the earliest arrival time for a given data item, based on the expected system resources available when the algorithm is run, and ignores any future competition for resources among the pending data requests. provided that there are no other data items competing for resources in the network. After Dijkstra's algorithm has been run for each requested data item (i.e., all data items in Rq), a single data item and one or more destination machines are selected through the use of a cost criterion presented in Section 5. This data item choice reflects a combination of its contribution to the effect of the schedule, and the amount of time between its arrival at a destination and its deadline at that destination. Network resources and machine storage are then allocated according to one of the procedures presented in Section 6, updating link availability times and available machine storage. This updating of network information will cause the arrival times and virtual link paths for some other data items to become invalid, so the heuristic process (using a cost and an allocation procedure) is repeated again (beginning with Dijkstra's algorithm) using the modified network information. This continues until there are no more satisfiable data items in the network, thus producing the communication schedule. Results from simulation studies using this approach, which only considers one version of each data item (i.e., considers only $Rq[i]$ where $0 \leq i < \rho$, not $Rq[j]$ where $\rho \leq j < 2\rho$), are found in Section 8. A modified approach considering

both versions of a data item is contained in Section 9.

4. Dijkstra's Shortest Path Algorithm

The heuristics presented here utilize Dijkstra's algorithm [10] for finding the shortest path from one or more source nodes to all other nodes in a directed graph. The version used calculates the earliest possible available time for a data item $Rq[i]$ (where $0 \leq i < 2\rho$) at each machine in M, given a subset of machines in M that already holds a copy of $Rq[i]$.

Define the available time $A_T[i,j]$ (where $0 \leq i < 2\rho$, $0 \leq j < m$) as the earliest possible time found, by executing Dijkstra's algorithm, when data item $Rq[i]$ could be present and available at machine $M[j]$. Define also the value of the predecessor $\pi[i,j]$ to be the two-tuple (s,k) (where $-1 \leq s < m$, $-1 \leq k < Nl[s,j]$) identifying the machine $M[s]$ as the machine that sends data item $Rq[i]$ to machine $M[j]$ via virtual link $L[s,j][k]$. This predecessor is also determined by the execution of Dijkstra's algorithm. If the value of $\pi[i,j]$ is $(-1,-1)$, this means that no machine sends data item $Rq[i]$ to machine $M[j]$ via any virtual link. This may happen if machine $M[j]$ is a source machine for data item $Rq[i]$, or it may happen if it is not possible for machine $M[j]$ to receive a copy of data item $Rq[i]$ (possibly due to the unavailability of network resources). For more information about the implementation of Dijkstra's algorithm, including pseudocode and examples, the reader is referred to [6].

5. Data Item Selection Cost Criteria

5.1. Introduction

Network resources must be allocated to data requests in some order; this order intuitively should include "more important" requests and requests that are "close" to their deadlines before "less important" requests and requests that are "not close" to their deadlines. Dijkstra's algorithm is used here for each data item individually, as if it were the only request in the system remaining to be satisfied. Thus, Dijkstra's algorithm is executed for each remaining data item separately. Some quantitative cost must therefore be applied so that an algorithm can evaluate the relative merit of any given request compared to any other request. Seven different cost criteria are detailed below; each attempts to take into consideration both the importance of a data request, and how close the data request is to its deadline.

Suppose $M[r]$ (where $0 \leq r < m$) is the next machine to receive data item $Rq[i]$ (where $0 \leq i < 2\rho$) on a path from $M[s]$ (where $(s,l) = \pi[i,r]$), which can be any machine already holding a copy of $Rq[i]$, to one or more requesting destination machines. That is, machine $M[s]$ holds a copy of data item $Rq[i]$, and $M[r]$ must be the next machine to receive $Rq[i]$ so that $M[Request[i,k]]$ (for one or more values of k, where $0 \leq k < Nrq[i]$) can ultimately receive $Rq[i]$. Let the set of values of k that satisfy this condition (i.e., destination machines that request $Rq[i]$ through $M[r]$) be called $Drq[i,r]$.

Assume that $Rq[i]$ is the next data item to be allocated network resources. Let the value $Sat[i,k]$ (where $0 \leq i < 2\rho$ and $0 \leq k < Nrq[i]$) be 1 if $Request[i,k]$ would be satisfiable, and 0 if it would not be satisfiable. For the simulations of Section 8, $Sat[i,k]$ is 0 for values of i such that $\rho \leq i < 2\rho$, thus ignoring the less desirable data item versions. Now, the effective priority $Efp[i,k]$ of data item $Rq[i]$ at the kth requesting location can be defined as $Sat[i,k] * W[Priority[i,k]] * Worth[i,k]$. An urgency term, indicating how close a data item's available time is to its deadline time (in seconds) at a destination is defined as $Urgency[i,k] = -Sat[i,k] * (Rft[i,k] - A_T[i, Request[i,k]] + 1)$. A smaller urgency here indicates that it is less urgent to get $Rq[i]$ to $M[Request[i,k]]$. The "+1" in the urgency term is so that the urgency never becomes a small number close to zero.

The next value that must be defined before detailing the cost criteria is the number of virtual links used to get from a machine $M[s]$ to a destination machine $M[Request[i,k]]$, where $k \in Drq[i,r]$. Let this value be called $Nlinks[i,k]$, and note that it reflects the number of links used in the path (generated by the most recent run of Dijkstra's algorithm) from a machine holding the data item to a machine requesting the data item.

All of the following cost functions take into account the priority and urgency of a data item. For all cost criteria, a smaller value indicates a more desirable use of communication resources; therefore, resource allocation is performed by the procedures in Section 6 for the data item and destination machine(s) with minimum cost.

Six of the costs allow the weight assigned to the priority term to be varied relative to the weight assigned to the urgency term. These weighting terms are W_E for the weight of the effective priority term, and W_U for the weight of the urgency term. The relative weight of these two terms compared to each other (W_E/W_U) is called the E-U ratio.

5.2. Costs $C1, C2,$ and $C3$

Four cost criteria were developed in the previous research that combine the above effective priority and urgency terms. The best performing cost, $C4$, was the basis for the work presented in this paper. The definitions of $C1, C2,$ and $C3$ are not discussed in detail in this paper. The reader is referred to [20] for more information about these three; $C4$ will be discussed in more detail below. The mathematical definitions of these three cost criterion are included for reference. Each cost is for sending data item $Rq[i]$ (where $0 \leq i < 2\rho$) to $M[r]$ (where $0 \leq r < m$) from $M[s]$ via link $L[s,r][k]$ (where $(s,k) = \pi[i,r]$), in order to ultimately try to satisfy the jth (where $0 \leq j < Nrq[i]$) requesting

destination machine:

$$C1[i,j][s,r][k] = -W_E * Efp[i,j] - W_U * Urgency[i,j]$$

$$C2[i][s,r][k] = \left(-W_E * \sum_{j \in Drq[i,r]} Efp[i,j]\right) + \left(-W_U * \max_{j \in Drq[i,r]} Urgency[i,j]\right)$$

$$C3[i][s,r][k] = \sum_{j \in Drq[i,r]} \frac{Efp[i,j]}{Urgency[i,j]}$$

5.3. Cost $C4$

The cost $\underline{C4}$ for transferring data item $Rq[i]$ (where $0 \leq i < 2\rho$) to $M[r]$ (where $0 \leq r < m$) from $M[s]$ via link $L[s,r][k]$ (where $(s,k) = \pi[i,r]$), in order to ultimately try to satisfy the jth (where $j \in Drq[i,r]$) requesting destination machine(s):

$$C4[i][s,r][k] = -W_E * \left(\sum_{j \in Drq[i,r]} Efp[i,j]\right) - W_U \left(\sum_{j \in Drq[i,r]} Urgency[i,j]\right).$$

This cost sums the weighted priorities of all satisfiable requests for data item $Rq[i]$ on a path through machine $M[r]$ and combines that with the sum of the urgency for those same satisfiable requests.

5.4. Cost $C4links$

Based on $C4$ because of its high performance in simulation tests, cost $\underline{C4links}$ is also defined for transferring data item $Rq[i]$ (where $0 \leq i < 2\rho$) to $M[r]$ (where $0 \leq r < m$) from $M[s]$ via link $L[s,r][k]$ (where $(s,k) = \pi[i,r]$), in order to ultimately try to satisfy the jth (where $j \in Drq[i,r]$) requesting destination machine(s):

$$C4links[i][s,r][k] = -W_E * \left(\sum_{j \in Drq[i,r]} \frac{Efp[i,j]}{Nlinks[i,j]}\right) - W_U * \left(\sum_{j \in Drq[i,r]} Urgency[i,j]\right).$$

Because, for example, a data request that can be satisfied by using three virtual links is using three times as much network resources as a data request of the same size that can be satisfied by using only one virtual link, this cost divides the effective priority term for each requesting destination by the number of links used to get to that destination. If the effective priority associated with a data request is considered as a measure of worth or importance to the user, then this first term would be considered a measure of worth per link. This should allow the cost criterion to better select data items to satisfy that will make the most effective use of the network resources available.

5.5. Cost $C4size$

Based again on $C4$ because of positive simulation results, the criterion $\underline{C4size}$ is also defined for transferring data item $Rq[i]$ (where $0 \leq i < 2\rho$) to $M[r]$ (where $0 \leq r < m$) from $M[s]$ via link $L[s,r][k]$ (where $(s,k) = \pi[i,r]$), in order to ultimately try to satisfy the jth (where $j \in Drq[i,r]$) requesting destination machine(s):

$$C4size[i][s,r][k] = -W_E * \left(\sum_{j \in Drq[i,r]} \frac{Efp[i,j]}{|Rq[i]|}\right) - W_U * \left(\sum_{j \in Drq[i,r]} Urgency[i,j]\right).$$

A data request with an effective priority p representing its worth to the recipient, and a size in bytes of q, then has an effective worth per byte of $\frac{p}{q}$. Because the goal of a cost criterion is to identify data requests that will make the most effective use of network resources, the first term in $C4size$ uses this effective priority divided by data request size to find data items that will transmit the maximum amount of worth per bandwidth byte.

5.6. Cost $C4sizlnk$

Cost $\underline{C4sizlnk}$ is a combination of the ideas in $C4size$ and $C4links$, and gives a cost for transferring data item $Rq[i]$ (where $0 \leq i < 2\rho$) to $M[r]$ (where $0 \leq r < m$) from $M[s]$ via link $L[s,r][k]$ (where $(s,k) = \pi[i,r]$), in order to ultimately try to satisfy the jth (where $j \in Drq[i,r]$) requesting destination machine(s):

$$C4sizlnk[i][s,r][k] = -W_E * \left(\sum_{j \in Drq[i,r]} \frac{Efp[i,j]}{|Rq[i]| * Nlinks[i,j]}\right) - W_U * \left(\sum_{j \in Drq[i,r]} Urgency[i,j]\right).$$

By combining the size and number of virtual links used, this cost gives a more accurate calculation of the resources used by a data request. For instance, consider two data items $Rq[i_1]$ and $Rq[i_2]$ of equal priority. Consider also that $Rq[i_2]$ is twice as large as $Rq[i_1]$, and that it requires the use of three virtual links versus $Rq[i_1]$'s single virtual link. In this case, $Rq[i_2]$ is requiring six times the total network resources required by $Rq[i_1]$ in order to satisfy the same priority level of request.

6. Resource Allocation Procedures

6.1. Introduction

The three procedures below allocate varying amounts of network resources for a single data item after each run of Dijkstra's algorithm, based on a cost function from Section 5. The performance of these procedures is shown in Section 8.

The resource allocations performed by these procedures update the following information in the system after scheduling $Rq[i]$ to move, and before running Dijkstra's algorithm again: (1) the list of virtual links and their start and stop times, (2) the available memory capacity on any machines that data item $Rq[i]$ has been placed, (3) the list of machines on which $Rq[i]$ is available, and (4) the time at which $Rq[i]$ can be removed from any intermediate machines.

6.2. Partial Path Procedure

Each iteration of this procedure involves: (1) performing Dijkstra's algorithm for each data request individually; (2) for the valid next communication steps, determining the "cost" to transfer a data item to its successor in the shortest path; (3) picking the lowest cost data request and transferring that data item to the successor machine (making this machine an additional source of that data item); (4) updating system parameters to reflect resources used in (3); and (5) repeating (1) through (4) until there are no more satisfiable requests in the system. In some cases, Dijkstra's algorithm would not need to be executed each iteration for a particular data transfer, i.e., if the data transfer did not use resources needed for any future transfers. In this study, only one data item is scheduled before rerunning Dijkstra's algorithm (this applies for all three procedures). This simplified the implementation of the procedures without changing the performance of the resulting schedules. The execution time of the procedures is affected; however, minimizing this is not the main goal of the work.

This procedure will schedule the transfer for the single "most important" request that must be transferred next, based on a cost criterion. The procedure (first described in [19]) is called the partial path procedure because only one successor machine in the path is scheduled at each iteration. If a data item is partially scheduled through the system and because of other scheduled transfers the requesting destination's deadline is no longer satisfied, the scheduled transfers remain in the system (the initial transfers were scheduled because the deadline could have been satisfied). Reasons the schedule for this now unsatisfiable request is not removed include: (1) in a dynamic situation, a change in the network could allow the request to be satisfied; and (2) removing the already scheduled transfers would require restarting the scheduling for all data requests because of conflicts that might have occurred.

6.3. Full Path/One Destination Procedure

The full path/one destination procedure uses a cost criterion to select a data request at an individual destination machine for resource allocation. The data item is then sent from its current location (machine $M[s]$ in each of the cost criteria) over as many virtual links as required to reach its destination machine (machine $M[j]$ for one value of j). For costs $C4$, $C4links$, $C4size$, and $C4sizlnk$, the data item $Rq[i]$ with minimum cost is sent first to machine $M[r]$, and if no request was satisfied, the cost is applied a second time for the same data item $Rq[i]$, but setting the new $M[s]$ (data source machine) to the old $M[r]$ (the machine to which the data was just scheduled). The minimum cost is then taken over all values of r (possible next storage locations). The value of r with minimum cost determines the machine $M[r]$ that the data is sent to next. This process continues until the data item has reached one requesting destination $M[j]$.

This produces a communication schedule using fewer executions of Dijkstra's algorithm than the partial path procedure. The behavior of the partial path procedure showed that if a data item $Rq[i]$ was selected for scheduling a transfer to its next intermediate location (a "hop"), in the following iteration, the same requested data item, $Rq[i]$, would typically be selected again to schedule its next hop. The full path/one destination procedure attempts to exploit this trend by selecting a requested data item with a cost criterion and scheduling all hops required for the data item to reach its lowest cost destination before executing Dijkstra's algorithm again.

The partial path procedure may construct a partial path (of many links) that it later cannot complete (due to network or memory resources being consumed by other requested data items). However, until this is determined, the part of the path constructed may block the paths of the other requested data items, causing them to take less optimal paths or causing them to be deemed unsatisfiable. The full path/one destination procedure avoids this problem. An advantage the partial path approach does have over the full path/one destination approach is that it allows the link-by-link assignment of each virtual link and each machine's memory capacity to be made based on the relative values of the cost criteria for the data items that may want the resource.

6.4. Full Path/All Destinations Procedure

The full path/all destinations procedure resembles the full path/one destination procedure but allocates more network resources after each run of Dijkstra's algorithm. This procedure satisfies all requests that would benefit from sending data item $Rq[i]$ from machine $M[s]$ to $M[r]$ (i.e., those in the set Drq[i,r]. Cost C1 is not used in conjunction with this procedure because it examines the cost of only one destination at a time. This approach was considered because it was expected to generate results comparable to the full path/one destination procedure, but with a smaller procedure execution time.

7. Upper and Lower Bounds

7.1. Introduction

Finding optimal solutions to data staging tasks with realistic parameter values are intractable problems. Therefore, it is currently impractical to directly compare the quality of the solutions found by the proposed heuristics with those found by exhaustive searches in which optimal answers can be obtained by enumerating all the possible schedules of communication steps. Also, to the best of the author's knowledge, there is no other work presented in the open literature that addresses the data staging problem and presents a heuristic for solving it (based on a similar underlying model). Thus, there is no other heuristic for solving the same problem with which to make a direct comparison of

the heuristics presented in this document. To aid in the evaluation of these heuristics, a lower bound and an upper bound on the performance of the heuristics are provided.

7.2. Full Path Random Dijkstra

The lower bound called the full path random Dijkstra method does take into account which data requests are satisfiable when it allocates resources, allowing it to improve over the random Dijkstra method used in [20]. It allocates enough resources in one scheduling step to take a data item from its current location all the way to one random satisfiable requesting destination before running Dijkstra's algorithm again. This method, is based on the full path/one destination procedure except that the next chosen transfer is randomly selected, instead of using a cost function. This bound differs from the single Dijkstra random method of [20] in that (1) this method checks that a requesting destination is satisfiable before allocating any resources toward fulfilling it, and (2) Dijkstra's algorithm is run with updated communication system information after each scheduling step.

7.3. Possible Satisfy Bandwidth

The possible satisfy bandwidth bound is a tighter bound than the possible satisfy bound of [20]. It considers satisfiable requests, and also the total amount of bandwidth available in the system, $NetBandwidth$. This value is calculated by adding together the number of bytes that could be transmitted over each virtual link in the system during the entire time interval being simultated. Consider the set of requests that would be satisfiable if each was the only request in the system. Then the one that has the largest ratio of priority weight to data item size is selected. Selecting the request that satisfies this condition guarantees that if a single link is used to satisfy this request, it will give the highest possible priority weight value per byte of network bandwidth used as compared to all requests remaining in the system. Each time a request is found, its size in bytes is added to the bandwidth used in the system (this assumes that only one virtual link is needed to satisfy this request) and its weighted priority is added to the weights of the other data items that have been selected. That particular request is then removed so that a new request can be found. This continues until the sum of bandwidths for the accepted requests exceeds $NetBandwidth$. This upper bound is unrealistic, however, because it does not take into account that more than one link may have to be used to satisfy a request, nor does it consider the time intervals that links are available, nor does it consider what machines have network bandwidth available between them.

8. Extended Simulation Study

8.1. Introduction

After the simulation study of [19] was completed, a new study was designed to examine the effects of varying some other parameters within the system. In particular, this new study introduces three new cost criteria and two new bounds, and it varies additional simulation parameters, including eight network loadings, three average numbers of links used to get from a source machine to a destination machine, and five priority weighting schemes.

The results of [20] indicated that $C4$ was the best-performing cost criterion. This led to the development of cost criteria $C4size$, $C4links$, and $C4sizlnk$, described in Section 5, for the new study. Because of the previous good performance of the full path/one destination procedure, it was implemented for the new study with all seven cost criteria described in Section 5. For comparison, the other two procedures in Section 6 (partial path and full path/all destinations) were also implemented for the new study with cost $C4$, for a total of nine heuristics. $C1$, $C2$, $C3$, $C4$,

In the previous study, all requests averaged traversing approximately 1.5 communication links (a communication link traversal count) from an initial source machine to a requesting destination machine. It was decided that the requests would be generated in a manner allowing this parameter to be controlled and varied with three different values in the new study. Another parameter concerning the data requests was the number of requests being made versus the number of requests that the network could possibly fulfill. Eight different "network loads" were decided upon for the new simulation study, as opposed to only one in the earlier [20] study. In combination with the three communication link traversal counts, there was a total of 24 different data request scenarios.

For this study, it was decided that a six-level priority scheme would be used in place of the three-level method used in the previous study. This was intended to better reflect the priority classes present in a military environment. Level 0 was generated with a 50% probability, level 1 with 25%, class 2 with 12%, level 3 with 7%, level 4 with 4%, and level 5 is generated with a 2% probability. These percentages were selected to reflect the fact that in a BADD/AICE-like environment, there would likely only be a small number of data requests in the highest priority class, and a large number of data requests at the lowest priority class.

The weighting of the priority levels was changed to a system where the weight of each priority level was a fixed multiple of the weight of the priority level immediately below it. Five different values for this multiple were used for this study, and each was evaluated with each of the 24 data request scenarios above, resulting in 120 testing scenarios for evaluation by the 79 heuristic/E-U ratio combinations.

As in the previous study, 40 individual test cases (each with a unique network configuration and set of data requests) were generated for each testing scenario, because a single case cannot reflect the range of possible data

Table 1. Network parameters used for the generation of test cases.

parameter	min. value	max. value
# machines	14	16
# srcs per data item	1	3
# dests per data item	1	5
src available time	1 sec	3600 sec
dest deadline delay	900 sec	3600 sec
data item size	10 kBytes	100 MBytes
machine storage	10 MBytes	20 GBytes
# outbound links	1	4
link bw	10 kBits/sec	1.5 MBits/sec

requests and network configurations. This resulted in the 379,200 simulation runs described in this section.

8.2. Generation of Test Cases

The network parameters used to create data sets for this simulation study are summarized in Table 1. Actual values were generated randomly with uniform probability between (and including) the minimum and the maximum values shown in the table. The "src available time" is the time the data item is available at all of its sources (the same time for all sources of that data item). The "dest deadline delay" is the deadline for the requested data item relative to the time it becomes available at its sources. These parameter values are intended to be representative of a subset of a BADD/AICE-like environment. For more information about how these parameters are used to generate the test cases, the reader is referred to [6].

For this simulation study, the number of data items generated for a network was 700 times the number of machines in the network. After all items were generated, Dijkstra's algorithm was run once for each item, establishing the individual satisfiability of each data item at each requesting destination along with a path of communication links used to reach each destination. The average number of communication links traversed from a source machine to a destination machine for all of the satisfiable requests is the "resulting average communication link traversal count."As indicated above, three different average link counts were generated (1.5, 2.5, and 3.5), and for each count, 40 different networks and associated data requests were created with the method given above, resulting in a total of 120 networks with associated data requests.

Now consider in the network all data requests that are determined to be satisfiable individually according the first execution of Dijkstra's algorithm. When considering each of these requests as if it were the only data request in the system, the resulting virtual link path from Dijkstra's algorithm and other known information can be used to calculate the bytes of bandwidth needed for each request. Then these bandwidths can be summed to give a value representing the total number of bytes of data bandwidth being requested in the system. Call this value $ReqBandwidth$. Recall now the value $NetBandwidth$ calculated by summing together the total number of bytes that could be transmitted on each of the virtual links within the network during the simulation period. An oversubscription rate can then be defined as $ReqBandwidth/NetBandwidth$. If this term is larger than 1, the network can clearly not satisfy all requests due to bandwidth limitations. If the term is less than 1, bandwidth may not exist between the correct machines or may not be available during the required time to satisfy all requests.

To examine system performance under various request loads, it was decided to consider networks with the following oversubscription rates: 25.0, 12.5, 6.2, 3.1, 1.6, 0.8, 0.4, and 0.2. These desired data sets were created by starting with one of the networks and its associated set of data requests, and removing random data requests until the desired oversubscription rate was achieved. This did not significantly affect the average communication link traversal counts. It resulted in data sets consisting of the same network with eight different oversubscription rates, for each of the 120 networks.

When applying the heuristics to these test cases, a variety of E-U ratios were used. For simulations run using the full path/one destination procedure with $C4size$ and $C4sizlnk$, the $log_{10}(W_E/W_U)$ used were inf, 9, 8, 7, 6, 5, 4, $-$ inf. The values of inf and $-$ inf represent considering only the priority term (the term weighted by W_E), and only the urgency term (the term weighted by W_U), respectively. For simulations run using the partial path procedure with $C4$, the full path/all destinations procedure with $C4$, and the full path/one destination procedure with $C4$, and $C4links$, the $log_{10}(W_E/W_U)$ used were inf, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0, $-$ inf.

The last parameter that was varied in this simulation study was the relative weight of one level of priority compared to another. With the six priority levels of data requests, the approach simulated was to make the weight of a priority level α (where $0 \leq \alpha \leq 5$) data request be ω^α (i.e., $W[\alpha] = \omega^\alpha$) for some fixed value of ω. The values of ω simulated were 1, 2, 4, 8, and 16, and this was done for each of the networks and loadings mentioned above. The results of the simulations using these parameters are now presented.

8.3. Evaluation of Simulations

Heuristic and bound labels used in the graphs at the end of this subsection are summarized in Table 2. As stated previously, because of the good performance of the full path/one destination procedure and Cost C4, they were the focus of these new experiments. The three new costs taking into account data item size and the number of communication links traversed from a source to a destination are shown

in Figure 1. The peak performance of the costs taking data item size into account are further to the right (signifying higher E-U ratios) in the graph because those costs divide the effective priority term by the data item size. Due to space constraints, only a subset of the results from [6] appears in this paper.

In Figures 2 through 5, the data points for the heuristics used correspond to the best E-U ratio for each testing scenario (for $\omega \neq 1$, this is a combination of the priority and urgency terms). The values for the normalized vertical axis in all of these graphs is computed as follows. For each test case, the sum of the satisfied requests' weighted priorities for a given heuristic or bound is divided by the sum of satisfied requests' weighted priorities given by the best E-U ratio for full_one_C4. This normalized sum is then averaged over the 40 network test cases to give the final value for each data point.

The relative performance of the heuristics are shown in Figures 2 and 3. The costs considering data item size will tend to allocate resources for all of the smaller data items first, resulting in many small time intervals of link bandwidth being allocated initially. In the lightly loaded cases, the remainder of the link bandwidth must be used by larger data items, but continuously available links may not exist for a long enough period of time for these larger data items to use. In the more heavily loaded network cases, there are enough smaller data items available to make use of all of the network bandwidth without sending any of the larger data items. The resulting trend is that the costs incorporating data item size have a relative decrease in performance for lightly oversubscribed networks, followed by a relative increase in performance for the heavily oversubscribed networks.

There is a general overall trend that as ω increases (and other factors are fixed), the performance of all heuristics is closer to each other. This is because more of the total sum of priority weights of requests in the system is contributed by a few highest priority requests.

The method full_one_C4links, performed very comparably to full_one_C4 in all tests. There was no situation indicated by these simulations where full_one_C4links should be chosen over full_one_C4, or vice versa. The partial_C4 method was also shown to perform comparably to the full_one_C4 method in all cases.

Table 2. Labels for heuristics and bounds used in the graphs of Section 8.3.

heuristic combination	label used
partial path w/ $C4$	partial_C4
full path/one dest. w/ C4	full_one_C4
full path/one dest. w/ C4links	full_one_C4links
full path/one dest. w/ C4size	full_one_C4size
full path/one dest. w/ C4sizlnk	full_one_C4sizlnk
full path/all dest. w/ C4	full_all_C4
possible satisfy bandwidth	possible_satisfy_bw
full path random Dijkstra	full_rand_Dijkstra

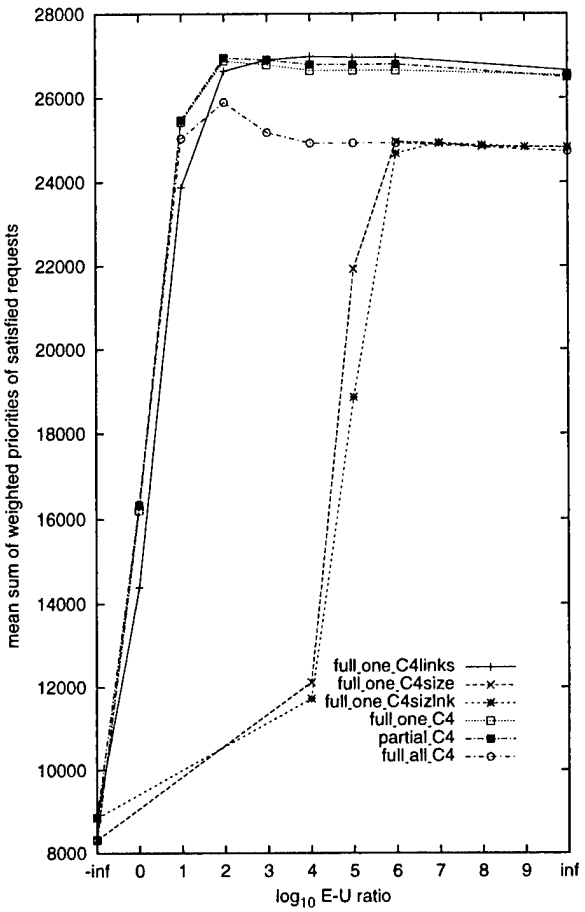


Figure 1. Sample graph of the effect of varying the E-U ratio (W_E/W_U). The data sets used had an average link traversal count of 2.5, a request over-subscription rate of 3.1, and an ω value of 4.

The full_all_C4 method is shown to perform well for small oversubscription rate, but as the oversubscription rate increases a clear decrease in performance is seen. This is due to the full path/all destinations procedure allocating resources for more than one destination simultaneously, where some requesting destinations may have very low priority.

Table 3 shows the average number of requests satisfied at each priority level by full_one_C4 as compared to a simple algorithm that schedules all requests of a higher priority level before any requests of a lower priority level. In particular, this algorithm was full_one_C1 with $W_U = 0$. For

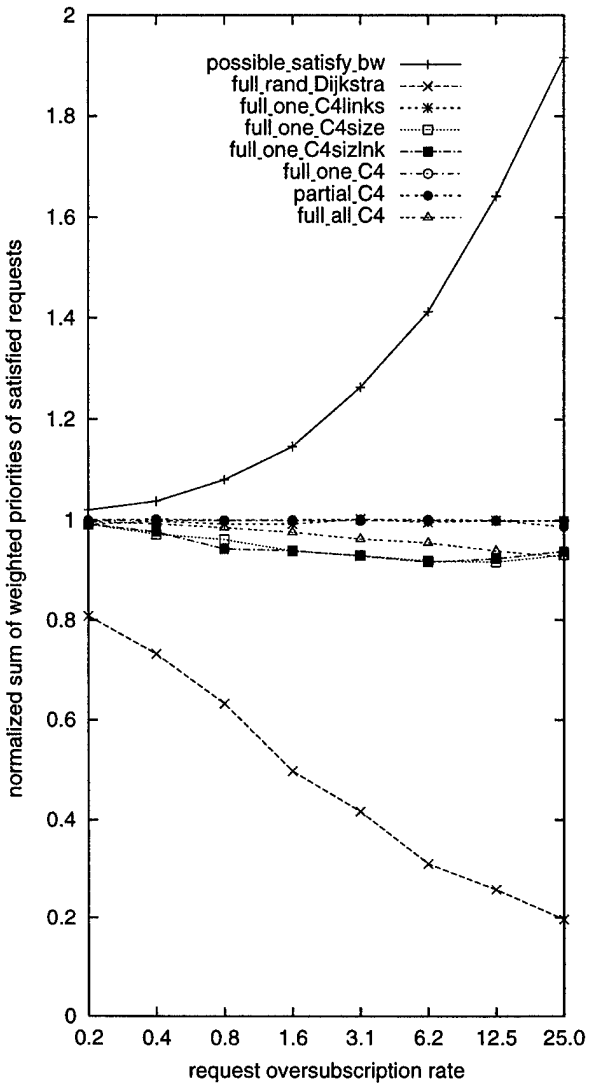


Figure 2. Weighted sum of satisfied requests' priorities normalized at each over-subscription rate to the performance of full_one_C4. The data sets had an average link traversal count of 2.5 and an ω value of 4.

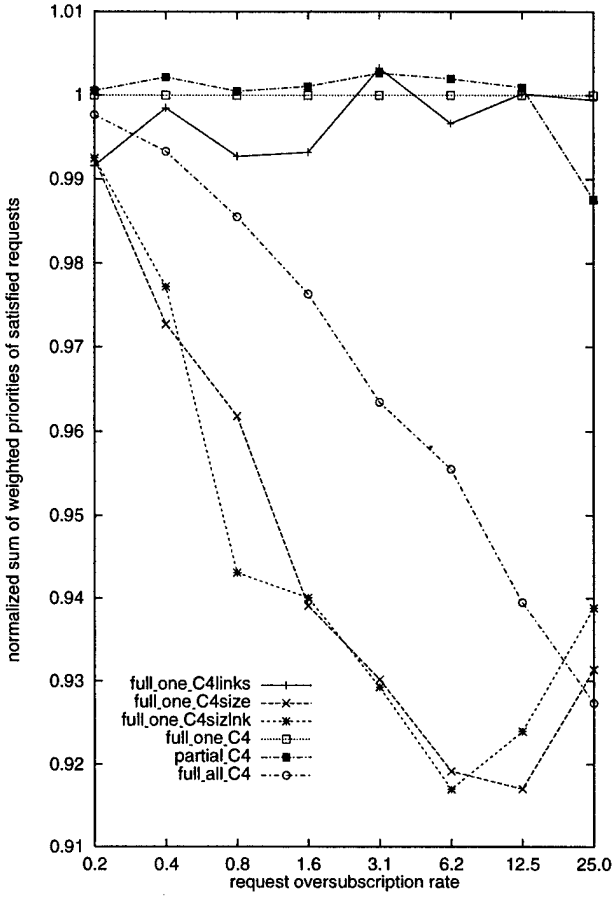


Figure 3. Weighted sum of satisfied requests' priorities normalized at each over-subscription rate to the performance of full_one_C4. The data sets had an average link traversal count of 2.5 and an ω value of 4.

$\omega > 1$ in these tables, more requests in the top three priority levels are being satisfied by full_one_C4 (which obeys the relative importance assigned to each of the priority levels set by the policy maker) than the level by level method (which ignores these policy requirements). The number of satisfied requests at the top priority level remains comparable for full_one_C4 and $\omega > 1$ because there are so few requests at that level that all are able to be satisfied. This is indicated by the fact that the level by level method cannot satisfy any more of the top priority requests. For example, even though the level by level method schedules all priority level 5 requests as if they were the only requests in the system, the total number scheduled does not exceed the results of full_one_C4 (for $\omega > 1$). This shows that full_one_C4 using urgency in addition to effective priority, is better than full_one_C1 without urgency. Furthermore, full_one_C4 results in a higher sum of weighted priorities of satisfied requests than the level by level method in almost all cases considered in Table 3.

In summary, a class of heuristics that compare well to upper and lower bounds has been developed and analyzed. Many heuristics perform within a few percentage points of each other, and this is why it is important to also consider the execution times of the different approaches. Furthermore, while in general several heuristics perform comparably, if a system is known to have a particular operating environment (e.g., ω value, oversubscription rate), there may

Table 3. Number requests satisfied at each priority level by full_one_C4 with an average link traversal count of 2.5 and an oversubscription rate of 1.6. The "level by level" column shows the effect of allocating resources for all priority class α requests before all priority class β requests where $\alpha > \beta$.

priority level	number requests satisfied					
	ω					level by level
	1	2	4	8	16	
5	4.2	8.3	8.4	8.4	8.4	8.2
4	9.8	16.0	16.1	16.1	16.1	16.0
3	16.4	23.4	23.4	23.1	23.5	22.8
2	28.8	33.8	31.2	27.0	32.5	32.5
1	57.6	50.5	44.8	43.1	43.0	50.1
0	118.8	81.6	82.0	85.9	84.9	78.6

be a preference for one heuristic over another. Confidence intervals for some of the data points generated by test cases in this section can be found in [6].

9. Data Items With Multiple Versions

9.1. Approach

In this section, a variable time, variable accuracy algorithm will be presented to deal with data items with a higher quality and lower quality version, as mentioned in Section 3. The higher quality data item is assumed for simplicity to be twice the size of the lower quality data item. The higher quality data item, however, has four times as much "worth" to the end user as the lower quality data item. This worth was chosen to indicate that the system should be penalized for selecting the lower quality data item over the higher one.

The approach used to incorporate these lower quality data item versions into the developed heuristics was to create an iterative algorithm that attempts to create a new schedule S_h with each iteration that has a smaller effect $E[S_h]$. In the first iteration, only the higher quality versions of the data items are considered satisfiable by the value $Sat[i,k]$ (where $0 \leq i < 2\rho$ and $0 < k \leq Nrq[i]$). That is, $Sat[i,k]$ (from the cost criteria of Section 5) can only be 1 if $0 \leq i < \rho$. A heuristic is then used with Dijkstra's algorithm to create a complete schedule of data transfers, which corresponds to the research described in Section 8.

After the first iteration schedule has been determined, the value of $Sat[j,k]$ (where $0 \leq j < \rho$) for the second iteration is only allowed to be 1 if $Request[j,k]$ was satisfied in the previous iteration. The value of $Sat[j+\rho,k]$ is then only allowed to be 1 if $Request[j,k]$ was not satisfied in the previous iteration. A complete new schedule is created using a heuristic with Dijkstra's algorithm. That is, if during iteration one a requesting destination does not receive its higher quality requested data item, then in the second iteration, it will request the lower quality version of that data item instead. The schedule produced by the second iteration will then likely satisfy at least a few lower quality data item requests (of higher priority) in place of higher quality data item requests (of lower priority). The higher quality data item requests that are not satisfied in the second iteration then request their respective lower quality versions for the third iteration. This iterative process can be repeated as many times as allotted execution time permits, and can stop at any time after the first iteration and output the best schedule that it has generated thus far. (This assumes that the best schedule is kept separately after each iteration and that the last iteration performance may not result in the best schedule.)

9.2. Costs C1, C2, and C3

Figures 4 and 5 include the full_one procedure with costs C1, C2, C3, and C4, where iteration 1 corresponds to the situation without consideration of versions. The full sets of experiments in [6] gave insights into the behaviors of these costs.

The full_one_C1 heuristic performs well except for the highest oversubscription rate test cases. Because cost $C1$ only considers the benefit of moving data to satisfy a single request, this suggests that in very highly oversubscribed networks, it helps to consider multiple requesting destinations that would collectively benefit from a data transfer.

The full_one_C2 method appears to suffer from its choice of destination machines; specifically, it allows a single destination's urgency (instead of a collective view of the urgencies) to affect which valid next step is selected. As system oversubscription rates increase, its relative performance decreases.

The full_one_C3 method performs consistently poorly for heavily oversubscribed networks. Its performance in the simulation studies of [20] indicated that it would not likely perform well, so this was expected. It is interesting to note, however that as ω was increased, the relative performance of full_one_C3 increased as well. This suggests that the problem with cost $C3$ is indeed due to allowing the urgency factor to dominate the cost equation, because as the priority weight is increased, it begins to perform well. This is especially true for the lower oversubscription rates.

9.3. Evaluation of Simulations

The data sets used for these experiments were a subset of the data sets created for the simulation study of Section 8. For very light loading (i.e., 0.2), all of the heuristics perform similarly after the second iteration. Only the data sets with average link traversal counts of 2.5 were used. Five iterations of the variable accuracy algorithm were run. Results from those runs under different loads is shown in Figures 2 and 3, where Figure 2 includes the upper and lower bounds. It should be noted that each graph is normalized to

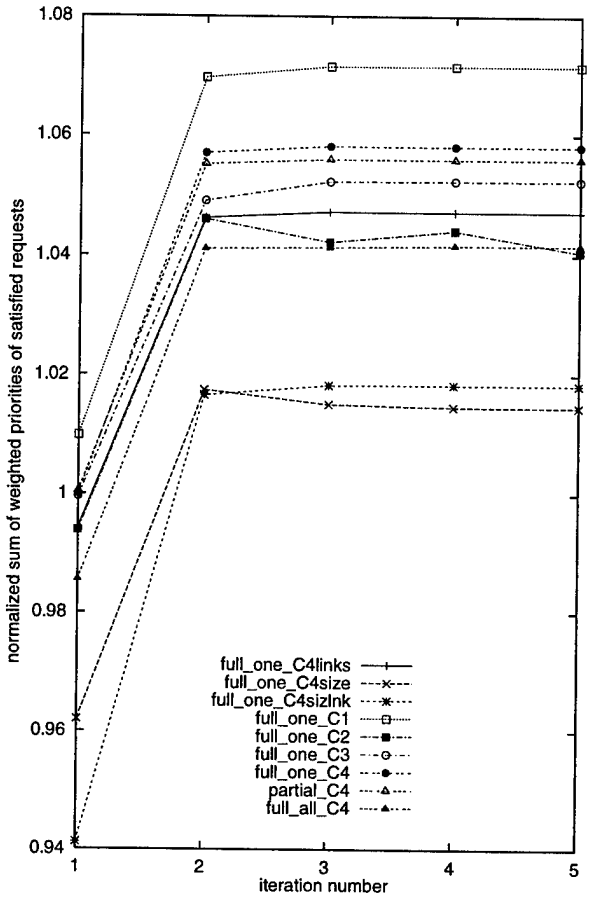


Figure 4. Weighted sum of satisfied requests' priorities normalized to the performance of full_one_C4 in iteration 1. The data set had an oversubscription rate of 0.8, an average link traversal count of 2.5, and an ω value of 4.

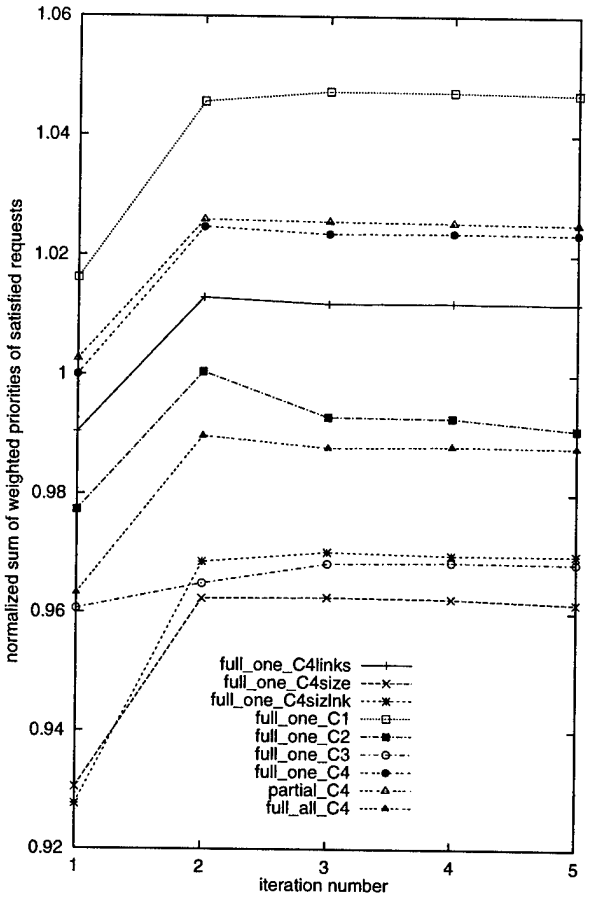


Figure 5. Weighted sum of satisfied requests' priorities normalized to the performance of full_one_C4 in iteration 1. The data set had an oversubscription rate of 3.1, an average link traversal count of 2.5, and an ω value of 4.

the performance of full_one_C4 at the end of its first iteration, which is the same as the performance of full_one_C4 in the study of Section 8.

For less oversubscribed networks, the heuristics are almost all able to increase their own respective performance with additional iterations (e.g., Figure 4). For more oversubscribed networks, this is not generally the case (e.g., Figure 5). All of the cost criteria used here except $C1$ consider more than one destination as part of the cost of sending a data item to its next machine. The implementation of the multiple versions approach works against this, particularly at higher oversubscription rates. For example, in iteration one, multiple requests may contribute to the overall sum for a transfer to destination d_1. When using multiple versions, the destinations that receive the second version will no longer contribute to the sum for destination d_1. Because of this, destination d_1's request may no longer have a large enough sum to obtain network resources. For this reason, full_one_C1 (which does not collectively consider multiple requesting destinations) is less inclined to decrease in performance in successive iterations after the second iteration.

An additional reason for a lack of improvement after each iteration for data sets with high oversubscription rates is related to the large number of requests of high priority in the system. There are already very many data items in these tests with a desirable priority to select from, and the secondary versions of data items are not any better of a choice than any of the primary versions of data items that are available.

In summary, the use of multiple versions will help some heuristics improve the sum of priorities satisfied in all but the most oversubscribed cases. The improvement obtained in some operator environments exceeds 10%. In almost all cases, the best improvement is given by the second iteration

of the variable time, variable accuracy algorithm.

10. Summary and Conclusions

Data staging is an important data management issue for distributed computer systems. It addresses the issues of distributing and storing over numerous geographically dispersed locations both repository data and continually generated data through an oversubscribed network, where not all data requests can be satisfied. When certain data with their corresponding priorities need to arrive at a site with limited storage capacities in a timely fashion, a heuristic must be devised to schedule the necessary communication steps efficiently.

The performance of nine heuristics were shown, and compared to an upper bound and a lower bound. Many different weighting schemes for the relative importance of different priority levels of requested data items were considered. Each procedure and cost criterion was designed with particular advantages in mind. The results presented showed that, for the system parameters considered (e.g., priority weighting, oversubscription rate), the combination of cost $C4$ or $C1$ with the full path/one destination procedure and $C4$ with the partial path procedure consistently performed the best, when using the measure of weighted sum of priorities satisfied. Because each heuristic has advantages, the procedure/cost criterion pair that performs best may differ depending on the system parameters (i.e., the actual environment where the scheduler heuristic will be deployed).

An additional novel approach using a variable time, variable accuracy method that considered multiple data item versions with different resource requirements was evaluated. The use of multiple versions was shown to help some heuristics in all but the most oversubscribed cases; in many cases, the improvement was over 10%.

In summary, a class of heuristics and cost criteria that compare well to upper and lower bounds were developed and analyzed. While in general several heuristics perform comparably, if a system is known to have a particular operating environment (e.g., ω value, oversubscription rate), there may be a preference for one pair over another.

Acknowledgments: The authors thank Joe Rockmore, Bob Beaton, Jose Fortes, and Edwin Chong for their valuable comments and suggestions.

[1] S. Acharya and S. B. Zdonik, "An efficient scheme for dynamic data replication," Technical Report CS-93-43, CS Dept., Brown Univ., Sep. 1993, 25 pp.

[2] "Agile Information Control Environment Proposers Information Package," BAA 98-26, http://web-ext2.darpa.mil/iso/aice/AICE98-26.htm, May 1998.

[3] M. Baentsch, L. Baum, G. Molter, S. Rothkugel, and P. Sturm, "Enhancing the web's infrastructure: From caching to replication," *IEEE Internet Computing*, Vol. 1, No. 2, Mar.-Apr. 1997, pp. 18-27.

[4] S. Balakrishnan and F. Özgüner, "A priority-driven flow control mechanism for real-time traffic in multiprocessor networks," *IEEE Trans. on Parallel and Distributed Systems* , Vol. 9, No. 2, July 1998, pp. 664-678.

[5] A. Bestavros, "WWW traffic reduction and load balancing through server-based caching," *IEEE Concurrency*, Vol. 5, No. 1, Jan.-Mar. 1997, pp. 56-67.

[6] N. B. Beck, M. D. Theys, H. J. Siegel, and M. Jurczyk, "Evaluation of Heuristics in a Distributed Data Staging Network," Technical Report TR-ECE 99-7, ECE School, Purdue Univ., May 1999, 141 pp.

[7] M. A. Bonuccelli and M. C. Clo, "EDD algorithm performance guarantee for periodic hard-real-time scheduling in distributed systems," *13th Int'l Parallel Processing Symp. and 10th Symp. Parallel and Distributed Programming (IPPS/SPDP'99)*, Apr. 1999, pp. 668-677.

[8] T. D. Braun, H. J. Siegel, N. Beck, L. L. Bölöni, M. Maheswaran, A. I. Reuther, J. P. Robertson, M. D. Theys, and B. Yao, "A taxonomy for describing matching and scheduling heuristics for mixed-machine heterogeneous computing systems," *IEEE Workshop on Advances in Parallel and Distributed Systems*, Oct. 1998, pp. 330-335.

[9] T. D. Braun, H. J. Siegel, N. Beck, L. L. Bölöni, M. Maheswaran, A. I. Reuther, J. P. Robertson, M. D. Theys, B. Yao, D. Hensgen, and R. F. Freund, "A comparison study of static mapping heuristics for a class of meta-tasks on heterogeneous computing systems," *8th IEEE Workshop on Heterogeneous Computing Systems (HCW '99)*, Apr. 1999, pp. 15-29.

[10] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms*, MIT Press, Cambridge, MA, 1990.

[11] P. Danzig, R. Hall, and M. Schwartz, "A case for caching file objects inside internetworks," Technical Report CU-CS-642-93, CS Dept., Univ. of Colorado, Mar. 1993, 15 pp.

[12] J. K. Kim, D. A. Hensgen, T. Kidd, H. J. Siegel, D. St. John, C. Irvine, T. Levin, V. K. Prasanna, and R. F. Freund, "A QoS performance measure framework for distributed heterogeneous networks," *8th Euromicro Workshop on Parallel and Distributed Processing*, Jan. 2000, pp. 18 - 27.

[13] P. C. Jones, T. J. Lowe, G. Muller, N. Xu, Y. Ye, and J. L. Zydiak, "Specially structured uncapacitated facility location problem," *Operations Research*, Vol. 43, No. 4, July-Aug. 1995, pp. 661-669.

[14] M. J. Lemanski and J. C. Benton, *Simulation for SmartNet Scheduling of Asynchronous Transfer Mode Virtual Channels*, Master's Thesis, CS Dept., Naval Postgraduate School, June 1997.

[15] A. J. Rockmore, "BADD functional description," Internal DARPA Memo, Febr. 1996, 9 pp.

[16] SmartNet/Heterogeneous Computing Team, "BC2A/TACITUS/BADD integration plan," Internal NRaD Naval Laboratory Report, Aug. 1996, 16 pp.

[17] J. A. Stankovic, M. Spuri, K. Ramamritham, and G. C. Buttazzo, *Deadline Scheduling for Real-Time Systems*, Kluwer Academic Publishers, Boston, MA, 1998.

[18] M. Tan, H. J. Siegel, J. K. Antonio, and Y. A. Li, "Minimizing the application execution time through scheduling of subtasks and communication traffic in a heterogeneous computing system," *IEEE Trans. on Parallel and Distributed Systems*, Vol. 8, No. 8, Aug. 1997, pp. 857-871.

[19] M. Tan, M. D. Theys, H. J. Siegel, N. B. Beck, M. Jurczyk, "A mathematical model, heuristic, and simulation study for a basic data staging problem in a heterogeneous networking environment," *7th IEEE Heterogeneous Computing Workshop (HCW '98)*, Apr. 1998, pp. 115-129.

[20] M. D. Theys, N. B. Beck, H. J. Siegel, M. Jurczyk, and M. Tan, "Scheduling heuristics for data requests in an oversubscribed network with priorities and deadlines," *Int'l Conf. Distributed Computing Systems*, Apr. 2000, to appear.

[21] L. Wang, H. J. Siegel, V. P. Roychowdhury, and A. A. Maciejewski, "Task matching and scheduling in heterogeneous computing environments using a genetic-algorithm-based approach," *J. of Parallel and Distributed Computing*, Vol. 47, No. 1, Nov. 1997, pp. 1-15.

Author Biographies

Mitchell D. Theys is an assistant professor in the Electrical Engineering and Computer Science Department at the University of Illinois at Chicago. He obtained his PhD in 1999 from the School of Electrical and Computer Engineering at Purdue University. While working on his PhD, he was supported by the DARPA BADD and AICE programs, and an Intel, AFCEA, and Meisner (through the school of ECE at Purdue) fellowship. He also obtained an MSEE and a BSCEE from the School of Electrical Engineering at Purdue. His research interests include parallel systems on a chip, distributed systems, communication systems, and computer architecture.

Noah B. Beck is an UltraSPARC Verification Engineer at the Sun Microsystems Boston Design Center. He received a Bachelor of Science in Computer Engineering in 1997 and a Master of Science in Electrical Engineering in 1999 from Purdue University. His research interests include microprocessor architecture and verification, parallel computing, and heterogeneous computing.

Howard Jay Siegel is a Professor in the School of Electrical and Computer Engineering at Purdue University. He is a Fellow of the IEEE and a Fellow of the ACM. He received BS degrees in both electrical engineering and management from MIT, and the MA, MSE, and PhD degrees from the Department of Electrical Engineering and Computer Science at Princeton University. Prof. Siegel has coauthored over 250 technical papers, has coedited seven volumes, and wrote the book *Interconnection Networks for Large-Scale Parallel Processing*. He was a Coeditor-in-Chief of the *Journal of Parallel and Distributed Computing*, and was on the Editorial Boards of the *IEEE Transactions on Parallel and Distributed Systems* and the *IEEE Transactions on Computers*. He was Program Chair/Co-Chair of three conferences, General Chair/Co-Chair of four conferences, and Chair/Co-Chair of four workshops. He is an international keynote speaker and tutorial lecturer, and a consultant for government and industry.

Michael Jurczyk is an assistant professor at the Computer Engineering and Computer Science Department at the University of Missouri-Columbia. He studied Electrical Engineering at Purdue University and the University of Bochum, Germany, where he received his Diploma in 1990. He obtained his PhD in Electrical Engineering from the University of Stuttgart, Germany, in 1996, where he studied parallel simulation and performance issues of interconnection networks. In 1996, he was a visiting assistant professor at the School of Electrical and Computer Engineering at Purdue University. His research interests include parallel and distributed systems, interconnection networks for parallel and communication systems, ATM-networking, and networked multimedia.

Fast Heterogeneous Binary Data Interchange

Greg Eisenhauer and Lynn K. Daley
College of Computing
Georgia Institute of Technology

Abstract

As distributed applications have become more widely used, they more often need to leverage the computing power of a heterogeneous network of computer architectures. Modern communications libraries provide mechanisms that hide at least some of the complexities of binary data interchange among heterogeneous machines. However, these mechanisms may be cumbersome, requiring that communicating applications agree a priori *on precise message contents, or they may be inefficient, using both "up" and "down" translations for binary data. Finally, the semantics of many packages, particularly those which require applications to manually "pack" and "unpack" messages, result in multiple copies of message data, thereby reducing communication performance. This paper describes PBIO, a novel messaging middleware which offers applications significantly more flexibility in message exchange while providing an efficient implementation that offers high performance.*

1 Introduction

As distributed applications have become more widely used, they often need to leverage the computing power of a heterogeneous network of computer architectures. Modern communications libraries provide mechanisms that hide at least some of the complexities of binary data interchange among heterogeneous machines. The features and semantics of these packages are typically a compromise between what might be useful to the applications and what can be implemented efficiently.

For example, many packages, such as PVM[8] and Nexus[7], support message exchanges in which the communicating applications "pack" and "unpack" messages, building and decoding them field by field, datatype by datatype. Other packages, such as MPI[6], allow the creation of user-defined datatypes for messages and message fields and provide some amount of marshalling and unmarshalling support for those datatypes internally.

The approach of requiring the application to build messages manually offers applications significant flexibility in message contents while ensuring that the pack and unpack operations are performed by optimized, compiled code. However, relegating message packing and unpacking to the communicating applications means that those applications must have *a priori* agreement on the contents and format of messages. This is not an onerous requirement in small-scale stable systems, but in enterprise-scale distributed computing, the need to simultaneously update all application components in order to change message formats can be a significant impediment to the integration, deployment and evolution of complex systems.

In addition, the semantics of application-side pack/unpack operations generally imply a data copy to or from message buffers. Such copies are known[11, 13] to have a significant impact on communication system performance. Packages which can perform internal marshalling, such as MPI, have an opportunity to avoid data copies and to offer more flexible semantics in matching fields provided by senders and receivers. However, existing packages have failed to capitalize on those opportunities. For example, MPIs type-matching rules require strict *a priori* agreement on the contents of messages. Additionally, most MPI implementations implement marshalling of user-defined datatypes via mechanisms that amount to interpreted versions of field-by-field packing.

This paper describes PBIO(Portable Binary Input/Output)[3], a multi-purpose communication middleware. In developing PBIO we have not attempted to recreate various higher-level communication abstractions offered by MPI or by the Remote Service Requests of Nexus. Instead, we provide flexible heterogeneous binary data transport for simple messaging of a wide range of application data structures, using novel approaches such as dynamic code generation (DCG) to preserve efficiency. In addition, PBIO's flexibility in matching transmitted and expected data

0-7695-0556-2/00 $10.00 © 2000 IEEE

types provides key support for *application evolution* that is missing from other communication systems.

This paper briefly describes PBIO semantics and features, and then illustrates performance metrics across a heterogeneous environment of Sun Sparc and X86-based machines running Solaris. These metrics are compared against the data communication measurements obtained by using MPI as a data communication mechanism across the same network architecture. The paper will show that the features and flexibility of PBIO do not impose overhead beyond that imposed by other communications systems. In the worst case PBIO performs as well as other systems, and in many cases PBIO offers a significant performance improvement over comparable communications packages.

Much of PBIO's performance advantage is due to its use of dynamic code generation to optimize translations from wire to native format. Because this is a novel feature in communications middleware, its impact on PBIO's performance is also considered independently. In this manner, we show that for purposes of data compatibility, PBIO, along with code generation, can provide reliable, high performance, easy-to-use, easy-to-migrate, heterogeneous support for distributed applications.

2 The PBIO Communication Library

In order to conserve I/O bandwidth and reduce storage and processing requirements, storing and transmitting data in binary form is often desirable. However, transmission of binary data between heterogeneous environments has been problematic. PBIO was developed as a portable self-describing binary data library, providing both stream and file support along with data portability.

The basic approach of the Portable Binary I/O library is straightforward. PBIO is a record-oriented communications medium. Writers of data must provide descriptions of the names, types, sizes and positions of the fields in the records they are writing. Readers must provide similar information for the records they wish to read. No translation is done on the writer's end, our motivation being to offload processing from data providers (e.g., servers) whenever possible. On the reader's end, the format of the incoming record is compared with the format expected by the program. Correspondence between fields in incoming and expected records is established by field name, with no weight placed on size or ordering in the record. If there are discrepancies in field size or placement, then PBIO's conversion routines perform the appropriate translations. Thus, the reader program may read the binary information produced by the writer program despite potential differences in: (1) byte ordering on the reading and writing architectures; (2) differences in sizes of data types (e.g. long and int); and (3) differences in structure layout by compilers.

Since full format information for the incoming record is available prior to reading it, the receiving application can make run-time decisions about the use and processing of incoming messages about whom it had no *a priori* knowledge. However, this additional flexibility comes with the price of potentially complex format conversions on the receiving end. Since the format of incoming records is principally defined by the native formats of the writers and PBIO has no *a priori* knowledge of the native formats used by the program components with which it might communicate, the precise nature of this format conversion must be determined at run-time.

Since high performance applications can ill afford the increased communication costs associated with interpreted format conversion, PBIO uses dynamic code generation to reduce these costs. The customized data conversion routines generated must be able to access and store data elements, convert elements between basic types and call subroutines to convert complex subtypes. Measurements[4] show that the one-time costs of DCG, and the performance gains by then being able to leverage compiled (and compiler-optimized) code, far outweigh the costs of continually interpreting data formats. The analysis in the following section shows that DCG, together with native-format data transmission and copy reduction, allows PBIO to provide its additional type-matching flexibility without negatively impacting performance. In fact, PBIO outperforms our benchmark communications package in all measured situations.

3 Evaluation

In order to thoroughly evaluate PBIO's performance and its utility in high-performance communication, we present a variety of measurements in different circumstances. Where possible, we compare PBIO's performance to the cost of similar operations in MPI. Additionally, we include measurements which evaluate PBIO's performance in situations which are not supported by other communications packages. In particular, we evaluate PBIO's support for application evolution and its ability to transmit dynamically sized data elements.

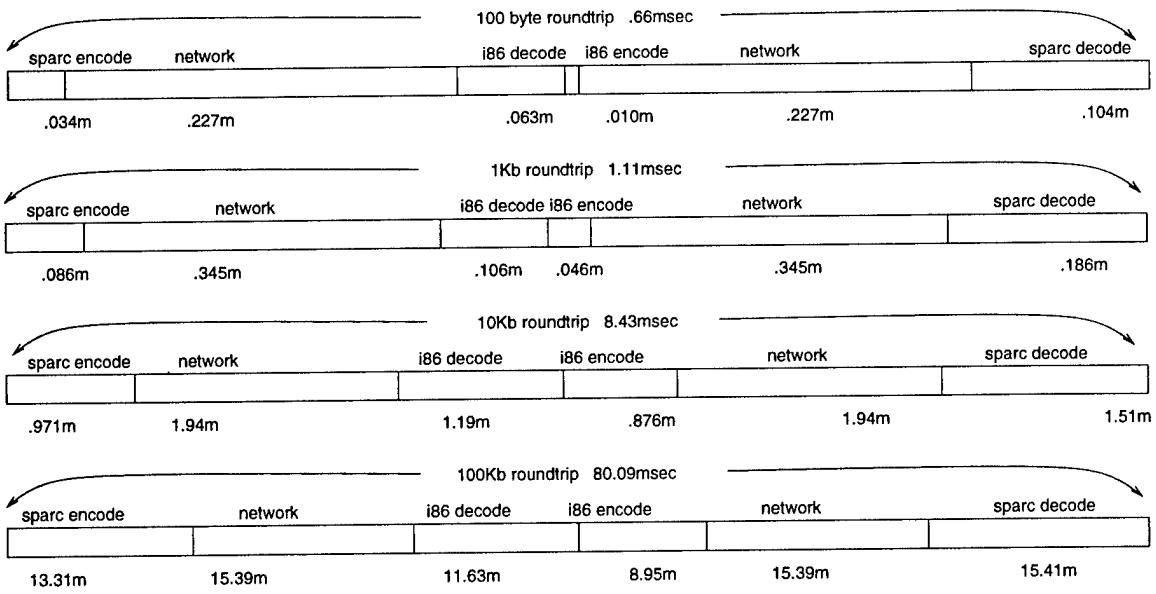


Figure 1: Cost breakdown for message exchange.

3.1 Analysis of costs in heterogeneous data exchange

Before analyzing PBIO costs in detail, it is useful to examine the costs in an exchange of binary data in a heterogeneous environment. As a baseline for this discussion, we use the MPICH[10] implementation of MPI, a popular messaging package in cluster computing environments. Figure 1 represents a breakdown of the costs in an MPI message round-trip between a x86-based PC and a Sun Sparc connected by 100 Mbps Ethernet.[1] The time components labeled "Encode" represent the time span between the application invoking `MPI_send()` and the eventual call to write data on a socket. The "Decode" component is the time span between the recv() call returning and the point at which the data is in a form usable by the application. In generating these numbers network transmission times were measured with NetPerf[9] and send and receive times were measured by substituting dummy calls for socket send() and recv(). This delineation allows us to focus on the encode/decode costs involved in binary data exchange. That these costs are significant is clear from the figure, where they typically represent 66% of the total cost of the exchange.

Figure 1 shows the cost breakdown for messages of a selection of sizes, but in practice, message times depend upon many variables. Some of these variables, such as basic operating system characteristics that affect raw end-to-end TCP/IP performance, are beyond the control of the application or the communication middleware. Different encoding strategies in use by the communication middleware may change the number of raw bytes transmitted over the network, but those differences tend to be negligible. Therefore, the remainder of our analysis will concentrate on the more controllable sending side and receiving side costs.

Another application characteristic which has a strong effect upon end-to-end message exchange time is the precise nature of the data to be sent in the message. It could be a contiguous block of atomic data elements (such as an array of floats), a stride-based element (such as a stripe of a homogeneous array), a structure containing a mix of data elements, or even a complex pointer-based structure. MPI, designed for scientific computing, has strong facilities for homogeneous arrays and strided elements. MPIs support for structures is less efficient than its support for contiguous arrays of atomic data elements, and it doesn't attempt to supported pointer-based structures at all. PBIO doesn't attempt to support strided array access, but otherwise supports all types with equal efficiency, including a non-recursive subset of pointer-based structures. The message type of the 100Kb message in Figure 1 is a non-homogeneous structure taken from the messaging requirements of a real application, a mechanical engineering simula-

[1] The Sun machine is an Ultra 30 with a 247 MHz cpu running Solaris 7. The x86 machine is a 450 MHz Pentium II, also running Solaris 7.

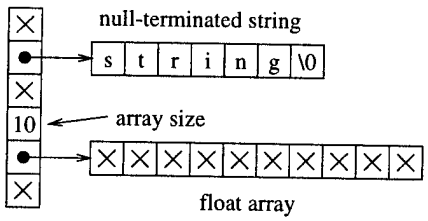


Figure 2: Strings and dynamic arrays in PBIO.

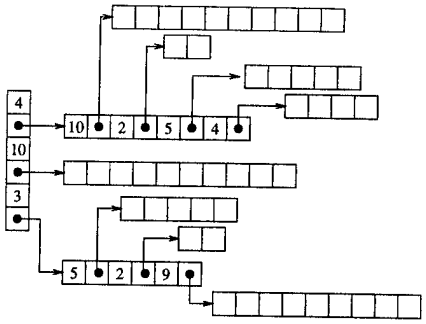


Figure 3: A multi-level pointer structure that can be transmitted by PBIO.

tion of the effects of micro-structural properties on solid-body behavior. The smaller message types are representative subsets of that mixed-type message.

The next sections will examine PBIO's costs in exchanging the same sets of messages. Subsequently, Section 3.5 will examine costs for other data types.

3.2 Sending side cost

As is mentioned in Section 2, PBIO transmits data in the native format of the sender. No copies or data conversions are necessary to prepare simple structure data for transmission. So, while MPICH's costs to prepare for transmission on the Sparc vary from 34μsec for the 100 byte record up to 13 msec for the 100Kb record, PBIO's cost is a flat 3 μsec. Of course, this efficiency is accomplished by moving most of the complexity to the receiver, where Section 3.3 tells a more complex story.

As mentioned above, PBIO also supports the transmission of some pointer-based structures. In particular, PBIO allows an element of a structure be to a null-terminated string, or a pointer to a dynamically sized array,[2] as shown in Figure 2. The array elements may be of an atomic data type or a previously registered structure. That there is no forward declaration mechanism or self-referentiality for structure types restricts PBIO from describing such things as linked lists. However, relatively complex structures, such as the one depicted in Figure 3 can be directly transmitted. The ability to directly transmit dynamically sized arrays is a feature that is not normally present in communications middleware.

Unlike contiguous structures, pointer-based entities do require some preparation before they are sent. In particular, PBIO must walk the structure to 1) prepare a transmission list of data blocks and their lengths, and 2) change all internal pointers from addresses to offsets within the message. The type semantics ensures that there can be no circularities in the structure, so the 'walk' is a simple tree descent which stops when it reaches the 'leaf' structures which contain no pointers. In order to avoid changing the data directly, structures containing pointers are copied to temporary memory and the pointers modified there. This imposes a cost on the sender that is proportional to the amount of data that must be copied and the number of pointers that must be adjusted. Because no similar features are included in common communications libraries, we don't include any representative measurements of these costs. However, we do observe that in the most common use of dynamic arrays, where a relatively small base structure holds pointers and sizes for one or more arrays, the 'walk' is a simple pass over the base structure, the majority of the data is in the 'leaves' which are *not* copied, and the additional sender-side processing is not overly significant.

[2] In the case of a dynamically sized array, the array size must be given by another, integer-typed, element in the base structure.

3.3 Receiving side cost

PBIO's approach to binary data exchange eliminates sender-side processing by transmitting in the sender's native format and isolating the complexity of managing heterogeneity in the receiver. Essentially, the receiver must perform a conversion from the various incoming 'wire' formats to the receiver's 'native' format. PBIO matches fields by name, so a conversion may require byte-order changes (byte-swapping), movement of data from one offset to another, or even a change in the basic size of the data type (for example, from a 4-byte integer to an 8-byte integer).

This conversion is another form of the "marshaling problem" that occurs widely in RPC implementations[1] and in network communication. That marshaling can be a significant overhead is also well known[2, 14], and tools such as USC[12] attempt to optimize marshaling with compile-time solutions.

Unfortunately, the dynamic form of the marshaling problem in PBIO, where the layout and even the complete field contents of the incoming record are unknown until run-time, rules out such static solutions. The conversion overhead is nil for some homogeneous data exchanges, but as Figure 1 shows, the overhead is high (66%) for some heterogeneous exchanges.

Generically, receiver-side overhead in communication middleware has several components which can be traded off against each other to some extent. Those basic costs are:

- byte-order conversion,
- data movement costs, *and*
- control costs.

Byte order conversion costs are to some extent unavoidable. If the communicating machines use different byte orders, the translation must be performed somewhere regardless of the capabilities of the communications package.

Data movement costs are harder to quantify. If byteswapping is necessary, data movement can be performed as part of the process without incurring significant additional costs. Otherwise, clever design of the communications middleware can often avoid copying data. However, packages that define a 'wire' format for transmitted data have a harder time being clever in this area. One of the basic difficulties is that the native format for mixed-datatype structures on most architectures has gaps, unused areas between fields, inserted by the compiler to satisfy data alignment requirements. To avoid making assumptions about the alignment requirements of the machines they run on, most packages use wire formats which are fully packed and have no gaps. This mismatch *forces* a data copy operation in situations where a clever communications system might otherwise have avoided it.

Control costs represent the overhead of iterating through the fields in the record and deciding what to do next. Packages which require the application to marshal and unmarshal their own data have the advantage that this process occurs in special-purpose compiler-optimized code, minimizing control costs. However, to keep that code simple and portable, such systems uniformly rely on communicating in a predefined wire format, incurring the data movement costs described in the previous paragraph.

Packages that marshal data themselves typically use an alternative approach to control, where the marshalling process is controlled by what amounts to a table-driven interpreter. This interpreter marshals or unmarshals application-defined data making

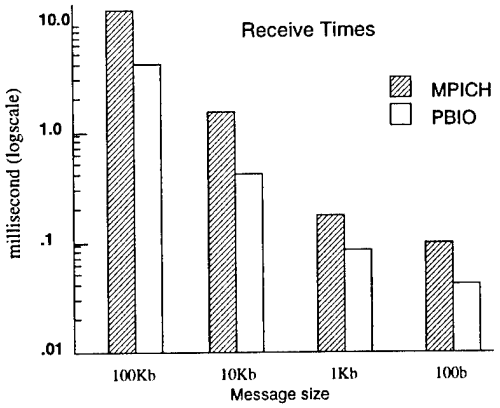


Figure 4: Receiver side costs for PBIO and MPI interpreted conversions.

data movement and conversion decisions based upon a description of the structure provided by the application and its knowledge of the format of the incoming record. This approach to data conversion gives the package significant flexibility in reacting to changes in the incoming data and was our initial choice for PBIO. Figure 4 shows a comparison of receiver-side processing costs on the Sparc for interpreted converters used by MPICH (via the `MPI_Unpack()`) call and PBIO. PBIO's converter is relatively heavily optimized and performs considerably better than MPI, in part because MPICH uses a separate buffer for the unpacked message rather than reusing the receive buffer (as PBIO does). However, PBIO's receiver-side conversion costs still contribute roughly 20% of the cost of an end-to-end message exchange. While a portion of this conversion overhead must be the consequence of the raw number of operations involved in performing the data conversion, we believed that a significant fraction of this overhead was due to the fact that the conversion is essentially being performed by an interpreter.

Our decision to transmit data in the sender's native format results in the wire format being unknown to the receiver until run-time, making a remedy to the problem of interpretation overhead difficult. However, our solution to the problem was to employ dynamic code generation to create a customized conversion subroutine for every incoming record type. These routines are generated by the receiver on the fly, as soon as the wire format is known, through a procedure that structurally resembles the interpreted conversion itself. However, instead of performing the conversion this procedure directly generates machine code for performing the conversion.

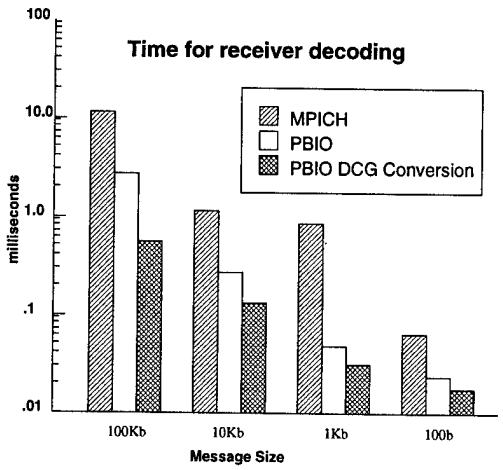


Figure 5: Receiver side costs for interpreted conversions in MPI and PBIO and DCG conversions in PBIO.

The execution times for these dynamically generated conversion routines are shown in Figure 5. The dynamically generated conversion routine operates significantly faster than the interpreted version. This improvement removes conversion as a major cost in communication, bringing it down to near the level of a copy operation, and is the key to PBIO's ability to efficiently perform many of its functions.

The cost savings achieved by PBIO through the techniques described in this section are directly reflected in the time required for an end-to-end message exchange. Figure 6 shows a comparison of PBIO and MPICH message exchange times for mixed-field structures of various sizes. The performance differences are substantial, particularly for large message sizes where PBIO can accomplish a round-trip in 45% of the time required by MPICH. The performance gains are due to:

- virtually eliminating the sender-side encoding cost by transmitting in the sender's native format, *and*
- using dynamic code generation to customize a conversion routine on the receiving side (currently not done on the x86 side).

3.4 Details of dynamic code generation

The dynamic code generation in PBIO is performed by Vcode, a fast dynamic code generation package developed at MIT by Dawson Engler[5]. We have significantly enhanced Vcode and ported it to several new architectures. The present implementation we can generate code for Sparc (v8, v9 and v9 64-bit), MIPS (old 32-bit, new 32-bit and 64-bit ABIs) and DEC Alpha architectures. An x86 port of Vcode is in progress, but not yet sufficiently advanced for us to generate PBIO's conversion routines. Vcode essentially provides an API for a virtual RISC instruction set. The provided instruction set is relatively generic, so that most Vcode instruction macros generate only one or two native machine instructions. Native machine instructions are generated directly into a memory buffer and can be executed without reference to an external compiler or linker.

Employing DCG for conversions means that PBIO must bear the cost of generating the code as well as executing it. Because the format information in PBIO is transmitted only once on each connection and data tends to be transmitted many times, conversion generation is not normally a significant overhead. Yet that overhead must still be considered to determine whether or not the use of DCG results in performance gains.

The proportional overhead encountered in actually generating conversion code varies dramatically depending upon the internal structure of the record. This differs from the situation in Figure 5, where the worst-case conversion run-time is more dependent upon the size of the message than its structure. To understand this variation, consider the conversion of a record that contains large internal arrays. In this case, the conversion code consists of a few *for* loops that process large amounts of data. In comparison, a record of similar size consisting solely of independent fields of atomic data types requires custom code for each field. The result is that for records consisting solely of arrays, DCG almost always improves performance. For array-based records of around 200 bytes the time to generate and execute dynamic conversion code is less than the time to perform an interpreted conversion. At that point, DCG is a performance improvement, even if the conversion routine is only used once.

The situation is less clear for record formats consisting mostly of individual atomic fields. For this type of record, dynamically generated conversions run nearly an order of magnitude faster than interpreted conversions, but the one-time cost of doing the code generation is relatively high. Obviously, if many records are exchanged, the costs will be amortized over the improved conversion times. But for one-time exchanges dynamic code generation for conversions may be more expensive than simple interpreted conversions.

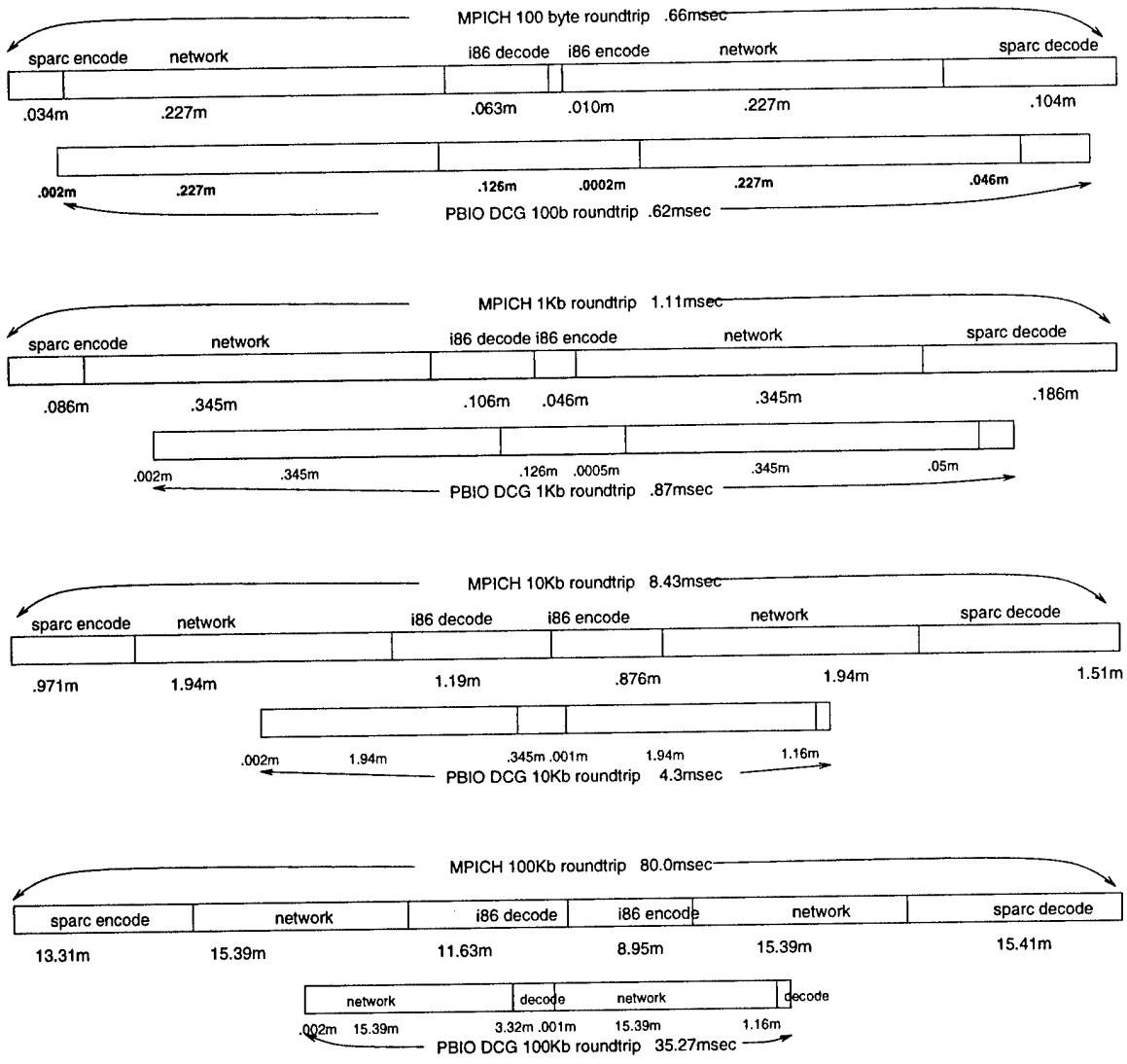


Figure 6: Cost comparison for PBIO and MPICH message exchange.

```
start of proceedure bookkeeping
    save  %sp, -360, %sp
byteswap load and store the 'ivalue' field.
    clr  %g1
    ldswa  [ %i0 + %g1 ] #ASI_P_L, %g2
    st  %g2, [ %i1 ]

byteswap load and store the 'dvalue' field
    mov  4, %g1
    ldswa  [ %i0 + %g1 ] #ASI_P_L, %g2
    mov  8, %g1
    ldswa  [ %i0 + %g1 ] #ASI_P_L, %g3
    st  %g3, [ %sp + 0x158 ]
    st  %g2, [ %sp + 0x15c ]
    ldd  [ %sp + 0x158 ], %f4
    std  %f4, [ %i1 + 8 ]

loop to handle 'iarray'
save 'incoming' and 'destination' pointers for later
restoration
    st  %i0, [ %sp + 0x160 ]
    st  %i1, [ %sp + 0x164 ]

make regs i0 and i1 point to start of incoming and
destination float arrays
    add  %i0, 0xc, %i0
    add  %i1, 0x10, %i1
setup loop counter
    mov  5, %g3

loop body.
    clr  %g1
    ldswa  [ %i0 + %g1 ] #ASI_P_L, %g2
    st  %g2, [ %i1 ]

end of loop, increment 'incoming' and 'destination',
decrement loop count, test for end and branch
    dec  %g3
    add  %i0, 4, %i0
    add  %i1, 4, %i1
    cmp  %g3, 0
    bg,a   0x185c70
    clr  %g1
reload original 'incoming' and 'destination' pointers
    ld  [ %sp + 0x160 ], %i0
    ld  [ %sp + 0x164 ], %i1

end-of-procedure bookkeeping
    ret
    restore
```

Figure 7: A sample DCG conversion routine.

For the reader desiring more information on the precise nature of the code that is generated, we include a small sample subroutine in Figure 7. This particular conversion subroutine converts message data received from an x86 machine into native Sparc data. The message being exchange has a relatively simple structure:

```
typedef struct small_record {
    int ivalue;
    double dvalue;
    int iarray[5];
};
```

Since the record is being sent from an x86 and PBIO always sends data in the sender's native data formats and layout, the "wire" and native formats differ in both byte order and alignment. In particular, the floating point value is aligned on a 4-byte boundary in the x86 format and on an 8-byte boundary on the Sparc. The subroutine takes two arguments. The first argument in register `%i0` is a pointer to the incoming "wire format" record. The second argument in register `%i1` is a pointer to the desired destination, where the converted record is to be written in native Sparc format.

The exact details of the code are interesting for a couple of points. First, we make use of the SparcV9 Load from Alternate Space instructions which can perform byteswapping in hardware during the fetch from memory. This yields a significant savings over byteswapping with register shifts and masks. Since this is not an instruction that is normally generated by compilers in any situation, being able to use it directly in this situation is one of the advantages of dynamic code generation.

Second, from an optimization point of view, the generated code is actually quite poor. Among other things, it performs two instructions when one would obviously suffice, and unnecessarily generates an extra load/store pair to get the double value into a float register. The are several reasons for this suboptimal code generation, including the generic nature of the virtual RISC instruction set offered by Vcode, the lack of an optimizer to repair it, and the fact that we have not seriously attempted to make the code generation better. Even when generating poor code, DCG conversions are a significant improvement over other approaches.

Examining the generated code may also bring to mind another lurking subtlety in generating conversion routines: data alignment. The alignment of fields in the incoming record reflects the restrictions of the sender. If the receiver has more stringent re-

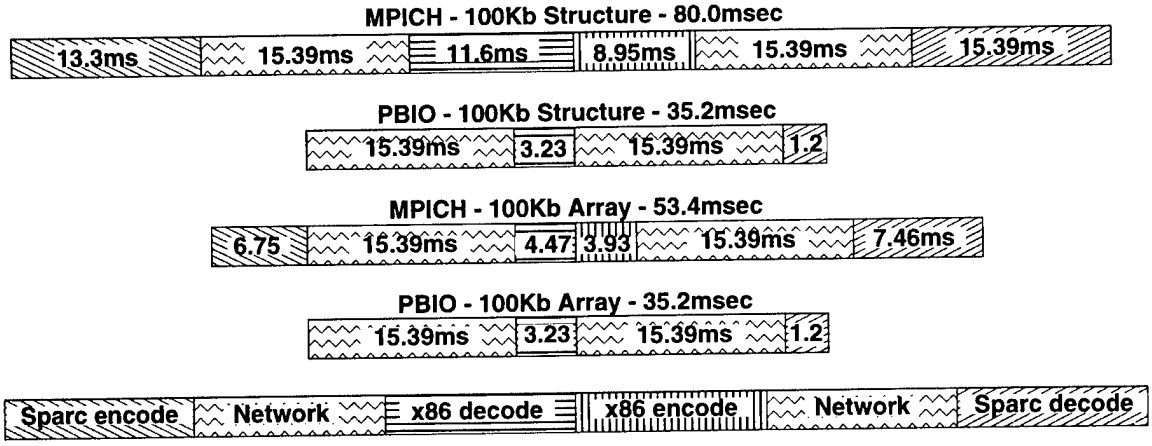


Figure 8: Comparison between PBIO and MPICH in structure and array exchange time.

	MPICH			PBIO		
data size	total time	send side overhead	receive side overhead	total time	send side overhead	receive side overhead
100Kb	20.8ms	0.46	0.78	18.3	0.0028	0.034
10Kb	3.02ms	0.083	0.20	2.52	0.0028	0.034
1Kb	1.06ms	0.0097	0.086	0.90	0.0028	0.034
100b	.63ms	0.0056	0.076	0.52	0.0028	0.034

Table 1: A comparison of PBIO and MPICH for homogeneous exchange of arrays

strictions, the generated load instruction may end up referencing a misaligned address, a fatal error on many architectures. This situation would actually have occurred in in the example shown in Figure 7, where the incoming `double` array is aligned on a 4 byte boundary because the Sparc requires 8 byte alignment for 8-byte loads. Fortunately, the suboptimal Sparc dynamic code generator loads the two halves of the incoming 8-byte doubles with separate `ldswa` instructions instead of a single `lddfa` instruction.

Data alignment is generally not an issue in storing to the native record because it is presumably aligned according to the requirements of the receiving machine. We also assume that the base addresses of the incoming and native records are strongly aligned. This leaves the offsets of the incoming record fields as the primary source of misalignment. Since these are known at code generation time, we can make static decisions about using efficient direct loads for aligned data or using potentially less efficient methods for unaligned data.[3]

[3] Our current code generator does not handle misaligned accesses, but the extension to handle them is straightforward.

3.5 Other data types and homogeneous systems

The previous sections compared PBIO's performance with that of MPICH in situations involving a heterogeneous exchange of structures containing mixed types. While PBIO shows clear and significant performance gains over MPICH in that situation, MPICH might be expected to perform better in dealing with messages consisting of contiguous arrays, or in a homogeneous exchange where it might not use an XDR-based encoding scheme.

Figure 8 shows a breakdown of MPICH and PBIO performance for heterogeneous transmission of a 100Kb floating point array and compares it to the previously presented breakdowns for the 100Kb structure. This figure shows that PBIO's performance remains essentially unchanged when the datatype is changed from structure to an array. MPICH performance does improve with contiguous arrays, but not to the point where it matches PBIO's performance. The results for smaller datatypes are similar.

A comparison of round-trip times for contiguous arrays between homogenous machines is shown in Table 1. This is one of the simplest cases in binary

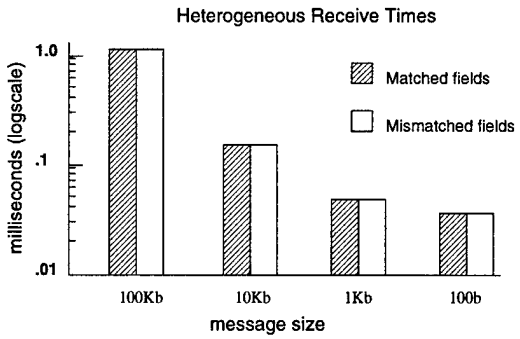


Figure 9: Receiver-side decoding costs with and without an unexpected field – heterogeneous case.

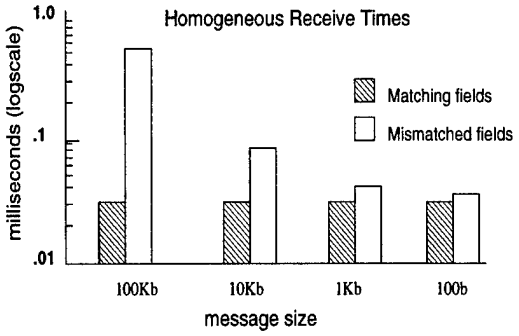


Figure 10: Receiver-side decoding costs with and without an unexpected field – homogeneous case.

communication, requiring no data conversion of any kind. The send and receive side overheads are tiny compared to the time required for network transmission, but PBIO retains a slight edge over MPICH in both receive and send side overheads. These differences largely account for the 10% or so better performance that PBIO achieves in round-trip time for these contiguous arrays.

That PBIO has better performance than MPICH even in situations where MPICH might be expected to prevail is convincing evidence that PBIO's extra flexibility in supporting application evolution does not negatively impact performance in other situations. The next section will examine PBIO's performance in the presence of application evolution.

3.6 Performance in application evolution

The principal difference between PBIO and most other messaging middleware is that PBIO messages carry format meta-information, somewhat like an XML-style description of the message content. This meta-information can be an incredibly useful tool in building and deploying enterprise-level distributed systems because it 1) allows generic components to operate upon data about which they have no *a priori* knowledge, and 2) allows the evolution and extension of the basic message formats used by an application without requiring simultaneous upgrades to all application components. In other terms, PBIO allows *reflection* and *type extension*. Both of these are valuable features commonly associated with object systems.

PBIO supports reflection by allowing message formats to be inspected before the message is received. It's support of type extension derives from doing field matching between incoming and expected records by name. Because of this, new fields can be added to messages without disruption because application components which don't expect the new fields will simply ignore them.

Most systems which support reflection and type extension in messaging, such as systems which use XML as a wire format or which marshal objects as messages, suffer prohibitively poor performance compared to systems such as MPI which have no such support. Therefore, it is interesting to examine the effect of exploiting these features upon PBIO performance. In particular, we measure the performance effect of type extension by introducing an unexpected field into the incoming message and measuring the change in receiver-side processing.

Figures 9 and 10 present receive-side processing costs for an exchange of data with an unexpected field. These figures show values measured on the Sparc side of heterogeneous and homogeneous exchanges, respectively, using PBIO's dynamic code generation facilities to create conversion routines. It's clear from Figure 9 that the extra field has no effect upon the receive-side performance. Transmitting would have added slightly to the network transmission time, but otherwise the support of type extension adds no cost to this exchange.

Figure 10 shows the effect of the presence of an unexpected field in the homogeneous case. Here, the overhead is potentially significant because the homogeneous case normally imposes no conversion overhead in PBIO. The presence of the unexpected field creates a layout mismatch between the wire and native record formats and as a result the conversion routine must relocate the fields. As the figure shows, the resulting overhead is non-negligible, but not as high as exists in the heterogeneous case. For smaller record sizes, most of the cost of receiving data is ac-

tually caused by the overhead of the kernel `select()` call. The difference between the overheads for matching and extra field cases is roughly comparable to the cost of `memcpy()` operation for the same amount of data.

The results shown in Figure 10 are actually based upon a worst-case assumption, where an unexpected field appears before all expected fields in the record, causing field offset mismatches in all expected fields. In general, the overhead imposed by a mismatch varies proportionally with the extent of the mismatch. An evolving application might exploit this feature of PBIO by adding any additional at the end of existing record formats. This would minimize the overhead caused to application components which have not been updated.

4 Conclusions

Current distributed applications rely heavily on leveraging the computing power of heterogeneous networks of computer architectures. The PBIO library is a valuable addition to the mechanisms available for handling binary data interchange among these heterogeneous distributed systems. PBIO performs efficient data translations, and supports simple, transparent system evolution of distributed applications, both on a software and a hardware basis.

Rather than relegating message packing and unpacking operations to the communicating applications, thus requiring *a priori* agreement on these data structures, PBIO efficiently layers and abstracts diversities in computer architectures. Applications need only agree on data by *name*, and previously exposed concerns such as byte ordering, architecture specifications, data type sizes, and compiler differences are no longer a concern. Since PBIO uses dynamic code generation rather than data interpretation, compiler optimizations are utilized without the cumbersome limitations of static data structures.

Enterprise-scale distributed computing can be implemented and deployed much more simply and efficiently using PBIO's flexibility, not only initially, but during the evolution of specific distributed components. Data elements can be incrementally to the basic message formats of distributed applications without disrupting the operation of existing application components.

The measurements in this paper have shown that PBIO's flexibility does not impact its performance. In fact, PBIO's performance is better than that of a popular MPI implementation in every test case, and significantly better in heterogeneous exchanges. Performance gains of up to 60% are largely due to:

- virtually eliminating the sender-side encoding cost by transmitting in the sender's native format, *and*
- using dynamic code generation to perform data conversion on the receiving side.

In short, PBIO is a novel messaging middleware that combines significant flexibility improvements with an efficient implementation to offer distributed applications fast heterogeneous binary data interchange.

References

[1] Guy T. Almes. The impact of language and system on remote procedure call design. In *Sixth International Conference on Distributed Computing Systems, Boston, Mass.*, pages 414–421. IEEE, May 1986.

[2] D.D.Clark and D.L.Tennenhouse. Architectural considerations for a new generation of protocols. In *Proceedings of the SIGCOMM '90 Synposium*, pages 200–208, Sept 1990.

[3] Greg Eisenhauer. Portable self-describing binary data streams. Technical Report GIT-CC-94-45, College of Computing, Georgia Institute of Technology, 1994. *(anon. ftp from ftp.cc.gatech.edu).*

[4] Greg Eisenhauer, Beth Schroeder, and Karsten Schwan. Dataexchange: High performance communication in distributed laboratories. *Journal of Parallel Computing*, 24(12-13), 1998.

[5] Dawson R. Engler. Vcode: a retargetable, extensible, very fast dynamic code generation system. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation (PLDI '96)*, May 1996.

[6] Message Passing Interface (MPI) Forum. MPI: A message passing interface standard. Technical report, University of Tennessee, 1995.

[7] I. Foster, C. Kesselman, and S. Tuecke. The nexus approach to integrating multithreading and communication. *Journal of Parallel and Distributed Computing*, pages 70–82, 1996.

[8] Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, and Vaidy Sunderam. *PVM 3 Users Guide and Reference manual.* Oak Ridge National Laboratory, Oak Ridge, Tennessee 37831, May 94.

[9] Hewlet-Packard. The netperf network performance benchmark. http://www.netperf.org.

[10] Argonne National Laboratory. Mpich-a portable implementation of mpi. http://www-unix.mcs.anl.gov/mpi/mpich.

[11] Mario Lauria, Scott Pakin, and Andrew A. Chien. Efficient layering for high speed communication: Fast messages 2.x. In *Proceedings of the 7th High Performance Distributed Computing (HPDC7)*, July 1998.

[12] S. W. O'Malley, T. A. Proebsting, and A. B. Montz. Universal stub compiler. In *Proceedings of the SIGCOMM '94 Symposium*, Aug 1994.

[13] Marcel-Catalin Rosu, Karsten Schwan, and Richard Fujimoto. Supporting parallel applications on clusters of workstations: The virtual communication machine-based architecture. *Cluster Computing, Special Issue on High Performance Distributed Computing*, 1, January 1998.

[14] M. Schroeder and M. Burrows. Performance or firefly rpc. In *Twelfth ACM Symposium on Operating Systems, SIGOPS, 23, 5*, pages 83–90. ACM, SIGOPS, Dec. 1989.

Greg Eisenhauer is a research scientist in the College of Computing at the Georgia Institute of Technology. He received his PhD from Georgia Tech in 1998 under the direction of Dr. Karsten Schwan. Dr. Eisenhauer previously worked at Honeywell's Systems and Research Center and received his BS and MS degrees from the University of Illinois, Champaign-Urbana. His research interests include interactive computational steering, performance evaluation and scientific computing.

Lynn K. Daley is a PhD student in the College of Computing at Georgia Institute of Technology, working under the direction of Dr. Karsten Schwan. Ms. Daley has over 20 years software development experience, having worked at Harris Government Systems, Digital Equipment Corp., and Atlanta Signal Processors. She holds BS/CS and MS/EE degrees from Georgia Institute of Tech, and a MS/EngMgmt from Florida Institute of Technology. Her research interests include parallel, distributed, and real-time processing.

A Heuristic Algorithm for Mapping Communicating Tasks on Heterogeneous Resources

Kenjiro Taura*
Computer Systems and Architecture Group
Computer Science and Engineering
University of California, San Diego
tau@csag.ucsd.edu

Andrew Chien
Computer Systems and Architecture Group
Computer Science and Engineering
University of California, San Diego
achien@csag.ucsd.edu

Abstract

*A heuristic algorithm that maps data-processing tasks onto heterogeneous resources (*i.e., *processors and links of various capacities) is presented. The algorithm tries to achieve a good* throughput *of the whole data-processing pipeline, taking both parallelism (load balance) and communication volume (locality) into account. It performs well both under compute-intensive and communication-intensive conditions. When all tasks/processors are of the same size and communication is negligible, it quickly distributes the compute load over processors and finds the optimal mapping. As communication becomes significant and reveals as a bottleneck, it trades parallelism for reduction of communication traffic. Experimental results using a topology generator that models the Internet show that it performs significantly better than communication-ignorant schedulers.*

1. Introduction

It is widely believed that future computing environment will consist of geographically distributed compute- and data-resources connected with diverse communication capacities, forming a so-called "computational Grid" environment [10]. Computational elements range from a desktop to clusters [4, 5] to supercomputers, and links range from phone lines to gigabits system area networks. Both CPU capacity and the network connectivity are improving in a rapid pace, but the recent trend indicates network bandwidth increases more rapidly than CPUs. As a consequence, communication-intensive parallel jobs, which we are currently able to run only on dedicated supercomputers or clusters, are likely to be hosted by a collection of desktops in laboratories or even home. This brings the Grid beyond just an aggregation of computational horsepower and enables a qualitatively different use of it. On the other hand, it presents significant resource management problems to all levels of parallel/distributed software developments.

One of the fundamental elements of such resource management problems is, given an application that consists of many communicating tasks, to select a suitable set of resources and map its tasks appropriately. To obtain a robust performance across a wide range of resource configurations, mapping algorithms must trade load balancing for the reduction of communication, and vice versa.

In this paper, we present a graph-theoretic formulation of this general problem and propose its heuristic algorithm. The algorithm takes as input a *task graph* and a *resource graph* and outputs the mapping from tasks to processors. A task graph models a data processing pipeline; a task in a pipeline continuously receives data from adjacent tasks, processes them, and sends processed data to other tasks. Weights of nodes and edges represent compute and communication *requirements* of these tasks, respectively. A resource graph models processors and links. Weights of nodes and edges represent their compute and communication *capacities*, respectively. If too many tasks are assigned on a processor or too much communication goes through a link, the processor or the link becomes a bottleneck and determines the overall throughput of the entire pipeline.

The key to achieving a good throughput is *clustering* of a task graph, a process which recognizes highly-connected components in a task graph. A cluster in a task graph represents a set of tasks that are intensively communicating with each other. These tasks should be placed in a single processor if available communication bandwidth is low. Among several graph clustering methods proposed in the literature [9, 13, 28], we use a simplified version of the stochastic flow injection method [29, 30] proposed by by Yeh et al.

Under a simple condition in which tasks and proces-

*Also affiliated to Departement of Information Science, University of Tokyo.

0-7695-0556-2/00 $10.00 © 2000 IEEE

sors are of a uniform weight and communication is negligible, it guarantees to quickly give the optimal solution, in which tasks are uniformly distributed over processors. As communication becomes significant and reveals as a bottleneck, it co-locates highly communicating tasks to reduce communication traffic. We have implemented the algorithm in scripting language Python [16] and performed experiments using a simplified version of an Internet topology generator [7, 12] to generate a realistic resource graph. As we expected, our algorithm significantly outperforms simpler, communication-ignorant algorithms on communication-intensive conditions.

The rest of the paper is organized as follows. Section 2 gives a practical motivating scenario that we envision will commonly occur in emerging Grid applications. Section 3 is devoted to the problem formulation and Section 4 describes its algorithm. Section 5 shows experimental results. Section 6 mentions relationship to other work and Section 7 summarizes the paper and states future work.

2. A Practical Scenario

Consider an application which reads a large volume of data from geographically distributed source (storage server), processes them, and displays the result on a desktop. An example of such application is SARA [22], in which the data is surface data of the earth. Emerging distributed applications that use geographically distributed data such as digital libraries [1] and scientific data archives [6] will have more or less this kind of structure.

Even in this fairly simple setting, one question that arises is where the data should be processed. The best decision clearly depends on how computationally expensive the processing is, how much data it reads from the source and writes to the display, how computationally powerful are the desktop and the storage server, and how much bandwidth we have between these nodes. The decision is much more complex when we have a more involved data processing pipeline and more available resources such as parallel compute-servers. Finally, the availability of all these resources changes over time. For example, processing should be done on the desktop when the storage server is highly loaded.

It can easily be seen that it is, if not impossible, difficult and time-consuming for individual application developers to implement a decision that works in a wide range of resource configurations, even in a very simple case like this. Application-specific solutions, if any, would not generalize to even more complex and dynamic cases, in which we have hundreds of tasks that are created and ceased over time.

3. Problem Description

3.1. Preliminary Definitions and Notations

Resource Graph and Task Graph: A *resource graph* is a weighted graph (both nodes and edges are weighted).[1] A node of a resource graph represents a processor and an edge a link between a pair of processors. The weight of a node represents the processor's compute capacity (the amount of computation that can be performed in a unit time) and that of an edge the link's communication capacity (the amount of data that can go through the link in a unit time).

A *task graph* is also a weighted graph. A node of a task graph represents a task and an edge a continuous communication (stream) between a pair of tasks. The weight of a node represents the task's compute requirement (the amount of computation that must be done for this task to make a unit progress) and that of an edge the communication requirement of the connected tasks (the amount of data that must be communicated for these tasks to make a unit progress).

Note that a task graph is *not* a traditional dependence graph, in which an edge $s \rightarrow t$ represents the fact that task t can start its computation only after s has finished. Rather, our task graph models a data processing *pipeline*, in which all tasks continuously receive pieces of data, process them, and then send the processed data. A typical example is a multimedia data processing pipeline such as Smart Kiosk [21, 20], in which the natural unit of work is a frame. Typical tasks include compression, decompression, color tracking, object detection, and so on. A weight of a node is the amount of computation performed by the task per single frame, whereas that of an edge the size of transferred data per frame.

Unlike other formulations [14, 26], our model does not have an explicit notion of *parallelized tasks*. That is, a single node of a task graph can be mapped only on a single node of a resource graph. A parallelized task can be to some extent modeled by many nodes that together represent a single logical task.

Notations: Let G be a weighted graph. G_i is the weight of node i and $G_{i,j}$ the weight of edge $i \rightarrow j$. G^i is the weighted graph isomorphic to G, in which the weight of node i is one and that of all other nodes/edges is zero. $G^{i,j}$ is the weighted graph isomorphic to G, in which the weight of the edges along the path from i to j is one and that of all other nodes/edges is zero (Figure 1). If there are multiple paths between a pair of nodes, we fix one such path.

Let G and H be isomorphic weighted graphs. We define $G + H$ as node- and edge-wise addition of their weights. We similarly define $G - H$ and G/H. Let k be a scalar, kG

[1] Graphs can either be directed or undirected, but the following discussion assumes directed graphs.

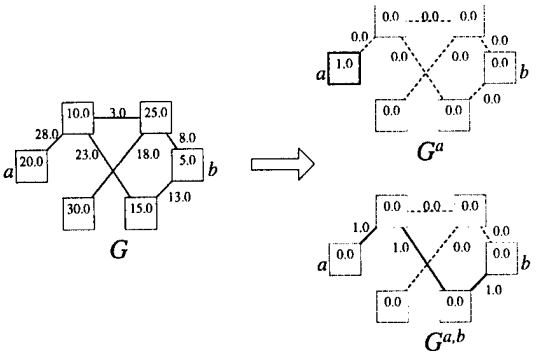


Figure 1. A weighted graph G, G^i, and $G^{i,j}$.

denotes a graph isomorphic to G whose weights are multiplied by k.

Interpretation: As we mentioned earlier, a task graph models a set of tasks each of which repeatedly receives data from other tasks, performs some computation on them to produce other data, and sends the produced data to other tasks. We make it more precise by showing the pseudo code for a task t in a task graph $G = \langle V, E \rangle$, as shown in Figure 2.

The progress rate of task t is determined by several factors. First, t will experience a certain amount of wait time at the **wait** phase, if tasks that are sending data to t cannot produce data fast enough or the bandwidth from these tasks to t are not enough. Second, more obviously, this task will spend some time at the **compute** step. Finally, the time taken at the **send** step will be determined by outgoing bandwidth and how fast receiving tasks can consume data.

As will be made clear in the next section, our problem formulation effectively makes idealizing assumptions that the progress rate of this task is determined by the maximum, rather than the summation, of these three factors. For example, if the wait step in isolation takes 5 time units, the compute step 3 time units, and the send step 2, then `unit_progress` as a whole takes only 5 time units, rather than $5 + 3 + 2 = 10$. This approximates a situation in which these three phases interleave in the infinitely fine-grained manner; that is, **compute** phase begins processing data when a single bit of data appears in the incoming stream, and the **send** phase sends data as soon as produced.

3.2. Formulation

We are interested in the throughput (the number of work units completed per unit time) of the system in equilibrium. Given a mapping from tasks to processors, it determines the amount of computation each processor must perform to

```
/* G = ⟨V, E⟩.
   a unit work task t repeats. */
unit_pregress(t)
{
  /* (1) wait */
  for s ∈ V s.t. s ↪ t ∈ E {
    wait for G_{s,t} units (e.g., bytes) of data
      to arrive from s;
  }
  /* (2) compute */
  perform G_t units of computation upon the
    received data;
  /* (3) send */
  for u ∈ V s.t. t → u ∈ E {
    send G_{t,u} bytes of data to u;
  }
}

/* a task t simply repeats unit_progress forever */
task(t)
{
  while (1) {
    unit_progress(t);
  }
}
```

Figure 2. Pseudo code for task t.

make all tasks complete a unit work; it is simply the summation of task weights mapped on the processor in question. It similarly determines the volume of data each link must transfer to have all tasks make a unit-progress. By dividing the requirement at each node (edge) by its corresponding computation (communication) capacity, we have the time required to service requested computation/communication at the node (edge). We call it *occupancy* at the node (edge). The maximum occupancy over the entire graph gives us the time required to unit-progress all tasks. The goal is to make the maximum occupancy of the mapping as small as possible. Note that an occupancy is the inverse of the number of unit works finished per a unit time. Thus, minimizing the occupancy is equivalent to maximizing the throughput.

A more formal description follows. Let $G = \langle V_G, E_G \rangle$ be a task graph and $P = \langle V_P, E_P \rangle$ a processor graph. Let m be a mapping from V_G to V_P. We define the *load graph* of the mapping, denoted by $L(G, P, m)$, as:

$$L(G,P,m) = \sum_{t \in V} G_t P^{m(t)} + \sum_{(s,t) \in E} G_{s,t} P^{m(s),m(t)}$$

That is, a load graph is a graph whose weights represent the amount of computation and communication required (at each node and edge) to unit-progress all tasks.

Occupancy graph of the mapping, denoted by $O(G, P, m)$, is obtained by simply dividing the load by the capacity at each node and edge:

$$O(G,P,m) = L(G,P,m)/P$$

The goal is to find a mapping m that minimizes $\mathtt{max}(O(G, P, m))$, where $\mathtt{max}(X)$ is the maximum weight over nodes and edges in graph X. Figure 3 shows an example of a load graph and an occupancy graph.

Note that the above formulation effectively assumes that all tasks progress in the same pace; when any of the tasks takes x unit time to make a unit progress, all the other tasks also take x. In other words, resources are never used to make some tasks go faster than the others. This is a practical assumption because, assuming finite communication buffers, any pair of communicating tasks must progress in the same pace in equilibrium. Consequently, for connected task graphs, tasks must eventually match their paces with all the other tasks.

Finally, we state that this problem is NP-hard. We show that the corresponding decision problem TASKMAP, which asks if a mapping whose maximum occupancy is no greater than a specified limit exists, is NP-hard. There are several NP-hard problems that straightforwardly reduce to TASKMAP. Reducing Knapsack problem is particularly simple; we however use a reduction from the two-way graph partitioning problem which is also NP-hard [19], because we believe it better illustrates the difficulty of the problem

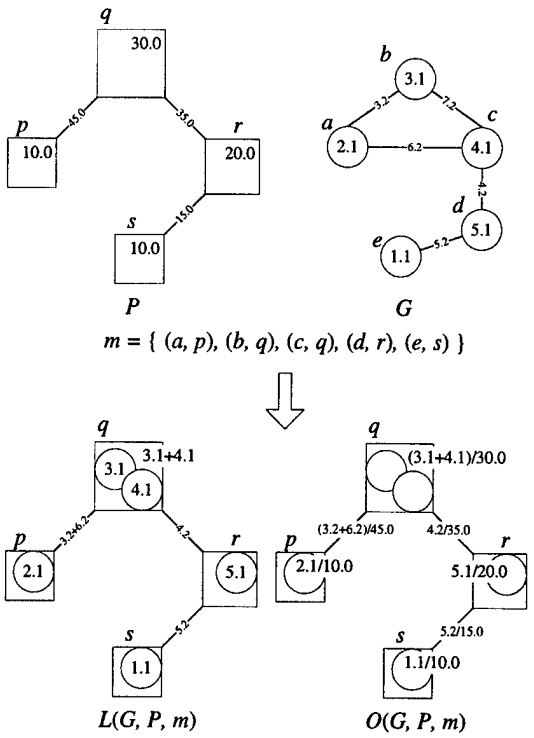


Figure 3. Load graph and occupancy graph.

(in particular it also shows the problem remains NP-hard even if we restrict all tasks to be the same size). The graph partitioning problem, PARTITION, takes an (unweighted) graph $G = \langle V, E \rangle$ and an integer c as input, and asks if there is a partition $V = V_1 + V_2$, such that V_1 and V_2 are disjoint and equal size (*i.e.*, $V_1 \cap V_2 = \emptyset$ and $|V_1| = |V_2| = |V|/2$) and the number of edges between V_1 and V_2 is $\leq c$.

Theorem 1 *TASKMAP is NP-hard.*

Proof: For a given instance of PARTITION $G = \langle V, E \rangle$ and c, we construct an instance of TASKMAP as follows.

- The task graph is a graph isomorphic to G, whose node weights and edge weights are all ones.
- The resource graph is a graph of two nodes, whose weights are both $|V|/2$, and the weight of the edge between the two is c.
- The maximum occupancy is one. That is, we ask if there is a mapping whose maximum occupancy is no greater than 1.

It is easily seen that if and only if there is such a mapping, there is a solution for the original graph partitioning problem, and the reduction can be performed in a polynomial time (Q.E.D).

4. The Algorithm

4.1. Motivating Example

If tasks are very compute-bound (communication is almost negligible), mapping is relatively straightforward, at least when task sizes are fairly uniform. It simply amounts to assigning each processor task weights roughly proportion to its compute capacity. Our main contribution is on cases where tasks are more communication intensive, thus such communication-ignorant mappings result in excess traffic that limits the performance. With increasing communication intensity of tasks, it becomes likely that mapping tasks that intensively communicate with each other on the same processor results in a significantly better performance.

As is the case in most combinatorial problems, the fundamental difficulty in achieving such mappings lies in the fact that the performance as a function of mappings is quite discontinuous and there are many local optima; the desired mapping is quite different from one communication intensity to another, and mappings that are in some sense 'between' these desired mappings are typically worse than both. Therefore it is difficult to move from one desired mapping to another by a series of greedy moves. To illustrate this, consider a task graph shown in Figure 4 where all nodes weigh one and all edges weigh c (a parameter). When c is very small, the desired mapping will typically be the one in which a single processor has a single task (assuming sufficient number of equally powerful processors). As c increases up to a certain threshold, the best mapping will typically become the one in which a single processor is assigned to a single cluster of tasks (as easily perceived by humans). Everything between these two extremes (for example, mappings in which a single processor has two tasks) are typically *worse than both*. This is because, when compared to the first extreme (one task per processor), the amount of traffic a single processor sends or receives increases, thus it does not reduce the communication bottleneck. The communication bottleneck can be eliminated only by moving *all* tasks of a cluster to a single processor. This property prohibits the use of a simple local search strategy which tries to find a task t and a processor p such that moving t to p improves the objective function. It is quite unlikely that a series of such moves eventually reaches the desired extreme, whichever is the better.

4.2. Overall Structure

As is easily seen from the example just discussed, the key to achieving a good mapping is to recognize highly-connected clusters, and use this clustering information to guide the mapping process. Our basic approach is to linearly order tasks in such a way that tasks within a cluster are close to each other, and put tasks to processors according to this order (as indicated by the labels in the figure). If a single processor is assigned to multiple tasks, they are likely to be in the same cluster, and therefore, when the tasks turn out to be communication-bound, processors can reduce communication simply by accommodating more tasks from the list.

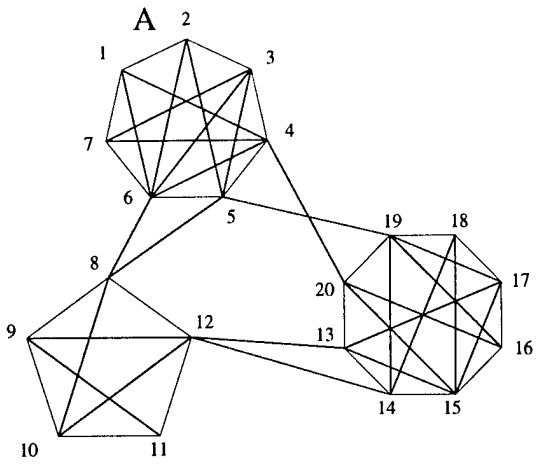


Figure 4. A graph with highly-connected subgraphs.

To continue the above example, we first pick up a processor and move tasks to it from the list. As tasks are ordered as shown in the figure, we exclusively choose tasks from a cluster (labeled A) in the beginning. The remaining problem is when we should stop this process and go onto the next processor. The best answer again depends on communication intensity; when c is small, it is typically when the compute-load is best balanced among processors, and otherwise when one or more clusters have just moved. Details are given in Section 4.4.

Our entire algorithm first obtains the appropriate order of tasks based on a simplified version of stochastic flow injection method proposed in [29, 30]. Given this information, it obtains an initial mapping and then improves it step by step. The elementary procedure mentioned above is used both to obtain the initial mapping and to improve it. The top-level structure of the algorithm is illustrated in Figure 5.

In the following sections, we first describe the clustering algorithm to obtain the order of tasks in Section 4.3, the elementary procedure that moves tasks to a processor from the list in Section 4.4, and how to improve the mapping once obtained in Section 4.5. Throughout the sections, $G = \langle V_G, E_G \rangle$ and $P = \langle V_P, E_P \rangle$ refer to the given task graph and the resource graph, respectively. As a convention, we do not update data structures in place (we always rebind a variable to signify an update). Variables assigned in one iteration of a loop and used in the next is subscripted

```
/* G = ⟨V_G, E_G⟩ : task graph.
   P = ⟨V_P, E_P⟩ : resource graph. */
taskmap()
{
  <_G = clustering(G); — (section 4.3)
  m = { }; /* empty map */
  m = map_tasks(m); — (section 4.4)
  repeat {
    m' = m;
    m = improve(m'); — (section 4.5)
  } while (O(G,P,m) < O(G,P,m'))
}
```

Figure 5. The overall structure of the algorithm.

```
1:  clustering(H)
    {
      T =recursive_clustering(H);
      <_H= depth-first traversal order of T;
5:    return <_H;
    }

    recursive_clustering(H)
      H = (V,E) /* a subgraph of the task graph */
10: {
      if (V is singleton (= {v})) {
        return leaf(v)
      } else {
        H_1,···,H_n =clusters obtained by
15:       stochastic flow injection (see text);
        return node(recursive_clustering(H_1),
                    ···, recursive_clustering(H_n));
      }
    }
```

Figure 6. Clustering Task Graphs.

by a loop index, even though it is a single variable in the real program.

Finally, we made various simplifications for the purpose of presentation. For example, the following algorithm calculates $L(G,P,m)$ many times, with m's that only slightly differ from each other. The actual program keeps track of $L(G,P,m)$ all the time and incrementally updates it as m changes. This kind of practical optimizations are not explicit in the description.

4.3. Clustering Task Graph

The clustering algorithm is shown in Figure 6. Given a graph H, it first creates a tree that hierarchically decompose the task graph into clusters (line 3). The root of the tree represents the entire set of nodes, whereas a leaf a singleton set of a node. Children of a node are partitions of the parent node, obtained by a simplified stochastic flow injection method as described below. Once such a tree is obtained, we determine a total order between nodes, $<_H$, simply by traversing the tree in a depth-first order (line 4).

The stochastic flow injection was originally proposed for VLSI circuit partitioning and works as follows:

1. Randomly pick up two nodes s and t of the given graph G.
2. Find the shortest path between s and t.
3. Decrement the weights of all the edges on the path by a (small) constant Δ (*i.e.*, inject a flow Δ between s and t).
4. Remove edges whose weight become zero or negative.
5. Repeat 1-4 until the graph becomes unconnected.
6. When graphs are disconnected, each connected component is a cluster.

The intuition is that if only a small number of edges bridge two (or more) large clusters, such edges are likely to be decremented often, and the graph soon becomes disconnected by these edges.

In the original stochastic flow injection method, another phase follows to merge some of the clusters hereby obtained, but we simply skip this phase, because our purpose is to recursively decompose clusters until each cluster becomes a singleton. We also slightly modified the above step 1, so that a task is chosen by a probability proportional to its weight; this is necessary because the original stochastic flow injection method assumes uniform weights (as in the case in their application).

4.4. The Elementary Move

Procedure `map_tasks` shown in Figure 7 takes as a parameter m, a partial mapping from tasks to processors (it is partial because some tasks are not mapped). It maps tasks not mapped in m onto V_P, by simply making a series of calls to a more elementary procedure `map_tasks_on`, which maps some tasks to a specified processor.

The procedure `map_tasks_on` takes three parameters, m, q, and Q; m is a partial mapping from tasks to pro-

```
1: map_tasks(m)
   {
     Q = V_P;
     while (Q ≠ {}) {
       q = a processor ∈ Q;
5:     Q = Q − { q };
       m = map_tasks_on(m, q, Q);
     }
     return m;
   }
10: /* move some of the tasks not mapped in m to
       processor q, taking open communication and
     the balance between { q } and Q into account */
   map_tasks_on(m, q, Q)
   {
15:  U_{m_0} = { t | not mapped in m };
     for i = 1, ···, |U_{m_0}| {
       t = the minimum task ∈ U_{m_{i-1}} w.r.t. <_G;
       m_i = m_{i-1}[t/q]; /* add mapping t → q */
       U_{m_i} = U_{m_{i-1}} − { t };
20:    O = O(G, P, m_i);
       O_comp = O_comp(G, P, m_i, Q);
       /* O_comp = ∞ if Q = {} */
       O_→ = O_→(G, P, m_i, U_{m_i}, q);
       O_← = O_←(G, P, m_i, q, U_{m_i});
25:    M_i = max(O, O_comp, O_→, O_←);
     }
     find i that gave minimum M_i (i = 1, ···, |U_{m_0}|);
       break ties by selecting largest i.
     return m_i;
30: }
```

Figure 7. The elementary move operation.

cessors, q a processor $\in V_P$ onto which some tasks are going to be mapped by the procedure, and Q a subset of V_P ($q \notin Q$) yet unused. The goal is to put an appropriate number of tasks on q, so that we are likely to reach a good final mapping, if the remaining tasks are mapped on Q. As mentioned earlier, it puts tasks one after another in the order obtained by the clustering; as we add more tasks to q, we obtain a series of mappings $m_0 = m, m_1 = m_0[t_1/q], m_2 = m_1[t_2/q], \cdots, m_n = m_{n-1}[t_n/q]$,[2] where $t_1 <_G t_2 <_G, \cdots, <_G t_n$ and m_n is the total mapping from V_G to V_P. So the only question is which m_i we should choose.

Let U_m denote the set of tasks that are not mapped in m. At each step, we keep track of the following four (three in case of undirected graphs) values to evaluate the situation.

- (Line 20): The current occupancy $O(G, P, m_i)$.
- (Line 21): A hypothetic occupancy O_{comp}. $O_{\text{comp}}(G, P, m_i, Q)$ is an occupancy estimated by assuming that tasks $\in U_{m_i}$ are perfectly mapped on Q, ignoring communication. That is, it is simply the total compute requirement of these tasks over the total compute capacity of Q:

$$O_{\text{comp}}(G, P, m, Q) = \frac{\sum_{t \in U_m} G_t}{\sum_{p \in Q} P_p},$$

  For convenience we define this to be ∞ when $Q = \{\}$.
- (Lines 23 and 24): Hypothetic occupancies $O_{\rightarrow}(G, P, m_i, U_{m_i}, q)$ and $O_{\leftarrow}(G, P, m_i, q, U_{m_i})$, which we call occupancies induced by *open communication*. Given a set of tasks T and a processor q, we define open communication from T to q (from q to T) to be the total communication volume from tasks in T to tasks on q (from tasks on q to tasks in T). $O_{\rightarrow}(G, P, m, T, q)$ refers to open communication from T to q divided by the total edge capacity adjacent to q. Similarly for $O_{\leftarrow}$. That is:

$$O_{\rightarrow}(G, P, m, T, q) = \frac{\sum_{x \in T, m(y)=q} G_{x,y}}{\sum_{(p,q) \in E_P} P_{p,q}}, \text{ and}$$

$$O_{\leftarrow}(G, P, m, q, T) = \frac{\sum_{x \in T, m(y)=q} G_{y,x}}{\sum_{(p,q) \in E_P} P_{q,p}}.$$

  When graphs are undirected, these two give the same value and are collectively referred to as $O_{\leftrightarrow}$.

At each step, we calculate the above four (or three in undirected case) values and record the *maximum* of them (M_i at line 25). The procedure returns m_i that minimizes M_i (lines 27-29).

[2] $m' = m[t/q]$ is an extension of m, s.t. $m'(t) = q$ and $m'(x) = m(x)$ for $x \neq t$.

The first item will be intuitive. The second one, O_{comp}, tries to estimate how much is the final occupancy going to be. Given this estimate, we determine how many tasks should be accommodated to the current processor q. For example, suppose compute capacity of q is 1, the total compute capacity of Q 99, and the total compute requirement of tasks yet to be mapped 1,000. Ideally, we like to obtain a mapping whose maximum occupancy is close to 1,000/(1 + 99) = 10. Put differently, when we compare a series of mappings $m_1, m_2, \cdots$, *any mapping whose occupancy is below 10 is equally good*; there is no points in quitting at m_i, when the occupancy of m_{i+1} is still below 10.

The third item, $O_{\rightarrow}$ ($O_{\leftarrow}$) or, *open communication metric* is to identify m_i at which the communication volume between tasks already mapped on q and those that are not is small. Keeping track of such communication is necessary because it is not taken into account by $O(G, P, m_i)$, which only counts tasks mapped in m_i. This guides the mapping process, by giving following pieces of information: "rather than choosing an m_5 at which open communication is so large, accommodate more tasks and choose m_8, at which the processor is more loaded, but communication traffic is much smaller." Accurate estimation clearly requires not only communication volume, but also the link bandwidth from q to processors that accommodate the other tasks. An obvious problem is we are yet to know how remaining tasks will be mapped, so we do not precisely know how much will the occupancy of these links be. We simply estimate this by: (1) calculating the total communication volume between tasks on q and the remaining tasks, and (2) dividing it by the total link capacity *adjacent to* q. This effectively assumes such communication will be routed evenly across all adjacent links and internal links (not adjacent to a processor) will not be bottleneck. These assumptions, the first one in particular, may be optimistic and need be more sophisticated when q has multiple adjacent links. In our experiments, a processor is adjacent only to a single link, thus this is not an issue.

To illustrate how the procedure works, let us look at a process that maps tasks to a processor as shown in Figure 8. We start from the empty mapping and add tasks to the left processor, in the order indicated by the numbers. Edges and nodes in the task graph weigh one. The edge of the resource graph weighs one and the two nodes five. Figure 9 plots O, O_{comp}, and $O_{\leftrightarrow}$ (graphs are undirected) at every step. Observe that the open communication metric goes up and down and that O_i (the maximum of the three values) minimizes at m_8, even though compute load between the two processors best balances at m_{11} (the point where two graphs O_{comp} and O intersect). Therefore the procedure will choose to put 8 tasks on the left processor, which is optimal. If edges of the task graph weigh much smaller (say, 0.1), on the other hand, the graph of $O_{\leftrightarrow}$ will become

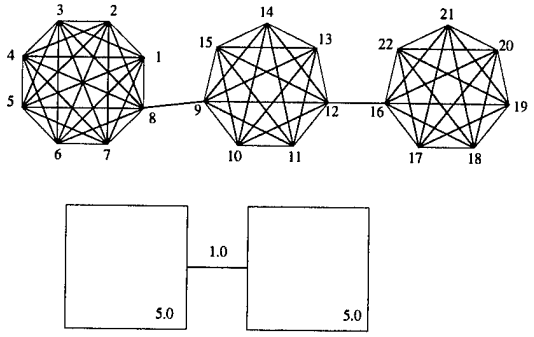


Figure 8. Example graph to illustrate `map_tasks_on`.

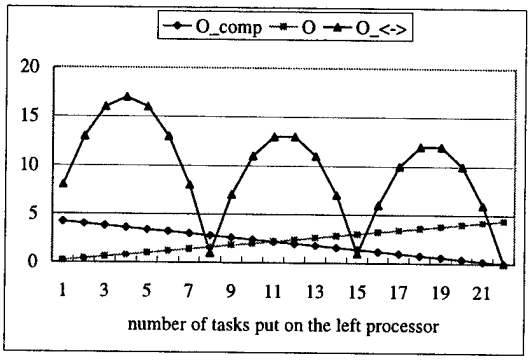


Figure 9. How O, O_{comp}, and $O_{\leftrightarrow}$ changes as we put tasks to the left processor in Figure 8.

much lower, giving the best M_i at m_{11}. So in this case, the first processor will get 11 tasks, which is again optimal.

Note that in general, for the easy case where communication is negligible and task and processors weigh uniformly (w.o.l.g. assume they weigh 1), the procedure `map_tasks({})` is guaranteed to return the optimal mapping in which no processors get more than $\lceil N/P \rceil$ tasks, where N is the number of tasks and P the number of processors. To see this, consider what happens in the first call to `map_tasks_on`$(\{\}, q, V_P - \{ q \})$. As communication is negligible, it simply amounts to finding the intersection of two graphs $O = i$ and $O_{\text{comp}} = (N - i)/(P - 1)$. Solving the equation $O = O_{\text{comp}}$ gives $i = N/P$ and thus the best value is obtained either at $\lceil N/P \rceil$ or $\lceil N/P \rceil - 1$. We can repeat this argument to show that this is the case for other processors. This property ensures our mapping procedure quickly gives a good solution for compute-mostly jobs without iterating improvements.

Other Implementation Notes: The actual implementation of the procedure is a bit more sophisticated to avoid useless computation.

- `map_tasks_on` quits as soon as $O(G, P, m_i)$ becomes greater than any of M_j $(j < i)$. Since $O(G, P, m_i)$ is monotonically non-decreasing with respect to i, once this condition is observed, we have:

  $$M_k \geq O(G, P, m_k) \geq O(G, P, m_i) > M_j \text{ for all } k > i.$$

  Thus there is no chance that we observe a better M_k in future. Again, this guarantees that in the easy case mentioned above, `map_tasks_on` quits as soon as it puts $\lceil N/P \rceil + 1$ tasks on a processor.

- Both `map_tasks` and `map_tasks_on` optionally take one more parameter, u, which specifies the occupancy they should at least achieve. `map_tasks_on` quits as soon as $O(G, P, m_i)$ becomes greater than this value. `map_tasks` aborts the entire process as soon as $O_{\text{comp}}(G, P, m_i, Q)$ gets larger than u in an iteration. This is useful when we already know a mapping and try to improve it. In such circumstances, we determine u based on the current occupancy (*e.g.*, $u =$ the current occupancy $\times$ 0.9) and give it to `map_tasks`.

4.5. Iterative Improvement

Procedure `improve` in Figure 10 tries to improve a given (total) mapping m by first removing some tasks from m (line 3) and then applying `map_tasks` to the partial mapping obtained this way. Obviously, the key is to identify a small subset of tasks whose removal gives us a good chance to improve the mapping. A silly selection algorithm could remove all the tasks from m, effectively applying `map_tasks` again from the empty mapping.

The selection algorithm works as follows.

1. First calculate the current max occupancy and multiply it by an acceleration factor (currently 0.75). We remove tasks until the resulting mapping gives max occupancy below this value (line 10).

2. We scan nodes and edges of the resource graph, trying to find an edge or a node whose occupancy is greater than it.

3. If such a node is found, let p be the node. Find tasks s_i $(i = 1, 2, \cdots)$, such that s_i is mapped on p and is not deleted yet. Among all such tasks, select the heaviest task.

4. If such an edge is found, let l be the edge. Find pairs of tasks (s_i, t_i) $(i = 1, 2, \cdots)$, such that the route between s_i and t_i (on the processor graph) uses l and either s_i

```
1: improve(m)
   {
     m = remove_bottlenecks(m);
     m = map_tasks(m);
5:   return m;
   }

   remove_bottlenecks(m)
   {
10:  o = 0.75 × max(O(G, P, m));
     D = {}; /* set of deleted mappings */
     while (max(O(G, P, m − D) > o)) {
       L = L(G, P, m − D);
       find if any p ∈ P and q ∈ P s.t. L_{p,q}/P_{p,q} > o;
15:    if found {
         select s, t ∈ V_G s.t. (s ∉ D or t ∉ D),
           P_{p,q}^{m(s),m(t)} = 1, and G_{s,t} is maximum;
         D = D + {(s, m(s)), (t, m(t))};
       } else {
20:      there must be p ∈ P s.t. L_p/P_p > o;
         select s ∈ V_G s.t. s ∉ D,
           m(s) = p, and G_s is maximum;
         D = D + {(s, m(s))};
       }
25:  }
     return m − D;
   }
```

Figure 10. The procedure to improve the current mapping.

or t_i is not deleted yet. Among all such pairs, select the most heavily communicating pairs and delete them.

5. Repeat steps 2-4 until the occupancy becomes less than the target value computed at step 1.

It basically tries to identify a set of tasks that form *bottlenecks*, tasks making the current occupancy so large. It finds an edge or node in the resource graph whose occupancy is larger than the target value calculated from the current occupancy. If found, tasks contributing to the edge or the node are candidates.

While reasonable, this algorithm still has a room for further improvements which we are yet to experiment with. It does not pay attention to communication induced between deleted tasks and undeleted tasks. If the communication between them is large, attempts to moving those deleted tasks unavoidably induce a large communication traffic and are likely to fail. Among many ways to select candidate tasks, we like to select a set of tasks that do not intensively communicate with the other tasks. If such selection cannot be obtained, it makes sense to co-migrate some of the other tasks too, even if they do not constitute the bottleneck.

5. Experiments

5.1. Graph Generation

We used a simplified version of the Internet topology model described in [7, 12] to generate resource graphs. While they model WAN, MAN, and LAN, we omit MANs for simplicity and model resource graphs by two level (WAN and LAN) hierarchy. Given a configuration that describes such parameters as the number of WAN nodes, LANs, nodes within a LAN, and compute capacity of a processor, it generates a graph as follows.

- First generate the specified number of WAN nodes and randomly place them in a specified rectangle. Create edges between all pairs of nodes, associating a cost proportional to its length with each edge. Then make the minimum spanning tree of the resulting complete graph.

- Generate the specified number of LANs. For each LAN, first create a gateway and randomly place it in the specified rectangle. Connect gateway to its nearest WAN node. Then generate a randomly chosen number of nodes in the LAN. LAN is modeled as a (shallow) tree whose root is connected to its gateway and each node has a randomly chosen number of children. The compute capacity within a single LAN is uniform and chosen randomly.

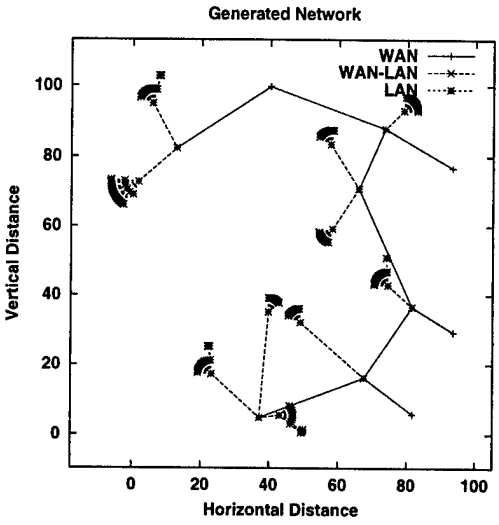


Figure 11. A typical resource graph used by the experiments. It was generated by a simplified Internet topology generator.

Table 1 lists relevant parameters and Figure 11 shows a typical graph generated by this model. A sector in the figure is a LAN, which has from 10 to 20 nodes. Depth of some sectors are one and that of others two.

For task graphs, we generate a pipeline of parallel jobs for each run as follows.

1. Randomly choose the number of tasks in a parallel job (m), and create a complete graph of m nodes. Nodes within a single parallel job are equally weighted and the weight is randomly chosen.

2. Repeat the step 1 a randomly chosen number (n) of times and obtain n complete graphs.

3. Connect these complete graphs to form a simple pipeline (without branches and merges). To connect two complete graphs A and B, we simply form a complete bipartite graph (create an edge between every task in A and every task in B). Each edge weighs $1.0/(a \times b)$, where a and b are the number of nodes in A and B, respectively. The total communication volume between two parallel jobs is always 1.0.

Resource Graph	
the number of WAN nodes	10
the number of LANs	10
bandwidth between WAN nodes	1000.0
WAN $\leftrightarrow$ LAN bandwidth	500.0
LAN bandwidth	50.0
the number of children for a LAN node	[5,20]
compute capacity of a processor	[3.0,15.0]
Task Graph	
the number of clusters in a task graph	[5,10]
the number of tasks in a cluster	[5,10]
compute requirement of a task	[1.0,3.0]
(total) comm. between a pair of clusters	1.0
communication intensity parameter	c (see text)

Table 1. Parameters used in the experiments. $[a, b]$ means that a value is chosen randomly from $[a, b]$ for each run.

Edges within a single parallel job are equally weighted and the weight is chosen randomly from $[1, c]$, where c is a parameter that controls the communication intensity of the tasks. We compare performance of several algorithms for various values of c.

Let us perform a rough calculation to see how communication intensity of tasks vary according to c. Since compute requirement per task is from 1.0 to 3.0, and capacity per processor is 3.0 to 15.0, the occupancy of a processor ranges from 1.0/15.0 to 1.0, assuming a single processor accommodates a single task. When sufficiently many tasks are created, one of the processors is likely to get an occupancy close to 1.0. On the other hand, since the number of tasks in a parallel job is from 5 to 10, the communication volume per task is from $5c$ to $10c$ (ignoring inter-cluster communication, which is a fraction). Considering LAN bandwidth, which is 50.0, occupancy of an edge adjacent to a processor is $0.1c$ to $0.2c$, again assuming a single task on a single processor. Comparing the expected node occupancy (≈ 1.0) and this value, clustering is unlikely to be necessary for $c \approx 1$. In this sense, for $c \approx 1$, tasks are hardly communication intensive. For $c \approx 16$, on the other hand, an edge occupancy will range from 1.6 to 3.2, much larger than the expected processor occupancy. Therefore when $c \approx 16$, a good solution is likely to use clustering.

5.2. Results

We compare the following four algorithms for $c = 1$, 2, 4, 8, 12, and 16.

Base: Do not use the open communication metric described in Section 4.4. Also do not perform the improvement phase described in Section 4.2.

Base + improve: Do not use the open communication metric. Apply the improvement phase after an initial mapping is obtained, again without open communication metric.

Open: Use the open communication metric. But do not perform the improvement phase.

Open + improve: Use the open communication metric and apply the improvement phase to the initial mapping.

For each value of c, we generate 32 instances of the problem and run the four algorithms. For each instance and for each algorithm, we calculate the improvement of the occupancy against **Base**. Graphs in Figure 12 show the result. A dot corresponds to an instance and the value represents the relative improvement over Base (Note that in **Base + improve**, the number of dots looks much smaller than 32. This is because results are in many cases 1; *i.e.*, no improvement is observed). Figure 13 shows the average improvement over 32 instances.

It is clear that taking open communication into account becomes significant as tasks become communication intensive. As we have expected, all four algorithms perform equally well for $c \approx 1$. Adding the iterative improvements to **Open** slightly improved performance, but not very much. As we have discussed in Section 4.5, our task selection algorithm is not very sophisticated yet, so we need more experiments to be conclusive.

6. Related Work

6.1. Task Scheduling

There are a number of studies on task scheduling in heterogeneous environments [8, 11, 15, 17, 18, 27]. To the author's knowledge, most of these work have been focusing on scheduling DAGs, in which a task graph represents dependencies between tasks. DAG scheduling problem and the throughput optimization problem discussed in this paper are quite different, both in terms of basic techniques employed and target applications. In terms of techniques, most algorithms for DAG scheduling are more or less based on a list scheduling, whereas the basic model of the throughput optimization is graph partitioning. For target application, DAG scheduling applies to a set of many tasks that rarely communicate with each other, whereas the throughput optimization problem to tasks communicating via high-bandwidth streams. While both are important, we believe the throughput optimization problem discussed in this paper will increasingly become important for emerging multimedia and data-intensive applications on wide area.

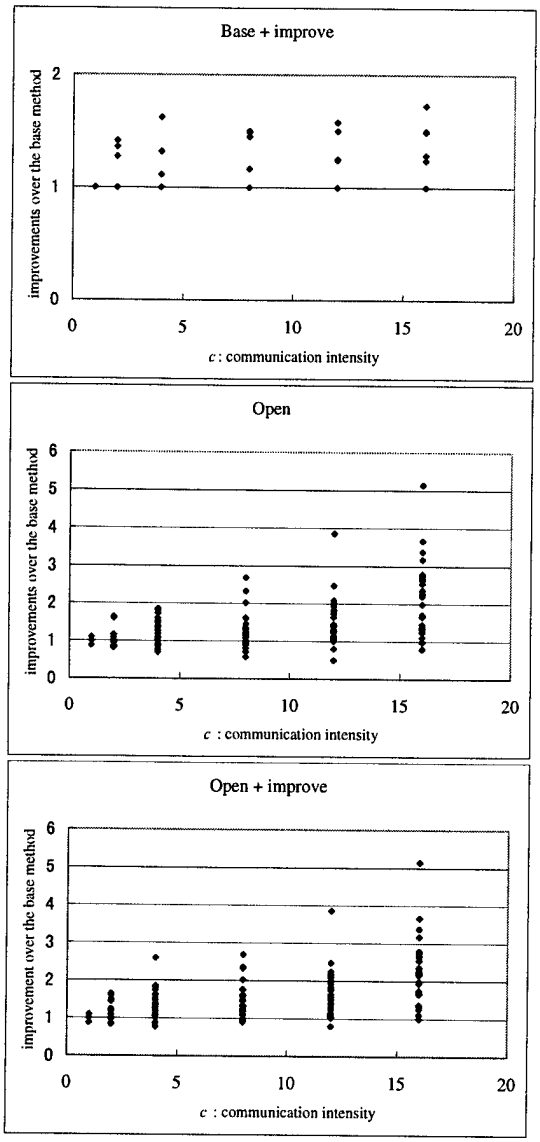


Figure 12. Improvements of the various methods over the Base method (Internet model).

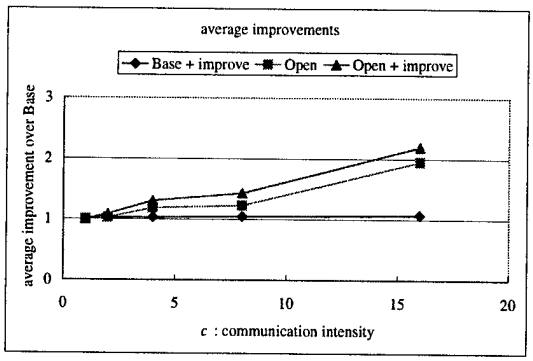


Figure 13. Average improvements.

Several studies on scheduling with bandwidth metrics have been done. Subhlok et al. [25, 26] studied optimal processor allocation for a set of communicating data parallel tasks, both with latency and bandwidth metrics. In their problem setting, performance of a task is a function of the number of processors allocated for that task and does not depend on which processors are used. They make a similar assumption on communication performance. Therefore the problem amounts to determining *how many* processors should be allocated for each task. This effectively assumes two things. One is that processor speed is uniform. The other is that link bandwidth is not only uniform but also very high, so the locations of communicating tasks do not matter. This will be a good model for system-area cluster, which is their target environment, but will not be directly applicable to multimedia/data-intensive applications on wide area.

Developing applications that exhibit robust performance over a wide range of resource conditions have become such an important issue. Several frameworks have been proposed [3, 24] and many practical studies on adaptive applications in heterogeneous environments have been conducted [2, 23]. While such studies are certainly instructive, it is difficult for individual programmers to perform such studies for every single application. We believe that task mapping should be much more automated.

6.2. Graph Partitioning

Graph partitioning tries to cut a graph into two ore more sub-graphs each of which is more connected than the entire graph. Our problem shares the common difficulty with this basic problem, in that moving any single node or exchanging any single pair of nodes is not likely to improve the objective function.

Kernighan and Lin [13] dealt with the basic two-way partitioning problem to cut the graph into two graphs of exactly the same size and gave the basic idea to overcome the local optima. Fidducia and Mattheyses [9] proposed a faster

algorithm for a slightly different problem, in which a certain amount of difference between the sizes of the two subgraphs is accepted. Wei et al. further proposed a ratio cut [28], which automatically achieves a balance between a low cut size and a good ratio of the sub-graph sizes. Finally, Yeh et al. proposed multi-way partitioning based on stochastic flow injection method [29, 30].

While our current algorithm can basically use any good partitioning algorithm as the preprocessing of a task graph, the following property of the Yeh's method is particularly attractive for our purpose; it can not only find highly connected components from a graph, but also finds the (negative) fact that no more natural clusters exist in a graph, in which case it typically divides the graph into many singletons. Having only two-way partitioning, we still have to apply two-way partitioning recursively. This is computationally expensive and does not improve quality.

7. Summary and Future Work

We have presented a heuristic algorithm for a task mapping problem, which takes compute and bandwidth requirements into account. The key to achieving good performance is clustering, a process that recognizes intensively-communicating tasks. We use this clustering information to obtain the order in which tasks should be put on processors. Open communication metric was introduced to decide how many tasks should be put in a processor. The algorithm is able to incrementally improve a given mapping, moving only those tasks that form the bottleneck. Therefore it can efficiently fix a significant load imbalance caused by a small number of tasks. We observed expected experimental results, indicating that our communication-sensitive algorithm significantly outperforms simpler, communication-ignorant algorithms for communication-intensive jobs.

We are planning to enhance this work in several ways. First, we are going to improve the task selection algorithm for incremental improvements, so that it moves clusters that do not intensively communicate with the rest of the tasks. Second, we will analyze computational complexity of the algorithm in detail. Third, we will try to identify other cases where this algorithm guarantees to produce a result within a constant of the optimal. Practical goals include developing a system that automatically selects resources and maps tasks on wide area, which helps Grid application designers develop performance-portable Grid code. We hope this work serves as a sound, logical step toward achieving this goal.

Acknowledgements The research described is supported in part by DARPA orders #E313 and #E524 through the US Air Force Rome Laboratory Contracts F30602-99-1-0534, F30602-97-2-0121, and F30602-96-1-0286. It is also supported by NSF Young Investigator award CCR-94-57809 and NSF EIA-99-75020. It is also supported in part by funds from the NSF Partnerships for Advanced Computational Infrastructure – the Alliance (NCSA) and NPACI. Support from Microsoft, Hewlett-Packard, Myricom Corporation, Intel Corporation, Packet Engines, Tandem Computers, and Platform Computing is also gratefully acknowledged. The paper was written when the first author was visiting UCSD as an exchange visitor, supported by the Ministry of Education of Japan.

References

[1] G. Aloisio, M. Cafaro, and R. Williams. The digital puglia project: An active digital library of remote sensing data. In *Proceedings of the 7th International Conference on High Performance Computing and Networking Europe*, volume 1593 of *Springer Lecture Notes in Computer Science*, pages 563–572, 1999. `http://www.cacr.caltech.edu/SDA/digital_puglia.html`.

[2] F. Berman and R. Wolski. Application-level scheduling on distributed heterogeneous networks. In *Proceedings of Supercomputing*, 1996.

[3] F. Berman and R. Wolski. The AppLeS project: A status report. In *Proceedings of the 8th NEC Research Symposium*, Berlin, Germany, 1997. `http://www-cse.ucsd.edu/groups/hpcl/apples/apples.html`.

[4] A. Chien, M. Lauria, R. Pennington, M. Showerman, G. Iannello, M. Buchanan, K. Connelly, L. Giannini, G. Koenig, S. Krishnamurthy, Q. Liu, S. Pakin, and G. Sampemane. Design and evaluation of an HPVM-based windows NT supercomputer. *The International Journal of High-Performance Computing Applications*, 13(3):201–209, Fall 1999. `http://www-csag.ucsd.edu/projects/hpvm.html`.

[5] D. E. Culler, A. Arpaci-Dusseau, R. Arpaci-Dusseau, B. Chun, S. Lumetta, A. Mainwaring, R. Martin, C. Yoshikawa, and F. Wong. Parallel computing on the berkeley NOW. In *Proceedings of the 9th Joint Symposium on Parallel Processing (JSPP)*, 1997. `http://now.cs.berkeley.edu/`.

[6] Digital Sky Project. Center for Advanced Computing Research (CACR) at California Institute of Technology. `http://www.cacr.caltech.edu/SDA/digital_sky.html`.

[7] K. C. Ellen W. Zegura and S. Bhattacharjee. How to model an Internetwork. In *Proceedings of IEEE Infocom '96*, 1996.

[8] M. Eshaghian and Y. Wu. Mapping heterogeneous task graphs onto heterogeneous system graphs. In *Proceedings of Heterogeneous Computing Workshop*, 1997.

[9] C. Fiduccia and R. Mattheyses. A linear-time heuristic for improving network partitions. In *Proeedings of 19th ACM/IEEE Design Automation Conference*, pages 175–181, 1982.

[10] I. Foster and C. Kesselman, editors. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers, 1998.

[11] R. F. Freund, M. Gherrity, S. Ambrosius, M. Campbell, M. Halderman, D. Hensgen, E. Keith, T. Kidd, M. Kussow, J. D. Lima, F. Mirabile, L. Moore, B. Rust, and H. J. Siegel. Scheduling resources in multi-user, heterogeneous, computing environments with SmartNet. In *Proceedings of Heterogeneous Computing Workshop*, 1998.

[12] M. D. Ken Calvert and E. W. Zegura. Modeling Internet topology. *IEEE Communications Magazine*, June 1997.

[13] B. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *Bell Systems Technical Journal*, 49(2):291–307, 1970.

[14] K. Knobe, J. M. Rehg, A. Chauhan, R. S. Nikhil, and U. Ramachandran. Scheduling constrained dynamic applications on clusters. In *Proceedings of SC'99*, 1999. `http://www.sc99.org`.

[15] Y.-K. Kwok and I. Ahmad. *High Performance Cluster Computing*, volume 1, chapter 23, Parallel Program Scheduling Techniques, pages 553–578. Prentice Hall, 1999.

[16] M. Lutz. *Programming Python*. O'Reilly & Associates, 1996.

[17] M. Maheswaran and H. J. Siegel. A dynamic matching and scheduling algorithm for heterogeneous computing systems. In *Proceedings of Heterogeneous Computing Workshop*, 1998.

[18] H. Nakada, A. Takefusa, S. Matsuoka, M. Sato, and S. Sekiguchi. A scheduling framework for global computing. In *Proceedings of Joint Symposium on Parallel Processing*, pages 277–284, 1999. `http://ninf.etl.go.jp/`.

[19] C. H. Papadimitriou and K. Steiglitz. *Combinatorial Optimization : Algorithms and Complexity*. Dover Publications, 1998.

[20] U. Ramachandran, R. S. Nikhil, N. Harel, J. M. Rehg, and K. Knobe. Space-time memory: A parallel programming abstration for interactive multimedia applications. In *Proceedings of the Seventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'99)*, 1999.

[21] J. M. Rehg, K. Knobe, U. Ramachandran, R. S. Nikhil, and A. Chauhan. Integrated task and data parallel support for dynamic applications. In *Proceedings of the Fourth Workshop on Languages, Compilers, and Run-Time Systems for Scalable Computers*, pages 167–180, 1998.

[22] SARA: The Synthetic Aperture Radar Atlas. University of Lecce and California Institute of Technology. `http://sara.unile.it/sara/`.

[23] N. Spring and R. Wolski. Application level scheduling of gene sequence comparison on metacomputers. In *Proceedings of the 12th ACM International Conference on Supercomputing*, 1998.

[24] P. Steenkiste. Adaptation models for network-aware distributed computations. In *3rd Workshop on Communication, Architecture, and Applications for Network-based Parallel Computing (CANPC'99)*, 1999.

[25] J. Subhlok and G. Vondran. Optimal mapping of sequences of data parallel tasks. In *Proceedings of ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 1995.

[26] J. Subhlok and G. Vondran. Optimal latency-throughput tradeoffs for data parallel pipelines. In *Proceedings of 8th Annual ACM Symposium on Parallel Alogorithms and Architectures (SPAA)*, pages 62–71, 1996.

[27] H. Topcuoglu, S. Hariri, and M.-Y. Wu. Task scheduling algorithms for heterogeneous processors. In *Proceedings of Heterogeneous Computing Workshop*, 1999.

[28] Y. Wei and C. Cheng. Ratio cut partitioning for hierarchical designs. *IEEE Transactions on Computer-Aided Design*, 10:911–921, 1991.

[29] C.-W. Yeh, C.-K. Cheng, and T.-T. Y. Lin. A probabilistic multicommodity-flow solution to circuit clustering problems. In *IEEE International Conference on Computer-Aided Design*, pages 428–431, 1992.

[30] C.-W. Yeh, C.-K. Cheng, and T.-T. Y. Lin. Circuit clustering using a stochastic flow injection method. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 14(2):154–162, 1995.

Kenjiro Taura is a research associate of Department of Information Science, University of Tokyo. He is also a visiting researcher of the Department of Computer Science and Engineering at the University of California at San Diego. He received BS, MS, and PhD from University of Tokyo. Contact him at UCSD/CSE-AP&M 6414, 9500 Gilman Drive, Dept. 0114 La Jolla, CA 92093-0114 USA; tau@csag.ucsd.edu.

Andrew A. Chien is the Science Applications International Corporation Chair Professor in the Department of Computer Science and Engineering at the University of California at San Diego. Andrew Chien leads the Concurrent Systems Architecture Group and is involved with joint projects with both NCSA and NPACI . He received BS, MS, and PhD from the Massachusetts Institute of Technology. Contact him at UCSD/CSE-AP&M 4808, 9500 Gilman Drive, Dept. 0114 La Jolla, CA 92093-0114 USA; achien@cs.ucsd.edu.

Design of a Framework for Data-Intensive Wide-Area Applications *

Michael D. Beynon, Tahsin Kurc, Alan Sussman, Joel Saltz
Department of Computer Science
University of Maryland, College Park, MD 20742
{beynon,kurc,als,saltz}@cs.umd.edu

Abstract

Applications that use collections of very large, distributed datasets have become an increasingly important part of science and engineering. With high performance wide-area networks becoming more pervasive, there is interest in making collective use of distributed computational and data resources. Recent work has converged to the notion of the Grid, which attempts to uniformly present a heterogeneous collection of distributed resources. Current Grid research covers many areas from low level infrastructure issues to high level application concerns. However, providing support for efficient exploration and processing of very large scientific datasets stored in distributed archival storage systems remains a challenging research issue.

We have initiated an effort that focuses on developing efficient data-intensive applications in a Grid environment. In this paper, we present a framework, called filter-stream programming, that represents the processing units of a data-intensive application as a set of filters, which are designed to be efficient in their use of memory and scratch space. We describe a prototype infrastructure that supports execution of applications using the proposed framework. We present the implementation of two applications using the filter-stream programming framework, and discuss experimental results demonstrating the effects of heterogeneous resources on application performance.

1. Introduction

Increasingly powerful computers have made it possible for computational scientists and engineers to model physical phenomena in greater detail. As a result, overwhelming amounts of experimental data are being generated by scientific and engineering simulations. In addition, large amounts of data are being gathered by sensors of various sorts, attached to devices such as satellites and microscopes. There are many examples of large useful datasets from simulations [26, 29, 33], sensor data [25, 28], and medical imaging [2] (pathology, MRI, CT scan, etc.). The primary goal of generating data through large scale simulations or sensors is to better understand the causes and effects of physical phenomena. Understanding is achieved through running analysis codes on the stored data, or by a more interactive visualization that relies on the ability to gain insight from looking at a complex system. Thus, both data analysis and visual exploration of large datasets plays an increasingly important role in many domains of scientific research. Decision support database applications are similar to scientific applications because they deal with large quantities of data (relational data), and need to perform significant computation in processing the data. The value provided by decision support systems and data-mining algorithms depend greatly on the amount of data, and hence businesses are inclined to retain as much data as possible.

Disks continue to become larger and cheaper making them commodity items. This helps to make it relatively easy to setup a large set of archival storage disks at a relatively low cost. For example, to build a large disk farm out of commodity PC components for the lowest current price: $400 for a motherboard with a Celeron or AMD K6-2 400MHz cpu and 64MB memory [9], four 40GB EIDE disks at $254 each [10] and a fast ethernet interconnect (100 Mbps), a farm of 8 PCs can present 1.25TB of disk space for less than $15K. The price point is sufficiently low to enable many such disk collections to be setup independently at multiple disparate locations, where local storage needs dictate. We anticipate that this trend will result in the emergence of *islands of data*, where cheap archival storage systems will be used to hold large locally generated datasets. Use of computation farms also is important for handling very large datasets in a reasonable amount of time. Oftentimes, high performance computation farms are where the data is generated (as in large scientific simulations), and the data may reside locally on the computation farm in an archival storage

*This research was supported by the National Science Foundation under Grants #ASC-9619020 (UC Subcontract #10152408), and by the Office of Naval Research under Grant #N66001-97-C-8534.

0-7695-0556-2/00 $10.00 © 2000 IEEE

system such as HPSS [22]. Thanks to high-performance networks, increasing numbers of computation farms have become accessible across a wide-area network. These computation farms span a spectrum of widely varying configurations and computation power, from relatively inexpensive network of workstations and PC clusters to very expensive high-performance machines, providing computing performance in the order of Teraflops.

These trends combine to present a new opportunity: *very large distributed datasets that can be used by applications for computationally and data intensive analysis, exploration and visualization.*

Consider the following scenario: A scientist wants to compare properties of a 3D reconstructed view of a raw dataset recently generated at a collaborating institution, with the properties of a large collection of reference datasets. The 3D reconstruction operation involves retrieving portions of 2D slices from the regions in question, and then performing feature recognition and interpolating between the slices to extract the important 3D features. A description of these features and the associated properties are then compared against a database of known features, and some appropriate similarity measure is computed. The final result is the set of reference features found that are close in some way to those found in the new raw dataset, along with the corresponding view renderings to visualize.

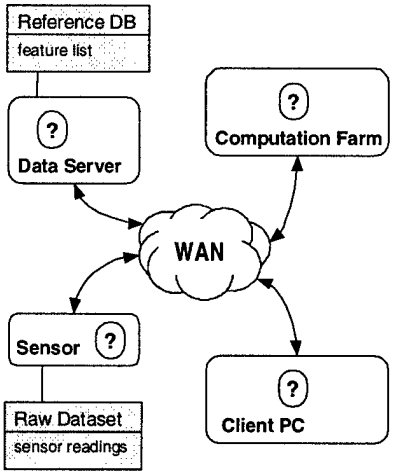


Figure 1. 3D reconstruction/visualization scenario on distributed collection of resources.

Consider the problems that can occur when the application is executed in a Grid [16] environment. That is, the required resources (new raw dataset, reference database, and the scientist) are all at distributed locations in a wide-area network as seen in Figure 1. The reference database is likely to be stored in an image library, since the dataset is large and useful to many users. The new raw dataset is stored at the site where the sensor readings were taken. If the hosts containing the data are low-power archival systems that make the execution of the 3D reconstruction code prohibitively expensive, it becomes unclear how to structure the application for efficient execution. Ideally we would like to execute portions of the application at strategic points in the collection of machines. A set of possible locations for performing computation is indicated in the figure by question marks. For example, if the portion of code that performs the range select on the new raw dataset could be run on the host where the data lives, the amount of data to be transmitted over the wide-area network (WAN) would be reduced. The computation farm is an ideal location for the feature recognition and 3D reconstruction due to the parallelism inherent in the codes. Given the set of features that were identified, it would be efficient to perform the selection of similar features from the reference database on the data server where the database is located. The low end PC where the scientist is located can be used to collect the 3D rendering and the similar feature information for interactive presentation to the scientist.

The success of this scenario depends on the application allowing portions of its computation to be executed in a distributed fashion. Beyond the mere possibility of execution in a distributed environment is the question of how efficient the application is. One interpretation of efficiency in this context is the ratio of useful data transmitted to the total amount of data transmitted between any two pieces of the application. For example, if an application transmitted a full dataset from a remote host, and discarded a large portion not required by subsequent processing, then this would not be considered efficient operation.

We have initiated an effort to investigate and develop methodologies and a framework for efficient execution of applications that make use of distributed collections of datasets in a Grid environment. There are two main challenges in developing efficient applications in a Grid environment:

- The Grid is composed of collections of heterogeneous resources. The characteristics, capacity and power of resources, including storage, computation, and network, vary widely. This requires that applications should be structured to accommodate the heterogeneous nature of the Grid.
- These distributed resources can be shared by many applications. This requires that applications should be designed to be optimized in their use of shared resources.

In order to address these challenges, we are investigating:

- Methodologies and a framework for structuring applications. In particular, we address decomposition of application processing into components and placement of these components onto a collection of heterogeneous resources that will aid efficient execution.

- Feasibility and effects of exposing application structure and characteristics. In particular, we address exposing resource requirements and the communication pattern between application components, and how this extra application structure information can be used.
- An infrastructure for providing execution of applications that conform to the developed framework.

In this paper, we present a framework, called *filter-stream programming*, that represents the processing in a data-intensive application as a set of processing units, referred to here as *filters*, which are designed to be efficient in their use of memory and scratch space. In this framework, data exchange between any two filters is described via *streams*, which are uni-directional pipes that deliver data in fixed size buffers. We describe a prototype infrastructure that provides support for execution of applications using the proposed framework. We present the implementation of two applications in filter-stream programming framework, and experimental results to demonstrate the effects of heterogeneous resources on the performance of the applications.

2. The Proposed Approach

In this section, we present a framework, called the *filter-stream programming model*. The basic ideas are to (1) constrain application components to allow for location independence, which is necessary for execution in a distributed environment, and (2) expose the application communication pattern and resource requirements, allowing a runtime system to aid in efficient execution. We should note that any programming model (e.g., message passing) modified to expose similar constraints could be employed in place of the filter-stream programming model we describe.

The programming model used in this work is derived from the *stream-based programming model*, originally developed for Active Disks [1, 35]. Many stream-based algorithms were developed and analyzed for Active Disks. These algorithms carry out a variety of data transformations that arise in earth science applications and applications of standard relational database sort, select and join operations. In this work we extend these algorithms and investigate the application of filters and the stream-based programming model in a Grid environment.

In the filter-stream programming model, an application is represented by a collection of *filters*. A filter is a portion of the full application that performs some amount of work. Filters are required to pre-disclose dynamic memory and scratch space needs. Communication with other filters is solely through the use of *streams*. A stream is a communication abstraction that allows fixed sized untyped data buffers to be transported from one filter to another. An example set of filters for the motivating example is shown in Figure 2. A simple example of this model is Unix system pipes, where the standard output of a process is used as standard input for another process. Unix pipes represent a linear chain of filters, each of which have a single input stream and a single output stream. The filter-stream model allows for arbitrary graphs of filters with any number of input and output streams.

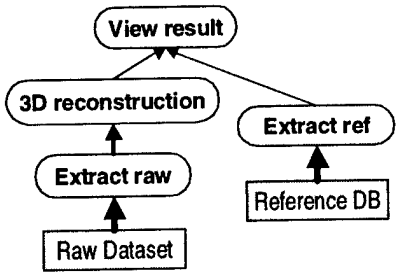


Figure 2. 3D reconstruction application decomposed into filters.

The process of manually restructuring an application using this model is referred to as *decomposing* the application. In choosing the appropriate decomposition, we need to consider the complete data flow path from data generation to ultimate consumption and the target machine configuration, which can be a distributed collection of heterogeneous machines. The main goal is to achieve efficient use of limited resources in a distributed and heterogeneous environment. The choice of decomposition can have a significant impact on efficiency and performance. Too many filters could mean there is not much work for individual filters, which would cause the system to spend much of its time moving data around and little time performing useful work on the data. Too few filters could limit the ability of the overall system to execute filters concurrently. Similarly, sending data over streams in very small pieces can make the overhead of the runtime system too large. If possible, an ideal granularity size should balance the amount of computation and communication such that the overall processing time across all filters does not exhibit a penalty merely because the computation is distributed.

Given a set of filters, the runtime mapping of filters onto various hosts in a wide-area grid environment is referred to as *placement*. Figure 3 shows a possible placement of the filters described for the motivating scenario. The choice of placement represents the main degree of freedom in affecting application performance by:

- placing filters with affinity to data sources near the sources,
- minimizing communication volume on slow links,
- co-locating filters with large communication volume,
- placing computationally intensive filters on less loaded hosts,

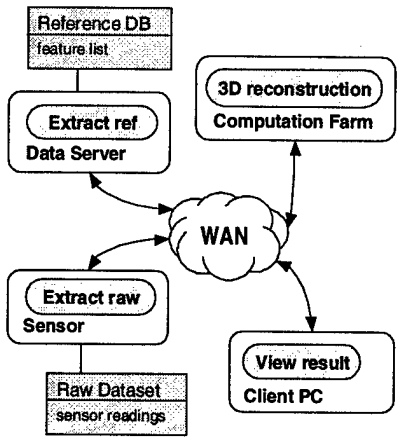


Figure 3. Possible placement of 3D reconstruction application filters.

- pipelining application filters by concurrent execution.

Note that a placement decision is not assumed to be static, and the programming model explicitly supports the notion of stopping a set of filters and replacing them with possibly a new set of filters with a different placement.

A runtime system infrastructure is used to support the execution of applications that are structured in the filter-stream programming model. In the following sections we present a prototype infrastructure for executing application filters, and present implementations of an image processing application and a database application using the filter-stream programming model.

3. Related Work

There is a large body of hardware and software research on archival storage systems, including distributed parallel storage systems [24], file systems [34], image servers [32], and data warehouses [23]. Several research projects have focused on digital libraries and geographic information systems [4, 20] that access collections of archival storage systems, high-performance I/O systems [8], tertiary storage systems [22] and remote I/O [19, 31]. Distributed storage systems attempt to provide large amounts of data to distributed clients. They present a uniform view of distributed data to applications, and transparently handle replicas and caching. This does not push the computation to the data as in our work, rather the data is migrated to the computation, but can achieve a similar result with an effective replacement policy and a warm cache. Another issue is *finding* the required data. The Storage Resource Broker (SRB) [31] provides uniform UNIX-like I/O interfaces and meta-data management services to locate and access collections of distributed data resources.

Distributed computing covers research that addresses ways to deal with distributed execution of application code in many different ways. Current work related to Grid computing [7, 14, 16] attempts to provide a uniform view into a collection of distributed computational, network and storage resources, and to provide services for unified, secure, efficient and reliable access. However, providing support for efficient exploration and processing of very large scientific datasets stored in archival storage systems at distributed locations remains a challenging research issue, and the necessity of infrastructure to provide such support was recognized in recent Grid forums [21]. This support of processing and retrieval for efficient operation is exactly what our work is attempting to provide.

There is a large body of classic work on dataflow systems. The macro dataflow model [30, 36] describes an application as a sets of tasks, communication edges, edge communication costs and task computation costs. PYRROS [37] uses this model of application behavior and manual annotations to cluster, map, and schedule computation to nodes of a homogeneous parallel machine. As we target a heterogeneous grid environment, we expand on assumptions such as constant computation regardless of placement, which makes sense in a tightly coupled environment. There is also task parallel work in systems such as STRAND [12], PCN, Fortran M [13], and HPF [18], which are related due to the dataflow model and/or task parallelism used. Our work is different in that we are considering remote datasource affinity as a primary reason for decomposition, rather than an attempt to extract paralellism.

4. A Prototype Infrastructure

In this section, we describe a prototype infrastructure implementation that provides support for execution of applications developed using the filter-stream framework. This work is part of the DataCutter project [6], that provides services for subsetting and processing multi-dimensional datasets stored on archival storage systems.

4.1. Filters

A filter is specified by the code to execute, and a description of the input and output streams it will use. Currently, filter code is expressed using a C++ language binding by sub-classing a provided filter base class. This base class provides a well-defined interface between the filter code and the system filter service. The description of input and output streams is specified in a separate configuration file (Figure 4).

Filters are constrained in several respects. First, *undisclosed dynamic allocation of memory and local disk space is not allowed. Instead, the filter must pre-disclose and be*

granted scratch memory and disk space by the runtime system. The granted scratch space is allocated on behalf of the filter by the runtime system when the filter is instantiated. Later, the filter may make use of the granted scratch space as needed. One of the potential benefits of exposing resource requirements in this way is that runtime system can achieve a better placement of filters. For example, a filter can be run on a machine with enough memory to avoid paging, and two filters requesting large scratch space can be placed on two different machines. In addition, the runtime system can potentially perform better scheduling of co-located filters on a machine. One of our goals in this project is to investigate and assess the potential benefits of pre-allocating memory, when it will really be important, and implications for structuring applications. In order to accomplish this, we plan to compare standard versions of target applications with filter-stream based implementations in subsequent work.

The interface for filters consists of an initialization function, a processing function, and a finalization function:

```
class MyFilter : public AS_Filter_Base {
public:
  int init(int argc, char *argv[]) { ... };
  int process(stream_t st) { ... };
  int finalize(void) { ... };
}
```

The **init** function is called when the filter is instantiated, and is passed parameters with the command line arguments used when the application was started. This is where a filter would request scratch memory space for use during later processing, for example. The **process** function is called to handle data arriving on the input streams in buffers from the sending filter. The parameter passed to the process function contains arrays of descriptors for the sets of input streams and output streams this filter can use. The filter can only read and write from/to the provided streams. No new streams can be created by the filter at runtime. The **finalize** function is called after all processing is finished and the filter is ready to terminate. This is where a filter would release any resources in use.

Another restriction is that a filter cannot change the source of its input streams nor the sinks of its output streams. This has two advantages. First, a filter does not need to handle buffering and scheduling for its own communication, thereby reducing the complexity of filters. Second, the location of filters is transparent, allowing filters to be placed at different locations initially and relocated as system resource constraints change.

Filters are the unit of placement. Each filter can potentially be executed on a different host. In addition, a filter's location may change at discrete application-defined intervals during the course of execution. Note this does not imply true migration of code and state, but rather placement can be recomputed and the filter can be stopped on the original host and a new copy re-instantiated on the new host. There is a limited mechanism for a final state transfer by a single buffer transfer from the old instance to the new instance. This approach avoids many of the details involved in checkpointing and process migration [11], while retaining most of the benefits. Filters need to be structured appropriately to handle such events. For cases when this is not desirable, a filter can be *pinned* to a particular host, which means the filter will always be placed on that host. This host affinity is useful for some situations, such as when runtime libraries or auxiliary data files only exist on a particular host, but does limit placement flexibility.

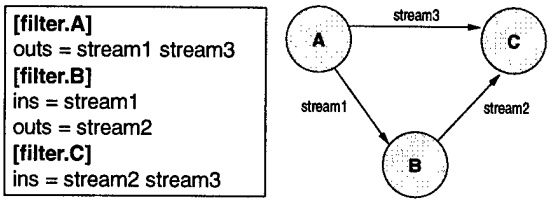
4.2. Streams

A stream is an abstraction used for filter communication. Since the placement of filters is largely unknown until runtime, this mechanism is used to achieve location-independent filter code because stream *names* are used rather than endpoint location on a specific host. A stream is used to specify how filters are logically connected, and to provide the glue at runtime to attach an input stream for one filter to an output stream of another. All transfers to and from streams are through a provided buffer abstraction. A buffer represents a contiguous memory region containing useful data. The buffer contains a pointer to the start, the length of the portion containing useful data, and the maximum size of the buffer. In the current prototype implementation we are using TCP for stream communication, but any point-to-point communication library could be added, such as Nexus [17].

The streams are specified in a global sense, separate from the application code. For each filter, a list of input and output streams is required. This discloses *all* potential filter communication pairs for the entire execution of the application. Given a set of filters with stream connectivity information, we can build a task graph where the nodes represent the filters, and the edges represent stream connections. For example, given three filters A, B and C, with data being sent from A to both B and C, and from B to C, the specification and resulting task graph are seen in Figure 4. Each filter in the spec appears in a section labeled [filter.<name>]. For each section, two optional entries *ins* and *outs* can appear containing the list of input and output stream names respectively.

In addition to the above inter-filter streams, we allow for two other types of streams:[1] File Streams and External Communication Streams. Files Streams are used to read and write to files stored in local scratch disk space or local permanent disk storage. The file stream abstraction further insulates the filter code from specifics about the host system. This provides a measure of safety between co-located filters,

[1] These are not yet implemented in the prototype.



(a) filter/stream spec (b) resulting graph

Figure 4. Sample filter/stream specification.

since one filter cannot access another filter's scratch disk space. The permanent disk storage presents a uniform file system to all filters, similar to a traditional file system. Thus a filter with sufficient authorization can read files in permanent disk storage written by another filter. External Communication Streams are used to connect to, and receive connections from legacy or other non-filter application code.

4.3. Execution Environment

The execution service performs all the steps necessary to instantiate filters on particular machines, connect all the logical stream endpoints, and call the interface functions to allow filters to run.

The description of where to instantiate filters is provided by a placement specification. Currently, this is statically generated before the application is started. An example placement specification for the sample filters is:

```
[placement]
A = host1.cs.umd.edu
B = host2.cs.umd.edu
C = host3.cs.umd.edu
```

The [placement] section is expected to contain one entry for each filter. The value is simply the host to execute the filter on. In general, this host can be a parallel machine, which implies multiple instances of the filter are created, but the prototype implementation does not yet support parallel filters. Security concerns have made it difficult to start processes on remote machines in a uniform manner. To solve this problem in the current prototype, an Application Execution Daemons (appd) must be run on every host used to execute filters. In the future, we plan to use existing Globus [15] services for process creation and authentication, in which case the Application Execution Daemons would not be needed. In addition, a single provided Directory Daemon (dird), which is similar to an LDAP server, is used to record the contact information (host, port, pid) for each appd. The dird is the only process that runs on a well-defined host and port. All other ports are ephemeral, and registered with the dird to later be queried. Based on a given placement specification, the execution of a filter-based application requires contacting the appd process on each host. A lookup is performed to find contact information for each required appd. Currently, we require an application binary to exist on every host, which must contain at least the code for the filters that will execute on that host. The binary can contain code for *all* the filters, and those filters not intended to run on a given host will not be instantiated at runtime. Currently we manually compile/copy the binaries as needed, but convenience procedures to do this will be added in the future.

The application is started by running the application binary on some host. This will become the *console* process, which performs no application processing such as running filters. The console process queries the dird process to get the relevant appd contact information, and then sends an execute command to each appd. The appd executes the application binary on that host, which in turn contacts the *console* process and performs some initial handshaking to setup the stream abstractions. In the current prototype, one POSIX thread is created for each filter that runs on the host, and a new instance of the application filter object is created. The thread calls the **init** interface function passing the command line arguments that were used when the *console* process was started. Next, the thread calls the **process** function. When this returns, all open streams are closed and the **finalize** function is called. Any remaining filter resources are released before the thread stops.

The multiple threads allow for fairness across filters on a single host, since all threads are executed with the same priority by the underlying operating system. No one filter can starve another due to the time sharing semantics of POSIX threads. Of course the filters do need to be thread-safe with respect to each other. Based on the filter-stream programming model, this should be natural for most applications. Filters in this model are inherently isolated and communicate via system provided buffers, thus should be fairly easy to make thread-safe due to the lack of shared resources. One problem could be common library routines. For the cases where no thread-safe implementations exist, we provide filter level locks that can be used to wrap the offending calls. This is only an issue when thread safety problems exist between filters that run on the same host, thus in the same process. For the sample placement, filters A, B and C can all have thread safety violations, since they are all actually run in separate processes on three different hosts.

For cases when thread safety is a problem and lock wrapping will not work, the infrastructure could be augmented to optionally use a single thread for all filters on a given host. Control could use a dataflow model where scheduling is performed by the infrastructure for filters based on the arrival of input. Another alternative re-design is to make each filter

execute as a separate process, thus avoiding all threading issues at the expense of increased filter communication costs on the same host. The use of a thread-per-filter-instance is a property of the current prototype implementation, and is not mandated by the overall model.

4.4. Applicability

Our approach is intended to be applicable to many common types of data-intensive applications that are emerging for use in a grid environment. The benefits of this approach result directly from two observations. The first is that the filter-stream framework exposes useful information, particularly application communication pattern and communication volume information. The second is that expressing the application processing as filters enables data volume reduction from remote data sources. These factors can be leveraged to improve application efficiency at runtime.

We recognize that the approach may not be effective for all application types, and are identifying characteristics that make applications ill-suited for this approach. Ill-suited in this case means performance will be no better than that of a generic message passing implementation, for example using MPI [27]. The first problem occurs when applications have high selectivity. This means nearly all the remote data is needed by the application, and no significant data reduction is achieved, which will nullify the benefits of application decomposition.

Applications that lack a clear task structure are also problematic. If the application cannot be divided cleanly into a set of filters, then placement choices are more limited for such a monolithic application. For example, if an application uses two remote data sources and cannot be divided into filters, we can execute the application at either data source (inputs), the client (output), or at an intermediate location. This will most likely be efficient only for data located at the execution site chosen, and inefficient for other input/output data sources/sinks.

The communication pattern and volume are significant characteristics that enable intelligent placement to overlap communication with computation and reduce high volume on slow network links. If the pattern or volume of communication is unknown, chaotic, very fine grained, or time varying, then it is difficult to perform an intelligent placement. For example, a communication pattern that involves all possible filters and is data dependent, where the destination for a piece of data is known only after its examination, will result in a conservative approximation of an all-to-all pattern with equal volume between all pairs of filters. There is no clear choice for placement in this case, because any possible good placement may only be known after execution has finished and the communication activity has been observed. Even worse, the observed communication pattern and volume may not be helpful for future runs, due to nondeterminism in such applications. Our approach assumes a single significant communication pattern and deterministic volume, which can be used for choosing placement for the entire execution. For the applications we are targeting, such as volume visualization, database decision support, and image processing, these assumptions appear to hold.

5. Application: Image Processing

The Virtual Microscope [2] is a query-response application that processes multi-dimensional image data to satisfy client queries. The dataset contains high power digitized images of microscope slides, which effectively forms a 3D dataset because each slide can contain multiple 2D focal planes at different depths. Images are stored at the highest magnification level, and the size of a single slide typically varies from $100MB$ to $5GB$, compressed. The system is required to provide interactive response times similar to a physical microscope, including continuously moving the stage and changing magnification. A typical query allows a client to request a 2D rectangular region at a particular magnification from within the bounds of a single focal plane. The processing for the query requires projecting high resolution data onto a grid of suitable resolution (governed by the desired magnification) and appropriately compositing pixels that map to a single grid point to avoid introducing spurious artifacts into the displayed image. The Virtual Microscope is useful for performing operations that are difficult with a physical microscope, such as simultaneous viewing and manipulation of a single slide by multiple users, or remote telepathology [2] where diagnosing pathologists are not required to be physically located near the slide.

5.1. Original Implementation

The original Virtual Microscope system is composed of two components; a client to generate queries and display the results (i.e. images), and a server to process the queries. The server is composed of a frontend and a backend. The frontend interacts with clients; it receives queries from clients and forwards them to the backend. The backend consists of one or more processes, typically one per node of a parallel machine. The processing of a query is carried out entirely in the backend.

In order to achieve high I/O bandwidth, each focal plane in a slide is regularly partitioned into data chunks, each of which is a rectangular subregion of the 2D image. Data chunks are declustered across all backend local disks to achieve I/O parallelism. Each pixel in a chunk is associated with a coordinate (in x- and y-dimensions) in the entire image. Each chunk has an associated minimum bounding rectangle (MBR) based on all the pixels in the chunk. An index

is created using the MBR of each chunk. Since the image is regularly partitioned into rectangular regions, a simple computation can be used instead of a complex index search.

During query processing, the backend process finds the chunks that intersect the query region, and reads them from the local disks. Each data chunk is stored in compressed form (JPEG format), and must be first *decompressed.* Then, it is *clipped* to the query region. Afterwards, each clipped chunk is subsampled to achieve the *zoom* level (magnification) specified in the query. The resulting image blocks are directly sent to the client. The client *viewer* assembles and displays the image blocks from each of the backend processes to form the query output.

5.2. Filter Implementation

The filter decomposition used for the Virtual Microscope system [6] is shown in Figure 5. This filter pipeline structure is natural for query-response applications. The figure only depicts the main dataflow path of image data through the system; other low-volume streams related to the client-server protocol are not shown for clarity. The thickness of the stream arrows indicate the relative volume of data that flows on the different streams.

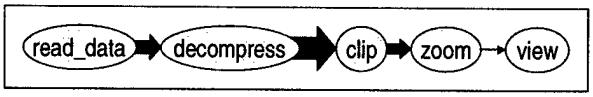


Figure 5. Virtual Microscope decomposition

In this implementation each of the main processing steps in the server is a filter:

- **read_data:** Full-resolution data chunks that intersect the query region are read from disk, and written to the output stream.
- **decompress:** Image blocks are read individually from the input stream. The block is decompressed using JPEG decompression and converted into a 3 byte RGB format. The image block is then written to the output stream.
- **clip:** Uncompressed image blocks are read from the input stream. Portions of the block that lie outside the query region are removed, and the clipped image block is written to the output stream.
- **zoom:** Image blocks are read from the input stream, subsampled to achieve the magnification requested in the query, and then written to the output stream.
- **view:** Image blocks are received for a given query, collected into a single reply, and sent to the client using the standard Virtual Microscope client/server protocol.

Figure 6 illustrates the high-level code for the zoom filter, which has two input streams and one output stream. It

```
VM_zoom::init() {
  // Allocate output buffer from pre-allocated scratch space
  bufOut = AllocFromScratch(getOutputStreamBufferSize());
}
VM_zoom::process(stream_t &st) {
  DC_StreamBuffer *buf;
  VMQuery *query;
  VMChunk *chunk;

  // recv the query
  buf = st.ins[0].read(); query = VMUnpackQuery(buf);
  // while there is data retrieved from input stream
  while ((buf = st.ins[1].read()) != NULL) {
    chunk = VMUnpackChunk(buf); // extract chunk information
    zoom_chunk(chunk, query); // perform zoom operation
    bufOut = VMPackChunk(chunk); // pack chunk into buffer
    st.outs[0].write(&bufOut); // write data to output stream
    FreeToScratch(chunk->Data);
  }
}
VM_zoom::finalize() {
  FreeToScratch(bufOut);
}
void VM_zoom::zoom_chunk(VMChunk *chunk, VMQuery *query) {
  int rel_zoom = query->Zoom/chunk->Zoom;
  int width = chunk->Width/rel_zoom;
  int height = chunk->Height/rel_zoom;
  int size = width*height*PIXELSIZE;

  char *pSrc = chunk->Data;
  char *pDst = chunk->Data = AllocFromScratch(size);
  // subsample the image block
  for (j = height; j>0; --j) {
    for (i = width; i>0; --i) {
      memcpy(pDst, pSrc, PIXELSIZE);
      pSrc += rel_zoom*PIXELSIZE;
      pDst += PIXELSIZE;
    }
    pSrc += rel_zoom*chunk.Width*PIXELSIZE;
  }
  // update chunk metadata
  chunk->Zoom = query->Zoom;
}
```

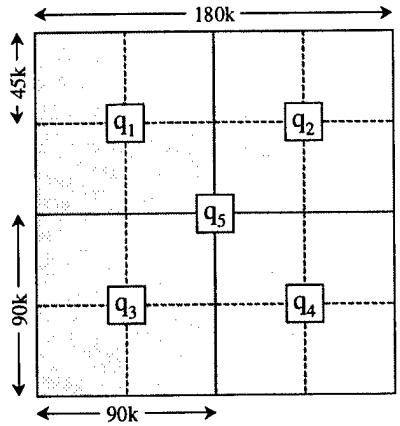
Figure 6. The high-level code for `zoom` filter.

reads the query from stream 0 (st.ins[0]) and data chunks from stream 1 (st.ins[1]), and subsamples the received data chunks using the `zoom_chunk` function. The zoom filter uses scratch space to store results during subsampling and to pack the subsampled chunk into the output buffer. The result is written to the output stream (st.outs[0]), which connects the filters *zoom* and *view*.

5.3. Experimental Results

Using the filters described in Section 5.2, we have implemented a simple data server for digitized microscopy images [6], stored in the IBM HPSS archival storage system at the University of Maryland. An existing Virtual Microscope client trace driver was used to drive the experiments. This driver was always executed on the same host as the *view* filter, which is referred to as the client host. The server host is where the *read_data* filter is run, which is the machine containing the disks with the dataset.

The HPSS setup has 10TB of tape storage space, 500GB of disk cache, and is accessed through a 10-node IBM SP. In



Filter	*Total Volume*	*Volume Per Chunk*
read data	3.60 MB	102.52 KB
decompress	83.42 MB	2373.04 KB
clip	57.83 MB	1645.02 KB
zoom (no)	57.83 MB	1645.02 KB
zoom (8)	.90 MB	25.70 KB

Figure 7. 2D dataset and query regions. The table lists transmitted data sizes for q_5. zoom (no) is for no subsampling and zoom (8) is for a subsampling factor of 8 (in each of the two spatial dimensions).

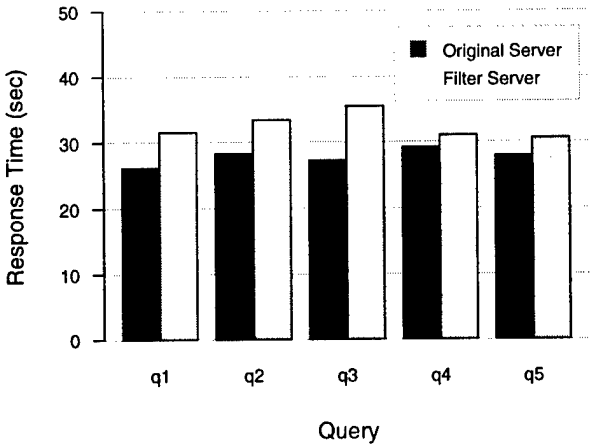


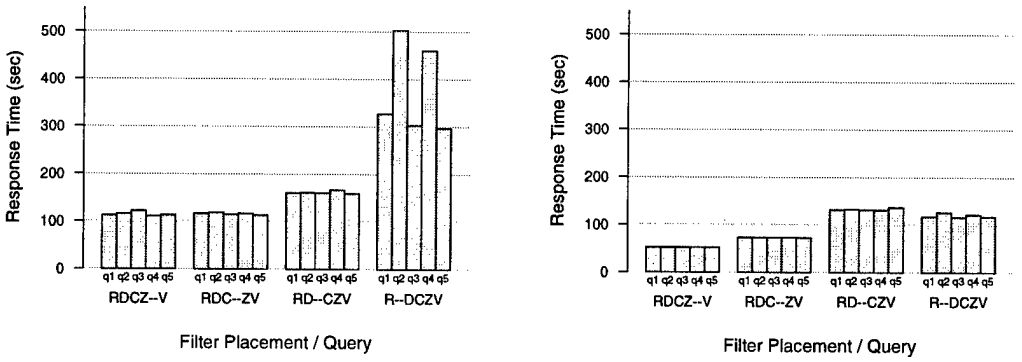
Figure 8. Query execution times for the original server and the filter implementation. (subsampling factor is 8)

all experiments we use a 4GB 2D compressed JPEG image dataset (90GB uncompressed), created by stitching together smaller digitized microscopy images. This dataset is equivalent to a digitized slide with a single focal plane that has $180K \times 180K$ RGB pixels. The 2D image is regularly partitioned into 200×200 data chunks and stored in HPSS in a set of files. We defined five possible queries, each of which covers 5×5 chunks of the image (see Figure 7). The execution times we will show are response times seen by the visualization client averaged over 5 repeated runs. For the presented experiments, we eliminated the effects of retrieving data stored on tape by insuring the data was staged to the HPSS disk cache before each run. We are using machines on our local area network for experimental repeatability, and will switch to hosts in a wide-area Grid environment once application behavior is sufficiently well-understood.

Overhead of Using Filters. The query execution times for the original optimized Virtual Microscope server versus the prototype filter implementation are shown in Figure 8. In this experiment the entire dataset is loaded from HPSS and stored on a single local disk attached to a SUN Ultra 1 workstation, because the original server can only access datasets stored on disks. The loading of the dataset took 4750 seconds (1 hour 19 minutes). The original server is run as a single process, and all filters in the filter-stream implementation are executed on the same uniprocessor SUN workstation where the dataset has been pre-loaded. In both cases the client is run on another SUN Ultra 1 workstation connected to the local Ethernet segment. As is seen from the figure, the filter implementation does not introduce much overhead compared to the optimized original server. The percent increase in query execution time ranges from 6% to 30% across all queries. The filter version contains extra work not present in the original server, such as flattening of the chunk and metadata into a linear buffer on the sending filter, and expanding the chunk and metadata into the same structure in the receiving filter. This overhead is necessary when filters are located on distributed machines, but could be eliminated for the co-located case by instead sending a pointer to an in-memory structure, which would eliminate much of the overhead. This experiment is designed to be biased against the filter implementation to see what the overhead is in the decomposed version. We should also note that the timings do not include the time for loading the dataset from tape, which can substantially increase for larger datasets and datasets stored in archival storage systems across a wide-area network.

Varying the Processing. One node of the IBM SP is used to access the stored dataset, and the client was run on a SUN workstation connected to the SP node through the department Ethernet. We experimented with different placements of the filters by running some of the filters on the same SP node where the data is accessed, as well as on the SUN workstation where the client is run.

In Figure 9 we consider varying the placement of the filters under different processing requirements. Fig-



(a) no zooming (no subsampling)

(b) subsampling by a factor of 8

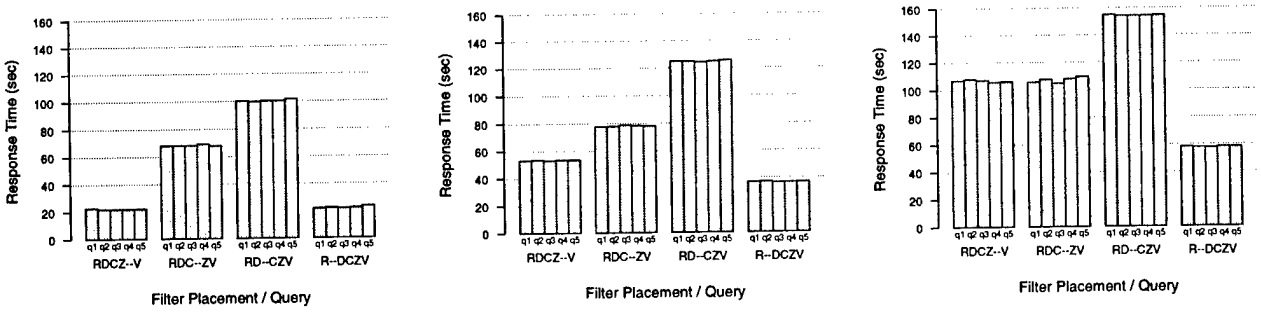
Figure 9. Execution time of queries under varying processing (subsampling). R,D,C,Z,V denote the filters *read_data*, *decompress*, *clip*, *zoom*, and *view*, respectively. <server>—<client> denotes the placement of the filters in each set.

ures 9(a) and (b) show the query execution times when the image is viewed at the highest magnification (no subsampling) and when the subsampling factor is 8 (i.e. every 8th pixel in each dimension is output), respectively. There are three predominant factors in these experiments. The first is the placement of the most computationally intensive filter (*decompress*). The second is the volume of data transmitted between the two machines. The final factor is the amount of data sent by the *view* filter to the client driver. Consider the first two groups of bars in the figures. The difference between the groups within each figure is the placement of the *zoom* filter on the server (RDCZ-V) or client host (RDC-ZV). When there is no subsampling, query execution times remain almost the same for both placements, because the volume of data transfer between the server and client is the same in both cases. In the case of subsampling, the placement of the zoom filter makes a difference, because the volume of data sent from the server to the client decreases if the zoom filter is executed at the server. Now consider the last two groups of bars in the figures. The difference between the groups within each figure is the placement of the *decompress* filter (RD-CZV or R-DCZV). For no subsampling case, the time increases substantially when decompress is placed on the client, because of the combined effects of the most computationally intensive filter (*decompress*) and the high amount of data being processed by *view* and sent to the client driver. When there is subsampling, the query execution time is not as high, because the amount of data processed by *view* and sent to the client driver is much lower. These experiments demonstrate the complex interactions between placement of computation and communication volume.

Varying the Server Load. In the next set of experiments (Figure 10), we consider varying the server load. We use the same experimental setup as for the previous experiment. In all experiments, we use a subsampling factor of 8. Figures 10(a), (b), and (c) show query execution times when the server load is doubled, tripled, and quintupled, respectively. The different loads were emulated by artificially slowing down the set of filters running on the server host such that the total running time was delayed. For example, the *zoom* filter runs twice as long in the $2\times$ case because the time is delayed. As server load increases (or the client host becomes relatively faster), running the filters on the client host achieves better performance. This result is not unexpected, but the experiment quantifies the effect for this particular configuration. The use of a different client to server network, or hosts with different relative speeds would significantly change the observed trends and trade-off points.

6. Application: External Sort

External sort has a long history of research in the database community and has resulted in many fast algorithms [3, 5]. The application starts with a large unsorted data file that is partitioned across multiple nodes, and the output is a new partitioned data file that contains the same data sorted on a key field. The sample data file is based on a standard sorting benchmark that specifies 100 byte tuples, with the first 10 bytes being the sort key. The distribution of the key values is assumed to be uniform, both in terms of the unsorted file as a whole and for each partition. A recent record holder



(a) 2x server load

(b) 3x server load

(c) 5x server load

Figure 10. Execution time of queries under varying server load. $2\times$ means the server computation is delayed to double the execution time of a filter on the server, etc. (subsampling factor is 8)

for the fastest external sort is NowSort [5], and we use the pipelined version of their two-pass parallel sort for our basic algorithm.

The algorithm proceeds in two phases. The first phase generates temporary sorted runs on each node, and the second phase produces the output sorted partition on each node. During the first phase, a *reader* reads chunks of tuples from the unsorted input file on disk, and partitions the records according to which node it will reside on when sorted, puts them into in-memory buffers, and when a buffer is full, sends it to the correct node. A *writer* collects tuples from all nodes, and when the in-memory buffer is full, sorts it using partial-radix sort[2], and writes the sorted run to disk. This first phase is over when all the unsorted input files have been processed, and written to disk as temporary sorted runs. For the second phase a *merge-reader* reads tuples from each local sorted run into merge input buffers. A *merger* selects the lowest-value key among all merge input buffers and copies it to an output buffer, from which the *merge-writer* appends buffers to the sorted output file on disk. This phase completes when tuples from all local runs have been merged.

6.1. Filter Implementation

The implementation of external sort using filters follows the above description. The location of the unsorted dataset dictates the number of nodes to be used for execution. There are two filters on each node, *Partitioner* and *Sorter*. The *Partitioner* filter reads buffers from the unsorted input file, and distributes the tuples into buckets based on the key value. When a bucket is full, it is sent over the stream that connects to the *Sorter* filter on the corresponding node. The *Sorter* continually reads buffers from the input streams, and extracts a portion of the key and appends it to a sort buffer. When the sort buffer becomes full, it is sorted and written to scratch space as a single temporary run. When all buffers have been read from the input streams, the merge phase begins with only the *Sorter* filters still executing. The *Sorter* filter then reads sorted tuples from each of the temporary run files and merges them into a single output buffer, and writes this buffer to the sorted output file on disk.

This application is essentially a parallel SPMD program, with an all-to-all communication pattern. This organization is in contrast to the Virtual Microscope application that was structured as a processing chain pipeline.

6.2. Experimental Results

The experimental setup is a 16 node cluster of dual 400MHz Pentium IIs with 256MB memory per node, running Linux kernel 2.2.12. There are two interconnects, a shared Ethernet segment, and a switched gigabit Ethernet channel. We use the faster switched interconnect for all experiments, and because of a problem with the network interface cards on some of the nodes, only use a maximum of 8 nodes in all experiments. The nodes are isolated from the rest of the network, and the cluster was not running other jobs during the experiments. Each node has a single Ultra2 SCSI disk. All data for a particular node, including temporary data, is stored on the single local disk. The dataset consists of a single 128MB unsorted file per node. The unsorted dataset was generated randomly with a uniform key distribution. The execution time for an experiment is the maximum time across all nodes used for the experiment. Each experiment is repeated for 5 trials, and the execution time shown represents the average of the trials. Both a *Partitioner* and

[2]Making two passes over the keys with a radix size of 11-bits [3] plus a cleanup.

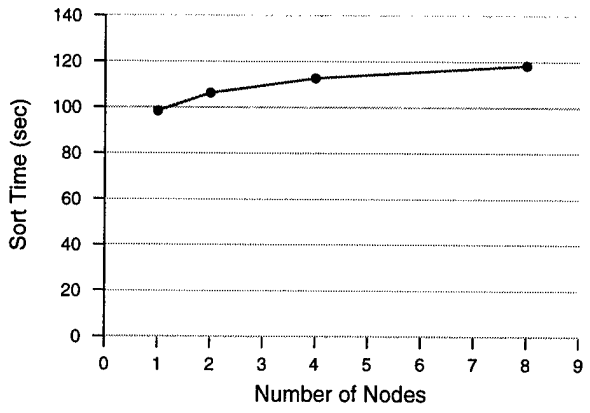


Figure 11. Sort execution time as number of nodes is increased. The dataset size is scaled with the number of nodes (128MB/node).

Filter	*Parameter*	*Full Memory*
Partitioner	read_size	256 KB
single disk buffer for reading tuples from the unsorted input file		
Partitioner	bucket_size	1 MB
shared space for all outgoing tuple buckets, before sending to Sorter filters		
Sorter (phase 1)	keybuf_size	1 MB
single buffer for storing extracted key and tuple pointer, before sorting and writing the temporary run		
Sorter (phase 2)	sharedbuf	768 KB
shared disk buffer for reading from all temporary runs during merge		
Sorter (phase 2)	outputbuf	512 KB
single disk buffer for writing sorted tuples to output file		

Figure 12. Memory parameters used by the sort filters. The *Full Memory* column contains the initial value for each parameter.

a *Sorter* filter are executed on each node used in the experiments for Figures 11 and Figures 13(a)–(d), and two Partitioner and Sorter filters are executed on some of the nodes in the experiments for Figures 13(e) and (f). The disk cache was cleared between executions to insure a cold disk cache for each run. Note that we are using a tightly coupled cluster for experimental repeatability, and will be switching to hosts on a wide-area Grid environment when application behavior is better understood.

Scaling. The first experiment examines the scalability of the sort application as we increase the number of nodes and total dataset size. As seen in Figure 11, the application is well-behaved. There is good scaling due to the fast interconnect not becoming saturated by the traffic generated by sort. This experiment demonstrates there is nothing inherent in the filter-stream based implementation that would otherwise limit its scalability.

Varying Memory Size. In this set of experiments we vary the amount of memory available for filters on some of the nodes while keeping it constant for filters on the remaining nodes. Our goal is to create a heterogeneous configuration in a controlled way, and observe the effects of heterogeneity on the application performance.

Figure 13 shows the execution times under varying memory constraints. The solid line in all of the graphs denotes the *base case*, in which the size of the memory is reduced equally across all nodes, and shows the change in the execution time. The amount of the *Full* memory case is determined empirically to minimize execution time while consuming the least memory (see Figure 12). Memory parameters are varied by halving the full memory amount for the 1/2 case, and halving again for the 1/4 case, etc. Constraining memory causes the filters to read/process/write data in smaller pieces, thus performance should suffer. As is shown by the solid line in the figure, the execution time increases as the size of the memory is decreased. In the experiments with heterogeneous memory configuration, we divide the eight nodes into two sets of four nodes. The first set of nodes retains the initial amount of memory (i.e., *Full* memory) for all runs, while the second set has their memory reduced for each case. The left bars for each case in each graph shows the maximum of the execution times on the nodes with full memory. Similarly, the right bar for each case in each graph shows the maximum of the execution times on the nodes with reduced memory. As is shown in Figure 13(a), we observe performance degradation similar to the base case. The nodes that use a constant amount of memory finish sooner, but the entire job runs no faster. In this experiment, both the input data to the *Partitioner* filter and the output of the Partitioner (i.e. the input data to the *Sorter* filter) on each node are regularly partitioned across all the nodes.

Notice that the total amount of memory across all nodes for this experiment is larger than that for the base case because half the nodes keep full memory. For example, for the 1/8 memory case, 350% more memory was being used in aggregate than for the 1/8 base case. Instead of a reduction in sort time, the extra memory results in a load imbalance between the two sets of four nodes. Hence, in the next experiment we partitioned the amount of input data for each node irregularly, to attempt to reduce overall execution time. Figure 13(b) shows that the execution time increases when we partition the input data based on available node memory, i.e., full nodes have more input data than nodes with reduced memory. This results from an increase in the time for the partitioning phase, because the Partitioner filters on the set of nodes with full memory have more input records to process. The execution time for the merge phase is effec-

tively unchanged, because the amount of data sent to each node is unchanged. Figure 13(c) shows the result of partitioning the output of the Partitioner filter (and thus the merge phase work) according to the memory usage of the receiving node. This experiment, however, moves too much work to the nodes with full memory, so that those nodes become the longest running node set. To improve performance further, we followed two different approaches. In Figure 13(d), the Partitioner filter output is adjusted to balance the performance of both sets of nodes (approximately a 10% reduction in the number of tuples assigned to a node for each 1/2 reduction in memory usage). In this case, we observe better performance than the previous cases. In the second approach, we partitioned both the input data and the output of the Partitioner filter as was done in the experiment for Figure 13(c), but executed two Sorter and two Partitioner filters on the nodes with Full memory to take advantage of the dual processors available in each node. As is seen in Figure 13(e), the performance is better than for the previous cases. Finally, Figure 13(f) shows the combined effect of running two sets of filters on the nodes with full memory, and adjusting the Partitioner output to balance the workload across both set of nodes. As expected, this configuration performs better than all other cases. These experimental results clearly show that application-level workload handling and careful placement of filters can deal with heterogeneity, which can have a significant impact on performance. Questions that require further investigation include (1) "can we develop cost models for filters and for the application performance so that the placement of filters and workload handling can be done by the runtime system, with little intervention from the user?" and (2) "can we make use of exposing resource requirements and communication characteristics to develop accurate and efficient cost models?". We plan to work on more applications and different configurations to seek answers to these questions in future work.

7. Conclusion and Future Work

We have presented a framework, called filter-stream programming, for developing data-intensive applications in a Grid environment. This framework represents the processing in an application as a set of processing components, called filters. The goal is to constraint application components to allow for location independence, and to expose communication characteristics and resource requirements, thus enabling a runtime system to support efficient execution of the application. We have described a prototype runtime infrastructure to execute applications using the filter-stream programming framework. We have discussed implementations of two data-intensive applications that make use of our filter-stream framework, and presented experimental results.

Our experimental results show that there exists a delicate balance, and sometimes subtle interactions with heterogeneous resources, that can have a large impact on application performance. We plan to further investigate such interactions to develop cost models that can aid in decomposition of applications into filters and placement of the filters. We also are in the process of implementing other applications to use the filter-stream programming framework from application areas such as volume visualization, database decision support, and image processing.

References

[1] A. Acharya, M. Uysal, and J. Saltz. Active disks: Programming model, algorithms and evaluation. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VIII)*, pages 81–91. ACM Press, October 1998. ACM SIGPLAN Notices, Vol. 33, No. 11.

[2] A. Afework, M. D. Beynon, F. Bustamante, A. Demarzo, R. Ferreira, R. Miller, M. Silberman, J. Saltz, A. Sussman, and H. Tsang. Digital dynamic telepathology - the Virtual Microscope. In *Proceedings of the 1998 AMIA Annual Fall Symposium*. American Medical Informatics Association, November 1998.

[3] R. Agarwal. A super scalar sort algorithm for RISC processors. In *Proceedings of 1996 ACM SIGMOD Conference*, pages 240–6, 1996.

[4] Alexandria Digital Library. *http://alexandria.ucsb.edu/*.

[5] A. Arpaci-Dusseau, R. Arpaci-Dusseau, D. Culler, J. Hellerstein, and D. Patterson. High-performance sorting on networks of workstations. In *Proceedings of 1997 ACM SIGMOD Conference*, Tucson, AZ, 1997.

[6] M. D. Beynon, R. Ferreira, T. Kurc, A. Sussman, and J. Saltz. DataCutter: Middleware for filtering very large scientific datasets on archival storage systems. In *Proceedings of the 2000 Mass Storage Systems Conference*, College Park, MD, March 2000. IEEE Computer Society Press. To appear.

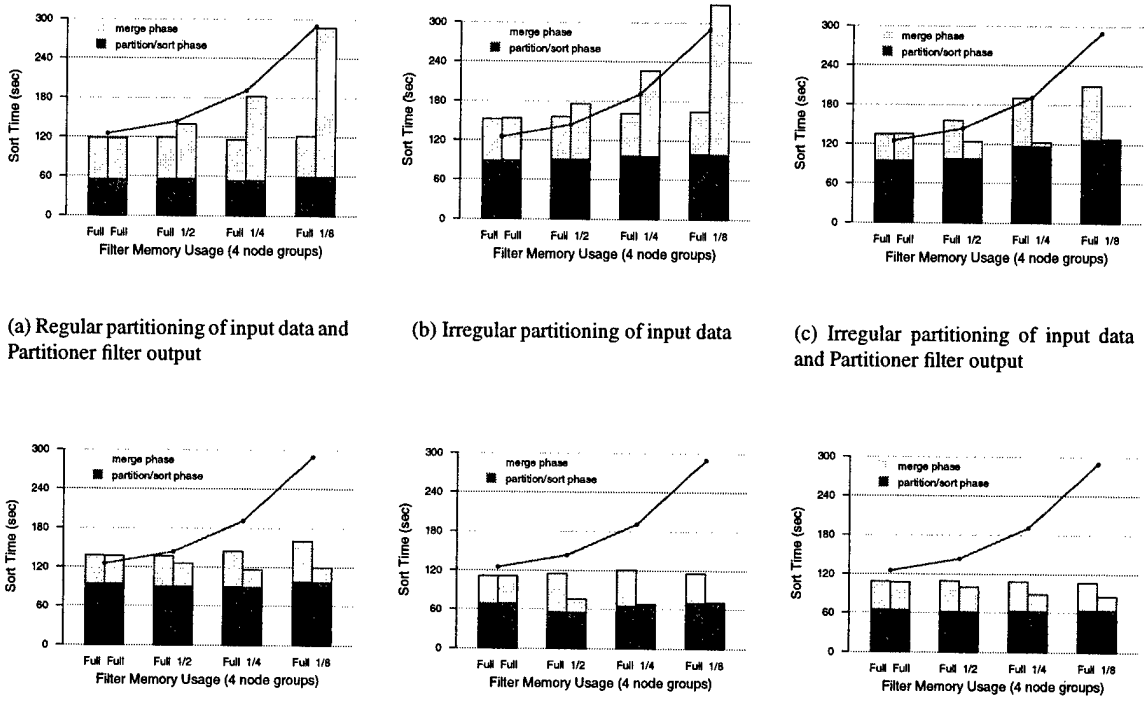
[7] A. Chervenak, I. Foster, C. Kesselman, C. Salisbury, and S. Tuecke. The Data Grid: Towards an architecture for the distributed management and analysis of large scientific datasets. (Submitted to NetStore '99), September 1999.

[8] A. Choudhary, R. Bordawekar, M. Harry, R. Krishnaiyer, R. Ponnusamy, T. Singh, and R. Thakur. PASSION: Parallel and scalable software for input-output. Technical Report SCCS-636, NPAC, September 1994. Also available as CRPC Report CRPC-TR94483.

[9] Phoenix Computers Catalog. *http://www.phoenixcomputers.net/*, Jan 2000. Following link from *http://www.pricewatch.com/*.

[10] ShopHere USA Catalog. *http://205.164.161.100/products.asp?dept=33*, Jan 2000. Following link from *http://www.pricewatch.com/*.

[11] F. Douglis and J. Ousterhout. Process migration in the Sprite operating system. In *Proceedings of the Seventh International Conference on Distributed Computing Systems*, pages 18–25, September 1987.



(a) Regular partitioning of input data and Partitioner filter output

(b) Irregular partitioning of input data

(c) Irregular partitioning of input data and Partitioner filter output

(d) Irregular partitioning of input data and Partitioner filter output (tuned)

(e) Irregular partitioning of input data and Partitioner output, 2 pair of filters per Full node

(f) Irregular partitioning of input data and Partitioner filter output (tuned), 2 pair of filters per Full node

Figure 13. Execution time of sorting a 1GB dataset using 8 nodes. Two groups of four nodes: (1) memory usage held constant (Full), (2) memory usage reduced (1/X).

[12] I. Foster. Automatic generation of self-scheduling programs. *IEEE Transactions on Parallel and Distributed Systems*, 2(1):68–78, January 1991.

[13] I. Foster. Compositional parallel programming languages. *ACM Trans. Prog. Lang. Syst.*, 18(4):454–476, July 1996.

[14] I. Foster. The Beta Grid: A national infrastructure for computer systems research. In *Network Storage Symposium*, October 1999.

[15] I. Foster and C. Kesselman. Globus: A metacomputing infrastructure toolkit. *International Journal of Supercomputer Applications and High Performance Computing*, 11(2):115–128, 1997.

[16] I. Foster and C. Kesselman. *The GRID: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, 1999.

[17] I. Foster, C. Kesselman, and S. Tuecke. The Nexus approach to integrating multithreading and communication. *Journal of Parallel and Distributed Computing*, 37:70–82, 1996.

[18] I. Foster, D. Kohr, R. Krishnaiyer, and A. Choudary. Double standards: Bringing task parallelisn to HPF via the message passing interface. In *Proceedings Supercomputing '96*. IEEE Computer Society Press, November 1996.

[19] I. Foster, D. Kohr, R. Krishnaiyer, and J. Mogill. Remote I/O: Fast access to distant storage. In *Fifth Workshop on I/O in Parallel and Distributed Systems (IOPADS)*, pages 14–25. ACM Press, 1997.

[20] Geographic Information Systems. *http://www.usgs.gov/research/gis/title.html*.

[21] Grid Forum. Birds-of-a-Feather Session, SC99, Nov 1999.

[22] The High Performance Storage System (HPSS). *http://www.sdsc.edu/hpss/hpss1.html*.

[23] T. Johnson. An architecture for using tertiary storage in a data warehouse. In *the Sixth NASA Goddard Space Flight Center Conference on Mass Storage Systems and Technologies, Fifteenth IEEE Symposium on Mass Storage Systems*, 1998.

[24] W. Johnston and B. Tierney. A distributed parallel storage architecture and its potential application within EOSDIS. In *the Fourth NASA Goddard Conference on Mass Storage Systems and Technologies*, 1995.

[25] Land Satellite Thematic Mapper (TM). *http://edcwww.cr.usgs.gov/nsdi/html/landsat_tm/ landsat_tm*.

[26] K.-L. Ma and Z. Zheng. 3D visualization of unsteady 2D airplane wake vortices. In *Proceedings of Visualization'94*, pages 124–31, Oct 1994.

[27] Message Passing Interface Forum. Document for a standard message-passing interface. Technical Report CS-93-214, University of Tennessee, November 1993.

[28] Microsoft Corp. Microsoft TerraServer. http://www.terraserver.microsoft.com, 1998.

[29] G. Patnaik, K. Kailasnath, and E. Oran. Effect of gravity on flame instabilities in premixed gases. *AIAA Journal*, 29(12):2141–8, Dec 1991.

[30] V. Sarkar. Partitioning and scheduling parallel programs for multiprocessors. In *Research Monographs in Parallel and Distributed Computing*. Cambridge, MA, 1989.

[31] SRB: The Storage Resource Broker. *http://www.npaci.edu/DICE/SRB/index.html*.

[32] N. Talagala, S. Asami, and D. Patterson. The Berkeley-San Francisco fine arts image database. In *the Sixth NASA Goddard Space Flight Center Conference on Mass Storage Systems and Technologies, Fifteenth IEEE Symposium on Mass Storage Systems*, 1998.

[33] T. Tanaka. Configurations of the solar wind flow and magnetic field around the planets with no magnetic field: calculation by a new MHD. *Jounal of Geophysical Research*, 98(A10):17251–62, Oct 1993.

[34] M. Teller and P. Rutherford. Petabyte file systems based on tertiary storage. In *the Sixth NASA Goddard Space Flight Center Conference on Mass Storage Systems and Technologies, Fifteenth IEEE Symposium on Mass Storage Systems*, 1998.

[35] M. Uysal. *Programming Model, Algorithms, and Performance Evaluation of Active Disks*. PhD thesis, Department of Computer Science, University of Maryland, College Park, 1999.

[36] M.-Y. Wu and D. D. Gajski. Hypertool: A programming aid for multicomputers. In *Proceedings of the 1989 International Conference on Parallel Processing*, pages II–15 – II–18. Pennsylvania State University Press, August 1989.

[37] T. Yang and A. Gerasoulis. PYRROS: static task scheduling and code generation. In *Proceedings of the 1992 International Conference on Supercomputing*, July 1992.

Michael D. Beynon is a Ph.D. candidate in the Department of Computer Science at the University of Maryland, College Park. His research interests include resource intensive application performance and high performance parallel and distributed systems.

Tahsin Kurc is a postdoctoral research associate in the Department of Computer Science at the University of Maryland College Park. His research interests include algorithms and systems software for high-performance and data-intensive computing.

Alan Sussman is a Research Scientist in the Computer Science Department at the University of Maryland College Park and a visiting scientist in Pathology at the Johns Hopkins School of Medicine. His research interests include high performance runtime and compilation systems for data- and compute-intensive applications.

Joel Saltz is a professor of Computer Science at the University of Maryland College Park and a professor of Pathology at the Johns Hopkins School of Medicine. His research interests are in the development of systems sofware, databases and compilers for the management, processing and exploration of very large datasets.

SESSION 2-B
MODELING AND METRICS

Chair: M. Baker, *University of Portsmouth, UK*

Toward Quality of Security Service in a Resource Management System Benefit Function

Cynthia E. Irvine
Department of Computer Science
Naval Postgraduate School
Monterey, CA 93943 USA
email: irvine@cs.nps.navy.mil

Timothy E. Levin
Anteon Corporation
2600 Garden Road
Monterey, CA 93940 USA
email: levin@cs.nps.navy.mil

Abstract

Enforcement of a high-level statement of security policy may be difficult to discern when mapped through functional requirements to a myriad of possible security services and mechanisms in a highly complex, networked environment. A method for articulating network security functional requirements, and their fulfillment, is presented. Using this method, security in a quality of service framework is discussed in terms of "variant" security mechanisms and dynamic security policies. For illustration, it is shown how this method can be used to represent Quality of Security Service (QoSS) in a network scheduler benefit function[1].

1 Introduction

Several efforts are underway to develop middleware systems that will logically combine network resources to construct a "virtual" computational system [4] [7] [8] [15] . These geographically distributed, heterogeneous resources are expected to be used to support a heterogeneous mix of applications. Collections of tasks with disparate computation requirements will need to be efficiently scheduled for remote execution. Large parallelized computations found in fields such as astrophysics [14] and meteorology will require allocation of perhaps hundreds of individual processes to underlying systems. Multimedia applications, such as voice and video will impose requirements for low jitter, minimal packet losses, and isochronal data rates. Adaptive applications will need information about their environment so they can adjust to changing conditions.

User acceptance of these virtual systems, for either commercial or military applications, will depend, in part, upon the security, adaptability, and user-responsiveness provided. Several of the projects engaged in building the middleware to create these networks are pursuing the integration of security [6] [10] [23] and quality of service [1] [17] into these systems. The need for virtual networked systems to both adapt to varying security conditions, and offer the user a range of security choices is apparent.

In the network computing context, users or user programs may request the execution of "jobs," which are scheduled by an underlying control program to execute on local or remote computing resources. The execution of the job may access or consume a variety of network resources, such as: local I/O device bandwidth, internetwork bandwidth; local and remote CPU time; local, intermediate (e.g., routing buffers) and remote storage. The resource usages may be temporary or persistent. As there are multiple users accessing the same resources, there are naturally various allotment, contention, and security issues regarding use of those resources.

The body of rules for resolving network security issues is called the network security policy, whereas the body of rules for resolving network contention and allotment comprise a network management policy (which is sometimes taken to include the network security policy). These policies consist of broad policy jurisdictions, such as scheduling, routing, access control, auditing, and authentication. Furthermore, these jurisdictions can be decomposed, typically, into functional requirements, such as, "users from network domain A must not access site B," and "user C must receive a certain quality of service." The network management and security policies, as mapped through the functional requirements, may be manifested in mechanisms throughout the network, including: host computer operating systems, network managers, traffic shapers, schedulers, routers, switches and combinations thereof. As these mechanisms are distributed and are often obscurely related, there has been some interest in the ability to express and quantify the level of support for security policy and Quality of Security Service (QoSS: managing security and security requests as a responsive "service" for which

1. This work was supported by the DARPA/ITO Quorum program.

0-7695-0556-2/00 $10.00 © 2000 IEEE

quantitative measurement of service "efficiency" is possible) provided in networked systems.

The purpose of this paper is to present the system developed for the MSHN resource management system [8] for describing network security policy functional requirements, to show how QoSS parameters and mechanisms can be represented in such a system, and to provide an example of the use of this system. The remainder of this paper is organized as follows. Section 2 discusses a "security vector" for quantifying functional support of network security policy. Section 3 describes how the security vector can be used for expressing the effects of QoSS in a network-scheduling benefit function; and a conclusion follows in Section 4.

2 Network Security Vector

A *network security policy* can be viewed as an n-dimensional space of functional security requirements. We represent this multidimensional space with a *vector* (S) of security components. Each component (S.c) specifies a boolean functional requirement, whereby the instantiation of a network job either meets (possibly trivially) or does not meet each of the requirements. By convention, a security vector's components are ordered, so they can be referenced ordinally (S.3) or symbolically (S.c). A component may indicate positive requirements (e.g., communications via node n must use encryption) as well as negative constraints (e.g., users from subnet s may not use node n). Components can also be hierarchically grouped. [22] Requirements for a given security service may be represented by one or more components (indicating a service *sub-vector*), and a security service may utilize functions and requirements of other services and their components.

Some jobs can produce output in different *formats*, where a given format (e.g., high resolution video) might be more resource consumptive than another format (e.g., low resolution video). Formats may have differing security requirements, even within the same job. For example, a video-stream format may require less packet authentication [19] , percentage-wise, than a series of fixed images based on the same data. A "quality of service" scheduling mechanism might choose one format for a job over another, depending on varying network conditions (e.g., traffic congestion). Further, adaptive applications may select formats depending upon changing conditions. For example, IPSec, security association (SA) processing using ISAKMP under IKE can permit complex security choices through an SA payload; and the payload recipient may be given transform choices regarding both Authentication Header and Encapsulating Security Protocol [13] .

2.1 Notation

The set of all jobs is represented by J. The set of all formats is represented by I. The notation S_{ij} identifies a vector containing the portions of S that are applicable to job j in format i, and $S_{ij}.c$ identifies a given component (c) of S_{ij}. The relation of S to S_{ij} is clarified further, below. The following are some informal examples of security-vector components:

- S.1: user access to resource is equal to read/write; based on table t
- S.2: % of packets authenticated >= 50, <= 90; inc 10
- S.3: clearance (user) = secrecy/integrity (resource)
- S.4: length of confidentiality encryption key >= 64, <= 256; inc 64
- S.5: authentication header transform *in* {HMAC-MD5, HMAC-SHA}
- S.6: packets from domain A to domain B must be encrypted
- S.7: packets from domain A cannot be sent through domain C

Here, "inc 10" indicates that the range from 50 through 90 is quantized into increments of 10, viz: 50, 60, 70, 80, 90. Later, we will need to indicate the number of quantized steps in the component; to do this, one more notational element is introduced, $[S.c]$. In the above examples, $[S.1] = 1$, and $[S.2] = 5$.

2.2 Variant Security Components

When [S.c] > 1, the underlying control program has a range within which it may allow the job to execute with respect to the policy requirement. We refer to this type of policy, and component, as "variant." Security-variant policies may be used within a resource management context, for example, to effect adaptation to varying network conditions. [18] Also, if the policy mechanism is variant, the control program may offer QoSS choices to the users to indicate their preferences with respect to a given job or jobs. Without variant mechanisms, neither security adaptability by the underlying control program nor QoSS are possible, since fixed policy mechanisms do not allow for changes to security within a fixed job/resource environment. While the expression S.c may contain a compound boolean statement (see Section 2.3), by convention it may contain only one variant clause.

2.3 Component Structure

For use in the examples in this discussion, a component has the following composition (see Table 1 for details):

- component ::= boolean expression, variant-range-specifier ; modifying-clause
- boolean _expression ::= boolean_statement [(or | and) boolean_statement]*
- boolean_statement ::= LHS boolean-operator RHS

Note that it is not the focus here to elaborate on a policy representation language. See other efforts and works in progress [2] [3] [5] [16].

A given policy component has a *value* which is a boolean *expression.* This component may also have an *instantiated value* with respect to a specific job and format, which is either "true" or "false." A component has a left hand side (LHS), which is the item that is being tested; of course the LHS has a *value* as well as an *instantiated value.* A component also has a right hand side (RHS), which is what the LHS is tested against, as well as zero or more modifying clauses. Similarly to the LHS, the RHS may have a value (or values) which is dependent on the instantiation of the component.

2.4 Dynamic Security Policies

With a dynamic security policy, the value of a vector's components may depend on the network "mode" (e.g., normal, impacted, emergency, etc.), where M is the set of all modes. There is, conceptually, a separate vector for each operational mode, represented as: S^{mode}. Access to a predefined set of alternate security policies allows their functional requirements and implementation mechanisms to be examined with respect to the overall policy prior to being fielded, rather than depending on *ad hoc* methods, for example, during an emergency.

Initially, every component of S has the same value in each of its modes. Ultimately, components may be assigned different values, depending on the network mode. For example:

- $S^{normal}.a$: % packets authenticated >= 50, <= 90; inc 10
- $S^{impacted}.a$: % of packets authenticated >= 20, <= 50; inc 10
  Note how [*S.a*] changes from 5 to 4 under the impacted mode
- S^{normal}.b: user access to network node = granted; based on table t
- $S^{impacted}$.b: user access to network node = granted; based on table t, OR UID in set of administrators
- $S^{emergency}$.b: UID in set of {administrators, policymakers}

Or, for example, policy makers might decide that the policy should remain in force regardless of network mode:

- $S^{normal}.c = S^{impacted}.c = S^{emergency}.c$: clearance (user) = classification (resource)

Table 1: Simple Component Elements

Element Name	Example S.1	Example S.2
Value	user access to resource r = RW, based on table t	% of packets authenticated >= 50, <= 90; inc 10
Instantiated value	false	true
Value of LHS	user access to resource r	% of packets authenticated
Instantiated value of LHS	W	70
Boolean operator	=	>=
Value of RHS	RW	50
variant range specifier	none applicable	<= 90
Modifying clause	based on table t	inc 10

If a mode is not specified for a component (e.g., "S.a"), normal mode is assumed. This will be the case (i.e., the mode is unspecified) for the remainder of this discussion.

2.5 Refinements to Security Vector

R is the set of resources $\{r_1.. r_n\}$. R_{ij} is the subset of R utilized in executing job j in format i.

T_j is the requested completion time of job j.

Security policies may be expressed with respect to principals (user, group or role, etc.,), applications, data sets (both destination and source), formats, etc., as well as resources in R_{ij}.

The definition of S_{ij} is finally refined as follows: S_{ij} is a vector that is an order-preserving projection of S, such that a component c from S is in S_{ij} if and only if the value of c depends on format i, job j, or any r in R_{ij}. The number of components in a security vector S_{ij} *is* $[S_{ij}]$.

2.6 Summary of Security Vector

S is a general purpose notational system suitable for expressing arbitrarily complex sets of network security mechanisms. S can express variant policies, to allow security expressions of quality of service requests, and can have dynamic security elements to accommodate multiple situation-based policies. In particular, S can represent both (1) static security requirements that may be implemented in a system, as well as (2) the results of running a particular job or set of jobs against such static requirements. The latter usage will be examined in the next section, to express QoSS in a resource management system benefit function.

3 Network-Scheduler Benefit Function

As discussed above, various mechanisms exist for managing contention for, and allotment of distributed network resources. One class of these mechanisms attempts to efficiently schedule the execution of multiple (possibly simultaneous) jobs on multiple distributed computers (e.g., the MSHN project [8] [23] [24] [11] [17]), where each job requires a determinable subset of the resources. Of interest is a benefit function for comparing the effectiveness of such job scheduling mechanisms when they are presented with real or hypothetical "data sets" of jobs.

Jobs are assigned *priorities* for use in resolving resource contention and allocation issues. In some systems, a job's priority may depend upon the particular operating *mode* of the network. [8] Also, the different data formats of a multiple-format job may have different *preferences* (e.g., assigned by a user or "hard wired" as part of the application or job-scheduler database), and different levels of resource usage. [10] [12] A network job scheduler should receive more credit in the benefit function for scheduling high priority and high preference jobs, as opposed to low priority or low preference jobs. That is to say, a scheduler is intuitively doing a better job if important jobs, as judged by priority and preference, receive more attention than unimportant jobs. How much weight the priorities and preferences are given is a matter of network scheduling policy.

For illustration, we introduce a simple benefit function, B, to measure how well a scheduler meets the goals of user preference and system priorities (see [4] , [12] and [21] for other approaches). This function averages preference (p) and priority (P) (use of a priority and preference in measuring network effectiveness have been introduced for the MSHN project [10]).

$$B = \frac{\sum_{j=1}^{n} \sum_{i=1}^{m_j} X_{ij}(P_{ij} + p_{ij})}{2n}$$

Where the characteristic function X is defined for i, j as:

$Xij = 1$ if format i was successfully delivered to job j within time T_j, else 0

and at most one format is completed per job:

$$\forall j \in J \left(\sum_{i=1}^{m_j} X_{ij} \leq 1 \right)$$

Jobs and formats are defined as above.

P_j is the priority of job j

$$0 \leq P_j \leq 1$$

The formats for a job are assigned preferences (p) by the user such that:

$0 <= p <= 1$

m_j is the number of {format, preference} pairs assigned for job j

p_{ij} is the preference the user has assigned to format i, job j

the preferences for a job add up to 1:

$$\forall j \in J : \sum_{i=1}^{m_j} p_{ij} = 1$$

This approach assumes that users will assign preference values that correspond to resource usage, since we want the benefit function to indicate a higher value when the scheduler succeeds in scheduling "harder" jobs [12] .

3.1 Adding Security to the Benefit Function

We wish the benefit function to reflect the effectiveness and restrictions of the security policy. First, we define the characteristic security function Z, for i and j:

$Z_{ij} = 1$ if the instantiated value of all components in S_{ij} are true, else 0

The numerator of the benefit function is multiplied by Z, so that no credit is given for jobs that fail to meet the security requirements:

$$B = \frac{\sum_{j=1}^{n} \sum_{i=1}^{m_j} X_{ij} Z_{ij} (P_j + p_{ij})}{2n}$$

Now, for variant components, we wish to be able to give less credit to the scheduler for fulfilling less "difficult" security requirements. One algorithm for expressing this is for each instantiated component (c) in S_{ij} to be assigned a security completion token (g) where $0 \le g \le 1$. g_c will indicate the completion token corresponding to component $S.c$. g_c is defined to be the "percentage" of $[S.c]$ met or exceeded by the *instantiated value* of the component's LHS (notated as $S.c$"):

$g_c = S.c" / [S.c]$

To illustrate the calculation of g_1, for component S.1:

S.1: % of packets authenticated >= 50, <= 90; inc 10

$[S.1] = 5$ (the number of quanta in *S.1*), $S.1" = 3$ (the job achieves the 3rd quantum (70))

$g_1 = 3/5 = 0.6$

For invariant components, $g = 1$ or $g = 0$. A token (g) whose value is 0 represents a job "failing" the component's security policy. Recall that Z will be 0 when the job/format fails to meet the requirement of any security component, meaning that the function returns no benefit value for that job/format. We introduce a function (A) which averages the tokens of a job:

$A_{ij} = (g_1 + g_2 + .. + g_n)/n$

where n = $[S_{ij}]$ -- the number of components in S_{ij}

and $(0 \le A_{ij} \le 1)$

Averages, such as A, over many different elements can tend to minimize the difference that is seen between different data sets. Therefore, we weight the tokens (g) assigned to individual security components to give more credence to components that are "more important" than others, e.g., reflecting network management policies. Each g_n has a corresponding integer weight (w_n), $w_c \ge 0$. So Aij becomes:

$A_{ij} = (g_1 w_1 + g_2 w_2 + .. + g_n w_n)/(w_1 + w_2 + .. + w_n)$

again $(0 \le A_{ij} \le 1)$

In the final expression of the network benefit function, A is added to the numerator, providing an average of security, priority and preference.

$$B = \frac{\sum_{j=1}^{n} \sum_{i=1}^{m_j} X_{ij} Z_{ij} (P_j + p_{ij} + A_{ij})}{3n}$$

$0 \le B \le 1$, where 1 indicates the maximum scheduling effectiveness.

3.2 Applicability

This technique for quantifying the variant security instantiated by a resource management system is being used in the MSHN project as a factor in representing the effectiveness of its resource assignments [10] . In the MSHN design, the security requirements of network resources (represented by S) are stored in a Resource Requirements Database. This database is consulted during the resource scheduling phase to effectively match jobs to resources. We expect that this measurement technique could also be applied to other resource management systems, such as Condor [15] and Globus [7] .

While different schedulers could be compared with respect to the individual components of B, a summary function such as B would be useful to automate and normalize the comparison process. Additionally, we expect that the security component (viz, A) in an operational system would be complex enough to evade effective manual analysis.

4 Discussion and Conclusion

A security vector has been presented for describing functional requirements of network security policies. It has been shown that this vector can be used for representing security with respect to both quality of service and a network scheduler benefit function.

We are involved in ongoing work to organize the security vector into a "normal form" with sub-vectors or hierarchies corresponding to security policy jurisdictions (such as: access control, auditing, and authentication) and to incorporate a costing methodology for security components, such as can be provided to a resource management system [9] . We are working to develop a means of adjusting the preference expression with a notion of the corresponding resource usage [12] . We are considering how to expand the security benefit function (A) to reflect user qual-

ity of security service choices within the range allowed by variant security components, and to reflect performance implications of redundant security mechanisms.

The organizational security policy [20] governing the network may allow individuals or principals representing them to override rules represented by invariant security vector components. For example, a military commander might decide to forgo cryptographic secrecy mechanisms for a job in an emergency (e.g., to improve network performance), even though the system has not been configured with "dynamic" or "variant" security mechanisms, as defined herein. From the perspective of the security vector S and the benefit function, this is a *change* to *or violation of the computer security policy.* It is recommended that this type of policy change be audited.

References

[1] Aurrecoechea, C., Campbell, A., and Hauw, L. "A Survey of Quality of Service Architectures", Multimedia Systems Journal, Special Issue on QoS Architectures, 1996.

[2] Badger, L., Stern, D. F., Sherman, D. L., Walker, K. M., and Haghighat, S. A., "Practical Domain and Type Enforcement for Unix," Proceedings of 1995 IEEE Symposium on Security and Privacy, 1995, Oakland, Ca., pp. 66-77

[3] Blaze, M., Feigenbaum, J., and Lacy, J., "Decentralized Trust Management," in Proceedings of 1996 IEEE Symposium on Security and Privacy, May 6-8, 1996, Oakland, Ca., pp 164-173

[4] Chatterjee, S., Sabata, B., Sydir, J. "ERDoS QOS Architecture," SRI Technical Report, ITAD-1667-TR-98-075, Menlo Park, CA, May 1998.

[5] Condell, M., Lynn, C. and Zao, J. "Security Policy Specification Language," INTERNET-DRAFT, Network Working Group, July 1, 1999, ftp://ftp.ietf.org/internet-drafts/draft-ietf-ipsec-spsl-01.txt, Expires January, 2000

[6] Foster, I, N. T. Karonis, N. T., Kesselman, C., Tuecke, S. Managing Security in High-Performance Distributed Computing. Cluster Computing 1(1):95-107, 1998.

[7] Foster, I., and Kesselman, C., Globus: A Metacomputing Infrastructure Toolkit. Intl J. Supercomputer Applications, 11(2):115-128, 1997.

[8] Debra Hensgen, Taylor Kidd, David St. John, Matthew C. Schnaidt, H. J. Siegel, Tracy Braun, Jong-Kook Kim, Shoukat Ali, Cynthia Irvine, Tim Levin, Viktor Prasanna, Prashanth Bhat, Richard Freund, and Mike Gherrity, An Overview of the Management System for Heterogeneous Networks (MSHN), 8th Workshop on Heterogeneous Computing Systems (HCW '99), San Juan, Puerto Rico, Apr.1999

[9] Irvine, C., and Levin, T., Toward a Taxonomy and Costing Method for Security Metrics, Annual Computer Security Applications Conference, Phoenix, AZ, Dec. 1999

[10] Kim, Jong-Kook, Hensgen, D., Kidd, T., Siegel, H.J., St.John, D., Irvine, C., Levin, T., Porter, N.W., Prasanna, V., and Freund, R., A QoS Performance Measure Framework for Distributed Heterogeneous Networks, Proceedings of the 8th Euromicro Workshop on Parallel and Distributed Processing, Rhodos, Greece, January 2000.

[11] Lee, C. Kesselman, C., Stepanek, j., Lindell, R., Hwang, S., Scott Michel, B., Bannister, J., Foster, I., and Roy, A. The Quality of Service Component for the Globus Metacomputing System. Proc. 1998 International Workshop on Quality of Service, Napa California, pp. 140-142, May, 1998.

[12] Levin, T., and Irvine C., An Approach to Characterizing Resource Usage and User Preferences in Benefit Functions, NPS Technical Report, NPS-CS-99-005

[13] Maughan, D., Schertler, M., Schneider, M., and Turner, J. Internet Security Association and Key Management Protocol, RFC 2408, http://info.internet.isi.edu/in-notes/rfc/files/rfc2408.txt

[14] Ostriker, J., and Norman. M. L., Cosmology of the Early Universe Viewed Through the New Infrastructure. C.A.C.M. 40(11):85-94.

[15] Raman, R., Livny, M., Solomon, M., "Matchmaking: Distributed Resource Management for High Throughput Computing," Proceedings the 7th IEEE International Symposium on High Performance Distributed Computing, July 28-31, 1998, Chicago, Ill.

[16] Ryutov, T. and Neuman, C. Access Control Framework for Distributed Applications. INTERNET-DRAFT, CAT Working Group, USC/Information Sciences Institute, draft-ietf-cat-acc-cntrl-frmw-00.txt, August 07, 1998, Expires February 1999, http://gost.isi.edu/info/gaa_api.html

[17] Sabata, B., Chatterjee, S., Davis, M., Sydir, J., Lawrence, T. "Taxonomy for QoS Specifications," Proceedings the Third International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS'97), February 5-7, 1997, Newport Beach, Ca., pages 100-107

[18] Schantz, R. E. "Quality of Service," to be published in "Encyclopedia of Distributed Computing," 1998.

[19] Schneck, P. A., and Schwan, K., "Dynamic Authentication for High-Performance Networked Applications," Georgia Institute of Technology College of Computing Technical Report, GIT-CC-98-08, 1998.

[20] Stern, D. F., On the Buzzword "Security Policy", Proceedings of 1991 IEEE Symposium on Security and Privacy, 1991, Oakland, Ca., pages 219-230.

[21] Vendatasubramanian, N. and Nahrstedt, K., "An Integrated Metric for Video QoS," ACM International Multimedia Conference, Seattle, Wa., Nov. 1997.

[22] Wang, C. and Wulf, W. A, "A Framework for Security Mea-

surement." Proc. National Information Systems Security Conference, Baltimore, MD, pp. 522-533, Oct. 1997.

[23] Wright, R., Integrity Architecture and Security Services Demonstration for Management System for Heterogeneous Networks, Masters Thesis, Naval Postgraduate School, Monterey, CA, Sept. 1998.

[24] Wright, R., Shifflett, D., and Irvine, C. E., "Security Architecture for a Virtual Heterogeneous Machine." Proc. Computer Security Applications Conference, Scottsdale, AZ, Dec. 1998, pp 167-17

Biographies

Cynthia E. Irvine is Director, Naval Postgraduate School Center for INFOSEC Studies and Research and an Assistant Professor of Computer Science at the Naval Postgraduate School. Dr. Irvine holds a Ph.D. from Case Western Reserve University. She has over thirteen years experience in computer security research and development. Her current research centers on architectural issues associated with applications for high assurance trusted systems, security architectures combining popular commercial and specialized multilevel components, and the design of multilevel secure operating systems.

Timothy E. Levin is a Senior Research Associate at the Naval Postgraduate School Center for INFOSEC Studies and Research. He received a BS degree in Computer and Information Science from the University of California at Santa Cruz, 1981. He has over fifteen years experience in computer security research and development. His current research interests include management and quantification of heterogeneous network security in the context of Resource Management Systems, development of costing frameworks and scheduling algorithms for the dynamic selection of QoS security mechanisms, and the application of formal methods to secure computer systems.

Optimising Heterogeneous Task Migration in the Gardens Virtual Cluster Computer

Ashley Beitz, Simon Kent and Paul Roe
School of Computing Science
Queensland University of Technology
Australia
ashley@citr.com.au, s.kent@student.qut.edu.au, p.roe@qut.edu.au

Abstract

Gardens is an integrated programming language and system designed to support parallel computing across non-dedicated cluster computers, in particular networks of PCs. To utilise non-dedicated machines a program must adapt to those currently available. In Gardens this is realised by over decomposing a program into more tasks than processors, and migrating tasks to implement adaptation. To be effective this requires efficient task migration. Furthermore, typically non-dedicated clusters contain different machines hence heterogeneous task migration is required. Gardens supports efficient task migration between heterogeneous machines via meta-information which completely describes a task's state. By identifying different degrees of heterogeneity and different kinds of tasks, we are able to optimise task migration. The main contribution of this paper is to show how heterogeneous task migration may be optimised.

1. Introduction

In the aggregate, networks of workstations represent a huge and cheap unused computing resource. By their very nature such non-dedicated cluster computers are dynamic. The workstations available to a computation will typically change during the execution of a program as workstation users come and go. Thus programs must adapt to the changing availability of workstations.

The Gardens system [28] is an integrated programming language and system targeted at non-dedicated cluster computers. The goals of Gardens are: adaptation, safety, abstraction and performance (ASAP!). These are realised in part by a modern object oriented programming language, Mianjin [27], a derivative of Pascal. The Gardens system and Mianjin programming language are custom designed and built; thus we have complete control over both of these.

Gardens utilises task migration to realise adaptation. A program is over decomposed into more tasks than processors and tasks are migrated in response to changing workstation loads. This adaptation is transparent to the programmer. Typically, workstation networks comprise a collection of different machines. Thus efficient use of such machines entails heterogeneous computing and heterogeneous task migration, the subject of this paper. Task migration is integrated into both Gardens and the Mianjin compiler.

Tasks communicate via a virtual shared object space. Tasks may reference objects belonging to other tasks, to communicate they invoke methods on such remote objects. This is the only way tasks may communicate, tasks cannot otherwise share data.

The main contribution of this paper is to show how efficient heterogeneous task migration may be achieved, to realise adaptive utilisation of workstation clusters; however, it should be noted that there are other uses for task migration e.g. to implement automatic fault tolerance through migrating tasks to disk.

The next section summarises our techniques for achieving heterogeneous task migration. Section 3 presents a more detailed look at the implementation. Some performance figures are reported in Section 4. Section 5 presents related work, and the final section discusses the work and future directions.

2. Task migration

To implement task migration Gardens uses meta-information which fully describes a task's state. This meta-information is generated by the Gardens Mianjin compiler. The meta-information is similar to that available in Java, although in addition to heap objects our meta-information also describes stack frames. A prerequisite for task migration is a safe language. For example we cannot allow a

0-7695-0556-2/00 $10.00 © 2000 IEEE

pointer to masquerade as an integer or vice versa since integers and pointers may require different translation under task migration (see Section 3).

Task migration is only supported at predetermined call points in the program. These migration points may be manually inserted by the programmer or automatically by the compiler. At these points the compiler generates the additional code and meta-information to support task migration. This can support both preemptive and non-preemptive task migration. Our current compiler does not support optimised code, although this is currently under investigation. To simplify migration we arrange for data structures to have common alignments and sizes across all platforms. We can do this since we have a custom system, language and compiler. A foreign language interface mechanism supports interoperability, but we do not support task migration within such code.

Meta-information is used to recover a task's state. A task's state comprises its stack, heap and global variables; registers and the PC are flushed to the stack. We do not migrate OS process state, our goal is to handle such state within wrapper libraries that can be migrated. The stack and heap are similar in that both can be viewed as collections of tagged records. The heap contains tagged objects, the stack contains tagged activation records, all objects in stack frames are statically known at compile time. When required, a task's state is transformed into a state suitable for the target machine. In general this is done lazily since the task may initially be saved to stable storage hence its destination may be unknown hence the information for transformation will only be available at task load time e.g. stack and heap base addresses.

To make task migration efficient we use optimisations based on different kinds of tasks and different degrees of platform heterogeneity. This is described in the following sections.

2.1. Different kinds of tasks

There are three kinds of tasks in Gardens which have different migration requirements. These different kinds of tasks can be distinguished by the runtime system.

Seed tasks are newly created tasks which have never been run. They comprise just the initial data passed in a create task operation, they have no stack nor heap and hence are trivial to migrate. Seed tasks are stored in a separate structure from other tasks until they are run, and hence are easily distinguished from other tasks.

Stackless tasks have no stack. They correspond to an inverted programming style as often used in event driven programming where control must periodically return to an event loop. Such tasks require only heap migration, since their stacks are empty, which can be considerably simpler than full task migration; some early work on this was reported in [29]. A stackless task may also be a task that has completed its main thread of execution but has "actions" to perform or objects in its heap which are referenced by other tasks.

Full tasks have stacks and heaps both of which must be migrated. These are the most expensive tasks to migrate.

2.2. Degrees of heterogeneity

There is a spectrum of degrees of system heterogeneity:

0. Same architecture, statically linked code, same heap and stack base addresses: a completely homogeneous platform. For such platforms no state transformation is required and migration corresponds to a straight memory copy from one machine to another. However in practice few modern platforms are this simple.

1. Same platform, but different base addresses (e.g. due to dynamic linking and loading): since all structures have common sizes and alignments, and all stack frames have the same representation, task migration only requires stack and heap pointers to be adjusted to deal with new stack and heap base offsets. This requires meta-information to locate all pointers in stack frames and heap objects.

2. Different architecture, but same word size: for heaps this requires stack and heap pointer adjustments as described above, endian adjustments, code and data pointer translations and type representation adjustments (e.g. for floating point types). In the case of migrating a stack, the stack must be rebuilt with different activation record conventions, e.g. stack mark information, register window flushing for SPARC etc. This can be very expensive to perform.

3. Different architecture and different word size: at present we do not address this level of heterogeneity. Note, some 64 bit processors are capable of running 32 bit code which may prove useful.

3. Implementation

3.1. Meta-information

All objects (records and arrays) in Gardens have an identifying "tag" located in the two words before the logical start of the object in memory. One word is used for garbage collection purposes while the second points to the object's type descriptor; these type descriptors serve two purposes.

Firstly, a type descriptor holds typical runtime information such as dimension and element size for arrays and method, ancestor and pointer tables for records to enable construction and polymorphism. Secondly, the type descriptors contain links to complete meta-information generated by the Gardens compiler to aid task migration. This meta-information maps out the size, location and type of fields or elements of the type in question. This allows the traversal of all objects at runtime. Furthermore, the meta-information contains a link to a descriptor for the type's module as well as a per-module index that is assigned to the type at compile time. From this, a unique module ID/per-module index pair may be obtained to identify the type across heterogeneous platforms.

In addition to the meta-information for types, meta-information for procedures is generated as well. Procedural meta-information maps out procedure entry addresses, local variable and parameter information (number, type and frame offset), possible frame offset location of saved display values, a module descriptor link and per-module index similar to those described for types. This information allows for traversal of a stack frame, given the location of a frame via a frame pointer and for unique identification for procedures, in function pointers or as instances in the stack, across platforms.

Finally, the module descriptor contains compilation and time stamps as well as three tables mapping per-module indexes to concrete addresses. The first two tables correspond to the type and procedure indexes while the third table maps potential migration points.

3.2. Heap migration

Each heap segment in Gardens comprises of two logical parts: the contiguous memory in which objects are allocated and the runtime information for managing that memory. Having designed and implemented the Gardens compiler and runtime system has allowed us to ensure that:

1. Objects of identical type are aligned identically across platforms.
2. Heap segment structure is identical across platforms.
3. Heap runtime information is logically identical across platforms.

This makes heap migration relatively simple; all that is required is a few changes to the heap segment's representation. Furthermore, since all hosts in Gardens environment have complete information as to the characteristics (architecture and operating system) of the other hosts, these representation changes may be made directly by the source or destination host; packing to and unpacking from an intermediate representation is avoided.

Representation changes fall into three categories:

1. Pointer rebasing
2. Endian adjustment
3. Code/Data segment address translation

The current platforms targeted by Gardens have similar representation for types (floating points, booleans, etc.) so type adjustments are currently not necessary.

Pointer rebasing is necessary when migrating between hosts with a degree of heterogeneity of (1) and (2) and involves traversal of all pointer fields within objects in the heap segment and adjusting any non-null pointers by an appropriate heap offset. Pointer rebasing is performed by the source host and, in the case of objects in the heap, requires only the pointer table found in the type descriptor.

Endian adjustment requires a full traversal of all objects in the heap segment and performing byte swapping on fields of necessary size. This is only required in degree (2) cases in which machines are of different endian. Endian swapping is generally performed by the source host.

Code/data segment address translation is also only required in degree (2) cases. Since the layout of code and data segments will not be identical across heterogeneous platforms, pointers into the code or data segments cannot be simply "rebased". In Gardens, however, the only candidates for code and data segment pointers in the heap are procedure variables and type descriptor addresses present in each object tag. These are replaced with procedure and type module ID/per-module index identifiers on the source side and replaced with the host specific address on the destination side.

Heap migration in Gardens thus breaks down into performing the above transformations to the objects in the heap segment and to the runtime information describing the heap itself.

The objects in the heap segment may be located by scanning a heap object bitmap that is maintained by the runtime system for memory management and garbage collection. The type and meta-information of each object is then obtained by inspecting the object's tag allowing the object to be traversed and transformed as necessary.

The runtime information for the heap consists of the portion of the heap object bitmap relevant to the heap segment, a heap descriptor located at the start of each heap segment and the list of free blocks for the heap. The heap object bitmap contains no pointers and only needs endian adjustments; however, the heap object bitmap remains in use on the source host after heap migration has occurred (to mark objects and heap segments as remote). Therefore, endian adjustments on the heap object bitmap are performed by the destination host if necessary. The heap descriptor and free

block list are indistinguishable from other objects and are transformed correspondingly.

3.3. Stack migration

Stack segments in Gardens comprise of a runtime stack and a task descriptor. The task descriptor stores stack and context information along with some programmer definable task property objects. As with heap descriptors, transformation of the task descriptor is straightforward. The method of runtime stack transformation, however, depends upon the degree of heterogeneity between hosts.

Degree (0), of course, requires no modifications to the stack.

Degree (1) requires only pointer rebasing as in degree (1) heap migration. Candidates for pointer rebasing in stack migration are no longer just pointer fields in structures but pointers in local variables and parameters, variable parameters, display pointers saved in stack frames and frame pointers themselves. The stack is traversed using the instruction pointer (or return address) for each stack frame to obtain the corresponding procedure's meta-information.

For degree (2) all three representation transformations described above need be performed. In addition to this, the layout of each of the stack frames needs to be restructured to match that of the destination host. To achieve this, the source host traverses the stack and deconstructs the stack into an abstract stack while performing the representation transformations. A list of all variable parameters that point into the stack and the address into the abstract stack at which they point is also constructed by the host.

The abstract stack is similar to the concrete stack in some ways. Each concrete frame has a corresponding abstract frame and each abstract frame has a parameter section, stack mark, local variable section and workspace (for value open arrays and value reference records). However, the abstract stack format does differ from concrete stacks in the following manners:

- Abstract frames are in opposite order to those in the concrete stack with the abstract frame pointers referring to the following frame rather than the proceeding frame.
- Along with an abstract frame pointer, each abstract stack mark contains module ID/per-module index identifier for the return address and a similar identifier for the procedure relevant procedure.
- Parameters and local variables are stored in order from the abstract stack mark as frame offsets of parameters and local variables differ across platforms.

Once the destination host has received the abstract stack, it rebuilds a stack specific to its architecture using a novel approach. For each stack frame, the parameters are first loaded from the abstract stack (into the concrete stack or the parameter registers). A context switch is performed to the new stack and a dummy procedure prologue is called for the appropriate procedure. This allocates appropriate stack space, updates the display vector and copies any value arrays into the appropriate position in the workspace. Context is switched back to the original stack and the correct return address is inserted into the newly allocated stack frame. Local variables are copied to their respective positions and code/data segment translations are performed. Finally, the variable parameter list is checked to see if memory pointed to by a variable parameter down the stack has been copied to the concrete stack. If so, the variable parameter value is adjusted to reflect the change. The stack frame is then complete.

Finally, the task descriptor requires some minor changes to reflect the state of the stack on the new host.

4. Performance

The measurements below were taken on: 233 MHz Pentium II, 96 Mb RAM running RedHat Linux v5.2 and Sun 4 Sparc, 32 Mb RAM running Solaris 2.51. For degrees of heterogeneity (0) and (1), figures are specified for machines running Linux as specified above. For degree (2), (LS) indicates migration from Linux to Solaris and (SL) indicates migration from Solaris to Linux.

The measurements are for a recursive sum program of approximately 40 stack frames and linked list program with 1000 objects in the heap. Each set of measurements presents the time taken for task transformation only (that is, no communication times are included) with the measurements split between the time taken by the source and destination hosts to perform the transformations necessary.

Seed Task Migration

Degree	Source (time μs)	Dest. (time μs)
0	6	4
1	8	4
2(LS)	9	8
2(SL)	11	4

Stackless Task Migration (Linked List)

	Heap		Stack	
Degree	Source (time ms)	Dest. (time ms)	Source (time ms)	Dest. (time ms)
0	0	0	0	0
1	0.64	0	0.03	0
2(LS)	4.49	27.65	0.64	0.95
2(SL)	57.20	2.46	7.83	0.05

Full Task Migration (Linked List)

	Heap		Stack	
Degree	Source (time ms)	Dest. (time ms)	Source (time ms)	Dest. (time ms)
0	0	0	0	0
1	0.65	0	1.06	0
2(LS)	4.73	30.08	1.21	7.37
2(SL)	59.38	2.47	12.47	0.83

Full Task Migration (Recursive Sum)

	Heap		Stack	
Degree	Source (time ms)	Dest. (time ms)	Source (time ms)	Dest. (time ms)
0	0	0	0	0
1	0.03	0	1.14	0
2(LS)	0.03	0.82	1.46	10.70
2(SL)	0.09	0.09	13.95	0.98

The above figures clearly show the advantages of identifying and targeting both the different degrees of heterogeneity and different classes of tasks to migrate.

In the context of task creation and initial load balancing, it is clear that seed task migration holds the greatest advantage for all degrees of heterogeneity, especially since seed tasks incur the smallest communication costs due to their size.

Similarly, the speed up for degree (1) migration from degree (2) migration (twice as fast for stack migration, ten times as fast for heap migration) is considerable. The heap migration figures reflect the use of pointer tables for pointer rebasing for degree (1) migration. This suggests a similar pointer map should be implemented for stack frames.

Of note are the times for the stack and task descriptor transformations for stackless and full task migration. The full task recursive sum stack transformation with 40 stack frames takes is only 10% to 20% slower than the full task linked list stack transformation. We believe this is due to inefficiencies in our current method of loading meta-information. This is further illustrated by the stack transformation (really task descriptor transformation) figure for the degree (2) stackless task migration; full meta-information information for the programmer defined task properties object must be loaded whereas only pointer tables are required for degree (1) migration.

5. Related Work

There are three main approaches to task migration across heterogeneous platforms [32]. The first approach assumes that all tasks will execute on a virtual machine that is available on all hosts in the system, for example the Java Virtual Machine. The second and third approaches both assume that tasks will execute on their host's native machine. To do this they need to generate meta-information on the executing task, so that they can translate the task's execution state from one native machine's format to another. The second approach relies on code that collects this meta-information to be included in the task's source code. This can either either be done manually by the programmer or automatically by a pre-processor. The third approach relies on the compiler and runtime system to generate the meta-information.

The first approach is much simpler than the other two, as the use of a common execution environment reduces the problem to one that can be solved via a homogeneous migration solution. This approach was initially used by Chameleon [12]. Today it is widely used by mobile agent systems, such as Agent TCL [16], Aglets [18], ARA [22], Concordia [11], Extended Facile [23], Liquid Software [13], Mole [2], Obliq [5], Odyssey [15], Omniware [20], Sumatra [1], TACOMA [39] and Telescript [40]. Despite this approach's simplicity it suffers performance penalties from the use of a virtual machine. Some solutions [20, 13] alleviate this problem by using "on-the-fly compilation" to translate parts of the task's code to native code. However, the native code produced is still 25 percent slower than regular native code [1], as they must include safeguards to protect the execution environment from being corrupted by the native code.

The second and third approaches provide better performance results than the first approach, as they allow the tasks to execute directly on the native machine. The second approach is more portable than the third approach, as it does not require a specialised compiler. However, the third approach delivers better runtime performance as it does not need to generate all of its meta-information at runtime. In addition to this, the third approach's migration mechanism is more transparent to the programmer. Examples of the second approach include: HMF [21], Process Introspection [14], HiCaM [25], Ythreads [30], Arachne [8], DOME [31], PMT [32] and MpPVM [6]. Examples of the third approach include: Emerald [35], Tui [34], Shub, Dubach and Rutherford's work [9, 10], Hollander and Silberman's work [17], Distributed C [24] and porch [36]. Our work is based on the third approach, as it provides the most optimal results.

With heterogeneous task migration, most research has focused on how to reconstruct the task's state [38, 3, 7, 30], the location of migration points [4, 35] and analysing the safety aspects of this approach [34, 19]. Very little research has been done on how to optimise migration based on different kinds of tasks and different degrees of heterogeneity. Most of the work in this area has been done by the University of Colorado at Colorado Springs [10, 33, 9]. The most significant contribution originating from their work is the

idea of ensuring that compilers generate code with the same data alignment rules. Our work builds on what they have done by providing optimised translation based on the task type and the degree of heterogeneity between platforms.

6. Discussion and Further Work

A basic heterogeneous task migration system has been implemented and initial results are promising. We are currently working on a revised system using local compiler back-end technology which supports the migration of tasks utilising optimised code. To make the migration process even more efficient we are looking at optimising our meta-information. Current performance figures suggest that our current meta-information and corresponding traversal techniques may be over complicated. To this end, a generalised version of the pointer map scheme, using bitmaps to plot required actions for stack frames and objects is being considered. Other methods of optimising meta-information include compression and lazy loading.

A general problem is how to deal with non-migrable resources such as I/O and file handles, our current solution is to retain remote references to them [26]. An interesting alternative, is to migrate tasks to the JVM where no target mapping is defined using e.g. Java Platform Debugger Architecture [37]. Task migration may also be generalised to encompass dynamic software reconfiguration. We have yet to study degree 3 heterogeneity where e.g. word sizes and alignments of data may differ between platforms; this is particularly challenging.

Acknowledgements

We would like to thank S-Y Chan for his help writing the Gardens meta-information facility and other Gardeners for their useful discussions concerning task migration. This study has been supported by an Australian Research Council grant and the Gardens research project (www.plasrc.qut.edu.au/Gardens) of the Programming Languages and Systems Research Centre at QUT.

References

[1] A. Acharya, M. Ranganathan, and J. Saltz. Sumatra: A Language for Resource-aware Mobile Programs. Number 1222 in LNCS, Springer-Verlag, 1997,pages 111–130.

[2] M. Strasser, J. Baumann and F. Hohl. Mole - a Java based agent system. In *Proceedings of the ECOOP'96 Workshop on Mobile Object Systems*, 1996.

[3] D. G. V. Bank. Gnu c language system support for dynamic native code heterogeneous process-originated migration in the v system. Master's thesis, University of Colorado at Colorado Springs, 1990.

[4] D. V. Bank, C. M. Shub, and R. W. Sebesta. A unified model of pointwise equivalence of procedural computations. *ACM Transactions on Programming Languages and Systems*, 14(6):1842–1874, Nov. 1994.

[5] L. Cardelli. A Language with Distributed Scope. *Computing Systems*, 8(1):27–59, Winter 1995.

[6] K. Chanchio and X. Sun. MpPVM: A Software System for Non-Dedicated Heterogeneous Computing. In *Proceedings of the International Conference on Parallel Processing*, 1996.

[7] K. Chanchio and X. H. Sun. Memory space representation for heterogeneous network process migration. In *12th International Parallel Processing Symposium*, Mar. 1998.

[8] B. Dimitrov and V. Rego. Arachne: A Portable Threads System Supporting Migrant Threads on Heterogeneous Network Farms. In *Proceedings of IEEE Parallel and Distributed Systems*, volume 9, 1998.

[9] F. B. Dubach. Code-Point and Data Mapping in Dynamic Native-Code Cross-Architectural Process Migration. Master's thesis, University of Colorado at Colorado Springs, 1990.

[10] F. B. Dubach, R. M. Rutherford, and C. M. Shub. Process-originated migration in a heterogeneous environment. In *ACM Seventeenth Annual Computer Science Conference*, pages 98–102. ACM Press, 1989.

[11] D. Wong, N. Paciorek, T. Walsh, J. DiCelie, M. Young and B. Peet. Concordia: An infrastructure for collaborating mobile agents. Number 1219 in LNCS, Spring-Verlag, 1997, pages 86-97.

[12] G. Attardi et al. Techniques for dynamic software migration. In *Proc, 5th Annual ESPRIT Conference*, pages 475–491, Brussels, Belgium, Nov. 1988. North-Holland.

[13] J. Hartman et al. Liquid software: A new paradigm for networked systems. Technical Report TR96-11, Dept. of Comp. Sci., University of Arizona, 1996.

[14] A. Ferrari. *Process State Capture and Recover in High-Performance Heterogeneous Distributed Computing Systems*. PhD thesis, School of Engineering and Applied Science, University of Virginia, 1998.

[15] General Magic Inc. Introduction to the Odyssey API, 20 North Mary Avenue, CA 94086

[16] R. Gray. *Agent TCL: A Flexible and secure mobile-agent system*. PhD thesis, Dartmouth College, Hanover, New Hampshire, 1997.

[17] Y. Hollander and G. M. Silberman. A Mechanism for the Migration of Tasks in Heterogeneous Distributed Processing Systems, *Parallel Processing and Applications*, Elsevier Science Publishers B.V. (North-Holland), 1988, pages 93–98.

[18] IBM. Aglets software development kit. Technical report, IBM Japan, 1998. http://www.trl.ibm.co.jp/aglets/.

[19] G. Q. M. Jr and J. M. Smith. Process migration: Effects on scientific computation. *ACM SIGPLAN Notices*, 23(3):102–106, Mar. 1988.

[20] S. Lucco, O. Sharp, and R. Wahbe. Omniware: A Universal Substrate for Web Programming. In *Proceedings of the 4th International World Wide Web Conference: The Web Revolution*, 1995.

[21] M. V. M Bishop and L. Wisniewski. Process migration for heterogeneous distributed systems. Technical Report PCS-TR95-264, Dept of Comp Sci, Dartmouth College, Janover, New Hampshire, Aug. 1995.

[22] H. Peine and T. Stolpmann. The Architecture of the ARA platform for Mobile Agents. Number 1219 in LNCS, Springer-Verlag, 1997.

[23] F. Knabe. *Language Support for Mobile Agents.* PhD thesis, School of Computer Science, Carnegie Mellon University, 1995.

[24] C. Pleier. *Prozessverlagerung in heterogenen Rechnernetzen basierend auf einer speziellen Ubersetzungstechnik.* PhD thesis, Technische Universitat Munchen, Institut fur Informatik, 1996.

[25] T. Redhead. *A High-level Checkpointing and Migration Scheme for Heterogeneous Distributed Systems.* PhD thesis, Dept of Comp Sci, University of Queensland, Australia, 1996.

[26] P. Roe and S.-Y. Chan. I/O in the Gardens Non-Dedicated Cluster Computing Environment. In *to appear in: First International Workshop on Cluster Computing*, Melbourne, Australia, Dec. 1999. IEEE Press.

[27] P. Roe and C. Szyperski. Mianjin is Gardens Point: A parallel language taming asynchronous communication. In *Fourth Australasian Conference on Parallel and Real-Time Systems (PART'97)*, Newcastle, Australia, Sept. 1997. Springer.

[28] P. Roe and C. Szyperski. The gardens approach to adaptive parallel computing. In R. Buyya, editor, *Cluster Computing*, volume 1, pages 740–753. Prentice Hall, 1999.

[29] P. Roe and C. Szyperski. Transplanting in gardens: Efficient heterogeneous task migration for fully inverted software architectures. In *Proc, Australasian Computer Architecture Conference (ACAC'99), Auckland, New Zealand*, Transactions of the CSA. Springer, 1999.

[30] J. Sang, G. W. Peters, and V. Rego. Thread Migration on Heterogeneous Systems via Compile-Time Transformation. In *Proceedings of the International Conference on Parallel and Distributed Systems (ICPADS'94)*, pages 634–639, Hsinchu, Taiwan, IEEE Computer Society Press, Dec. 1994.

[31] E. Seligman and A. Beguelin. DOME: Distributed Object Migration Environment. Technical Report CMU-CS-94-153, Carnegie-Mellon University, 1994.

[32] C. Shao and B. Schnabel. A Task Migration System for Parallel Scientific Computations in Heterogeneous NOW Environments. In *Proceedings of the Ninth SIAM Conference on Parallel Processing for Scientific Computing, San Antonio, TX*, 1999.

[33] C. M. Shub. Native code process-oriented migration in a heterogeneous environment. In *Proc, 18th Annual Computer Science Conf*, pages 266–270, Washington, DC, ACM Press, Feb. 1990.

[34] P. Smith. *The Possibilities and Limitations of Heterogeneous Process Migration.* PhD thesis, University of British Columbia, 1997.

[35] B. Steensgaard and E. Jul. Object and Native Code Thread Mobility Among Heterogeneous Computers. *Operating Systems Review*, 29(5):69–78, Dec. 1995.

[36] W. Strumpen. Compiler Technology for Portable Checkpoints, submitted for publication (http://theory.lcs.mit.edu/ strumpen/porch.ps.gz), 1998.

[37] Sun Microsystems. Java platform debugger architecture. http://java.sun.com/products/jpda/.

[38] M. Theimer and B. Hayes. Heterogeneous Process Migration by Recompilation. In *Proc, 11th Intl Conf on Distributed Computing Systems*, pages 18–25. IEEE Comp Soc Press, 1991.

[39] D. Johansen, R. van Renesse and F. Schneider. An Introduction to the TACOMA Distributed System, Version 1.0. Technical Report 95-23, University of Tromso, Norway, June 1995.

[40] J. White. Mobile agents, White Paper, General Magic Inc., 1996, http://www.genmagic.com/agents/Whitepaper/whitepaper.html.

Biographies

Ashley Beitz is a software architect at CiTR (http://www.citr.com.au) and is a part time PhD student at Queensland University of Technology (QUT). Simon Kent is a PhD student also at QUT; he was formerly a research assistant working on heterogeneous task migration. Paul Roe is a senior lecturer at QUT interested in cluster computing, programming languages and component technology. For further information concerning our research see: www.plasrc.qut.edu.au.

Linear Algebra Algorithms in a Heterogeneous Cluster of Personal Computers

J. Barbosa*, J. Tavares† and A.J. Padilha
FEUP-INEB
Grupo de Arquitecturas e Sistemas
Praça Coronel Pacheco, 1, 4050 Porto (P)
{jbarbosa,tavares,padilha}@fe.up.pt

Abstract

Cluster computing is presently a major research area, mostly for high performance computing. The work herein presented refers to the application of cluster computing in a small scale where a virtual machine is composed by a small number of off-the-shelf personal computers connected by a low cost network. A methodology to determine the optimal number of processors to be used in a computation is presented as well as the speedup results obtained for the matrix-matrix multiplication and for the symmetric QR algorithm for eigenvector computation which are significant building blocks for applications in the target image processing and analysis domain. The load balancing strategy is also addressed.

1. Introduction

Several personal computer or workstation based cluster systems have been developed, from commercial off-the-shelf processors to high performance ones such as SMP architectures [3] and using high performance networks like Myrinet [2, 19]. Most of the work is devoted to the high performance computing aiming to achieve the performance of a specific supercomputer at a lower cost.

Our aim is not to build a cluster of personal computers for parallel processing but to do parallel processing on already existing group clusters, where each node is a desktop computer running the Windows operating system. These clusters are characterized by having a low cost network, such as a 10 Mbits/s Ethernet, connecting different types of processors, of variable processing capacity and amount of memory, thus forming a heterogeneous parallel virtual computer. Due to network restrictions, which do not allow simultaneous communication among several nodes, the application domain is restricted to one or two dozens of processors.

The need for a methodology to determine the ideal number of processors comes also due to network restrictions, since as the number of processors increases the network acts as a communication bottleneck and the time spent in data exchange can overcome the benefits of more processing power. This is not usually referred in the high performance clusters literature, due to the usually huge problem size, however, in [17] a scheduling policy is studied for multiprocessor systems based on that some applications cannot exploit the computational power available, due to hardware and software constraints. In [4] a performance model for heterogeneous processing was proposed but not in the context of processor co-operation to solve a task.

Our motivation for a parallel implementation of linear algebra algorithms comes from image and image sequence analysis needs, posed by various application domains, which are becoming increasingly more demanding in terms of the detail and variety of the expected analytic results, requiring the use of more sophisticated image and object models (e.g., physically-based deformable models), and of more complex algorithms, while the timing constraints are kept very stringent.

A promising approach to deal with the above requirements consists in developing parallel software to be executed, in a distributed manner, by the machines available in an existing computer network, taking advantage of the well-known fact that many of the computers are often idle for long periods of time. It is quite common in many organizations that a standard network connects several general purpose workstations and personal computers, accumulating a very substantial computing power that, through the use of appropriate managing software, could be put at the service of the more computationally demanding applications.

Existing software, such as the Windows Parallel Virtual Machine (WPVM) [1], allows building parallel virtual computers by integrating in a common processing environment a set of distinct machines (nodes) connected to the network.

*PhD grant BD/2850/94 from PRAXIS XXI
†PhD grant BD/3243/94 from PRAXIS XXI

0-7695-0556-2/00 $10.00 © 2000 IEEE

Although the parallel virtual computer nodes and the underlying communication network were not designed for optimized parallel operation, very significant performance gains can be attained if the parallel application software is conceived for that specific environment.

This paper addresses the problem [22] of determining, from a pool of available nodes, which ones should be selected for building a parallel virtual computer that achieves the fastest application response time, and it also discusses the issue of computational load distribution; the study considers that the nodes available prior to running the application may differ from time to time, as different users and machines are active. At every program initiation phase, the highest performance computers from the available set are selected, in a number that is computed for optimizing the processing time.

The test cases presented, a parallel matrix multiplication algorithm and the QR algorithm, while pertinent to many advanced image analysis methods, are also a common module in many other fields, such as in simulation problems. In a previously reported work [5], the step edge operator proposed by Shen and Castan [20] was also tested.

2. Computational model

Several computational models [23, 7, 14] were presented in order to estimate the processing time of a parallel program. Although they could be adapted for the cluster of personal computers, a specific and simplified model is presented below. The target machine is composed by nodes with different processing capacities, resulting from different amounts of available memory and from various processor types and versions, connected by a standard interconnection network, such as the Ethernet. Each node of the machine is characterized by the processor capacity S, measured in Mflops. The network is characterized by the number of messages that are allowed simultaneously, the bandwidth LB measured in Mbits/s, and by the existence or not of broadcasting capacity.

The computational model for the virtual machine, describing the behavior for a given algorithm, is obtained by summing the time spent in sequential operations T_S and the time spent in parallel operations T_P. Sequential operations include communications, data input/output and other processing that cannot occur in parallel due to each particular algorithm characteristics. Parallel operations are those that the time spent by one processor can be divided by p if p processors are used. The total processing time, as a function of the number of processors p and the problem size n is given by equation 1.

$$T_T(n,p) = T_S(n,p) + T_P(n,p) \quad (1)$$

The interconnection network is modeled by a temporal expression, T_C, representing the time required to transmit a message of nb bits between two network nodes, assuming a distance 1 network.

$$T_C = T_L + nb(\frac{1}{LB} + T_E) \quad (2)$$

The latency time T_L represents the time gap between the processor order to transmit and the beginning of transmission and T_E the packing time. The logical topology of an Ethernet provides a single channel, or bus, that carries Ethernet signals to all stations, allowing broadcast communications. There is only one signal channel delivering packets over the network to all stations. Each message is divided into packets of length 46 to 1500 bytes of data ($packetsize$), to be sent sequentially and individually onto the shared channel. For each packet the computer has to gain access to the channel [21]. This division of a message into packets leads to a latency time for each message that is proportional to the number of packets (K) into which it is split, resulting equation 3.

$$T_{Comm} = KT_L + nb(\frac{1}{LB} + T_E) \quad (3)$$

The value of K is given by equation 4. A typical value for $packetsize$ is 1024 bytes.

$$K = \lceil \frac{nb/8}{packetsize} \rceil \quad (4)$$

For a heterogeneous virtual machine T_L and T_E depend on the processor speed S. Several experiments were conducted in order to measure these parameters, for the network referred to in the results section, which is composed by processors as illustrated in table 1. The values were measured for the matrix multiplication algorithm over different matrix sizes, resulting the average values of table 1.

$S(Mflops)$	244	161	60	50	49
$T_L(\mu s/byte)$	70	130	180	180	180
$T_E(\mu s/byte)$	0.05	0.07	0.13	0.13	0.13

Table 1. Processors parameters

Although the Ethernet physically allows broadcasting the WPVM converts a broadcast in a p processor machine to $p-1$ messages [1]. Therefore, to model correctly a broadcast the time spent in one message has to be multiplied by $p-1$.

Independent communications over rows or columns, either for 1-D or 2-D grids, can originate network collisions. Examples are all slave processes trying to send results to the master process at the same time; or for the matrices multiplication algorithm, in each step, the distribution of the matrices are independent over rows, for one matrix, and

over columns for the other matrix. To avoid collisions a set of communication routines using a ring communication pattern, as shown in figure 1, were developed. They allow processes to synchronize by establishing the order of communications according to the processes position on the grid.

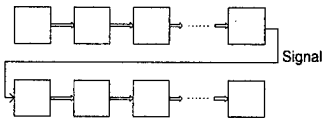


Figure 1. Communication pattern for two independent row broadcasts

The parallel component T_P of the computational model, equation 5, represents the operations that can be divided over a set of p processors obtaining a speedup of p, i.e. operations without any sequential part.

$$T_P(n,p) = \frac{\psi(n)}{\sum_{i=1}^{p} S_i} \quad (5)$$

The numerator $\psi(n)$ is the cost function of the algorithm measured in floating point operations ($flops$) as a function of the problem size n. For example, to multiply square matrices of size n, the cost is $\psi(n) = 2n^3$ [8]. This operation count does not include memory operations resulting, therefore, a higher complexity. To obtain a correct operation count one should consider the memory references made and have an estimation of the memory access time. The nodes of the virtual machine have different levels of memory (cache, main memory, and disk) with different access times, and one cannot predict how many accesses are made to each one.

Figure 2 shows the processing capacity achieved by an 161 Mflop peak performance processor for the matrix multiplication algorithm. The computational cost is $\psi(n) = 21.5n^3 flops$. Figure 2 also shows that a non block oriented algorithm cannot assure a constant coefficient of $\psi(n)$, which is a requirement in order to be able to estimate the time the processors will take to execute the algorithm. From this point on the coefficient of $\psi(n)$ will be referred to as the algorithm constant β. The value β does not depend on the processor but rather is a characteristic of the algorithm.

The denominator of equation 5 is the processing capacity used which is obtained by summing the individual processing capacities of the machines. For this equation to be valid each machine should not take more than T_P seconds to process its part. This assumes a perfect load balancing in the heterogeneous machine.

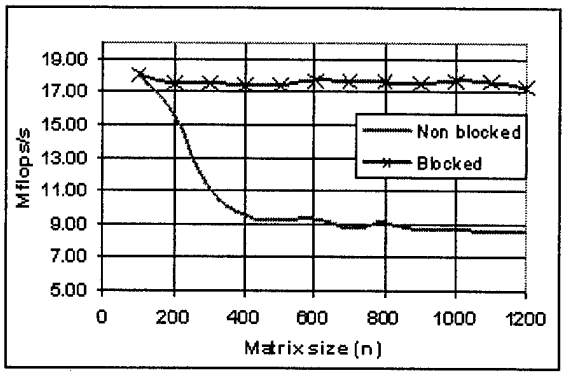


Figure 2. Performance of the matrix multiplication algorithm on a 161 Mflop peak performance processor as a function of matrix size

3. Load balancing strategy

In this section a static load distribution algorithm is presented and issues related to the optimization of processing time in a heterogeneous environment are discussed.

3.1. Data distribution

To avoid the slowest processors to determine the parallel processing time, the load should be distributed proportionally to the capacity of each processor. The aim is to assign the same amount of processing time which may not correspond to the same amount of data.

The matrices are organized in square blocks of data which are assigned to the processor grid. To achieve a balanced distribution in the heterogeneous machine the number of blocks assigned to each processor should be proportional to its processing capacity compared to the entire machine:

$$l_i = \frac{S_i}{\sum_{k=1}^{p} S_k} \quad (6)$$

The load index l_i although theoretically correct, is not fully applicable in practice since the number of blocks assigned has to be an integer value. As an example, for a machine composed by 6 processors of capacity {244, 244, 161, 161, 60, 50} Mflops, l_i would be {0.265, 0.265, 0.175, 0.175, 0.065, 0.054}. To distribute a matrix of size 1800 over a (1,6) processor grid the assignment would be 1800 rows by {477, 477, 315, 315, 118, 98} columns respectively.

The strategy implemented is to compute the number of blocks to assign to each processor rounding the real value obtained down to the nearest integer, so that some blocks are left to be assigned. Then, to obtain an optimal solution the remaining blocks are assigned one at a time to the grid

of processors, choosing the one that will take less time to finish the job.

For the test case presented, consider a block size of 25 elements, which lead to an assignment in terms of number of blocks, of {19, 19, 12, 12, 4, 3} summing 69 in a total of 72 blocks, leaving 3 blocks unassigned. Using the time complexity analysis presented in the next section for the matrix multiplication algorithm, $T_P = 21.5n^3/S$, the estimated computational time per processor is {135.6, 135.6, 129.8, 129.8, 116.1, 104.5} seconds. Each block will take {7.14, 7.14, 10.82, 10.82, 29.03, 34.83} seconds in each processor respectively. This block processing time is summed to the total time each remaining block being assigned to the processor that would finish first. The first block is assigned to processor 6 and the last two blocks to processors 3 and 4, resulting an estimated processing time of {135.6, 135.6, 140.6, 140.6, 116.1, 139.3} seconds. A perfect load balancing cannot be achieved, however for this block size it is the optimal assignment, i.e. the assignment that leads to the minimum processing time. Figure 9 shows the processing time measured for each processor.

Another issue in data distribution for a heterogeneous machine is to keep the load balance in the whole algorithm. For some algorithms, such as tridiagonal reduction and LU factorization, in each iteration part of the matrix is fully computed and not visited again, the working matrix being smaller from step to step. This can lead to an imbalance load if the distribution is not cyclic. For the example above, if contiguous blocks are assigned to each processor, one of the fastest processors would be idle after computing 19 blocks of the matrix , remaining 53 blocks to be processed.

To overcome this load imbalance, blocks are organized in balanced groups. Being l_i the load index of processor i, one define group block G_B as:

$$G_B = \frac{1}{min(l_i)} \tag{7}$$

If $G_B/Q < 2$ then $G_B = 2/min(l_i)$, where Q is the number of column processors. For a (P, Q) grid the algorithm is applied to columns and rows independently, considering the processing capacity by column and row respectively, as shown in table 2.

For the example given above $G_B = 1/0.054 = 18$ blocks, giving a group block of {5, 5, 3, 3, 1, 1}.

With this strategy it is guaranteed that from the beginning to the end of the algorithm all processors are involved in proportion of their load indices l_i, allowing an effective load balancing. When the last group block is being processed, the last 8 blocks would be computed by the slowest processors; it is reasonable that in cases where some processors cannot participate due to the lack of data, it should be the fastest ones doing the computation. Therefore, the cyclic distribution is used inside each group block.

3.2. Data redistribution

In order to exploit the computational capacity of the target machine, the algorithms must be implemented in order to increase the computation to communication ratio, mainly due to the slow network. Therefore, data redistribution is allowed in order to switch to the optimal grid computed for each algorithm. Data distribution is represented by system independent objects, allowing the system to switch between two unrelated processor grids.

The cost of redistribution is estimated by the communication of n^2 elements for a matrix of size n, which is the worst case, i.e. every element being allocated to a different processor. The redistribution algorithm starts from the first processor (1,1) to the last, changing data synchronously with the remaining processors.

For related grids, e.g. switching from (1,6) to (1,7), the system evaluates if the gain in time due to the addition of one processor is overcome by the data redistribution time. In that case the grid change does not occur.

3.3. Block size

The block size should be chosen according to the following conditions: first, it should maximize the individual processing capacity, that as shown in figure 2 degrades for a block size 1, and second, to allow the implementation of a load balancing distribution. For the machines tested a block size in the range 15 to 40 ensure an almost constant processing capacity.

For a sequence of parallel algorithms, e.g. for eigenvector computation where different grids are used, the block size should satisfy all grids in terms of load balancing since, although there is data redistribution, this parameter remains unchanged.

3.4. Processor selection policy

The system keeps a record of the computers enrolled in the parallel virtual machine ordered by decreasing computational capacity. If only part of the machine is needed to execute the algorithm the computers are selected from the fastest to the slowest one.

A computer is considered available for parallel processing if there is no user activity for at least half the processing time of the last parallel algorithm. If a user starts using his/her computer during a parallel execution, the system does not transfer the work to another computer; it completes the current job and then marks the computer as unavailable. For the problem size addressed, whose processing time is expected to be of a few minutes, this policy is satisfactory.

4. Optimization of the processing time

A parallel algorithm may have two aims: to obtain a better accuracy of results by using a more detailed domain which could not be possible in a single processor, usually due to memory limitations, or, for a given accuracy, to obtain a reduction in the processing time. The time gain obtained with the parallel algorithm is usually called *Speedup* and is defined as the quotient of the serial algorithm time (T_1) over the parallel algorithm time (T_T).

$$Speedup = \frac{T_1}{T_T} \tag{8}$$

Depending on how the serial processing time is measured one can have different definitions of *Speedup*. *Relative Speedup* is obtained if the serial time is the processing time of the parallel algorithm in a single node of the parallel computer. *Real Speedup* is obtained if the serial time is the processing time of the most efficient sequential algorithm in a single node of the parallel computer. *Absolute Speedup* is defined when the serial time is obtained for the fastest sequential algorithm executed in the fastest sequential computer available [18]. In the context of the envisaged applications of the parallel virtual machine, we define *Speedup* as the ratio between the processing time of the serial version in the computer that controls the parallel execution (master), over the processing time of the parallel program. This is the effective gain as seen by the user, who has a choice between his/her own single machine (master) or the parallel virtual machine; the definition is also globally fair when the master computer is one of the fastest available, which is the case in the test cases presented below. In a parallel virtual machine it is quite common that each node of the computer network is not fully available for the user that is running a parallel application. The application should not schedule work for nodes that are in use by other users, and therefore it should have a record of the ones that are free. The aim in scheduling work for distributed processing is to obtain a processing time that is as small as it can be obtained for that particular network, even if some of the nodes are left in the idle state. Therefore, the relevant parameter to be considered is the Speedup in detriment of the Efficiency, which is often used in other contexts.

Given the above definitions, one can state the goal of the work herein reported as the determination of the optimum number of processors using a criterion of minimum processing time. The optimal number of processors p, which minimizes $T_T(n,p)$, is the one for which an increase on the serial component, due to the addition of one more processor, will be balanced by the gain obtained on the processing time of the parallel component.

4.1. Application to a homogeneous machine

For a given algorithm, characterized by the constant β, and for size n matrices, p can be obtained by solving equation 9 in order to p [5].

$$\frac{\partial T_T}{\partial p} = 0 \tag{9}$$

For a homogeneous machine equation 5 simplifies to $T_P(n,p) = \frac{\psi(n)}{pS}$ and the communication parameters T_L and T_E assume the same value for all machines, allowing a straightforward solution.

4.2. Application to a heterogeneous machine

For a heterogeneous machine another degree of complexity is added to equation 9: first, processors have different computational capacities (S) and second, the communication parameters T_L and T_E also vary with S, as shown in table 1.

To tackle this problem one first orders the nodes by decreasing value of S_i (the capacity of node i), and then schedules the work from the fastest to the slowest free node, resulting the denominator of equation 5: $S_T(p) = \sum_{i=1}^{p} S_i$. To compute the first derivative of T_T in order to p it is required to find the sum $S_T(p)$, which cannot be computed beforehand since one does not know how many processors will be used. The function $S_T(p)$ increases monotonically with p, having a growth rate that decreases with increasing p, as shown in figure 3 for a machine composed by processors of capacities {244, 244, 161, 161, 60, 50, 49} Mflops in decreasing order.

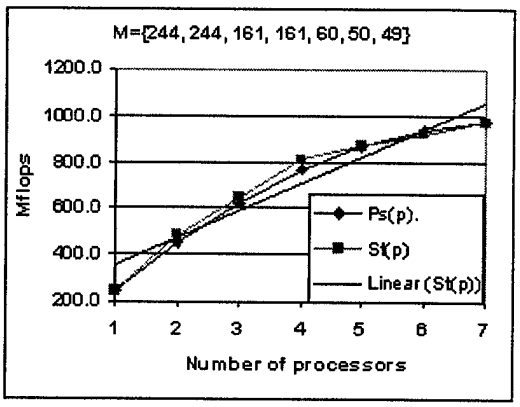


Figure 3. Processing capacity of the heterogeneous machine as a function of the processors used

The aim is to approximate $S_T(p)$ by a polynomial function in p in order to be able to solve equation 9. A first

order polynomial function as used for a homogeneous machine is not adequate here. The ideal polynomial function would be one that passes in all points of $S_T(p)$; however, its computation time may be significant for a large number of processors. The solution adopted was an iterative quadratic approximation. The first function is defined by zero and the extreme points of $S_T(p)$. The iterative process allows the reevaluation of the cost function $T_T(n,p)$ in the neighborhood of the solution computed. In each iteration only half of the processors used in the last iteration are considered being the polynomial function defined by: if P is the total number of processors, $p^{(i-1)}$ the solution for iteration $(i-1)$ then in iteration i the function is defined by the three points of equation 10.

$$S_T(p^{i-1} \pm P/2^{i+1}) \quad and \quad S_T(p^{(i-1)}) \quad i = 1, 2, ... \tag{10}$$

The second degree polynomial function has the same behavior as $S_T(p)$ and is written as:

$$P_S(p) = ap^2 + bp + d \tag{11}$$

resulting the first derivative of $T_P(n,p)$ in order to p in:

$$\frac{\partial T_P(n,p)}{\partial p} = \frac{\partial}{\partial p}\left(\frac{\psi(n)}{ap^2 + bp + d}\right) = 0 \tag{12}$$

which must be solved in order to obtain the number of processors p that minimizes the total processing time.

If the logical grid of processors affects the processing time, then changing to a 2D grid (e.g. (r,c) grid) or 3D (e.g. hypercube), one or two dimensions are added to the problem respectively. For the 2D grid the quadratic approximation with $p = rc$ becomes:

$$P_S(r,c) = a(rc)^2 + b(rc) + d \tag{13}$$

The communication parameters T_L and T_E also need to be modeled by a function of p in order to solve $\partial T_S(n,p)/\partial p$. To transmit a message from computer A to B the latency and packing time depend on the speed of processor A. If one can predict the amount of data each processor will be responsible to transmit, one can estimate the time spent in communications by the whole machine. According to the data distribution algorithm to each processor is allocated an amount of data proportional to its relative speed in the heterogeneous machine: $l_i = S_i / \sum_{k=1}^{p} S_k$. Therefore, functions to model these parameters are defined by equations 14 and 15, corresponding to an weighted mean of these values for each possibility of p processors. The values of $(T_L)_i$ and $(T_E)_i$ are shown in table 1.

$$T_{TL}(p) = \sum_{i=1}^{p} l_i \times (T_L)_i \tag{14}$$

$$T_{TE}(p) = \sum_{i=1}^{p} l_i \times (T_E)_i \tag{15}$$

For the machine considered (figure 3), the functions $T_{TL}(p)$ and $T_{TE}(p)$ are shown in figures 4 and 5 respectively. In those figures it is also shown a first degree polynomial approximation to be included in $T_S(n,r,c)$.

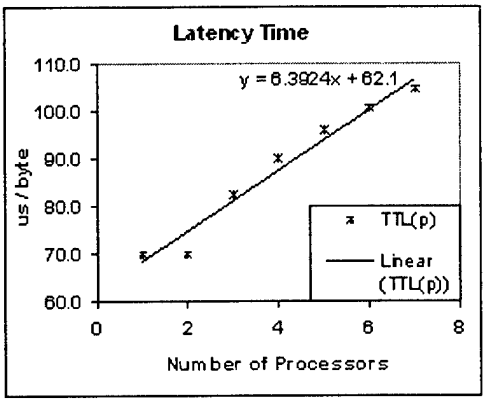


Figure 4. Approximation for $T_{TL}(p)$ per byte

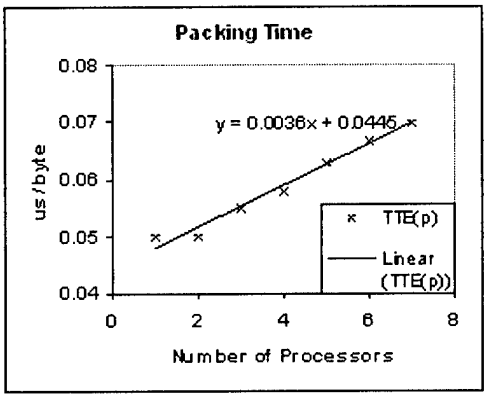


Figure 5. Approximation for $T_{TE}(p)$ per byte

The (r,c) configuration that minimizes the processing time is obtained by $\nabla T_T(n,r,c) = 0$. Since one wants to compute the ideal grid (r,c) for a given problem size n, the first derivative of $T_T(n,r,c)$ in order to n is zero. Thus, the optimal configuration is obtained by solving the system of equations 16.

$$\begin{cases} \frac{\partial T_T(n,r,c)}{\partial r} = 0 \\ \frac{\partial T_T(n,r,c)}{\partial c} = 0 \end{cases} \tag{16}$$

4.3. Applying the methodology to a matrix multiplication algorithm

The methodology presented above will be tested with an improved implementation of the matrix multiplication operations [11]. Figure 6 shows an hypothetical data assignment for a $(2,3)$ processor grid. For simplicity, the blocks displayed are formed by contiguous data, although the block cyclic data distribution is used [10].

To compute the matrix product $C = A \times B$, in each iteration of the algorithm each processor multiplies one column block of A by one row block of B, updating the correspondent block of C. The shadowed area in matrix C represents the block that processor $(0,0)$ has to update in each iteration.

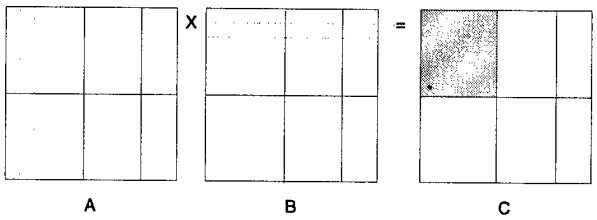


Figure 6. Matrix multiplication operations

Considering a grid (r,c) of processors, the matrices $A = (m,k)$, $B = (k,l)$ and $C = (m,l)$ the amount of data required to broadcast matrix A over the rows of processors is:

$$\frac{m}{r}\frac{k}{c}(c-1)rc = mk(c-1) \tag{17}$$

Note that $(c-1)$ appears because the broadcast is in fact performed by sequential communications. To broadcast matrix B over the column of processors it is required to transmit:

$$\frac{k}{r}\frac{l}{c}(r-1)cr = kl(r-1) \tag{18}$$

The time required to compute the inner loop products is given by:

$$T_P = \beta\frac{mlk}{S_T(r,c)} \tag{19}$$

where $S_T(r,c)$ is the processing capacity of the heterogeneous machine when rc processors are used. The value β for the matrix multiplication is 21.5, as given in section 2. The total estimated processing time, assuming square matrices of size n, is expressed as:

$$\begin{aligned} T_T(n,r,c) &= (\frac{n^2(r+c-2)}{packetsize})T_{TL}(r,c) \\ &+(n^2(r+c-2))(LB^{-1}+T_{TE}(r,c)) \\ &+\beta\frac{n^3}{P_S(r,c)} \end{aligned} \tag{20}$$

Depending on the data types used (float or double) the correspondent communication factors have to represent the amount of data in bytes. LB is the bandwidth per byte.

For the machine of figure 3 the quadratic approximation, equation 11, becomes $P_S(r,c) = -17.595(rc)^2 + 261.6rc$. This approximation is close to the real curve $S_T(r,c)$ for values $0 \leq rc \leq 7$. Outside this domain the polynomial function may introduce false minima in the processing time function. Therefore, the minimization must be restricted to the allowed domain by the number of processors available. This can be accomplished by introducing the Lagrange multipliers [15] in the system of equations 16. An additional function to restrict the domain is included:

$$\begin{cases} \frac{\partial T_S(n,r,c)}{\partial r} + \frac{\partial T_P(n,r,c)}{\partial r} = -\lambda c \\ \frac{\partial T_S(n,r,c)}{\partial c} + \frac{\partial T_P(n,r,c)}{\partial c} = -\lambda r \\ \lambda(rc-7) = 0 \end{cases} \tag{21}$$

The following figures, 7 and 8, present results for matrices of size 1800. Figure 7 displays the communication estimated (Est.) and measured (Meas.) time for one and two rows of processors, limited to 7 processors. And figure 8 displays the total processing time $T_T(n,r,c)$ obtained by estimation with the quadratic approximation for machine processing capacity (Tot. E), by estimation using the exact processing capacity (Tot. R), and the measured time (Tot. M). The communication times are modeled correctly, existing only a slight difference for some grids. The total estimated processing time differs from the measured one due to the quadratic approximation which underestimates the processing capacity in some cases and overestimates in others, although the behavior is similar to the measured curve and it does not introduce false minima in the processing time function. The curve obtained with the real processing capacity of the heterogeneous machine shows that the overall model is correct and that the processing time can be accurately estimated.

Solving the system of equations 21, the values of $r = c = 2.65$ are obtained for $n = 1800$ and $LB = 100Mbits/s$. Since one wants an integer solution, it can be assumed $c = 3$ which implies $r = 2$, since $rc \leq 7$. The grid (3,2) would be equivalent. Figure 8 shows that the minimum is obtained for grid (2,3), confirming the system solution, although there is an increase in the processing time compared to the estimation. This is the consequence of an imbalance grid which cannot be overcome for that machine. Table 2 shows the processor layout for grid (2,3). The first two columns of processors are equilibrated what does not happen for column 3, in which either processor (1,3) will

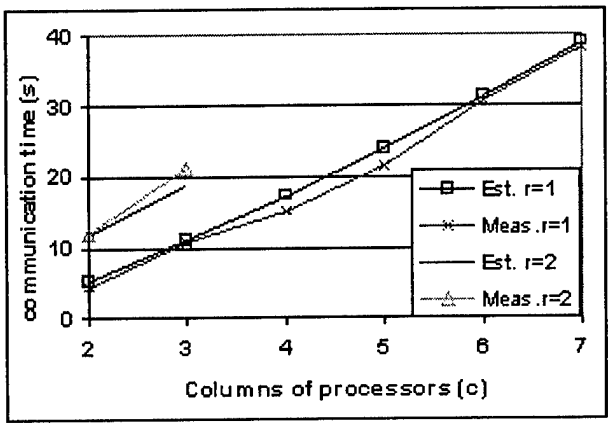


Figure 7. Communications for the matrix multiplication algorithm (matrix size 1800)

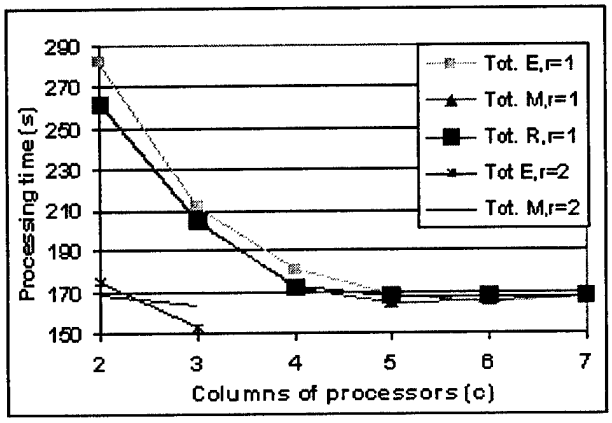


Figure 8. Processing time for the matrix multiplication algorithm (matrix size 1800)

be underloaded or processor (2,3) will be overloaded, delaying all other processors as they will be always waiting to communicate.

Figure 9 shows the processing time for all processors, where it can be seen that processor 6 is delaying the process for grid (2,3). Grid (1,6) is better balanced but the ideal load balance is not achieved due to the data blocks indivisibility. For this network, due to processor relation in processing speed, a balanced load can only be achieved with small blocks of data. The squared block size used was 25. A smaller block size, e.g. 10, while improving the load balance, would decrease the individual performance of processors due to a sub-utilization of the processors cache memory.

Note that although grid (2,3) is less balanced and there is one processor that takes more time, it makes a better solution than grid (1,6) due to the fact that this grid requires more communication time, as it can be seen in figure 7.

244	161	60	=465
244	161	50	=455
=488	=322	=110	

Table 2. Processor layout for grid (2,3)

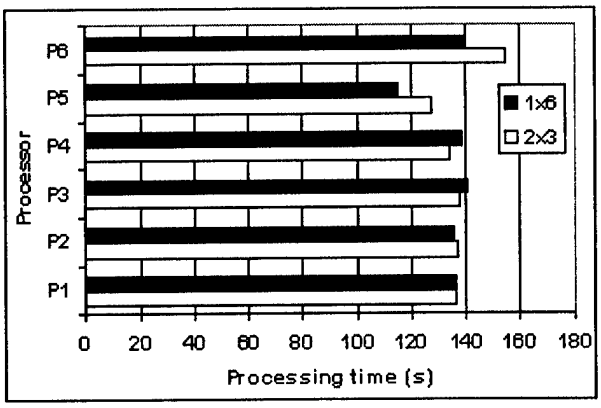


Figure 9. Matrix multiplication processing time

5. Results

In this section results for tridiagonal reduction (TRD), LU and QR factorization algorithms in the heterogeneous machine represented in figure 3 for an Ethernet network at 100 Mbits/s are presented. Figure 10 shows the performance of each algorithm in a single processor. The QR performance is divided by 2 for displaying purposes. As shown before for the matrix multiplication algorithm, the processor performance is kept almost constant for the block versions of these algorithms, for matrices greater than 400 elements. The correspondent β value considered for each algorithm is the average in that domain. The square block size used in all cases varies from 15 to 40. There is some variation in the processor performance for a given matrix size, mainly due to the operating system (Windows NT) which stochastically has some activity; however, this represents a variation in the processing time below 1%.

The estimated values presented below are obtained by applying the system of equations 21 using the time function of each algorithm respectively.

5.1. LU factorization algorithm

The LU factorization algorithm is applied in order to solve directly a system of equations. The implementation

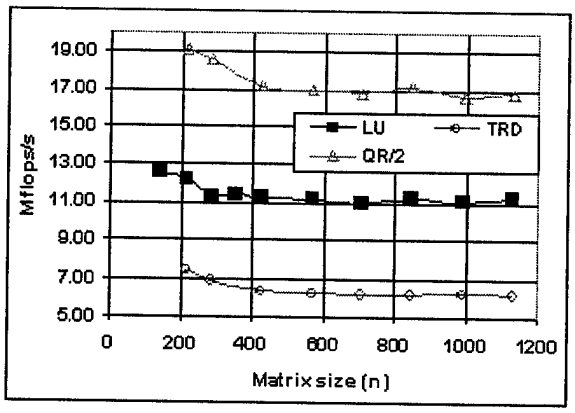


Figure 10. Performance of LU, QR and TRD block algorithms on a 161 Mflop peak performance processor as a function of matrix size

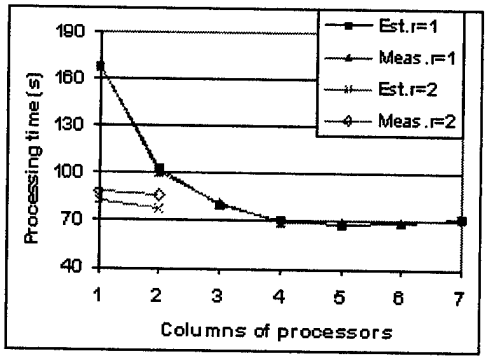


Figure 11. LU processing time for a matrix of size 1800

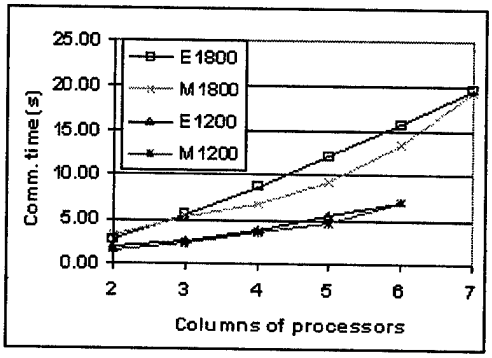


Figure 12. Estimated (E) and measured (M) communications for LU algorithm

is the right-looking variant where algorithm details can be found in [9]. For a (r, c) grid of processors, the amount of data (double/float) transmitted in the parallel matrix update is:

$$(r + c - 2)\frac{n^2}{2} \tag{22}$$

and the parallel processing time is:

$$T_P(n, r, c) = \beta \frac{n^3}{S_T(r, c)} + \Theta(n^2) \tag{23}$$

The β for LU is 7.5. There is a component of complexity n^2 correspondent to the computations made by the pivot processor. Figure 11 shows the processing time estimated and measured for a matrix of size 1800. Although it is hardly perceptible in the figure, the optimum value estimated for (r, c) is (1,5). In practice the optimum is grid (1,4), which outperforms grid (1,5) by only 0.5 seconds. In this case the difference is due to the quadratic approximation for machine processing capacity. If the real values are used the estimated optimum is (1,4).

Figure 12 shows the estimated (E) and measured (M) communication times for matrices of size 1200 and 1800. In general the communications are well modeled. The differences observed are less than 3 seconds. This can lead to a grid selection that is not the optimal one; however, since the processing times obtained for grids (1,4), (1,5) and (1,6) are 69.1, 69.6 and 70.0, the main drawback would be to have unnecessary processors allocated.

Figure 13 shows the load distribution for the matrix of size 1800. For up to 5 processors a good load balancing is achieved, with processors taking almost the same time to process the data allocated to them. The block size from processor (1,1) to (1,5) is 1800 rows by 500, 500, 340, 340 and 120 columns respectively. Ideally they should receive 504, 504, 333, 333 and 124 columns.

5.2. Tridiagonal reduction algorithm

The tridiagonal reduction algorithm (TRD) is a step in the computation of the eigenvalues and eigenvectors of a symmetric matrix. Details of the algorithm can be found in [6]. For a (r, c) grid of processors, the amount of data to transmit is:

$$2n(r - 1) + 4n^2(rc - 1) \tag{24}$$

for computation and broadcast of Householder vectors, parallel matrix update and matrix vector products. The parallel processing time is:

$$T_P(n, r, c) = \beta \frac{n^3}{S_T(r, c)} + \Theta(n^2) \tag{25}$$

The β for TRD is 28 and there is also a negligible term in n^2. Figure 14 shows the processing time for a matrix

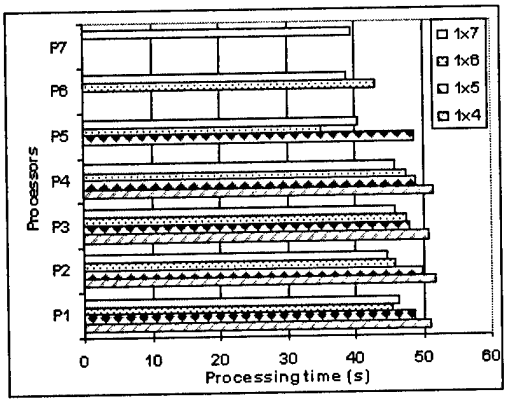


Figure 13. LU load distribution for a matrix of size 1800

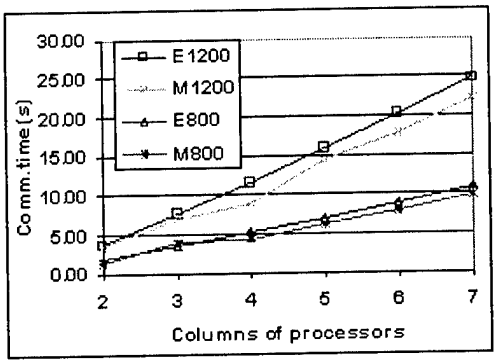


Figure 15. Estimated (E) and measured (M) communications for Tridiagonal reduction algorithm

of size 1200. For grids (1,1) to (1,4) the estimated time is higher than the measured one; the maximum error occurs for grid (1,4) which coincides with the maximum error in the quadratic approximation of computational capacity. The minimum is correctly determined as grid (1,4). Again if grid (1,3) was chosen the total time would be marginally higher: 104.7 s instead of 100.0 s. To guarantee the selection of the best grid the scheduler can operate with real values of processing capacity for estimating the processing time in the neighborhood of the solution obtained by the system of equations 21.

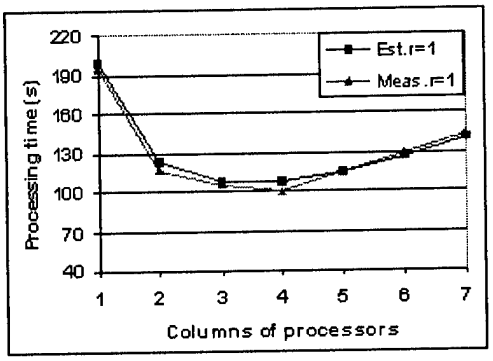


Figure 14. Tridiagonal reduction processing time for a matrix of size 1200

Figure 15 shows the estimated (E) and measured (M) communication times for matrices of size 800 and 1200. The more significant differences are for matrix of size 1200 where communications are overestimated. In all cases the difference is below 1.1 second.

Figure 16 shows the load distribution for the matrix of size 1800. For grid (1,4) a good load balancing is achieved. For grid (1,5) one process takes 3 seconds less than the others because it was assigned one block less of size 20. The data allocated to each processor was 1200 rows by 340, 320, 220, 220 and 100 columns respectively; ideally it should be 1200 by 336, 336, 222, 222, 83. Grids (1,6) and (1,7) are not well balanced also due to block indivisibility.

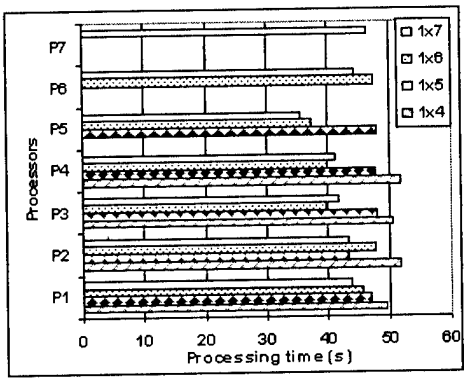


Figure 16. Tridiagonal reduction load distribution for a matrix of size 1200

5.3. QR iteration algorithm

The QR iteration is the last step in the eigenvector computation sequence, preceded by the tridiagonal reduction of a symmetric matrix and orthogonal matrix computation.

Synthetically, the procedure is to compute Givens rotations in order to reduce the tridiagonal matrix into a diagonal one whose elements are the eigenvalues. Eigenvectors are computed by updating the orthogonal matrix, resulting from the tridiagonal operation, with the rotations. Each rotation affects only two columns of the orthogonal matrix; a detailed explanation is given in [12].

The parallelization implemented takes advantages of the fact that one rotation updates only two columns without inter-row dependencies. For the tridiagonal reduction a column oriented distribution is more favorable; however, that data allocation will imply communications between boundary columns, with the additional drawback of using cyclic distributions, which increase the boundary columns drastically. A column oriented algorithm applying the technique of considering multiple bulges [13] was implemented, but only a marginal speedup, below 1.5, was obtained due to the fact that multiple bulges increase the number of iterations required which, associated to boundary communications, is not suited for the slow bus network of the target machine.

Alternatively, it was given the possibility of data redistribution in order to match the ideal processor grid for each algorithm. In this case, QR iteration was a row oriented strategy.

The QR iteration has two computational tasks: one, to do the bulge chase of order n^2, and the other to update the orthogonal matrix of order n^3:

$$T_P = \beta \frac{n^3}{T_T(r,c)} + \Theta(n^2) \qquad (26)$$

The β for QR is 43. The time to compute the chases is in fact negligible compared to the $\Theta(n^3)$ term (e.g., for the matrix of size 1600 used it takes 2.1 seconds to compute the chases and 721 seconds to update the matrix in a 244 Mflop computer). Therefore, the solution adopted was to do the chases in one computer (1,1), the fastest one, which at the end of a chase transmits the correspondent rotations to the remaining processors. Then, all processors update the part of the orthogonal matrix allocated to them without requiring any data exchange, i.e. true parallelism.

Figure 17 shows the estimated and measured processing time for a matrix of size 1000. The difference for grid (1,4) is mainly due to error of the quadratic approximation which is maximum for 4 processors. The estimated minimum is 6 processors; in practice it is 7 processors. This is due to a load imbalance occurring for 6 processors, in which there is a processor that takes 2 seconds more than the others, as shown in figure 18.

The communications involved are only to distribute the Givens rotations, estimated assuming a convergence rate of γ, as:

$$\gamma n^2 (r-1) \qquad (27)$$

This is an estimation because the number of chases depends on the rate of convergence of the QR iteration. This rate is expected to be less than 2 [8]. The estimated values of figure 19 were obtained with $\gamma = 0.9$ obtained experimentally with the matrix used. In this algorithm the communication parameters T_E and T_L refer to the machine that computes

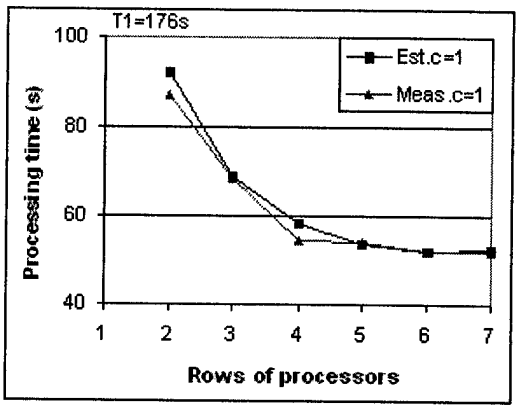


Figure 17. QR iteration processing time for a matrix of size 1000

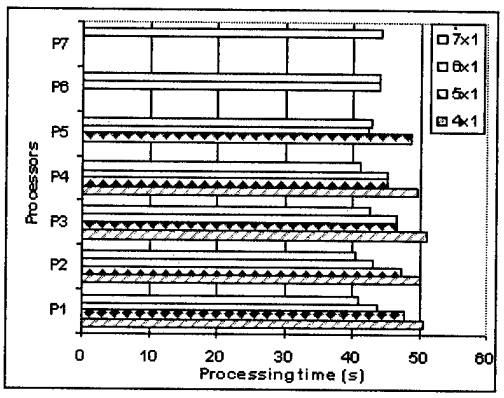


Figure 18. QR iteration load distribution for a matrix of size 1000

the Givens rotations, since it is the only emitter in the QR iteration.

5.4. Symmetric eigenvector computation

In this subsection the whole algorithm for eigenvector computation executed in the heterogeneous machine is compared to a serial version [16] when executed in the fastest node.

The performance metrics used to evaluate the parallel application is, first, the runtime, and second the speedup achieved. To have a fair comparison in terms of speedup, one defines the Equivalent Machine Number ($EMN(p)$) which considers the power available instead of the number of machines that, for a heterogeneous environment, is an ambiguous information. Equation 28 defines $EMN(p)$ for p processors used, and S_1 is the computational capacity of the processor that executed the serial code, also called the

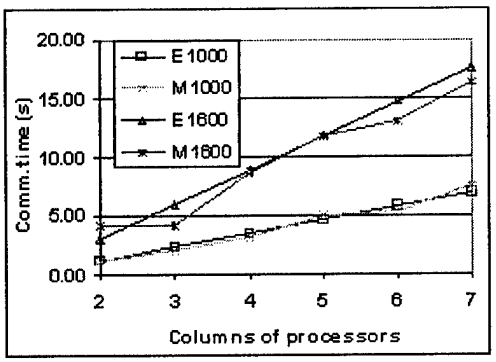


Figure 19. Estimated (E) and measured (M) communications for QR iteration

master processor.

$$EMN(p) = \frac{\sum_{i=1}^{p} S_i}{S_1} \tag{28}$$

For the machine presented in figure 3 $EMN(6) = 3.77$ and $EMN(7) = 3.97$, i.e. using 6 processors of the heterogeneous machine is equivalent to 3.77 processors identical to the master processor and to 3.97 if 7 processors are used.

Figure 20 and table 3 compare the virtual machine to the fastest node of the machine used to run the sequential code. Different grid configurations are used for the different algorithms, according to the optimal grid computed by equation 21.

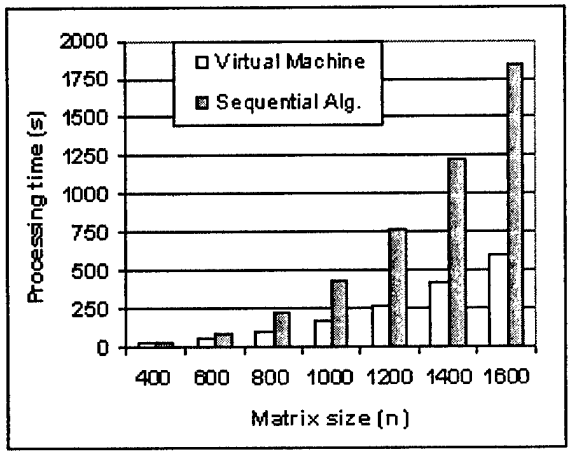


Figure 20. Eigenvector computation in a 7 processor heterogeneous machine compared to the sequential algorithm executed in the fastest node

Stage	Number of processors used							GRID
(n)	400	600	800	1000	1200	1400	1600	(pxq)
TRD	1	2	3	3	4	4	4	1xq
Q (Orth.)	5	6	6	6	7	7	7	1xq
QR it	5	6	6	6	6	6	6	px1
Speedup	1.0	1.7	2.3	2.6	2.9	3.0	3.1	
EMN	3.6	3.8	3.8	3.8	4.0	4.0	4.0	
Efficiency	0.3	0.5	0.6	0.7	0.7	0.8	0.8	

Table 3. Processors used in each stage of the eigenvector computation

6. Conclusions

Briefly stated, the methodology presented in this paper was designed to address problems arising in the context of using image processing and analysis algorithms for interactively extracting important data and information from images of a specific application domain, e.g. medical imaging.

Currently, this activity is often conducted by exploring the functionality (hardware and software) of general purpose systems, which usually trade off algorithm sophistication and user comfort; this means that more advanced image tools may be absent in these systems due to practical considerations.

The main goal of the work herein presented was to take advantage of the existence of a network of computers (this is a very frequent situation in many user organizations) to try and move the aforementioned trade-off in the direction of allowing the provision of more advanced and sophisticated algorithms without sacrificing user comfort.

The results presented show that, for the important linear algebra building blocks of many advanced image analysis methods, the stated goal may be accomplished; an improvement has been achieved in the execution time, by a factor of about 3, which may bring more image analysis tools into the feasible condition for new general-purpose software.

A collection of machines with a wide range of processing capacities, from 244 to 49 Mflops in the case presented, can cooperate and achieve a considerable speedup in linear algebra algorithms. The load balancing strategy proved to be a determinant condition for the quality of the results.

A methodology to determine in a computer network the number of active processors that minimizes the total processing time for a specific parallelized algorithm was presented. The main objective is that the user of a computationally demanding application may benefit from the computational power distributed over the network, while keeping other active users undisturbed.

This goal can be achieved in a transparent manner for the user, once the modules of his/her application are correctly

parallelized for the target network and the performance of the machines in the network is known. The application, before initiating a parallel module, determines the best available computer composition for a parallel virtual computer to execute it, and then launches the module, achieving the best response time possible in the actual network conditions.

Practical tests of the methodology were conducted both on homogeneous and heterogeneous networks, using basic algorithms from linear algebra; in both cases, the theoretical values computed were confirmed by the measured performance. It was shown that a good load balancing could be achieved even for a heterogeneous environment, by using an appropriate processor layout. Other generic modules will be parallelized and tested, so that an ever increasing number of image analysis methods may be assembled from them. Application domains other than image analysis may also benefit from the proposed methodology.

References

[1] A. Alves, L. Silva, J. Carreira, and J. Silva. Wpvm: Parallel computing for the people. In *HPCN'95 High Performance Computing and Network Conference*, Milan (http://dsg.dei.uc.pt/wpvm), 1995. Springer-Verlag.

[2] T. Anderson, D. Culler, D. Patterson, and T. N. Team. A case for now (network of workstations). *IEEE Micro*, February 1995.

[3] M. Baker, R. Buyya, and D. Hyde. Cluster computing: A high-performance contender. *IEEE Computer*, 32(7):79–83, July 1999.

[4] S. Balsamo, L. Donatiello, and N. V. Dijk. Bound performance models of heterogeneous parallel processing systems. *IEEE Transactions on Parallel and Distributed Systems*, 9(10), October 1998.

[5] J. Barbosa and A. Padilha. Algorithm-dependent method to determine the optimal number of computers in parallel virtual machines. In *VECPAR'98, 3rd International Meeting on Vector and Parallel Processing (Systems and Applications)*, volume 1573, Porto, 1998. Springer-Verlag.

[6] J. Choi, J. Dongarra, and D. Walker. The design of parallel dense linear software library: Reduction to hessenberg, tridiagonal and bidiagonal form. Technical Report LAPACK Working Note 92, University of Tennessee, Knoxville, January 1995.

[7] D. Culler, R. Karp, D. Patterson, A. Sahay, K. Schauser, E. Santos, R. Subramonian, and T. von Eicken. Logp: Towards a realistic model of parallel computation. In *4 ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, San Diego, CA, 1993.

[8] J. W. Demmel. *Applied Numerical Linear Algebra*. SIAM, 1997.

[9] J. Dongarra, S. Hammarling, and D. W. Walker. Key concepts for parallel out-of-core lu factorization. Technical Report CS-96-324, LAPACK Working Note 110, University of Tennessee Computer Science, Knoxville, April 1996.

[10] J. Dongarra and D. Walker. The design of linear algebra libraries for high performance computers. Technical Report LAPACK Working Note 58, University of Tennessee, Knoxville, June 1993.

[11] R. Geijn and J. Watts. Summa: Scalable universal matrix multiplication algorithm. Technical Report CS-95-286, University of Tennessee, Knoxville, 1995.

[12] G. Golub. *Matrix Computations*. The Johns Hopkins University Press, 1996.

[13] G. Henry, D. Watkins, and J. Dongarra. A parallel implementation of the nonsymmetric qr algorithm for distributed memory architectures. Technical Report Technical Report CS-97-352 and LAPACK Working Note 121, University of Tennessee, March 1997.

[14] J. JáJá and K. Ryu. The block distributed memory model. Technical Report CS-TR-3207, University of Maryland, January 1994.

[15] J. E. Marsden and A. J. Tromba. *Vector Calculus*. W. H. Freeman and Company, 1981.

[16] W. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical Recipes in C: The Art of Scientific Computing*. Cambridge University Press, 1997.

[17] E. Rosti, E. Smirni, L. Dowdy, G. Serazzi, and K. Sevcik. Processor saving scheduling policies for multiprocessor systems. *IEEE Transactions on Computers*, 47(2), February 1998.

[18] S. Sahni and V. Thanvantri. Performance metrics: Keeping the focus on runtime. *IEEE Parallel & Distributed Technology*, pages 43–56, Spring 1996.

[19] C. Seitz. Myrinet - a gigabit per second local-area network. *IEEE Micro*, February 1995.

[20] J. Shen and S. Castan. An optimal linear operator for step edge detection. *CVGIP: Graphical Models and Image Processing*, 54(2):112–133, 1992.

[21] C. Spurgeon. *Ethernet Configuration Guidelines*. Peer-to-Peer Communications, Inc, 1996.

[22] A. Steen. Methodology, metrics and presentation of results. In *Tutorial in VECPAR'98, 3rd International Meeting on Vector and Parallel Processing (Systems and Applications)*, Porto, 1998.

[23] L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, August 1990.

Biographies

Jorge Barbosa got a diploma in Electrical Engineering from FEUP (Faculdade de Engenharia do Porto), a MSc. in Digital Systems from UMIST, UK, and he is currently a PhD. student at FEUP, researching the application of parallel computing in image processing. João Tavares got a diploma in Mechanical Engineering and a MSc. in Electrical Engineering from FEUP, and he is currently a PhD. student at FEUP researching DEFORMABLE object models in image processing. Armando Padilha is ASSOCIATE PROFESSOR at FEUP and GROUP research leader at INEB (BIOMEDICAL ENGINEERING INSTITUTE, http://ineb.fe.up.pt).

New Cost Metrics for Iterative Task Assignment Algorithms in Heterogeneous Computing Systems

Raju D. Venkataramana & N. Ranganathan
Department of Computer Science and Engineering
University Of South Florida
Tampa, FL 33620, USA
venkatar@csee.usf.edu & ranganat@csee.usf.edu

Abstract

Task assignment and scheduling algorithms for Heterogeneous computing systems can be classified as iterative and non-iterative techniques, and are designed to optimize a specific cost function defined on the system. The quality of the solutions generated is controlled by the nature of this cost metric. The common metrics that are used include minimizing the overall execution time or minimizing the load on the maximum loaded processor. In this work, a new set of cost metrics have been proposed that can be used by iterative task assignment algorithms. These metrics exploit the fact that in iterative algorithms the mapping of the subtasks to the processors is known at every iteration. They reflect the actual scheduling cost of the application, thereby improving the quality of the solutions generated by the algorithm. The proposed metrics are evaluated using the learning automata based iterative algorithm [15]. Observations are made regarding the nature of the metrics from the results obtained.

Key Words: *Task assignment and scheduling, Heterogeneous computing, Cost function.*

1 Introduction

Efficient task assignment and scheduling is critical to achieving high performance in Heterogeneous Computing(HC) systems [2, 10]. In these systems, applications are represented as a directed acyclic graph called the task flow graph(TFG), and the processing resources are represented as a directed graph called the processor graph(PG). The purpose of scheduling is to map the tasks to the available processors and order their execution, so that the task precedence requirements are satisfied and the schedule length is minimized. It has been shown that the scheduling problem in general is an NP-complete problem [14], and hence a number of heuristic algorithms have been proposed to solve it.

These algorithms can be broadly classified as iterative and non-iterative algorithms. Proposed works in the former category include [12, 13, 15, 17], and the algorithms in the latter are [1, 3, 4, 5, 6, 9, 11, 16]. The non-iterative algorithms work by exploiting the graph-theoretic properties of the TFG to generate a solution that optimizes a specific cost function. The iterative algorithms on the other hand, proceed by generating an initial random solution and then progressively improving it, subject once again to optimizing the cost criterion. Due to the difference in approach of the two classes of algorithms, the influence of the cost metric on their ability to generate efficient solutions varies. But, traditionally, generic cost metrics like minimization of overall execution time or minimization of the load on the maximum loaded processor have been used for both classes of algorithms. In this work, we propose a new set of cost metrics that are applicable to the iterative algorithms. The proposed metrics generate solutions that are closer to the actual schedule time.

The material in this paper is organized as follows. Section 2 begins with the required preliminaries and explains the system model within which the metrics have been defined. The next section describes the proposed set of cost functions. Section 4 evaluates these functions and makes observations about the efficiency based on the results obtained. The last section concludes the work.

2 Preliminaries

This section introduces the required preliminaries and describes the system model within which the proposed cost metrics have been defined.

It is assumed that the application has been partitioned into subtasks and modeled by means of a directed acyclic

0-7695-0556-2/00 $10.00 © 2000 IEEE

graph called the task flow graph. The nodes of the TFG correspond to the subtasks and the edges represent the data-dependencies between them. The nodes are represented by the set S and the edges by the set E^{TFG}. Hence,

$S = \{s_i, 0 \leq i < |S|\}$ and

$E^{TFG} = \{(i,j) \,/\, s_i, s_j \in S \text{ and } s_i \text{ dependent on } s_j\}$

Every directed edge in the graph indicates the flow of data from one subtask to another. The edges are assigned a weight that corresponds to the number of data-units exchanged between the corresponding pair of subtasks. If $e_{i,j}^{TFG}$ represents the edge weight, then:

$$e_{i,j}^{TFG} = \begin{cases} \#\ of\ data\ units\ exchanged\ between\ s_i\ and\ s_j, & \text{if } (i,j) \in E^{TFG}; \\ 0, & \text{otherwise.} \end{cases}$$

The processor configuration is assumed to be modeled as a directed graph called the processor graph. Here, the nodes correspond to the processors and the edges to the communication links between them. Let M represent the set of nodes and E^{PG} the set of edges, then:

$M = \{m_i, 0 \leq i < |M|\}$ and

$E^{PG} = \{(i,j) \,/\, m_i, m_j \in M \text{ and } m_i \text{ is connected to } m_j\}$

Here again, the edges which indicate whether or not a set of processors are connected, are assigned weights that correspond to the cost of communicating a single unit of data from one processor to another. Let $e_{i,j}^{PG}$ represent the edge weight, then:

$$e_{i,j}^{PG} = \begin{cases} cost\ of\ communicating\ a\ data\ unit\ between\ m_i\ and\ m_j, & \text{if } (i,j) \in E^{PG}; \\ \infty, & \text{otherwise.} \end{cases}$$

In addition to this information, it is assumed that the cost of executing each of the subtasks on the processors are known. These values are stored as a matrix called E_T. The matrix can be represented as:

$E\_T = \{e\_t(i,j),\ 0 \leq i < |S|,\ 0 \leq j < |M|\}$.

$e\_t(i,j) =$ *execution time of subtask* s_i *on machine* m_j.

Since in this work we are concerned only with the iterative algorithms, there should be a means of representing the solution generated at every iteration. In general, the solution can be conceived as a mapping, π, from the set of subtasks to the set of processors.

$\pi : S \rightarrow M$.

Let 'n' represent the iteration number. Then the solution generated at iteration 'n' can be represented as $\pi_n(i)$, where:

$\pi_n(i) \rightarrow$ *indicates the machine* 'm_j' *to which subtask* 's_i' *is assigned to at iteration* 'n'.

The representation of the system model is now complete and the cost functions can be defined on the system. This forms the subject of the next section.

3 Proposed Cost Metrics

The objective of iterative task assignment algorithms is to explore the solution space efficiently in order to seek the global optimal solution. The solution space is characterized by the cost function defined on the system, and hence becomes critical to determining its performance. In these assignment algorithms, the process of determining the final solution proceeds by initially generating a random solution. This is then evaluated for its merit, based on which a new improved solution is generated in the next iteration. The process is repeated until the solution converges, or in other words when there is no further improvement in the quality of the solution. Hence, at each iteration of the algorithm, the mapping of the subtasks to the machines is known. In previous works, this information is neglected when trying to determine the solution to the assignment problem. But it can be utilized to explore the solution space more efficiently and bring the final solution closer to the actual scheduling cost. In this work, we propose precisely such a set of metrics.

To begin with, let us define terminologies that will help develop the cost metrics. For each node $s_i \in S$ in the TFG we associate three schedule times, for each of the machines $m_j \in M$ a machine start time, and arrays of nodes. These are defined as:

$MST[j,l] \rightarrow$ the machine start time for machine m_j at the beginning of level 'l'.

$EST[i] \rightarrow$ the earliest time at which the subtask s_i can begin its execution.

$WT[i] \rightarrow$ the amount of time the subtask s_i has to wait before it can begin its execution.

$CT[i] \rightarrow$ the time at which the subtask s_i completes its execution.

$pred_i[] \rightarrow$ an array consisting of the predecessor nodes of subtask s_i.

$order_{j,l}[] \rightarrow$ an array that specifies the order in which the subtasks in level 'l' and assigned to machine m_j are executed.

$TT_{j,l} \rightarrow$ represents the number of subtasks in $order_{j,l}[]$.

It can be readily inferred that for any node s_i in the TFG, the completion time $CT[i]$ at iteration 'n' can be computed as:

$$CT[i] = EST[i] + WT[i] + e\_t(i, \pi_n(i))$$

In order to compute MST, EST and WT of the nodes however, the information about the structure of the TFG is

needed. This is achieved by leveling the TFG. The levelizing process is similar to the levelized min time heuristic proposed in [7]. The root node or nodes are first assigned, '*level* 0'. It's successor nodes, if all of their predecessors are in '*level* 0', are assigned '*level* 1'. The next successor nodes, if all of their predecessors are either in '*level* 0' or '*level* 1', are assigned '*level* 2' and so on. Finally the leaf nodes are assigned a level number equal to the height of the TFG. Now, the machine start time, earliest start time and the wait times of the nodes can be computed.

The machine start time MST, refers to the time at which a particular machine can begin executing tasks from a particular level. This is important, since nodes or tasks at a later level have to wait until all the tasks in previous levels have completed execution on the machines they were assigned to. Therefore we have:

$$MST[j,l] = \begin{cases} 0, & \text{if } l = 0; \\ Max_{k=0}^{TT_{j,l-1}} \{CT[order_{j,l-1}[k]]\}, & \text{otherwise.} \end{cases}$$

Assume that a subtask s_i has 'p' predecessor nodes, represented as defined previously by the array $pred_i[k]$ with $0 \leq k < p-1$. Then,

$$EST[i] = \begin{cases} 0, & \text{if } p = 0; \\ Max_{k=0}^{p-1} X(k), & \text{otherwise.} \end{cases}$$

where $X(k) = Max\{ \{CT[pred_i[k]] + e^{TFG}_{pred_i(k),i} * e^{PG}_{\pi_n(pred_i(k)),\pi_n(i)}\}, MST[\pi_n(i),l] \}$

To complete the calculation of the completion time for each of the subtasks, the wait time has to be defined. The wait time determines when a task will begin its execution and hence determines the efficiency of the cost metrics. If a subtask is the only task that has been mapped to a particular processor, then it does not have to wait to begin its execution. It's wait time therefore will be equal to zero. But if more than one subtask is mapped to the same machine, then it's possible to order their execution so that a more optimal solution can be obtained. Since the TFG has task precedence constraints, only the subtasks in the same level can be considered for this ordering. The ordering of the subtasks determines the wait time for each of them. Here, three different orderings of the subtasks that result in three cost metrics, named CM_1, CM_2, CM_3 are proposed.

Cost Metric CM_1:
The subtasks from the same level and assigned to a particular machine, are executed in the non-decreasing order of their earliest start times. For instance, if we have three subtasks s_1, s_2 and s_3, and assume that $EST[1] < EST[2] < EST[3]$, then the subtask s_1 will be executed first, followed by s_2 and then s_3.

Cost Metric CM_2:
In the second cost metric, the subtasks are executed in the non-decreasing order of their expected execution times. For the aforementioned three tasks if we assume that $e\_t(3,j) < e\_t(2,j) < e\_t(1,j)$, where '$m_j$' is the machine to which they are assigned, then the subtask s_3 is executed first, followed by s_2 and s_1.

Cost Metric CM_3:
The last cost metric that is proposed here, in a way combines the ideas of the previous two cost metrics. It orders the subtasks in the non-decreasing order of the sum of the earliest execution time and the expected execution time of the subtasks.

The difference in these cost metrics can be clearly understood by means of an illustrative example. Assume that three subtasks s_1, s_2 and s_3 belong to the same level in a TFG, and are assigned to the machine m_j. Let their earliest start times and expected execution times be:

$EST[1] = 4$, *and* $e\_t[1,j] = 11$ *time units.*
$EST[2] = 7$, *and* $e\_t[2,j] = 6$ *time units.*
$EST[3] = 18$, *and* $e\_t[3,j] = 3$ *time units.*

Figure 1 shows the schedule times for the three subtasks based on the ordering of CM_1. Here, the maximum of the completion times amongst the subtasks is 24 time units. The schedule times corresponding to the order of CM_2 is shown in Figure 2. For this metric the maximum of the completion times is 38 time units. The final metric CM_3, results in the schedule times shown in Figure 3. Here, the maximum of the completion times is 27 time units. Therefore, for the example case, the ordering of CM_1 offers the best solution as it results in the minimum schedule time(24), amongst the three metrics.

The different orderings of the subtasks represented by the metrics can be incorporated into the generic cost metric by means of the wait time. The array, $order_{j,l}[]$, specifies the ordering of the subtasks in level l and assigned to machine m_j, as shown in the initial definitions. Now the wait time can be generically defined as:

$$WT[order_{j,l}[k]] = \begin{cases} 0, & \text{if } k = 0; \\ (Mod(CT[k-1] - EST[k]) + (CT[k-1] - EST[k])) / 2, & \text{otherwise.} \end{cases}$$

where $0 \leq k < TT_{j,l}$.

It can be observed that the proposed metrics are very similar to scheduling heuristics proposed previously in the literature, and can be easily confused with them. But there are important distinctions between the cost metrics proposed in this work and these heuristics. A scheduling heursitic works on a fixed assignment and tries to generate an optimal schedule of the mapped tasks so that the overall completion time of the application is minimized. In our approach on the other hand, the cost metric is used to as a measure of efficiency for the solutions generated. This is then used by the iterative algorithm to move towards a better and more optimal solution for the assignment problem. There is also a clear distinction between dynamic scheduling algorithms proposed in [8] and the metrics presented in this work. The dynamic scheduling algorithms begin with an intial assignment and attempt to determine the optimal remapping for each of the subtasks in turn by using the information on subtasks that have already completed execution and those that need to be executed. The cost metrics CM_1, CM_2 and CM_3, at every iteration consider the entire mapping of subtasks to the various machines and compute a cost that is as close as possible to the actual scheduling cost. This metric is then exploited by the algorithm to explore the solution space more efficiently. Hence, although the construction of the proposed metrics are similar to scheduling heuristics, there are important distinctions between them.

At every iteration of any iterative algorithm therefore, the completion times of each of the subtasks can be computed using one of the three proposed cost metrics. The objective of the algorithm would then be to minimize the completion time of the subtask that has the maximum completion time. The next section present the results that were obtained using these cost metrics.

4 Results and Observations

To evaluate the proposed cost metrics, the learning automata based iterative algorithm [15] is used, though the metrics can be used by any iterative assignment algorithm. The primary reason for using this algorithm is because it can be adapted for any user specified cost metric without requiring a change in the construction of the algorithm. A short description of the algorithm is presented first.

The task assignment algorithm proposed in [15] works on a framework consisting of a HC system model and a learning automata model. The system model abstracts the application as a TFG and the suite of machines as a PG, similar to the model presented in this work. The algorithm can be adapted to work for any cost metric that can be defined on the system by the user. This feature is realized by means of the learning automata model. It is constructed by associatong every task in the TFG with a variable structure stochastic automaton. The HC system model serves as the external environment for these automata. Six heuristics were investigated to construct the learning algorithm. The best of these heuristics is used in this work.

The simulation environment consists of the task flow graph and the processor graph which are generated at random. The edge weights of the graphs are also assumed to be generated at random with equal probability over some predefined ranges. The values for these ranges are presented in Table 1. It is assumed that the iterations of the algorithm are continued until the probability of the actions of the automata reach 0.99, or if the number of iterations reaches a specified user limit. In all the experiments that were conducted, the algorithm terminated due to the former reason indicating that the solutions were convergent.

For the first set of experiments, the communication complexity was assumed to be low. In other words, the number of edges in the TFG is equal to one-third the number of tasks. The number of processors were varied between 2,5,10 and 20. The results obtained are presented in Figures 4 - 7. It can be observed from the graphs that the costs generated by the metrics differ in their optimality. They can therefore be used to deliver better solutions to the assignment problem.

The second set of experiments was conducted with a medium communication complexity, where the number of edges were equal to two-thirds that of the tasks. These results are presented in Figures 8 - 11. A similar observation as the first set of experiments can be made here, proving once again the utility of the metrics proposed. From both sets of experiments, it can be seen that when the communication complexity increases it leads to an increased difference between the solutions generated by the metrics. The reason for this is that when the communication complexity increases the number of tasks being assigned to the same level also increases. Hence the different orderings of the proposed metrics have a greater impact on the solutions generated.

On the same lines, in both sets of experiments when the number of tasks are low, the costs generated are equal between the different metrics. This is due to the fact that very few of the tasks are at the same level number when the total number of tasks are low and hence the ordering of the subtasks does not have a big impact on the solutions generated.

In general it can be seen that the proposed metrics are affected by the number of subtasks in the TFG, their data dependencies represented as the communication complexity, and the number of processors available for scheduling. Since all these factors directly affect the actual scheduling cost of an application task executing in the HC system,

Number of tasks	10,25,50,75 and 100
Number of machines	2,5,10 and 20
Number of edges	$\|S\|/3$ and $2\|S\|/3$
Expected execution time range	1000
Communication time range	4
TFG edge weight range	500

Table 1. Parameters for generating TFG's and PG's

the proposed metrics can help in providing better solutions. Since in the results shown, the graphs, the expected execution time and communication time are all generated at random, the difference in efficiency of these metrics is not discernible.

5 Conclusions

A new set of cost metrics for iterative task assignment algorithms in HC systems were proposed. The metrics were developed by exploiting the fact that in iterative algorithms the mapping of the subtasks to the processors is known at all iterations. The proposed functions were evaluated using the learning automata based iterative algorithm [15]. The results obtained show that when the number of tasks in the TFG is low or when the communication complexity is low, there is not much of difference in the costs generated by the metrics. This difference increased when the communication complexity was increased. Since the performance of the proposed metrics depend on the factors that affect the scheduling cost, they reflect the actual scheduling time of the application. Therefore they can be used to improve the quality of solutions for the task assignment problem in heterogeneous computing systems.

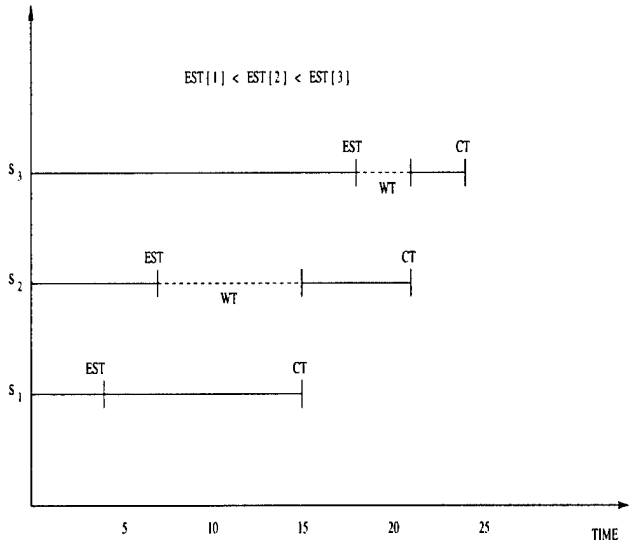


Figure 1. Example schedule times for CM_1

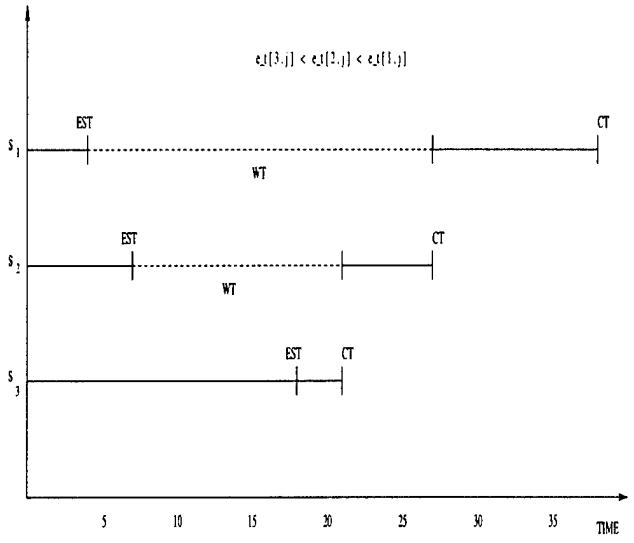


Figure 2. Example schedule times for CM_2

References

[1] Song Chen et al, "Selection Theory for Methodology for Heterogeneous Supercomputing," *Proceedings of Heterogeneous Computing Workshop,* pp. 15-22, Apr. 1993.

[2] M.M. Eshaghian, *Heterogeneous Computing,* Artech House, Norwood, MA, 1996.

[3] Chi-Chung Hui and S.T. Chanson, "Allocating Task Interaction Graphs to Processors in Heterogeneous Networks," *IEEE Trans. on Parallel and Distributed Systems,* vol. 8, no. 9, pp. 908-923, Sept. 1997.

[4] R.F. Freund, "Optimal Selection Theory for Superconcurrency," *Proceedings Supercomputing '89,* pp. 699-703, Nov. 1989.

[5] R.F. Freund "SuperC or Distributed Heterogeneous HPC," *Computing Systems in Engineering,* vol. 2, no. 9, pp. 349-355, 1991.

[6] R.F. Freund and H. J. Seigel, "Heterogeneous Processing," *IEEE Computer,* vol. 26, no. 6, pp. 13-17, Jun. 1993.

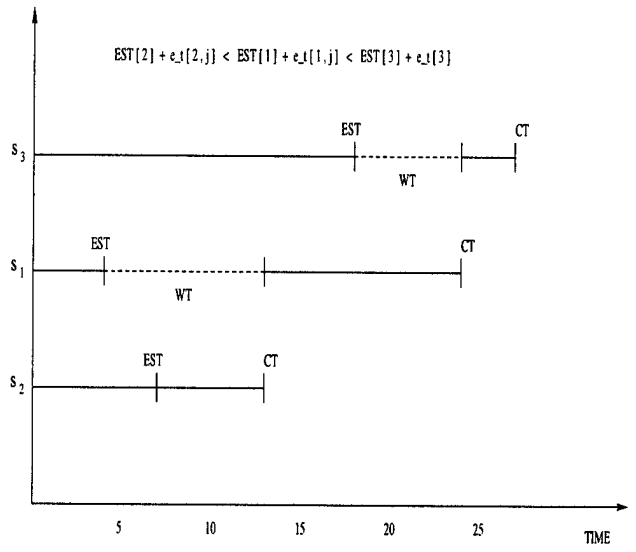


Figure 3. Example schedule times for CM_3

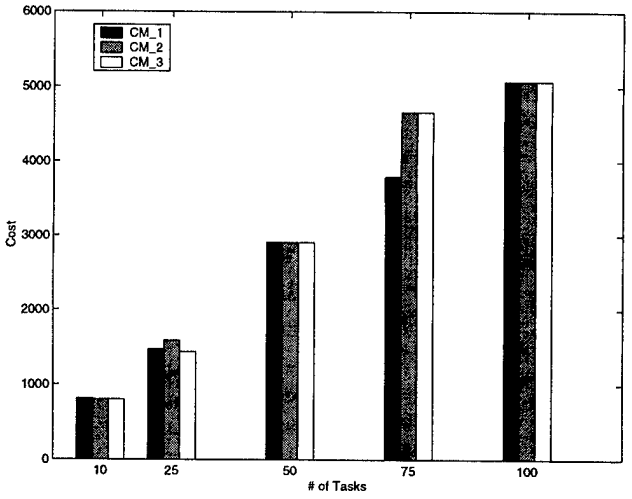


Figure 5. Low communication complexity, $|M| = 5$

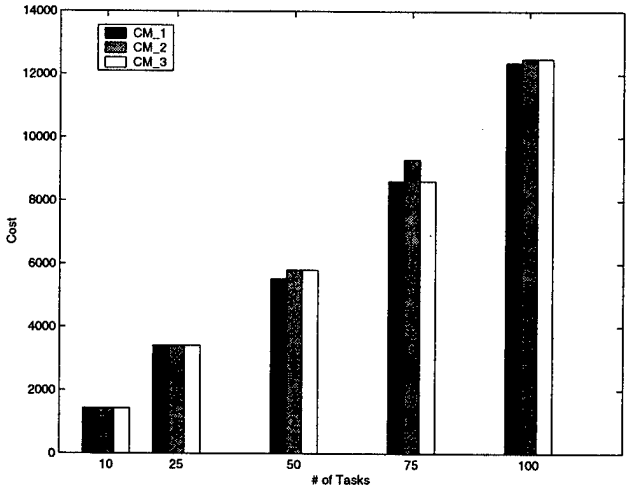


Figure 4. Low communication complexity, $|M| = 2$

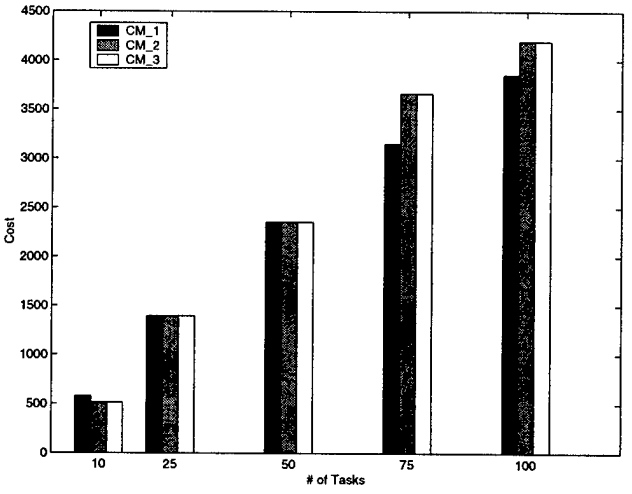


Figure 6. Low communication complexity, $|M| = 10$

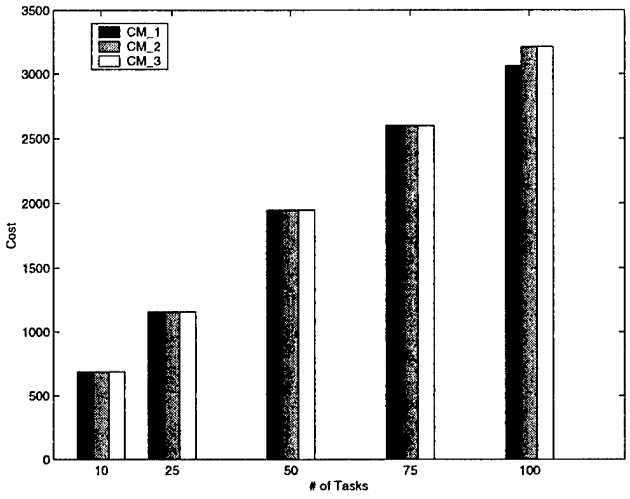


Figure 7. Low communication complexity, $|M| = 20$

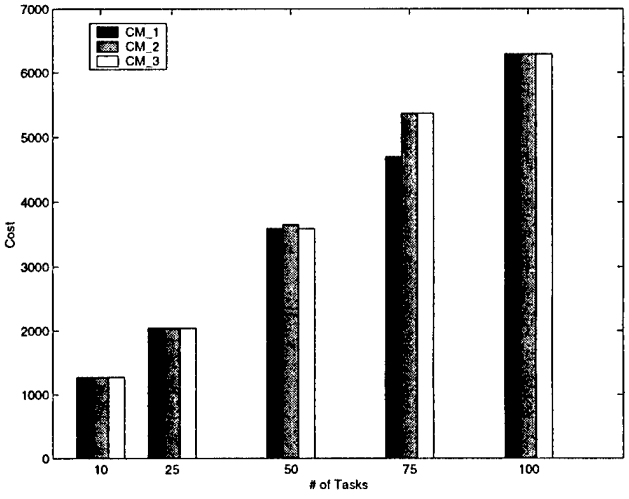


Figure 9. Medium communication complexity, $|M| = 5$

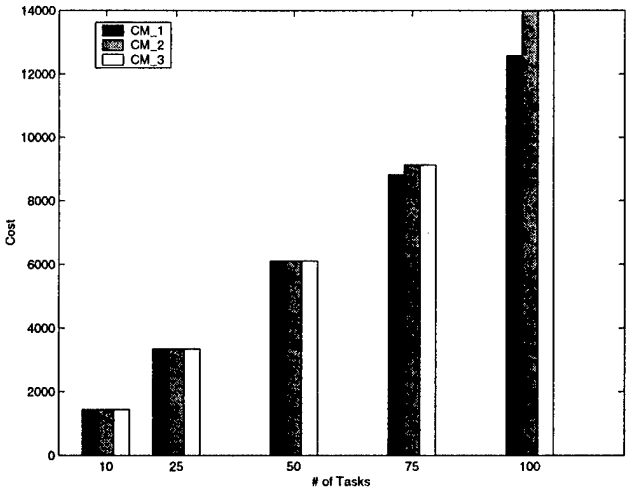


Figure 8. Medium communication complexity, $|M| = 2$

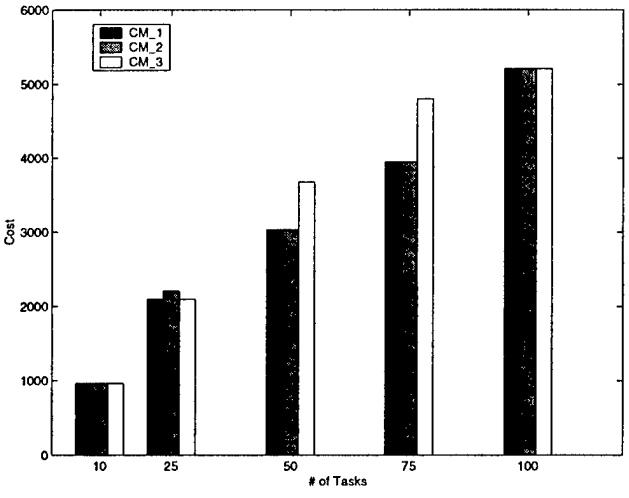


Figure 10. Medium communication complexity, $|M| = 10$

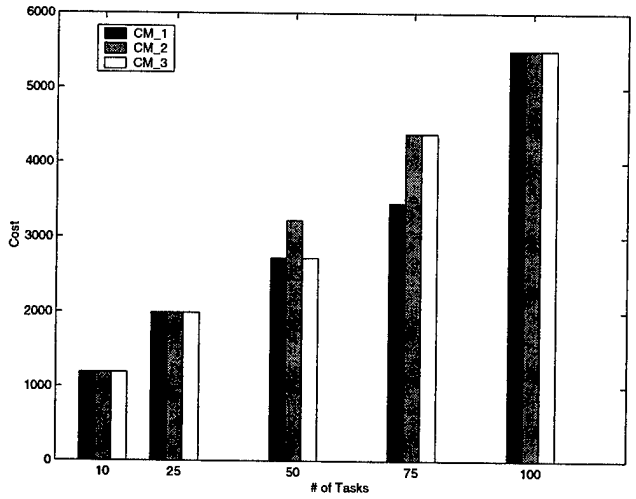


Figure 11. Medium communication complexity, $|M| = 20$

[7] M. Iverson, F. Ozguner, and G. Follen, "Parallelizing Existing Applications in a Distributed Heterogeneous Environment," *Proceedings of Heterogeneous Computing Workshop,* pp. 93-100, Apr. 1995.

[8] M. Maheswaran and H.J. Seigel, "A Dynamic Matching and Scheduling Algorithm for Heterogeneous Computing Systems," *Proceedings of Heterogeneous Computing Workshop,* Mar. 1998.

[9] B. Narahari, A. Youssef, and H.A. Choi, "Matching and Scheduling in a Generalized Optimal Selection Theory," *Proceedings of Heterogeneous Computing Workshop,* pp. 3-8, Apr. 1994.

[10] H.J. Seigel et al, "Heterogeneous Computing," *Parallel and Distributed Computing Handbook,* A.Y. Zomaya, McGraw-Hill, New York, pp. 725-761, 1996.

[11] C.C. Shen and W.H. Tsai, "A Graph Matching Approach to Optimal Task Assignment in Distributed Computing Systems Using a Minimax Criterion," *IEEE Trans. on Computer,* vol. 34, no. 3, pp. 197-203, Mar 1985.

[12] P. Shroff et al, "Genetic Simulated Annealing for Scheduling Data-dependent tasks in Heterogeneous Environments," *Proceedings of Heterogeneous Computing Workshop,* pp. 98-117, Apr. 1996.

[13] H. Singh and A. Youssef, "Mapping and Scheduling Heterogeneous Task Graphs using Genetic Algorithms," *Proceedings of Heterogeneous Computing Workshop,* pp. 86-97, Apr. 1996.

[14] H.S. Stone, "Multiprocessor Scheduling with the aid of Network Flow Algorithms," *IEEE Trans. Software Eng.,* vol. 3, no. 1, pp. 85-93, Jan. 1977.

[15] Raju D. Venkataramana and N. Ranganathan, "Multiple Cost Optimization for Task Assignment in Heterogeneous Computing Systems Using Learning Automata," *Proceedings of Heterogeneous Computing Workshop,* pp 137-145, Apr. 1999.

[16] M. Wang et al, "Augmenting the Optimal Selection Theory for Superconcurrency," *Proceedings of the Workshop on Heterogeneous Processing,* pp. 13-22, 1992.

[17] Lee Wang et al, "Task Matching and Scheduling in Heterogeneous Computing Environments Using a Genetic-Algorithm-Based Approach," *Journal of Parallel and Distributed Computing,* vol. 47, pp. 8-22, Nov. 1997.

Raju D. Venkataramana is currently pursuing a PhD in the Dept. of Computer Science and Engineering at the University of South Florida, Tampa. He received his B.E. in Computer Science and Engineering from Sri Venkateswara College of Engineering, University of Madras, India and Masters in Computer Science from the University of South Florida,Tampa in 1995 and 1997 respectively. His research interests include heterogeneous computing, parallel and distributed processing, interconnection networks and VLSI design.

N. Ranganathan is currently a Professor of Computer Science and Engineering at The Univ of South Florida, Tampa and was an Associate Professor from 1988-1998. He was also a Professor of Electrical and Computer Engineering at the University of Texas at El Paso, El Paso from 1998-1999. His research interests include VLSI design and hardware algorithms, computer architecture and parallel processing.

SESSION 3-A
HETEROGENEOUS ENVIRONMENT

Chair: M. Theys, *University of Illinois at Chicago, USA*

Reliable Cluster Computing with a New Checkpointing RAID-x Architecture

Kai Hwang, Hai Jin, Roy Ho, and Wonwoo Ro
Internet and Cluster Computing Laboratory
University of Southern California
Email : {kaihwang, hjin, wro}@usc.edu, and scho@csis.hku.hk

Abstract

In a serverless cluster of PCs or workstations, the cluster must allow remote file accesses or parallel I/O directly performed over disks distributed to all client nodes. We introduce a new distributed disk array, called the RAID-x, for use in serverless clusters. The RAID-x architecture is based on an *orthogonal striping and mirroring* (OSM) scheme, which exploits full-bandwidth and protects the system from all single disk failures.

The performance of the RAID-x is experimentally proven superior to RAID-1 and NFS in the Linux cluster environment. We propose a new *striped checkpointing* scheme, leveraging on striped parallelism and pipelined writing of successive disk stripes. This RAID-x architecture greatly enhances the throughput, reliability, and availability of scalable clusters. It appeals especially to I/O-centric cluster applications.

Keywords: Scalable computing, RAID architectures, parallel I/O, Linux clusters, disk mirroring, single system image, checkpointing, staggered writing, and fault tolerance

1. Introduction

Many *redundant arrays of inexpensive disks* (RAID) [6] use independent disks under the control of a single or multiple controllers. The TickerTAIP [3] pioneered the *Parallel RAID* architecture for supporting parallel disk I/O with multiple controllers. Still, these parallel disk arrays are implemented as a centralized I/O subsystem. These RAID subsystems are often attached to a storage server or used as network-attached disks [10].

For this reason, we consider the classic disk arrays as a *centralized* RAID. In contrast, this paper deals only with *distributed* RAID architectures. This concept was investigated by Stonebraker and Schloss [25]. The actual prototyping of distributed RAIDs did not start until the Petal [17] and the Tertiary Disk project [26].

A distributed RAID is constructed out of dispersed disks, which are physically attached to different computer hosts through the network connections. The Petal was built with a chained declustering [12]. The Tertiary Disk was built with a RAID-5 architecture using software support by the serverless xFS file system [2].

The architecture and performance of a new distributed RAID architecture, namely the RAID-x, are reported here. The level x is yet to be rectified with an appropriate code assignment by the RAID Advisory Board [22]. Our RAID-x differs from existing distributed RAID architectures in many aspects.

First, the RAID-x is built with a new disk mirroring technique, called *orthogonal striping and mirroring* (OSM). The small write problem associated with RAID-5 is completely eliminated in this OSM approach. Second, we use cooperative disks instead of independent disks.

To enable true cooperation among dispersed disks, we have developed *cooperative disk drivers* (CDD) at the kernel level. Data consistency is maintained inside the CDD, instead of using a central network file system. Therefore, unmodified file system interface is available to users. Third, the RAID-x was specially designed over distributed disks for I/O-centric cluster computing.

The rest of the paper is organized as follows: Section 2 describes the Trojans cluster architecture and also presents an overview of distributed RAID architectures. Our RAID-x approach is compared with the architectural designs in Berkeley Tertiary Disks running the xFS,

0-7695-0556-2/00 $10.00 © 2000 IEEE

Digital Petal system and Princeton TickerTAIP parallel RAID system. Section 3 introduces the OSM scheme and the RAID-x architecture. We also compare RAID-x with RAID-1 for designing distributed disk arrays. Section 4 describes the architecture of the cooperative disk drivers and data consistency checking mechanisms.

Section 5 presents the benchmark performance results obtained on the Trojans cluster. Section 6 explains the striped staggering checkpointing scheme we developed on top of RAID-x. Section 7 gives out the preliminary experimental results on striped checkpointing overhead and the analysis of reliability issue of proposed checkpointing scheme. Section 8 summaries the contributions and identifies extended research work.

2. USC Trojans Cluster Architecture

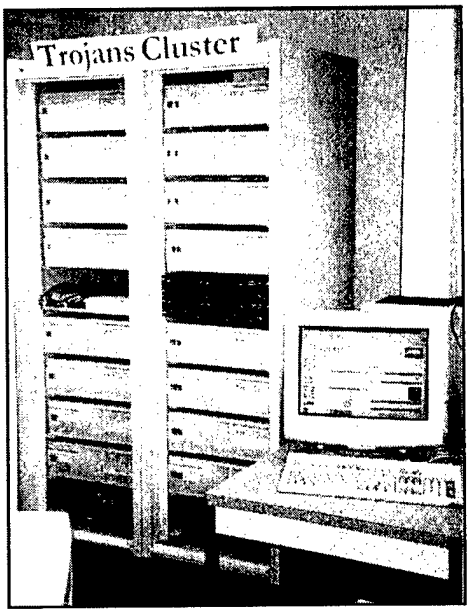
The prototype Trojans cluster was built with 16 Pentium PCs (Pentium II 400MHz) running the Linux operating system (Redhat Linux 6.0 with kernel 2.2.5). These PC nodes are connected by a 100 Mbps Fast Ethernet switch.

At present, each node is attached with a 10-GB disk. With 16 nodes, the total capacity of the disk array is 160 GB. All 16 disks form a single I/O space. Figure 1a shows the front view of the prototype Trojans cluster. This cluster is connected to Internet over fiber links.

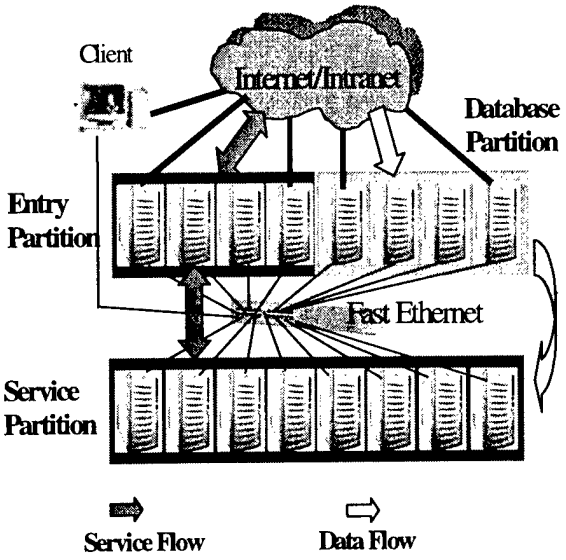
As illustrated in Fig.1b, we subdivide the cluster nodes into three functional classes. The *entry partition* is for the user to access the cluster through Internet/Intranet. Nodes in the *service partition* provide the services requested by users. The *database partition* supports database or information accesses operations. Nodes in the three partitions can be dynamically reconfigured to suit special application demands.

To build a distributed RAID with a SIOS, our research objectives are identified in three aspects: (i) A single address space for all data blocks in the cluster. This means that the users can utilize all disk storage in a cluster without knowing the physical locations of the data blocks referenced or of the files used. (ii) High scalability, availability, and compatibility with current cluster architectures and applications must be maintained. (iii) Remote disk I/O operations should have performance at least comparable to that of local disk I/O operations.

Previous approaches to achieve SIOS were attempted at the user level, file-system level, and device-driver level. The *user-level approach* has the lowest cost and higher portability across different platforms. The Parallel Virtual File System (PVFS) [18] and the Remote I/O project [9] are two examples. However, this approach does introduce two problems: First, users still have to use specific APIs and identifiers to exploit full functionality of the packages. Second, using system calls to perform network and file I/O are too expensive to meet real-time or cluster computing requirements.



(a) A front-view of the Trojans Cluster



(b) I/O-centric cluster architecture

Fig. 1. Trojans cluster built at USC Internet and Cluster Computing Laboratory

Distributed file systems provide another approach to achieving SIOS to the users. Users can access remote data as if it is accessed locally. The serverless xFS system developed at Berkeley [2] and the Solaris MC project are good examples. However, this approach has its own shortcomings.

Changing the file system does not guarantee high compatibility with current applications. This will discourage the deployment of the distributed file systems in clusters. What we want to achieve is a SIOS with an unmodified file system to achieve high portability with a low cost/performance ratio.

Device-driver level designs provide SIOS not only to the users, but also to the file system. We choose this approach, because it solves most of the above problems and shortcomings. Digital Petal project [17] uses user level device driver design to enable remote I/O access.

All physically distributed disks can be viewed as a collection of virtual disks. Each virtual disk can be accessed as if it is a local disk. Petal developed a distributed file system, called Frangipani [27]. In Petal, the actual data transfer is handled at the user level.

We have developed *Cooperative Device Drivers* (CDD). These drivers work cooperatively at the kernel level. Data consistency is maintained by the CDD. Unmodified file system is used to achieve high portability and compatibility.

The development of the RAID-x architecture was inspired by previous projects. The pioneering RAID work at Berkeley [2][6][8] and at CMU [10], the TickerTAIP project [3], the Tertiary Disk project [25], chained declustering [12], and Petal project [17] all have influenced our design philosophy.

Our RAID-x design appeals especially to serverless clusters. The major innovation in our design lies in the cooperation of distributed disks in a serverless cluster environment. The cooperation is established at the Linux kernel level, rather in the user space.

Petal and Tertiary Disk achieve the SIOS at the levels of user level device drivers and xFS file system, respectively. The Digital Petal virtual disks was built in 1996, the Berkeley Tertiary Disk project was reported in 1998, the Princeton TickerTAIP parallel RAID was designed at 1993, and our RAID-x built at USC Trojans project in 1999.

The entries in Table 1 distinguish the four parallel and distributed RAID architectures in four aspects. All four I/O subsystems support SIOS, however by quite different mechanisms. All four parallel and distributed RAIDs support parallel disk I/O at the block level.

The first distinction among the four distributed RAIDs lies in their architectures. The Petal virtual disk array uses chained declustering, Tertiary Disk applies the RAID-5, TickerTAIP uses parallel disk array controllers within single RAID server to implement parallel RAID-5, and we use the new RAID-x architecture.

Our major contributions lie in the creation of the OSM and CDD mechanisms. The enabling mechanisms for SIOS are also quite different among the four architectures. TickerTAIP achieves SIOS by event-driven simulation among all the worker nodes. We realize the SIOS with cooperative device driver at the Linux kernel level.

Table 1 Parallel and Distributed RAID Projects at USC, Princeton, Digital and Berkeley

System Attributes	USC Trojans RAID-x	Princeton TickerTAIP [3]	Digital Petal [17]	Berkeley Tertiary Disk [26]
RAID Architecture Environment	Orthogonal striping and mirroring over The RAID-x in a Linux cluster	RAID-5 with multiple controllers in a single server	Chained declustering in an Unix cluster	RAID-5 built with a Solaris PC cluster
Enabling Mechanism for SIOS	Cooperative device drivers in Linux kernels	Single server implements the SIOS directly	Petal device drivers at user level	xFS storage servers at file system level
Data Consistency Checking	Locks at device driver level	Sequencing of user requests	Supported by Frangipani file system	Locks in the xFS file system
Communication Mechanism	TCP/IP Sockets	Not Available	UDP/IP Sockets	RPC at user level

Even both Petal and RAID-x choose the device driver approach, their implementations are very different under UNIX user level and Linux kernel level. Petal does provide a global name space for logical disks in the cluster. We want to extend the global name space to each data block in the cluster.

The four RAID architectures differ in their handling of the data consistency problem in establishing a distributed file management system. We implemented the lock mechanisms within the device drivers. Our performance results are generated in Linux cluster environment.

For inter-node communications, we use the TCP/IP sockets. Regardless of their differences, we believe that hardware and software experiences learned from distributed RAID projects will be complementary to each other in many aspects.

For parallel writes, the RAID-x has lower access times than RAID-1. These claims are based on benchmark results to be presented in section 5. To sum up, the RAID-x scheme demonstrates scalable I/O bandwidth with much reduced latency in a cluster environment.

Using the CDDs, a cluster can be built serverless and offers remote disk access directly at the kernel level. Parallel I/O is made possible on any subset of local disks, because all distributed disks form a SIOS. No heavy cross-space system calls are needed to perform remote file accesses.

3. Orthogonal Striping and Mirroring

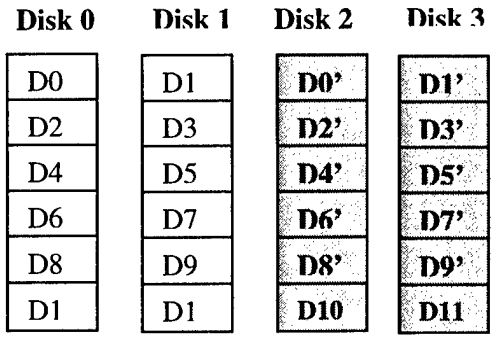
Over the years, many techniques have been developed to overcome the small-write problem [6][22], such as parity logging [24], floating parity and data [20], parity striping [7], disk caching disk [13], log-structured disk subsystem [19] and chained declustering [12]. The concept of OSM started with our earlier work [16].

In this paper, we present the design details of RAID-x and prove its effectiveness through experimentation. Figure 2 shows the architecture of RAID-x (Fig.2b) along with RAID-1 (Fig.2a) architectures. The original *data blocks* are denoted as Di in the unshaded boxes. The corresponding *image blocks* are distinguished with primes, such as Di' in the shaded boxes. The RAID-x completely avoids the small write problem.

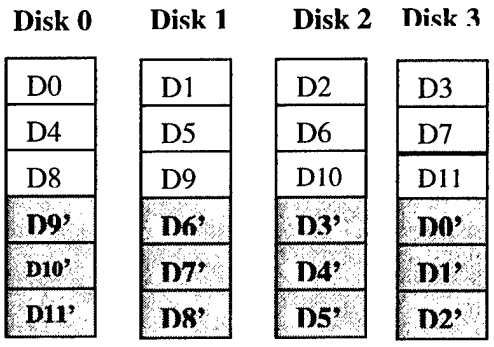
As shown in Fig.2b, data blocks in RAID-x are striped across the disks on the top half of the disk array. Low latency and high bandwidth of RAID-0 are preserved in RAID-x architecture. The image blocks of other data blocks in the same stripe are clustered in the same disk vertically. All image blocks occupy the lower half of the disk array. On a RAID-x, the images are copied and updated at the background, thus saving the overhead time.

Consider the top stripe of data blocks D0, D1, D2, and D3 in Fig.2b. Their image blocks D0', D1', and D2' are stored in Disk 3, while the image block D3' in disk 2. The rule is that no data block and its image should be mapped in the same disk. Full bandwidth is achievable in parallel disk I/O across the same stripe.

For large write, the data blocks are written in parallel to all disks simultaneously. The image blocks are gathered as a long block written into the same disk with a reduced latency. In case of the small write of a single block, the writing is directed to the data block, while the image block is postponed to write to the disk until all the clustered image blocks are ready.



(a) Duplicated striping in RAID-1



(b) Orthogonal striping and mirroring in RAID-x

Fig. 2 The mirroring schemes in RAID-1 and RAID-x

We define a pair of functions for the logical data block to physical RAID mapping: *data-mapping-function* and

mirror-mapping-function. The data-mapping-function is a one-to-one function which maps a logical RAID block address A to a physical disk address (DiskNo, StripeNo). Mirror-mapping-function maps the corresponding image block of a logical block address to a physical disk address. The A, DiskNo, and StripeNo count from 0.

We define n as the number of disks in the array, k as the number of blocks per disk, and A as the logical RAID block address. Table 2 gives out the data-mapping function and mirror-mapping function for RAID-1 and RAID-x. The notation *mod* stands for arithmetic modulo operation. Table 2 also lists the expected peak performance of two RAID architectures.

The *maximum bandwidth* of a disk array reflects the ideal case of parallel accesses of all useful data blocks. B stands for the bandwidth per disk. In the best case, a full bandwidth of nB can be delivered by RAID-x. The RAID-1 can only deliver half of the full bandwidth. The parallel read or parallel write time of a file of m blocks depends on the read or write latencies (R and W) per block, the array size n, and the file size m.

The entries given in Table 2 are expected peak performance of parallel disk I/O operations, excluding all software overhead or network delays. In case of large reads, mR/n latency is expected to perform m/n reads simultaneously for RAID-x, while RAID-1 needs to double the latency. For small read of a single block, both require R time to finish the read.

For parallel writes, as in RAID-x, the image blocks are clustered in one disk, written to the disk at the same time. That is, $m/n(n\text{-}1)$ image blocks are written together to each disk. Therefore, the large write latency is reduced to $mW/n + m/n(n\text{-}1)$.

For small writes, our RAID-x takes only W time to write the data block. The writing of the image blocks will be done later when all the stripe images are clustered at the same disk. This clustered writing can be done at the background, overlapping with the regular data writes.

Table 2 also shows the maximum number of disk failures that each disk array can tolerate. The RAID-x can tolerate single-disk failures, RAID-1 is more robust than RAID-x. The experimental results in section 6 will verify the accuracy of the expected performance.

Figure 3 illustrates an example of the two-dimensional RAID-x architecture with 3 disks attached to each node. The maximum number of disks attached to each SCSI controller is determined by the SCSI controller used. For Wide/Fast SCSI-II, 15 disks can be connected to one single SCSI controller.

In order to implement SIOS, addresses of all the data blocks are linearly continuous among all the member disks. Only the disks with same position corresponding to each node belong to one stripe group. All the disks within stripe group can be accessed in parallel.

Different stripe groups are independent. As all the disks within one node are connected through SCSI bus, different stripe group can be accessed in pipeline. The overlap degree for the different stripe group is depends on the property of SCSI bus used.

The Trojans cluster is presently being upgraded to 4 disks per node. Using 20 GB SCSI disks, the next RAID-x array will have 1.28 TB on 64 disks. In the future, the Trojans cluster will scale to hundreds of PC nodes or more, using next generation of microprocessors and Gigabit switched connections.

Using the Fast Ethernet, the aggregate I/O bandwidth is at most 12.5 MB/s. As reported in section 5, we have achieved 9.7 MB/s bandwidth for large parallel reads. This represents 78% efficiency in the cluster utilization.

Table 2 Architectural Characteristics of RAID-1 and RAID-x

Performance Indicators		RAID-1	RAID-x
Data Block mapping	**DiskNo.**	$A \bmod n/2$	$A \bmod n$
	StripeNo.	$(2A/n) \bmod k$	$(2\,A/n) \bmod k$
Mirror-mapping function	**DiskNo.**	$n/2 + A \bmod n/2$	$(-(A/(n-1)) \bmod k/2 - 1) \bmod n$
	StripeNo.	$(2A/n) \bmod k$	$k/2 + (A/(n-1)\,n)\,(n-1) + A \bmod (n-1)$
Max. Bandwidth	**Read/Write**	$nB/2$	$n\,B$
Estimates of Parallel Read/Write Time	**Large Read**	$2mR/n$	mR/n
	Small Read	R	R
	Large Write	$2mW/n$	$mW/n + m/n(n\text{-}1)$
	Small Write	W	$\approx W$
Max. Fault Coverage		$n/2$ disk failures	Single disk failure

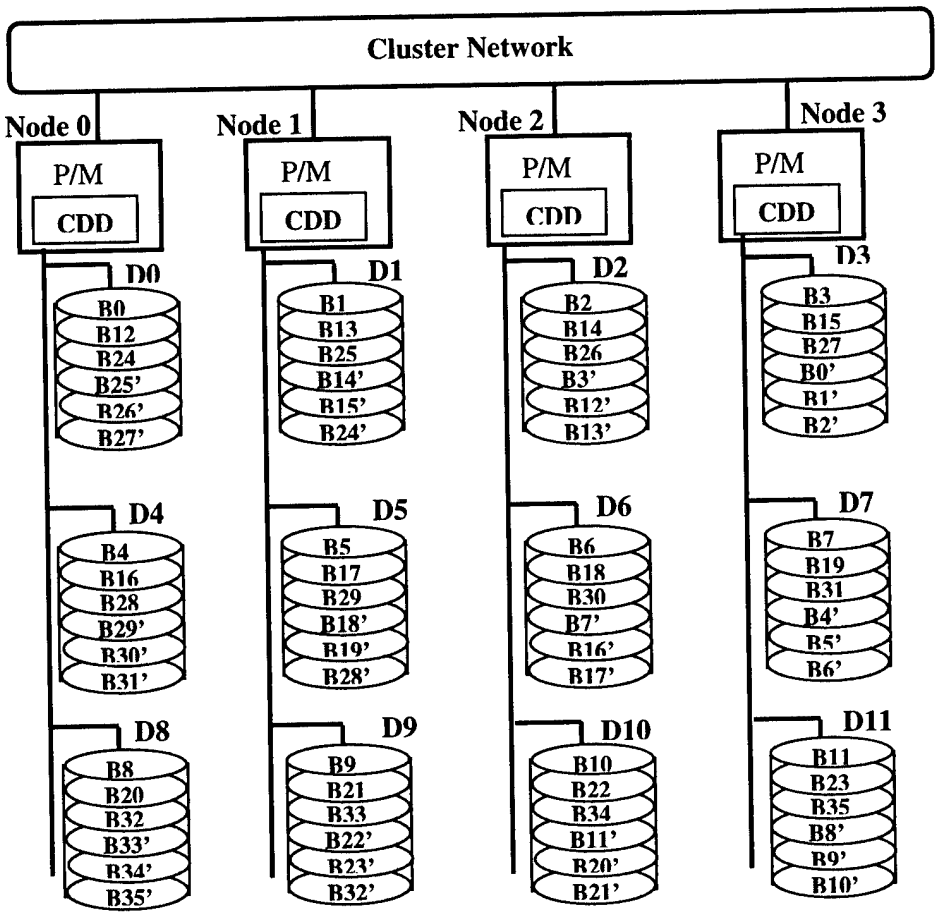


Figure 3. Distributed RAID-x architecture, shown with a 4 x 3 configuration

(P: processor, M: memory, CDD: cooperative disk drivers. All shaded blocks are mirrored images of the corresponding unshaded data blocks)

With a 128-node cluster and 8 disks per node, the disk array could be enlarged to have a total capacity exceeding 20 TB, suitable for any large-scale, database or multimedia applications. With an enlarged array of 128 disks, the cluster must be upgraded to a Gigabit switched connection. Based on the growing I/O bandwidth, the Trojans cluster and its RAID-x architecture show a very promising future in term of scalability and availability.

4. Cooperative Disk Drivers

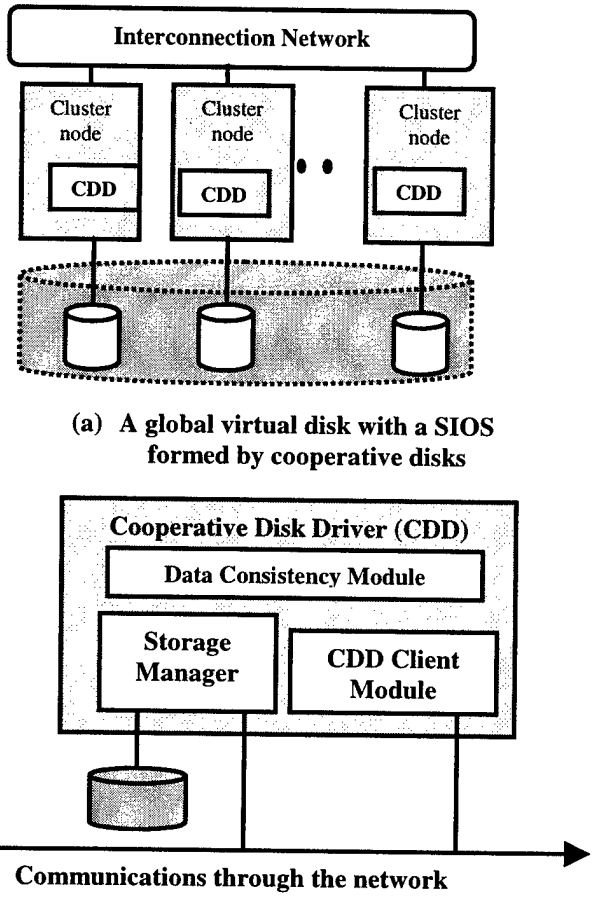
The *Single I/O space* (SIOS) is crucial to building scalable cluster of computers. A loosely coupled cluster use distributed disks driven by different hosts independently. The *independent* disk drivers handle distinct I/O address spaces. Without the SIOS, remote disk I/O must be done by a sequence of time-consuming system calls through a centralized file server (such as the use of NFS) across the cluster network.

On the other hand, the CDDs work together to establish the SIOS across all physically distributed disks. Once the SIOS is established, all disks are used collectively as a single *global virtual disk* shown in Fig.4a.

Each node perceives the illusion that it has several physical disks attached locally. Figure 4b shows the internal design of a CDD. Each CDD is essentially made from three working modules. The *storage manager* receives and processes the I/O requests from remote *client modules*. The client module redirects local I/O requests to

remote disk managers.

The *consistency module* is responsible for maintaining data consistency among distributed disks. A CDD can be configured to run as a storage manager or as a client, or both at the same time. There are three possible states of each disk: (1) a manager to coordinate use of local disk storage by remote nodes, (2) a client accessing remote disks through remote disk managers, and (3) both of the above functions.



(a) A global virtual disk with a SIOS formed by cooperative disks

(b) The CDD architecture

Figure 4 Single I/O space in RAID-x built at Linux kernel level.

The Petal virtual disk array uses chained declustering, Tertiary Disk applies the RAID-5, and we use the new RAID-x architecture. The major innovations in RAID-x architecture lie in the creation of the orthogonal striping and mirroring in mapping the data blocks and their images on the distributed disks.

The OSM scheme outperforms the chained declustering scheme mainly in parallel write operations. The RAID-x scheme demonstrates scalable I/O bandwidth with much reduced latency in a cluster environment. Both Petal and Tertiary Disk achieve the SIOS at the user level. We achieved the SIOS at the Linux kernel level. Using the CDDs, the cluster can be built serverless and offers remote disk access directly at the kernel level.

Parallel I/O is made possible on any subset of local disks, because all distributed disks form SIOS. No heavy cross-space system calls are needed to perform remote file access. A device masquerading technique is adopted here. Multiple CDDs run cooperatively to redirect I/O requests to remote disks.

Data consistency problems arise when multiple cluster nodes have cached copies of the same set of data blocks. The xFS approach and the Frangipani approach maintain the data consistency at the file system level. In our design, data consistency checking is maintained at the disk driver level.

Our approach simplifies the design and implementation of distributed file management services. Data consistency is maintained by all CDDs with higher speed and efficiency at the data block level. We introduced a special lock-group table for developing distributed file management services.

Each record in this table corresponds to a group of data blocks that have been granted to a specific CDD client with write permissions. The write locks in each record are granted and released atomically. This lock-group table is replicated among the data consistency modules in the CDDs. Which guarantee that file management operations are performed atomically.

5. Benchmark Performance Results

To test the cooperative operations among the CDDs residing on individual PCs, we use all 16 PCs as I/O storage servers. We use the same hardware platform to compare the relative performance of two disk array architectures: RAID-1 and RAID-x, all supported by CDDs. The NFS is used as a baseline for comparison purposes. Presently, Linux kernel version 2.2.5 supports the RAID-0, RAID-1, and RAID-5 configurations.

We implemented the RAID-x based on the RAID-0 implementation supported in the Linux kernel. This poses no difficulty in mapping the data blocks onto the top half of each disk. The mapping of the image blocks in the

RAID-x configuration is done by a special address translation subroutine residing in each CDD. To study the maximum I/O bandwidth of the disk array, the caches in the storage servers are bypassed by issuing a special *sync* command in the Linux kernel.

For reads or writes, the file size chosen was 10MB. Each block (stripe unit) in the disk is 4 KB. This means that a 10-MB file is striped uniformly across all 16 disks in consecutive stripe groups. We have performed three benchmark experiments.

The first two experiments measure the parallel I/O performance in terms of the *throughput* or the *aggregate I/O bandwidth*. The first experiment tests the throughput of RAID-x, RAID-1 and the NFS against the number of *client requests*. The second test checks the bandwidth against the disk array size for RAID-1 and RAID-x.

The distributed file system is evaluated in the third experiment using the standard Andrew Benchmark [11] consisting of a sequence of basic file system testing programs. There are five phases in the Andrew benchmark.

The first phase recursively creates subdirectories. The second phase measures the data transfer capabilities by copying files. The third phase recursively examines the status of directories and the associated files. The fourth phase scans the contents of each file. The final phase compiles the files and links them together.

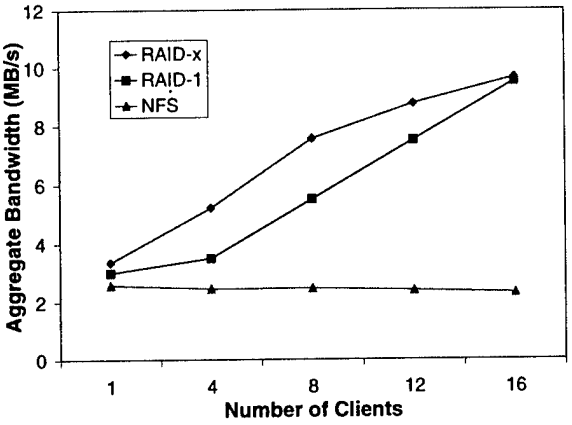
5.1. Bandwidth Results and Analysis

Figure 5 shows the performance of RAID-x, RAID-1 and NFS architectures. The results on parallel read are given in Fig. 5a. In this test, each client reads a 10MB-long file from all the disks. Therefore, the test is truly focused on the parallel I/O capability of the disk array. All the files are set to be uncached and each client only reads its own private file. All read operations are performed simultaneously, with the help of an MPI_Barrier() call.

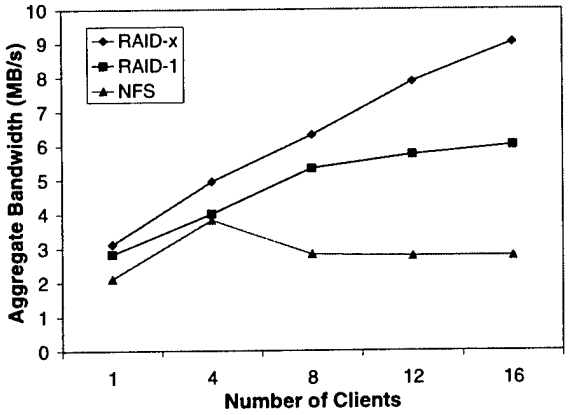
The NFS throughput is limited at 2.6 MB/s regardless of the number of clients, due to the fact that sequential I/O is performed by the NFS on a central server. As the request number increases, the NFS becoming the bottleneck shows a declining performance. RAID-x architectures scale up to a bandwidth of 9.7 MB/s for 16 clients. RAID-1 lags behind with a show of 6.33 MB/s for 16 clients.

Fig. 5b shows the write bandwidths of the RAID-x, RAID-1 and NFS subsystems. In this test, each client writes a 10MB-long file to the cache and issues a special *sync*() call to flush the data blocks to the disks. All write operations among the clients are also synchronized in these experiments.

The NFS scales in performance up to 4 requests. As the requests exceed 4, the NFS bandwidth drops to a low 2.77MB/s. For writes of a large file, RAID-x achieves the better scalability with a 9.02MB/s for 16 clients. RAID-1 saturates early to a 5.95MB/s, due to the fact that only half of the disks are used for data storage.



(a) Parallel read



(c) Parallel write

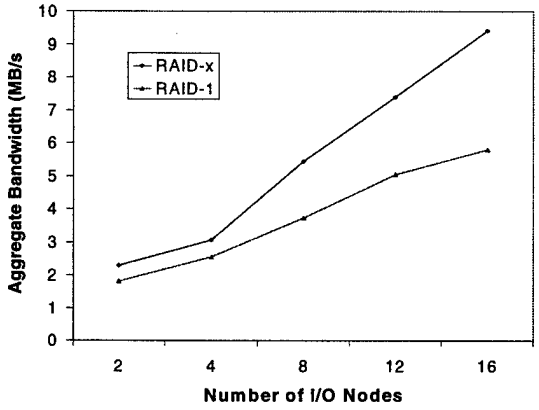
Fig. 5 Aggregate I/O bandwidth of RAID-x, RAID-1 and NFS with increasing clients

5.2. Raw I/O Performance of RAID-x

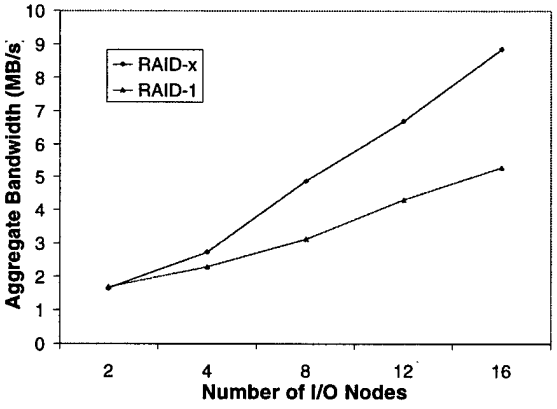
Raw I/O performance is plotted in Fig.6 against the disk array size. The results are shown for two RAID architectures. Again, all caches are bypassed in the experiments and the number of client processes is fixed at 16. The read ranking differs from the write ranking

sharply in these plots.

For parallel reads (Fig. 6a), the data size has very little effects on the relative standings of two RAIDs. It is important to note that the read bandwidth of RAID-x approaches 9.7 MB/s, about 78% of 12.5 MB/s, the limit of a 100 Mbps Fast Ethernet. The difference is attributed mainly to the CDD protocol and TCP/IP overheads incurred.



(a) Parallel read



(b) Parallel write

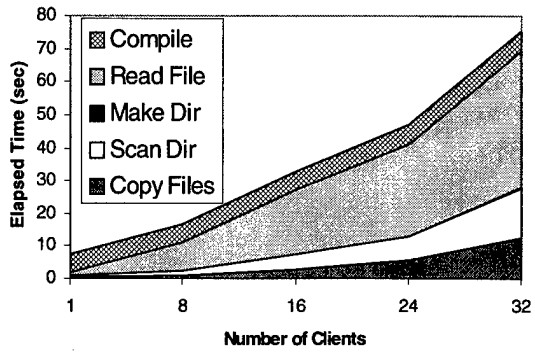
Fig. 6 Aggregate I/O bandwidth of RAID-x and RAID-1 with increasing disk numbers

For parallel writes, the large write bandwidths of RAID-x and RAID-1are 9.02MB/s and 5.72MB/s, respectively. Table 3 shows the improvement factor of 16 clients over 1 client in using the 16-node Trojans cluster. Comparing with Berkeley xFS results, our 1-client bandwidth is quite high due to well-exploited parallelism in 16-way striping across the disk array.

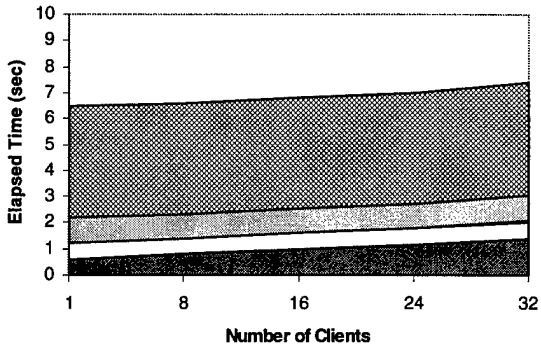
For this reason, the improvement factor is lower than that achieved by the xFS system. Again, the RAID-x demonstrated the highest improvement factor among the three distributed RAID architectures and the NFS.

5.3. Andrew Benchmark Results

Andrew benchmark tests the performance of a network file system. In this experiment, the Andrew benchmark was executed on four I/O subsystems with respect to increasing number of client requests up to 32. The performance is indicated by the elapsed time in executing Andrew benchmark on the target I/O subsystem. Figure 7 shows the benchmark results for RAID-x and NFS.



(a) NFS performance



(b) RAID-x performance

Fig. 7 Elapsed time to execute the Andrew benchmark on the Trojans cluster

These tests demonstrate how the underlying storage structures can affect the performance of the file system being supported. Each local file system on the I/O nodes mounts the "virtual" storage device provided by the CDD. The number of I/O nodes is fixed at 16. Each client only executes its own private copy of Andrew benchmark. We use the Linux ext2 local file system to keep the operations on metadata atomic.

Table 3 Achievable I/O Bandwidth and Improvement Factor on Trojans Cluster

I/O Operations	NFS			RAID-x		
	1 Client	16 Clients	Improve	1 Client	16 Clients	Improve
Large Read	2.58 MB/s	2.3 MB/s	0.89	3.36 MB/s	9.65 MB/s	2.87
Large Write	2.11 MB/s	2.77 MB/s	1.31	3.12 MB/s	9.02 MB/s	2.89
Small Write	2.47 MB/s	2.81 MB/s	1.34	3.22 MB/s	9.13 MB/s	2.84

Figure 7a shows the benchmark result of NFS, while Figures 7b shows the results of RAID-x. It is obvious that the elapsed time in using NFS increases sharply with the number of clients, while the RAID-x scheme can sustain the same workload. For 16 clients, the elapsed times for RAID-x and NFS are 6.8 and 33 seconds, respectively.

For 32 clients, these numbers increase to 7.41 and 75.5, respectively. From Fig. 7a, NFS shows a worsening performance especially in reading the files, scanning directories, and copying files operations. The RAID-x architectures, in contrast, do not share this weakness.

6. Striped and Staggered Checkpointing

The parallel I/O characteristic of distributed RAID-x architecture can be applied to achieve fast checkpointing in the cluster system. Striped checkpointing method is storing checkpointing file over distributed RAID-x system. To alleviate the network contention, the staggered writing skill is combined to striped checkpointing.

Simultaneous writing of multiple processes in coordinated checkpointing may cause a network contention and I/O bottleneck problem to a central stable storage. As suggested by Vaidya [28], staggered writing of the checkpoints taken by different nodes reduces the above contentions. The time lag between staggered checkpointers can alleviate the bottleneck problem associated with the central stable storage.

The basic concept of staggered checkpointing allows only one process to store the checkpoint at a time. A token is passed around to determine the timing. When a node receives the token, the node starts to store the checkpoint. After finishing checkpointing, the node passes the token to the next node.

Our work on coordinated checkpointing was inspired by the previous works by Cao [4] and associates, Chandy and Lamport [5], and Vaidya [28]. In our scheme, several nodes within the cluster form a striped group. Only the nodes within the same striped group checkpoint simultaneously and each of the groups checkpoints in a staggered way.

Figure 8 shows the concept of striped staggering in coordinated checkpointing on the RAID-x disk array. The drawing shows a 12-disk RAID-x array configured as a 2-dimensional structure, i.e. a 4 x 3 configuration. Each *stripe* corresponds to the *degree of parallelism* (DOP) in concurrent accesses of four disks in the 4 x 3 disk array.

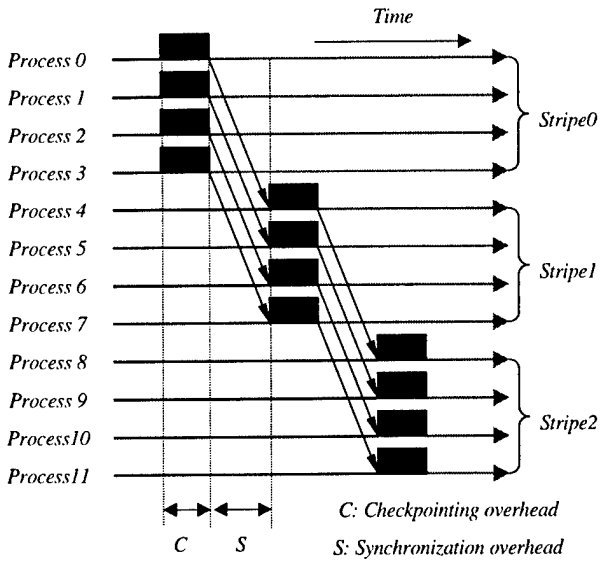


Fig. 8. Striped checkpointing with staggering on a distributed RAID-x

Successive stripes are accessed in a staggered manner from different stripes on successive 4-disk groups, as demonstrated in Fig.3. Staggering implies pipelined accesses of the disk array. We first proposed the idea of striped checkpointing in [23]. There exists trade-off between stripe parallelism and staggering depth.

For example, the layout in Fig.8 can be reconfigured

from 4 x 3 to a 6 x 2 configuration, if needed. Higher DOP leads to higher aggregate disk bandwidth. Higher staggering degree can cope better the network contention problem. The staggered writing way can reduce the average checkpointing overhead. However, in the case of blocking algorithm, the staggered writing method also introduces the synchronization time.

Although blocking algorithm is the simpler than non-blocking algorithm to achieve coordinated checkpointing in parallel processing, it suffers from large amount of overhead. Every node should be blocked during the checkpointing procedure. The basic idea is to shut down all processes temporary to define consistent state. After all the processes are blocked and all the messages are clearly delivered, the global checkpoints are stored. In the staggered writing case, the blocked time increases according to the number of node.

7. Overhead and Reliability Analysis

Figure 9 shows the advantage of striped staggering on distributed disk array, as compared with staggering in Vaidya scheme [28] on a centralized disk and the conventional approach using the NFS server. These preliminary results were measured on the small prototype Trojans cluster.

Our striped checkpointing scheme has the lowest overhead, especially when the checkpoint files becomes very large. Through continued experiments on the enlarged 64-disk RAID-x cluster, we will reveal more experimental results on the checkpointing overhead and rollback recovery latency.

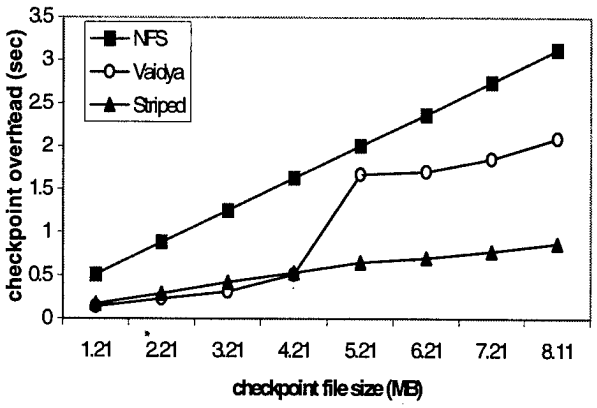


Fig. 9 Checkpointing overhead of staggered writing on distributed RAIDs

Table 4 summarizes three checkpointing schemes we have compared in this paper. Their advantages and shortcomings are identified. Suitable applications for each checkpointing scheme are also elaborated.

Using the OSM, each striped checkpointing file has its mirrored image in its local disk. For each node, transit failure can be recovered from its mirrored image in local disk. Permanent failure of a disk can be recovered from the striped checkpointing among the distributed disks.

The I/O performance in a degraded mode of OSM is the same as the RAID-0 performance in a normal mode. The striped checkpointing can be read in parallel from RAID-x. The checkpointing recovery latency can be shortened greatly.

Table 4 Summary of Three Coordinated Checkpointing Schemes

Checkpointing Scheme	Advantages	Shortcomings	Suitable applications
Simultaneous writing to a central storage (The NFS scheme)	Simple, no inconsistent state	Has network and I/O contentions, NFS is single point of failure	Small size of checkpoint, small number of nodes, low I/O operation
Staggered writing to a central storage (Vaidya scheme)	Eliminate the network and I/O contention	Network bandwidth is wasted, NFS is a single point of failure	Small size of checkpointers, small number of nodes, low I/O operations
Striped staggering checkpointing on any distributed RAID (Our scheme)	Eliminate network and I/O contentions, low checkpoint overhead, fully utilize network bandwidth, tolerate multiple failures among stripe groups	Can not tolerate more node failures within each stripe group	Large size of checkpointers, large number of nodes, low communication, I/O intensive applications

According to the mirror mapping of the OSM, the proposed RAID-x architecture can recover from any single disk failure in each stripe group. The total number of disk failure depends on the number of stripe groups to be accessed. For the 4 x 3 configuration in Fig.3, three disk failures in three stripe groups can be tolerated. An indepth analysis of the reliability of the proposed checkpointing RAID-x architecture is given in [23].

8. Conclusions

The development of the new RAID-x architecture was inspired by several research projects. The xFS and the Tertiary Disk projects at Berkeley [26], and the Petal project at Compaq Digital [17], all have influenced our design philosophy. The main difference between our approach and these projects is that we use the orthogonal striping and mirroring (OSM) to preserve both parallel disk accesses and staggered (pipelined) checkpointing of successive stripes.

We built data consistency checking in the device driver level. The CDDs work cooperatively to perform data transferring and consistency checking. With the support of CDDs, the design of a distributed file system can be focused on the concurrent file access policies and the related performance issues. In this case, the complexity of the distributed file system can be greatly reduced Our SIOS disk array separates the I/O subsystem into a distributed file system and a set of distributed CDDs.

All SSI services are provided by the CDDs while the file system modification is reduced to a minimum. Furthermore, some desired SSI services for cluster computing can be built on top of the SIOS. In this aspect, the SIOS is a very powerful middleware infrastructure to achieve single-system image. Benchmark performance results show that our distributed RAID can achieve scalability, performance, and availability in cluster computing.

The RAID-x outperforms the RAID-1 in the Linux cluster environment. For parallel reads with 16 active clients, the RAID-x achieved 9.7 MB/s throughput, 1.5 and 3.7 times higher than using RAID-1 and NFS, respectively. Running the Andrew benchmark, RAID-x results in a 17% cut in elapsed time, compared with that experienced on a RAID-1. The achieved throughput corresponds to 78% of the peak bandwidth deliverable by the Fast Ethernet. Scalable I/O bandwidth makes the RAID-x especially appealing to I/O-centric cluster applications.

The OSM mechanisms can be built not only on Linux PC clusters, but also on any Unix workstation clusters. These architectural features differ from the user-level designs in Berkeley Tertiary Disk and Digital Petal virtual disks. The new mechanisms support not only *single I/O space*, but also *distributed shared memory, checkpointing,* and *distributed file management* at the kernel level without using cross-space system calls.

The prototype RAID-x has the following open issues yet to be solved in future R/D efforts. These extended works are among the tasks planned in the next phase of our Trojans cluster project.

(1). We expect even higher performance as we continue improving the CDD protocol. The current hand shaking protocol could be improved with prefetching techniques. The TCP/IP used in our prototype is known for its high overhead. Plan is underway to port the whole cluster system with a low-latency protocol, expecting to further reduce the communication overhead.

(2). We plan to design a distributed file system with I/O load balancing capabilities along with an enlarged distributed disk array onto our Trojans cluster in the future. In addition to consider the RAID-1, RAID-5, and RAID-x configurations, we will also consider other configurations, such as RAID-10 and chained declustering.

(3). Our PC nodes in the Trojans cluster act as clients as well as storage servers at the same time. These dual roles affect the performance of the I/O nodes. We believe that the I/O performance can be further improved with an enlarged cluster size.

(4). We plan to develop a suite of middleware with striped staggering checkpointing to support process migration. Based on future Trojans cluster configuration, more detailed analysis of the DOP and depth of staggering will be conducted.

(5). New message logging algorithms for non-blocking striped checkpointing will be developed to reduce checkpointing overhead furthermore. We also plan to design an application dependent checkpointing scheme to elaborate the efficiency of striped checkpointing.

Lots of interesting research work can be generated out of a very large disk array in real-life applications. Potential applications are encouraged in biological sequence analysis, collaborative engineering design, clusters or grids for E-commerce, specialized digital libraries, and distributed multimedia processing.

References

[1] C. Amza, A. L. Cox, S. D. Wakadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel, "TreadMarks: Shared Memory Computing on Networks of Workstations", *IEEE Computer*, Vol.29, No.2, 1996, pp.18-28.

[2] T. Anderson, M. Dahlin, J. Neefe, D. Patterson, D. Roselli, and R. Wang. "Serverless Network File Systems". *ACM Trans. on Computer Systems*, Jan. 1996, pp.41-79.

[3] P. Cao, S. B. Lim, S. Venkataraman, and J. Wilkes, "The TickerTAIP Parallel RAID Architecture", *ACM Trans. on Computer System*, Vol.12, No.3, August 1994, pp.236-269.

[4] G. Cao and M. Singhal, "On Coordinated Checkpointing in Distributed Systems", *IEEE Transactions on Parallel and Distributed Systems*, Vol. 9, No. 12, pp. 1213-1225, Dec. 1998.

[5] K. M. Chandy and L. Lamport, "Distributed Snapshots: Determining Global States of Distributed Systems," *ACM Trans. Computer Systems*, pp. 63-75, Feb. 1985.

[6] P. M. Chen, E. K. Lee, G. A. Gibson, R. H. Katz and D. A. Patterson; "RAID: High-Performance, Reliable Secondary Storage", *ACM Computing Surveys*, Vol.26, No.2, June 1994, pp.145-185.

[7] S. Chen and D. Towsley, "The Design and Evaluation of RAID 5 and Parity Stripping Disk Array Architecture", *Journal of Parallel and Distributed Computing*, Vol.17, 1993, pp.58-74.

[8] M. Dahlin, R. Wang, T. Anderson, D. Patterson. "Cooperative Caching: Using Remote Client Memory to Improve File System Performance". *Proceedings of Operating System Design and Implementation*, 1994.

[9] I. Foster, D. Kohr, Jr., R. Krishnaiyer, and J. Mogill. "Remote I/O: Fast Access to Distant Storage". *Proc. of the Fifth Annual Workshop on I/O in Parallel and Distributed Systems*, November 1997, pp.14-25.

[10] G. Gibson, D. Nagle, K. Amiri, F. Chang, H. Gobioff, E. Riedel, D. Rochberg and J. Zelenka, "A Cost-effective, High-bandwidth Storage Architecture", *Proc. of the 8th Conf. on Architectural Support for Programming Langagues and Operating Systems*, 1998.

[11] J. H. Howard, M. L. Kazar, S. G. Menees, D. A. Nichols, M. Satyanarayanan, R. N. Sidebotham, and M. J. West, "Scale and Performance in a Distributed File System". *ACM Trans. on Computer System*, Vol.6, No.1, pp.51-81, February 1988.

[12] H. I. Hsiao and D. DeWitt, "Chained Declustering: A New Availability Strategy for Multiprocessor Database Machines", *Proc. of 6th International Data Engineering Conf.*, 1990, pp.456-465.

[13] Y. Hu and Q. Yang, "DCD - Disk Caching Disk: A New Approach for Boosting I/O Performance", *Proc. of the 23rd International Symp. On Computer Architecture*, 1996, pp.169-177.

[14] K. Hwang, H. Jin, E. Chow, C.L. Wang, and Z. Xu. "Designing SSI Clusters with Hierarchical Checkpointing and Single I/O Space". *IEEE Concurrency Magazine*, March 1999, pp.60-69.

[15] K. Hwang and Z. Xu. *Scalable Parallel Computing: Technology, Architecture, Programming*. McGraw-Hill, New York, 1998.

[16] H. Jin and K. Hwang, "Striped Mirroring Disk Array", *Journal of Systems Architecture,* Elsevier Science, The Netherlands, March 2000.

[17] E. K. Lee and C. A. Thekkath. "Petal: Distributed Virtual Disks". *Proceedings of the Seventh International conf. on Architectural Support for Programming Languages and Operating Systems*, Cambridge, MA, October 1996, pp.84-92.

[18] W. B. Ligon and R. B. Ross. "An Overview of the Parallel Virtual File System". *Proceedings of the 1999 Extreme Linux Workshop*, June 1999.

[19] B. McNutt, "Background Data Movement in a Log-Structured Disk Subsystem", *IBM Journal of Research and Development*, Vol.38, No.1, January 1994, pp.47-58.

[20] J. Menon, J. Roche and J. Kason, "Floating Parity and Data Disk Arrays", *Journals of Parallel and Distributed Computing*, January 1993.

[21] G. F. Pfister. "The Varieties of Single System Image", *Proceedings of IEEE Workshop on Advances in Parallel and Distributed System*, IEEE CS Press, 1993, pp.59-63.

[22] RAID Advisory Board, *The RAIDbook*, Seventh Edition, December 1998.

[23] W. Ro and K. Hwang, "Striped Staggering for Coordinated Checkpointing on Distributed RAIDs", *Technical Report*, Department of EE-Systems, University of Southern California, Feb.10, 2000.

[24] D. Stodolsky, M. Holland, W. V. Courtright II and G. A. Gibson, "Parity-Logging Disk arrays", *ACM Trans. on Computer Systems*, Vol.12, No.3, August 1994, pp.206-235.

[25] M. Stonebraker and G. A. Schloss, "Distributed RAID – a New Multiple Copy Algorithm", *Proc. of the Sixth*

International Conf. on Data Engineering, Feb. 1990, pp.430-437.

[26] N. Talagala, S. Asami, D. Patterson, and K. Lutz, "Tertiary Disk: Large Scale Distributed Storage", *UCB Technical Report No. UCB//CSD-98-989.*

[27] C. A. Thekkath, T. Mann, and E. K. Lee. "Frangipani: A Scalable Distributed File System". *Proceedings of ACM Symp. of Operating Systems Principles*, Oct. 1997, pp.224-237.

[28] N. H. Vaidya, "Staggered Consistent Checkpointing", *IEEE Transactions on Parallel and Distributed Systems*, 1999, Vol. 10, No. 7, pp. 694- 702.

Biographical Sketches:

Kai Hwang is a Professor of Electrical Engineering and Computer Science at the University of Southern California. An IEEE Fellow, he specializes in computer architecture, digital arithmetic, and parallel processing. He is a founding Editor of the *Journal of Parallel and Distributed Computing.* He received the Outstanding Achievement Award from the PDPTA in 1996.

He has published six books and over 160 technical papers in computer science and engineering. His current research interest lies in network-based PC or workstation clusters and the middleware support for security, availability, and single-system-image services. He can be reached by Email: kaihwang@usc.edu.

Hai Jin is an Associate Professor of Computer Science at Huazhong University of Science and Technology (HUST) in China. He received his Ph.D. in computer engineering from HUST in 1994. He has worked at the University of Hong Kong, where he participated in the HKU Cluster project. Presently, he works as a visiting scholar at the Internet and Cluster Computing Laboratory at USC.

A member of IEEE and ACM, he served as program committee chair of APSCC'2000. He has co-authored three books and published more than 30 papers in international journals and conferences. His research interests cover parallel I/O, RAID architecture design, fault tolerance, and cluster benchmark experiments. Contact him at hjin@ceng.usc.edu.

Roy Ho is currently working on his M. Phil. degree in the Computer Science and Information Systems Department, University of Hong Kong. He receives the B.S. degree in Computer Engineering from the University of Hong Kong in 1998. He has participated in this work, while he was visiting USC in Fall 1999. His research interest lies in computer architecture and scalable cluster computing. He can be reached by Email: scho@csis.hku.hk.

Wonwoo Ro presently works as a Ph.D. research assistant in the Department of Electrical Engineering at USC. He received the B.S. degree in Electrical Engineering from Yonsei University, Seoul, Korea, in 1996. He received the M.S. degree in Electrical Engineering from USC in 1999. His current research interest includes scalable cluster computing, checkpointing and fault tolerance. He can be reached by Email: wro@usc.edu.

Task Execution Time Modeling for Heterogeneous Computing Systems

Shoukat Ali[†], Howard Jay Siegel[†], Muthucumaru Maheswaran[‡],
Debra Hensgen[◇], and Sahra Ali[†]

[†]Purdue University
School of Electrical and Computer Engineering
West Lafayette, IN 47907-1285 USA
{alis@ecn., hj@}purdue.edu

[‡]Department of Computer Science
University of Manitoba
Winnipeg, MB R3T 2N2 Canada
maheswar@cs.umanitoba.ca

[◇]OS Research and Evaluation
OpenTV
Mountain View, CA 94043 USA
dhensgen@opentv.com

Abstract

A distributed heterogeneous computing (HC) system consists of diversely capable machines harnessed together to execute a set of tasks that vary in their computational requirements. Heuristics are needed to map (match and schedule) tasks onto machines in an HC system so as to optimize some figure of merit. This paper characterizes a simulated HC environment by using the expected execution times of the tasks that arrive in the system onto the different machines present in the system. This information is arranged in an "expected time to compute" (ETC) matrix as a model of the given HC system, where the entry(i, j) is the expected execution time of task i on machine j. This model is needed to simulate different HC environments to allow testing of relative performance of different mapping heuristics under different circumstances. In particular, the ETC model is used to express the heterogeneity among the runtimes of the tasks to be executed, and among the machines in the HC system. An existing range-based technique to generate ETC matrices is described. A coefficient-of-variation based technique to generate ETC matrices is proposed, and compared with the range-based technique. The coefficient-of-variation-based ETC generation method provides a greater control over the spread of values (i.e., heterogeneity) in any given row or column of the ETC matrix than the range-based method.

1. Introduction

A distributed heterogeneous computing (HC) system consists of diversely capable machines harnessed together to execute a set of tasks that vary in their computational requirements. Heuristics are needed to map (match and schedule) tasks onto machines in an HC system so as to optimize some figure of merit. The heuristics that match a task to a machine can vary in the information they use. For example, the current candidate task can be assigned to the machine that becomes available soonest (even if the task may take a much longer time to execute on that machine than elsewhere). In another approach, the task may be assigned to the machine where it executes fastest (but ignores when that machine becomes available). Or the current candidate task may be assigned to the machine that completes the task soonest, i.e., the machine which minimizes the sum of task execution time and the machine ready time, where machine ready time for a particular machine is the time when that machine becomes available after having executed the tasks previously assigned to it (e.g., [13]).

The discussion above should reveal that more sophisticated (and possibly wiser) approaches to the mapping problem require estimates of the execution times of all tasks (that can be expected to arrive for service) on all the machines present in the HC suite to make better mapping decisions. One aspect of the research on HC mapping heuristics explores the behavior of the heuristics in different HC environments. The ability to test the relative performance of different mapping heuristics under different circumstances necessitates that there be a framework for generating simulated execution times of all the tasks in the HC system on all the machines in the HC system. Such a framework would, in turn, require a quantification of heterogeneity to express the variability among the runtimes of the tasks to be executed, and among the capabilities of the machines in the HC system. The goal of this paper is to present a methodology for synthesizing simulated HC environments with quantifiable levels of task and machine heterogeneity. This paper characterizes the HC environments so that it will be easier for the researchers to describe the workload and the machines used in their simulations using a common scale.

Given a set of heuristics and a characterization of HC

This research was supported by the DARPA/ITO Quorum Program under the NPS subcontract numbers N62271-98-M-0217 and N62271-98-M-0448, and under the GSA subcontract number GS09K99BH0250. Some of the equipment used was donated by Intel.

0-7695-0556-2/00 $10.00 © 2000 IEEE

environments, one can determine the best heuristic to use in a given environment for optimizing a given objective function. In addition to increasing one's understanding of the operation of different heuristics, this knowledge can help a working resource management system select which mapper to use for a given real HC environment.

This research is part of a DARPA/ITO Quorum Program project called MSHN (pronounced "mission") (Management System for Heterogeneous Networks) [7]. MSHN is a collaborative research effort that includes the Naval Postgraduate School, NOEMIX, Purdue, and University of Southern California. It builds on SmartNet, an implemented scheduling framework and system for managing resources in an HC environment developed at NRaD [5]. The technical objective of the MSHN project is to design, prototype, and refine a distributed resource management system that leverages the heterogeneity of resources and tasks to deliver the requested qualities of service. The methodology developed here for generating simulated HC environments may be used to design, analyze and evaluate heuristics for the Scheduling Advisor component of the MSHN prototype.

The rest of this paper is organized as follows. A model for describing an HC system is presented in Section 2. Based on that model, two techniques for simulating an HC environment are described in Section 3. Section 4 briefly discusses analyzing the task execution time information from real life HC scenarios. Some related work is outlined in the Section 5.

2. Modeling Heterogeneity

To better evaluate the behavior of mapping heuristics, a model of the execution times of the tasks on the machines is needed so that the parameters of this model can be changed to investigate the performance of the heuristics under different HC systems and under different types of tasks to be mapped. One such model consists of an expected time to compute (ETC) matrix, where the entry(i, j) is the expected execution time of task i on machine j. The ETC matrix can be stored on the same machine where the mapper is stored, and contains the estimates for the expected execution times of a task on all machines, for all the tasks that are expected to arrive for service over a given interval of time. (Although stored with the mapper, the ETC information may be derived from other components of a resource management system (e.g., [7])). In an ETC matrix, the elements along a row indicate the estimates of the expected execution times of a given task on different machines, and those along a column give the estimates of the expected execution times of different tasks on a given machine.

The exact actual task execution times on all machines may not be known for all tasks because, for example, they might be a function of input data. What is typically assumed in the HC literature is that estimates of the expected execution times of tasks on all machines are known (e.g., [6, 10, 12, 16]). These estimates could be built from task profiling and machine benchmarking, could be derived from the previous executions of a task on a machine, or could be provided by the user (e.g., [3, 6, 8, 14, 18]).

The ETC model presented here can be characterized by three parameters: machine heterogeneity, task heterogeneity, and consistency. The variation along a row is referred to as the machine heterogeneity; this is the degree to which the machine execution times vary for a given task [1]. A system's machine heterogeneity is based on a combination of the machine heterogeneities for all tasks (rows). A system comprised mainly of workstations of similar capabilities can be said to have "low" machine heterogeneity. A system consisting of diversely capable machines, e.g., a collection of SMP's, workstations, and supercomputers, may be said to have "high" machine heterogeneity.

Similarly, the variation along a column of an ETC matrix is referred to as the task heterogeneity; this is the degree to which the task execution times vary for a given machine [1]. A system's task heterogeneity is based on a combination of the task heterogeneities for all machines (columns). "High" task heterogeneity may occur when the computational needs of the tasks vary greatly, e.g., when both time-consuming simulations and fast compilations of small programs are performed. "Low" task heterogeneity may typically be seen in the jobs submitted by users solving problems of similar complexity (and hence have similar execution times on a given machine).

Based on the above idea, four categories were proposed for the ETC matrix in [1]: (a) high task heterogeneity and high machine heterogeneity, (b) high task heterogeneity and low machine heterogeneity, (c) low task heterogeneity and high machine heterogeneity, and (d) low task heterogeneity and low machine heterogeneity.

The ETC matrix can be further classified into two categories, consistent and inconsistent [1], which are orthogonal to the previous classifications. For a consistent ETC matrix, if a machine m_x has a lower execution time than a machine m_y for a task t_k, then the same is true for any task t_i. A consistent ETC matrix can be considered to represent an extreme case of low task heterogeneity and high machine heterogeneity. If machine heterogeneity is high enough, then the machines may be so much different from each other in their compute power that the differences in the computational requirements of the tasks (if low enough) will not matter in determining the relative order of execution times for a given task on the different machines (i.e., along a row). As a trivially extreme example, consider a system consisting of Intel Pentium III and Intel 286. The Pentium III will almost always run any given task from a certain set of tasks faster than the 286 provided the computational requirements of all tasks in the set are similar (i.e.,

low task heterogeneity), thereby giving rise to a consistent ETC matrix.

In inconsistent ETC matrices, the relationships among the task computational requirements and machine capabilities are such that no structure as that in the consistent case is enforced. Inconsistent ETC matrices occur in practice when: (1) there is a variety of different machine architectures in the HC suite (e.g., parallel machines, superscalars, workstations), and (2) there is a variety of different computational needs among the tasks (e.g., readily parallelizable tasks, difficult to parallelize tasks, tasks that are floating point intensive, simple text formatting tasks). Thus, the way in which a task's needs correspond to a machine's capabilities may differ for each possible pairing of tasks to machines.

A combination of these two cases, which may be more realistic in many environments, is the partially-consistent ETC matrix, which is an inconsistent matrix with a consistent sub-matrix [2, 13]. This sub-matrix can be composed of any subset of rows and any subset of columns. As an example, in a given partially-consistent ETC matrix, 50% of the tasks and 25% of the machines may define a consistent sub-matrix.

Even though no structure is enforced on an inconsistent ETC matrix, a given ETC matrix generated to be inconsistent may have the structure of a partially consistent ETC matrix. In this sense, partially-consistent ETC matrices are a special case of inconsistent ETC matrices. Similarly, consistent ETC matrices are special cases of inconsistent and partially-consistent ETC matrices.

It should be noted that this classification scheme is used for generating ETC matrices. Later in this paper, it will be shown how these three cases differ in generation process. If one is given an ETC matrix, and is asked to classify it among these three classes, it will be called a consistent ETC matrix only if it is fully consistent. It will be called inconsistent if it is not consistent.

Often an inconsistent ETC matrix will have some partial consistency in it. For example, a trivial case of partial-consistency always exists; for any two machines in the HC suite, *at least* 50% of the tasks will show consistent execution times.

3. Generating the ETC Matrices

3.1. Range Based ETC Matrix Generation

Any method for generating the ETC matrices will require that heterogeneity be defined mathematically. In the range-based ETC generation technique, the heterogeneity of a set of execution time values is quantified by the range of the execution times [2, 13]. The procedures given in this section for generating the ETC matrices produce inconsistent ETC matrices. It is shown later in this section how consistent and partially-consistent ETC matrices could be obtained from the inconsistent ETC matrices.

```
(1) for i from 0 to (t − 1)
(2)     τ[i] = U(1, R_task)
(3)     for j from 0 to (m − 1)
(4)         e[i, j] = τ[i] × U(1, R_mach)
(5)     endfor
(6) endfor
```

Figure 1. The range-based method for generating ETC matrices.

Assume m is the total number of machines in the HC suite, and t is the total number of tasks expected to be serviced by the HC system over a given interval of time. Let $U(a, b)$ be a number sampled from a uniform distribution with a range from a to b. (Each invocation of $U(a, b)$ returns a new sample.) Let R_{task} and R_{mach} be numbers representing task heterogeneity and machine heterogeneity, respectively, such that higher values for R_{task} and R_{mach} represent higher heterogeneities. Then an ETC matrix $e[0..(t-1), 0..(m-1)]$, for a given task heterogeneity and a given machine heterogeneity, can be generated by the range-based method given in Figure 1, where $e[i, j]$ is the estimated expected execution time for the task i on the machine j.

As shown in Figure 1, each iteration of the outer **for** loop samples a uniform distribution with a range from 1 to R_{task} to generate one value for a vector τ. For each element of τ thus generated, the m iterations of the inner **for** loop (Line 3) generate one row of the ETC matrix. For the i-th iteration of the outer **for** loop, each iteration of the inner **for** loop produces one element of the ETC matrix by multiplying $\tau[i]$ with a random number sampled from a uniform distribution ranging from 1 to R_{mach}.

In the range-based ETC generation, it is possible to obtain high task heterogeneity low machine heterogeneity ETC matrices with characteristics similar to that of low task heterogeneity high machine heterogeneity ETC matrices if $R_{task} = R_{mach}$. In realistic HC systems, the variation that tasks show in their computational needs is generally larger than the variation that machines show in their capabilities. Therefore it is assumed here that requirements of high heterogeneity tasks are likely to be more "heterogeneous" than the capabilities of high heterogeneity machines (i.e., $R_{task} \gg R_{mach}$). However, for the ETC matrices generated here, low heterogeneity in both machines and tasks is assumed to be same. Table 1 shows typical values for R_{task} and R_{mach} for low and high heterogeneities. Tables 2 through 5 show four ETC matrices generated by the range-based method. The execution time values in Table 2 are

Table 1. Suggested values for R_{task} and R_{mach} for a realistic HC system for high heterogeneity and low heterogeneity.

	high	low
task	10^5	10^1
machine	10^2	10^1

much higher than the execution time values in Table 5. The difference in the values between these two tables would be reduced if the range for the low task heterogeneity was changed to 10^3 to 10^4 instead of 1 to 10.

With the range-based method, low task heterogeneity high machine heterogeneity ETC matrices tend to have high heterogeneity for both tasks and machines, due to method used for generation. For example, in Table 5, original τ vector values were selected from 1 to 10. When each entry is multiplied by a number from 1 to 100 for high machine heterogeneity this generates a task heterogeneity comparable to machine heterogeneity. It is shown later in Section 3.2 how to produce low task heterogeneity high machine heterogeneity ETC matrices which do show low task heterogeneity.

3.2. Coefficient-of-Variation Based ETC Matrix Generation

A modification of the procedure in Figure 1 defines the coefficient of variation, V, of execution time values as a measure of heterogeneity (instead of the range of execution time values). The coefficient of variation of a set of values is a better measure of the dispersion in the values than the standard deviation because it expresses the standard deviation as a percentage of the mean of the values [11]. Let σ and μ be the standard deviation and mean, respectively, of a set of execution time values. Then $V = \sigma/\mu$. The coefficient-of-variation-based ETC generation method provides a greater control over spread of the execution time values (i.e., heterogeneity) in any given row or column of the ETC matrix than the range-based method.

The coefficient-of-variation-based (CVB) ETC generation method works as follows. A task vector, q, of expected execution times with the desired task heterogeneity must be generated. Essentially, $q[i]$ is the execution time of task i on an "average" machine in the HC suite. For example, if the HC suite consists of an IBM SP/2, an Alpha server, and a Sun SPARC 5 workstation, then q would represent estimated execution times of the tasks on the Alpha server.

To generate q, two input parameters are needed: μ_{task} and V_{task}. The input parameter, μ_{task} is used to set the average of the values in q. The input parameter V_{task} is the desired coefficient of variation of the values in q. The value of V_{task} quantifies task heterogeneity, and is larger for higher task heterogeneity. Each element of the task vector q is then used to produce one row of the ETC matrix such that the desired coefficient of variation of values in each row is V_{mach}, another input parameter. The value of V_{mach} quantifies machine heterogeneity, and is larger for higher machine heterogeneity. Thus μ_{task}, V_{task}, and V_{mach} are the three input parameters for the CVB ETC generation method.

A direct approach to simulating HC environments should use the probability distribution that is empirically found to represent closely the distribution of task execution times. However, no standard benchmarks for HC systems are currently available. Therefore, this research uses a distribution which, though not necessarily reflective of an actual HC scenario, is flexible enough to be adapted to one. Such a distribution should not produce negative values of task execution times (e.g., ruling out Gaussian distribution), and should have a variable coefficient of variation (e.g., ruling out exponential distribution).

The gamma distribution is a good choice for the CVB ETC generation method because, with proper constraints on its characteristic parameters, it can approximate two other probability distributions, namely the Erlang-k and Gaussian (without the negative values) [11, 15]. The fact that it can approximate these two other distributions is helpful because this increases the chances that the simulated ETC matrices could be synthesized closer to some real life HC environment.

The uniform distribution can also be used but is not as flexible as the gamma distribution for two reasons: (1) it does not approximate any other distribution, and (2) the characteristic parameters of a uniform distribution cannot take all real values (explained later in the Section 3.3).

The gamma distribution [11, 15] is defined in terms of characteristic shape parameter, α, and scale parameter, β. The characteristic parameters of the gamma distribution can be fixed to generate different distributions. For example, if α is fixed to be an integer, then the gamma distribution becomes an Erlang-k distribution. If α is large enough, then the gamma distribution approaches a Gaussian distribution (but still does not return negative values for task execution times).

Figures 2(a) and 2(b) show how a gamma density function changes with the shape parameter α. When the shape parameter increases from two to eight, the shape of the distribution changes from a curve biased to the left to a more balanced bell-like curve. Figures 2(a), 2(c) and 2(d) show

Table 2. A high task heterogeneity low machine heterogeneity matrix generated by the range-based method using R_{task} and R_{mach} values of Table 1.

	m_1	m_2	m_3	m_4	m_5	m_6	m_7
t_1	333304	375636	198220	190694	395173	258818	376568
t_2	442658	400648	346423	181600	289558	323546	380792
t_3	75696	103564	438703	129944	67881	194194	425543
t_4	194421	392810	582168	248073	178060	267439	611144
t_5	466164	424736	503137	325183	193326	241520	506642
t_6	665071	687676	578668	919104	795367	390558	758117
t_7	177445	227254	72944	139111	236971	325137	347456
t_8	32584	55086	127709	51743	100393	196190	270979
t_9	311589	568804	148140	583456	209847	108797	270100
t_{10}	314271	113525	448233	201645	274328	248473	170176
t_{11}	272632	268320	264038	140247	110338	29620	69011
t_{12}	489327	393071	225777	71622	243056	445419	213477

Table 3. A high task heterogeneity high machine heterogeneity matrix generated by the range-based method using R_{task} and R_{mach} values of Table 1.

	m_1	m_2	m_3	m_4	m_5	m_6	m_7
t_1	2425808	3478227	719442	2378978	408142	2966676	2890219
t_2	2322703	2175934	228056	3456054	6717002	5122744	3660354
t_3	1254234	3182830	4408801	5347545	4582239	6124228	5343661
t_4	227811	419597	13972	297165	438317	23374	135871
t_5	6477669	5619369	707470	8380933	4693277	8496507	7279100
t_6	1113545	1642662	303302	244439	1280736	541067	792149
t_7	2860617	161413	2814518	2102684	8218122	7493882	2945193
t_8	1744479	623574	1516988	5518507	2023691	3527522	1181276
t_9	6274527	1022174	3303746	7318486	7274181	6957782	2145689
t_{10}	1025604	694016	169297	193669	1009294	1117123	690846
t_{11}	2390362	1552226	2955480	4198336	1641012	3072991	3262071
t_{12}	96699	882914	63054	199175	894968	248324	297691

Table 4. A low task heterogeneity low machine heterogeneity matrix generated by the range-based method using R_{task} and R_{mach} values of Table 1.

	m_1	m_2	m_3	m_4	m_5	m_6	m_7
t_1	22	21	6	16	15	24	13
t_2	7	46	5	28	45	43	31
t_3	64	83	45	23	58	50	38
t_4	53	56	26	42	53	9	58
t_5	11	12	14	7	8	3	14
t_6	33	31	46	25	23	39	10
t_7	24	11	17	14	25	35	4
t_8	20	17	23	4	3	18	20
t_9	13	28	14	7	34	6	29
t_{10}	2	5	7	7	6	3	7
t_{11}	16	37	23	22	23	12	44
t_{12}	8	66	47	11	47	55	56

Table 5. A low task heterogeneity high machine heterogeneity matrix generated by the range-based method using R_{task} and R_{mach} values of Table 1.

	m_1	m_2	m_3	m_4	m_5	m_6	m_7
t_1	440	762	319	532	151	652	308
t_2	459	205	457	92	92	379	60
t_3	499	263	92	152	75	18	128
t_4	421	362	347	194	241	481	391
t_5	276	636	136	355	338	324	255
t_6	89	139	37	67	9	53	139
t_7	404	521	54	295	257	208	539
t_8	49	114	279	22	93	39	36
t_9	59	35	184	262	145	287	277
t_{10}	7	235	44	81	330	56	78
t_{11}	716	601	75	689	299	144	457
t_{12}	435	208	256	330	6	394	419

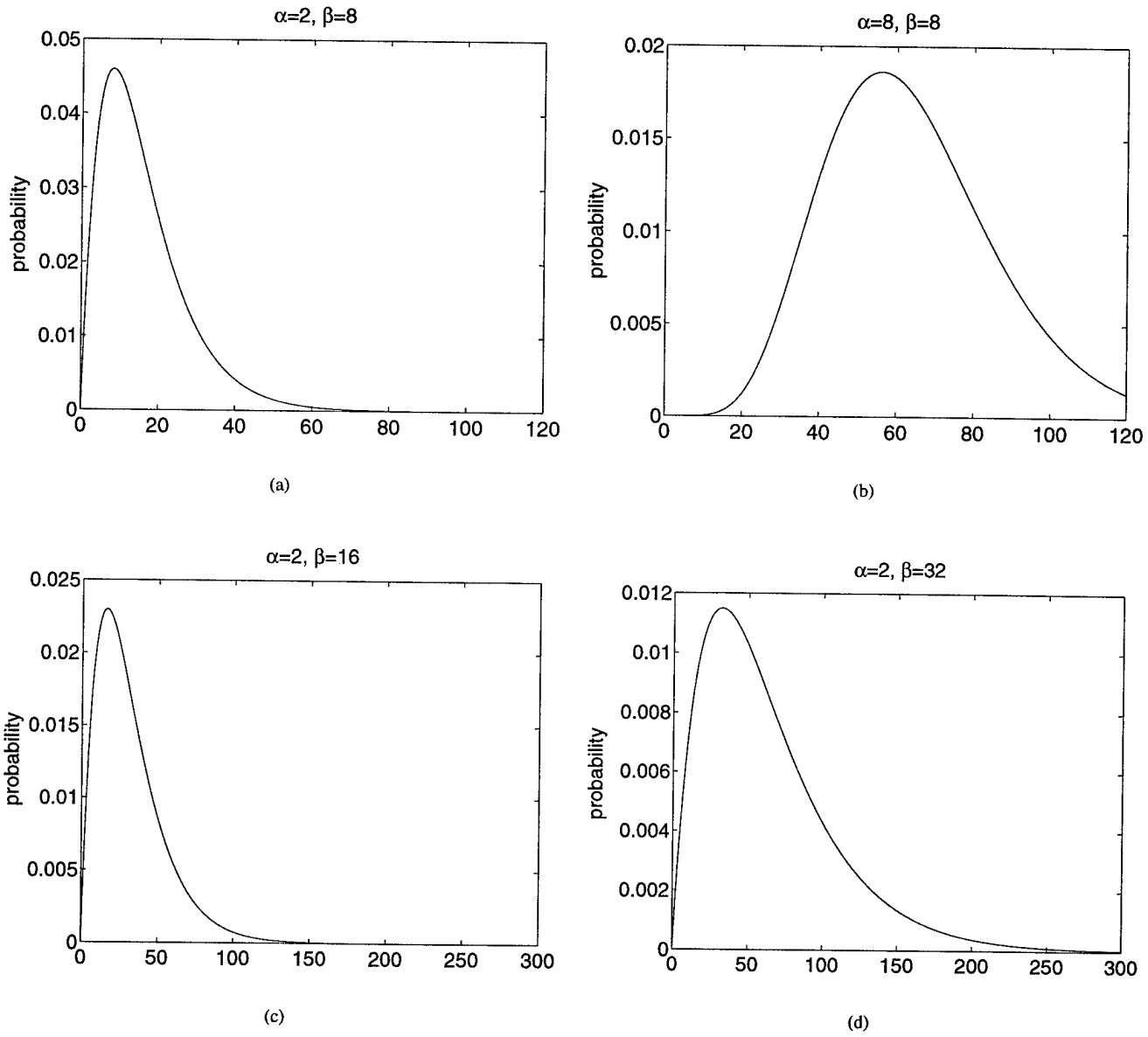


Figure 2. Gamma probability density function for (a) $\alpha = 2$, $\beta = 8$, (b) $\alpha = 8$, $\beta = 8$, (c) $\alpha = 2$, $\beta = 16$, and (d) $\alpha = 2$, $\beta = 32$.

(1) $\alpha_{task} = 1/V_{task}^2$; $\alpha_{mach} = 1/V_{mach}^2$;
    $\beta_{task} = \mu_{task}/\alpha_{task}$
(2) for i from 0 to $(t-1)$
(3) $q[i] = G(\alpha_{task}, \beta_{task})$
     /* $q[i]$ will be used as mean
     of i-th row of ETC matrix */
(4) $\beta_{mach}[i] = q[i]/\alpha_{mach}$
     /* scale parameter for i-th row */
(5) for j from 0 to $(m-1)$
(6) $e[i,j] = G(\alpha_{mach}, \beta_{mach}[i])$
(7) endfor
(8) endfor

Figure 3. The general CVB method for generating ETC matrices.

(1) $\alpha_{task} = 1/V_{task}^2$; $\alpha_{mach} = 1/V_{mach}^2$;
    $\beta_{mach} = \mu_{mach}/\alpha_{mach}$
(2) for j from 0 to $(m-1)$
(3) $p[j] = G(\alpha_{mach}, \beta_{mach})$
     /* $p[j]$ will be used as mean
     of j-th column of ETC matrix */
(4) $\beta_{task}[j] = p[j]/\alpha_{task}$
     /* scale parameter for j-th column */
(5) for i from 0 to $(t-1)$
(6) $e[i,j] = G(\alpha_{task}, \beta_{task}[j])$
(7) endfor
(8) endfor

Figure 4. The CVB method for generating low task heterogeneity high machine heterogeneity ETC matrices.

the effect on the distribution caused by an increase in the scale parameter from 8 to 16 to 32. The two-fold increase in the scale parameter does not change the shape of the graph (the curve is still biased to the left); however the curve now has twice as large a domain (i.e., range on x-axis).

The gamma distribution's characteristic parameters, α and β, can be easily interpreted in terms of μ_{task}, V_{task}, and V_{mach}. For a gamma distribution, $\sigma = \beta\sqrt{\alpha}$, and $\mu = \beta\alpha$, so that $V = \sigma/\mu = 1/\sqrt{\alpha}$ (and $\alpha = 1/V^2$). Then $\alpha_{task} = 1/V_{task}^2$ and $\alpha_{mach} = 1/V_{mach}^2$. Further, because $\mu = \beta\alpha$, $\beta = \mu/\alpha$, and $\beta_{task} = \mu_{task}/\alpha_{task}$. Also, for task i, $\beta_{mach}[i] = q[i]/\alpha_{mach}$.

Let $G(\alpha, \beta)$ be a number sampled from a gamma distribution with the given parameters. (Each invocation of $G(\alpha, \beta)$ returns a new sample.) Figure 3 shows the general procedure for the CVB ETC generation.

Given the three input parameters, V_{task}, V_{mach}, and μ_{task}, Line (1) of Figure 3 determines the shape parameter α_{task} and scale parameter β_{task} of the gamma distribution that will be later sampled to build the task vector q. Line (1) also calculates the shape parameter α_{mach} to use later in Line (6). In the i-th iteration of the outer **for** loop (Line 2) in Figure 3, a gamma distribution with parameters α_{task} and β_{task} is sampled to obtain $q[i]$. Then $q[i]$ is used to determine the scale parameter $\beta_{mach}[i]$ (to be used later in Line (6)). For the i-th iteration of the outer **for** loop (Line 2), each iteration of the inner **for** loop (Line 5) produces one element of the i-th row of the ETC matrix by sampling a gamma distribution with parameters α_{mach} and $\beta_{mach}[i]$. One complete row of the ETC matrix is produced by m iterations of the inner **for** loop (Line 5). Note that while each row in the ETC matrix has gamma distributed execution times, the execution times in columns are not gamma distributed.

The ETC generation method of Figure 3 can be used to generate high task heterogeneity high machine heterogeneity ETC matrices, high task heterogeneity low machine heterogeneity ETC matrices, and low task heterogeneity low machine heterogeneity ETC matrices, but cannot generate low task heterogeneity high machine heterogeneity ETC matrices. To satisfy the heterogeneity quadrants of Section 2, each column in the final low task heterogeneity high machine heterogeneity ETC matrix should reflect the low task heterogeneity of the "parent" task vector q. This condition would not necessarily hold if rows of the ETC matrix were produced with a high machine heterogeneity from a task vector of low heterogeneity. This is because a given column may be formed from widely different execution time values from different rows because of the high machine heterogeneity. That is, any two entries in a given column are based on different values of $q[i]$ and α_{mach}, and may therefore show high task heterogeneity as opposed to the intended low task heterogeneity. In contrast, in a high task heterogeneity low machine heterogeneity ETC matrix the low heterogeneity among the machines for a given task (across a row) is based on the same $q[i]$ value.

One solution is to generate what is in effect a transpose of a high task heterogeneity low machine heterogeneity matrix to produce a low task heterogeneity high machine heterogeneity one. The transposition can be built into the procedure as shown in Figure 4. The procedure in Figure 4 is very similar to the one in Figure 3. The input parameter μ_{task} is replaced with μ_{mach}. Here, first a machine vector, p, (with an average value of μ_{mach}) is produced. Each element of this "parent" machine vector is then used to generate one low task heterogeneity column of the ETC matrix, such that the high machine heterogeneity present in p is reflected in all rows. This approach for generating low task heterogeneity high machine heterogeneity ETC matrices can also be used with the range-based method.

Tables 6 through 11 show some sample ETC matrices generated using the CVB ETC generation method. Tables 6 and 7 both show high task heterogeneity low machine heterogeneity ETC matrices. In both tables, the spread of the execution time values in columns is higher than that in rows. The ETC matrix in Table 7 has a higher task heterogeneity (higher V_{task}) than the ETC matrix in Table 6. This can be seen in a higher spread in the columns of matrix in Table 7 than that in Table 6.

Tables 8 and 9 show high task heterogeneity high machine heterogeneity and low task heterogeneity low machine heterogeneity ETC matrices, respectively. The execution times in Table 8 are widely spaced along both rows and columns. The spread of execution times in Table 9 is smaller along both columns and rows, because both V_{task} and V_{mach} are smaller.

Tables 10 and 11 show low task heterogeneity high machine heterogeneity ETC matrices. In both tables, the spread of the execution time values in rows is higher than that in columns. ETC matrix in Table 11 has a higher machine heterogeneity (higher V_{mach}) than the ETC matrix in Table 10. This can be seen in a higher spread in the rows of matrix in Table 11 than that in Table 10.

3.3. Uniform Distribution in the CVB Method

The uniform distribution could also be used for the CVB ETC generation method. The uniform distribution's characteristic parameters a (lower bound for the range of values) and b (upper bound for the range of values), can be easily interpreted in terms of μ_{task}, V_{task}, and V_{mach}. (Recall that $V_{task} = \sigma_{task}/\mu_{task}$ and $V_{mach} = \sigma_{mach}/\mu_{mach}$). For a uniform distribution, $\sigma = (b-a)/\sqrt{12}$ and $\mu = (b+a)/2$ [15]. So that

$$a + b = 2\mu \tag{1}$$

$$a - b = -\sigma\sqrt{12} \tag{2}$$

Adding Equations (1) and (2),

$$a = \mu - \sigma\sqrt{3} \tag{3}$$

$$a = \mu(1 - (\sigma/\mu)\sqrt{3}) \tag{4}$$

$$a = \mu(1 - V\sqrt{3}) \tag{5}$$

Also,

$$b = 2\mu - a \tag{6}$$

The Equations (5) and (6) can be used to generate the task vector q from the uniform distribution with the following parameters:

$$a_{task} = \mu_{task}(1 - V_{task}\sqrt{3}) \tag{7}$$

$$b_{task} = 2\mu_{task} - a_{task} \tag{8}$$

Once the task vector q has been generated, the i-th row of the ETC matrix can be generated by sampling (m times) a uniform distribution with the following parameters:

$$a_{mach} = q[i](1 - V_{mach}\sqrt{3}) \tag{9}$$

$$b_{mach} = 2q[i] - a_{mach} \tag{10}$$

The CVB ETC generation using the uniform distribution, however, places a restriction on the values of V_{task} and V_{mach}. Because both a_{task} and a_{mach} have to be positive, it follows from Equations (7) and (9) that the maximum value for V_{mach} or V_{task} is $1/\sqrt{3}$. Thus, for the CVB ETC generation, the gamma distribution is better than the uniform distribution because it does not restrict the values of task or machine heterogeneities.

3.4. Producing Consistent ETC Matrices

The procedures given in Figures 1, 3, and 4 produce inconsistent ETC matrices. Consistent ETC matrices can be obtained from the inconsistent ETC matrices generated above by sorting the execution times for each task on all machines (i.e., sorting the values within each row and doing this for all rows independently). From the inconsistent ETC matrices generated above, partially-consistent matrices consisting of an $i \times k$ sub-matrix could be generated by sorting the execution times across a random subset of k machines for each task in a random subset of i tasks.

It should be noted from Tables 10 and 11 that the greater the difference in machine and task heterogeneities, the higher the degree of consistency in the inconsistent low task heterogeneity high machine heterogeneity ETC matrices. For example, in Table 11 all tasks show consistent execution times on all machines except on the machines that correspond to columns 3 and 4. As mentioned in Section 1, these degrees and classes of mixed-machine heterogeneity can be used to characterize many different HC environments.

4. Analysis and Synthesis

Once the actual ETC matrices from a real life scenario are obtained, they can be analyzed to estimate the probability distribution of the execution times, and the values of the model parameters (i.e., V_{task}, V_{mach}, and μ_{task} (or μ_{mach}, if a low task heterogeneity high machine heterogeneity ETC matrix is desired)) appropriate for the given real life scenario. The above analysis could be carried out using common statistical procedures [9]. Once a model of a particular HC system is available, the effect of changes in the workload (i.e., the tasks arriving for service in the system) and the system (i.e., the machines present in the HC system) can be studied in a controlled manner by simply changing the parameters of the ETC model.

Table 6. A high task heterogeneity low machine heterogeneity matrix generated by the CVB method. $V_{task} = 0.3$, $V_{mach} = 0.1$.

	m_1	m_2	m_3	m_4	m_5	m_6	m_7	m_8	m_9	m_{10}
t_1	628	633	748	558	743	684	740	692	593	554
t_2	688	712	874	743	854	851	701	701	811	864
t_3	965	1029	1087	1020	921	825	1238	934	928	1042
t_4	891	866	912	896	776	993	875	999	919	860
t_5	1844	1507	1353	1436	1677	1691	1508	1646	1789	1251
t_6	1261	1157	1193	1297	1261	1251	1156	1317	1189	1306
t_7	850	928	780	1017	761	900	998	838	797	824
t_8	1042	1291	1169	1562	1277	1431	1236	1092	1274	1305
t_9	1309	1305	1641	1225	1425	1280	1388	1268	1290	1549
t_{10}	881	865	752	893	883	813	892	805	873	915

Table 7. A high task heterogeneity low machine heterogeneity matrix generated by the CVB method. $V_{task} = 0.5$, $V_{mach} = 0.1$.

	m_1	m_2	m_3	m_4	m_5	m_6	m_7	m_8	m_9	m_{10}
t_1	377	476	434	486	457	486	431	417	429	428
t_2	493	370	400	420	502	472	475	440	483	576
t_3	745	646	922	650	791	878	853	791	756	788
t_4	542	490	469	559	488	498	509	431	547	542
t_5	625	666	618	710	624	615	618	599	522	540
t_6	921	785	759	979	865	843	853	870	939	801
t_7	677	767	750	720	797	728	941	717	686	870
t_8	428	418	394	460	434	427	378	427	447	466
t_9	263	289	267	231	243	222	283	257	240	247
t_{10}	1182	1518	1272	1237	1349	1218	1344	1117	1122	1260
t_{11}	1455	1384	1694	1644	1562	1639	1776	1813	1488	1709
t_{12}	3255	2753	3289	3526	2391	2588	3849	3075	3664	3312

Table 8. A high task heterogeneity high machine heterogeneity matrix generated by the CVB method. $V_{task} = 0.6$, $V_{mach} = 0.6$.

	m_1	m_2	m_3	m_4	m_5	m_6	m_7	m_8	m_9	m_{10}
t_1	1446	1110	666	883	1663	1458	653	1886	458	1265
t_2	1010	588	682	1255	3665	3455	1293	1747	1173	1638
t_3	1893	2798	1097	465	2413	1184	2119	1955	1316	2686
t_4	1014	1193	275	1010	1023	1282	559	1133	865	2258
t_5	170	444	500	408	790	528	232	303	301	480
t_6	1454	1106	901	793	1346	703	1215	490	537	1592
t_7	579	1041	852	1560	1983	1648	859	683	945	1713
t_8	2980	2114	417	3005	2900	3216	421	2854	1425	1631
t_9	252	519	196	352	958	355	720	168	668	1017
t_{10}	173	235	273	176	110	127	93	276	390	103
t_{11}	115	74	251	71	107	479	153	138	274	189
t_{12}	305	226	860	554	394	344	68	86	223	120

Table 9. A low task heterogeneity low machine heterogeneity matrix generated by the CVB method. $V_{task} = 0.1$, $V_{mach} = 0.1$.

	m_1	m_2	m_3	m_4	m_5	m_6	m_7	m_8	m_9	m_{10}
t_1	985	1043	945	835	830	1087	1009	891	1066	1075
t_2	963	962	910	918	1078	1091	881	980	1009	981
t_3	782	837	968	960	790	800	947	1007	1115	845
t_4	999	953	892	986	958	1006	1039	1072	1090	1030
t_5	971	972	913	1030	891	873	898	994	1086	1122
t_6	1155	1065	800	1247	980	1103	1228	1062	1011	1005
t_7	1007	1191	964	860	1034	896	1185	932	1035	1019
t_8	1088	864	972	984	736	950	944	994	970	894
t_9	878	967	954	917	942	978	1046	1134	985	1032
t_{10}	1210	1120	1043	1093	1386	1097	1202	1004	1185	1226
t_{11}	910	958	1046	1062	952	1054	1020	1175	850	1060
t_{12}	930	935	908	1155	991	997	828	1062	886	831

Table 10. A low task heterogeneity high machine heterogeneity matrix generated by the CVB method. $V_{task} = 0.1$, $V_{mach} = 0.6$.

	m_1	m_2	m_3	m_4	m_5	m_6	m_7	m_8	m_9	m_{10}
t_1	1679	876	1332	716	1186	1860	662	833	534	804
t_2	1767	766	1327	711	957	2061	625	626	642	800
t_3	1870	861	1411	932	1065	1562	625	976	556	842
t_4	1861	817	1218	865	1096	1660	587	767	736	822
t_5	1768	850	1465	764	1066	1585	663	863	579	757
t_6	1951	807	1177	914	939	1483	573	961	643	712
t_7	1312	697	1304	921	1005	1639	562	831	633	784
t_8	1665	849	1414	795	1162	1593	577	791	709	774
t_9	1618	753	1283	794	1153	1673	639	787	563	744
t_{10}	1576	964	1373	752	950	1726	699	836	633	764
t_{11}	1693	742	1454	758	961	1781	721	988	641	793
t_{12}	1863	823	1317	890	1137	1812	704	800	479	848

Table 11. A low task heterogeneity high machine heterogeneity matrix generated by the CVB method. $V_{task} = 0.1$, $V_{mach} = 2.0$.

	m_1	m_2	m_3	m_4	m_5	m_6	m_7	m_8	m_9	m_{10}
t_1	4784	326	1620	1307	3301	10	103	4449	228	40
t_2	4315	276	1291	1863	3712	11	91	5255	200	47
t_3	6278	269	1493	1181	3186	12	93	4604	235	46
t_4	4945	294	1629	1429	2894	14	87	4724	231	45
t_5	5276	321	1532	1516	2679	12	102	4621	205	46
t_6	4946	293	1467	1609	2661	10	96	3991	255	39
t_7	4802	327	1317	1668	2982	10	90	5090	252	42
t_8	5381	365	1698	1384	3668	12	99	5133	242	38
t_9	5011	255	1491	1386	3061	10	94	3739	216	42
t_{10}	5228	296	1489	1515	3632	12	107	4682	203	38
t_{11}	5367	319	1332	1363	3393	12	72	4769	221	43
t_{12}	4621	258	1473	1501	3124	12	96	4091	199	44

This experimental set-up can then be used to find out which mapping heuristics are best suited for a given set of model parameters (i.e., V_{task}, V_{mach}, and μ_{task} (or μ_{mach})). This information can be stored in a "look-up table," so as to facilitate the choice of a mapping heuristic given a set of model parameters. The look-up table can be part of the toolbox in the mapper.

The ETC model of Section 2 assumes that the machine heterogeneity is the same for all tasks, i.e., different tasks show the same general variation in their execution times over different machines. In reality this may not be true; the variation in the execution times of one task on all machines may be very different from some other task. To model the "variation in machine heterogeneity" along different rows (i.e., for different tasks), another level of heterogeneity could be introduced. For example, in the CVB ETC generation, instead of having a fixed value for V_{mach} for all the tasks, the value of V_{mach} for a given task could be variable, e.g., it could be sampled from a probability distribution. Once again, the nature of the probability distribution and its parameters will need to be decided empirically.

5. Related Work

To the best of the authors' knowledge, there is currently no work presented in the open literature that addresses the problem of modeling of execution times of the tasks in an HC system (except the already discussed work [13]). However, below are presented two tangentially related works.

A detailed workload model for parallel machines has been given in [4]. However the model is not intended for HC systems in that the machine heterogeneity is not modeled. Task execution times are modeled but tasks are assumed to be running on multiple processing nodes, unlike the HC environment presented here where tasks run on single machines only.

A method for generating random task graphs is given in [17] as part of description of the simulation environment for the HC systems. The method proposed in [17] assumes that the computation cost of a task t_i, averaged over all the machines in the system, is available as $\overline{w_i}$. The method does provide for characterizing the differences in the execution times of a given task on different processors in the HC system (i.e., machine heterogeneity). The "range percentage" (β) of computation costs on processors roughly corresponds to the notion of machine heterogeneity as presented here. The execution time, e_{ij}, of task t_i on machine m_j is randomly selected from the range, $\overline{w_i} \times (1 - \beta/2) \leq e_{ij} \leq \overline{w_i} \times (1 + \beta/2)$. However, the method in [17] does not provide for describing the differences in the execution times of all the tasks on an "average" machine in the HC system. The method in [17] does not tell how the differences in the values of $\overline{w_i}$ for different machines will be modeled. That is, the method is [17] does not consider task heterogeneity. Further, the model in [17] does not take into account the consistency of the task execution times.

6. Conclusions

To describe different kinds of heterogeneous environments, an existing model based on the characteristics of the ETC matrix was presented. The three parameters of this model (task heterogeneity, machine heterogeneity, and consistency) can be changed to investigate the performance of mapping heuristics for different HC systems and different sets of tasks. An existing range-based method for quantifying heterogeneity was described, and a new coefficient-of-variation-based method was proposed. Corresponding procedures for generating the ETC matrices representing various heterogeneous environments were presented. Sample ETC matrices were provided for both ETC generation procedures. The coefficient-of-variation-based ETC generation method provides a greater control over the spread of values (i.e., heterogeneity) in any given row or column of the ETC matrix than the range-based method. This characterization of HC environments will allow a researcher to simulate different HC environments, and then evaluate the behavior of the mapping heuristics under different conditions of heterogeneity.

Acknowledgments: The authors thank Anthony A. Maciejewski, Tracy D. Braun, and Taylor Kidd for their valuable comments and suggestions.

References

[1] R. Armstrong, *Investigation of Effect of Different Run-Time Distributions on SmartNet Performance* Master's thesis, Department of Computer Science, Naval Postgraduate School, 1997 (D. Hensgen, Advisor).

[2] T. D. Braun, H. J. Siegel, N. Beck, L. Bölöni, M. Maheswaran, A. I. Reuther, J. P. Robertson, M. D. Theys, B. Yao, R. F. Freund, and D. Hensgen, "A comparison study of static mapping heuristics for a class of meta-tasks on heterogeneous computing systems," *8th IEEE Heterogeneous Computing Workshop (HCW '99)*, Apr. 1999, pp. 15–29.

[3] H. G. Dietz, W. E. Cohen, and B. K. Grant, "Would you run it here... or there? (AHS: Automatic Heterogeneous Supercomputing)," *1993 International Conference on Parallel Processing (ICPP '93)*, Vol. II, Aug. 1993, pp. 217–221.

[4] D. G. Feitelson and L. Rudolph, "Metrics and benchmarking for parallel job scheduling," in *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph, eds., Vol. 1459, Lecture Notes in Computer Science, Vol. 1459, Springer-Verlag, New York, NY, 1998, pp. 1–15.

[5] R. F. Freund, M. Gherrity, S. Ambrosius, M. Campbell, M. Halderman, D. Hensgen, E. Keith, T. Kidd, M. Kussow, J. D. Lima, F. Mirabile, L. Moore, B. Rust, and H. J. Siegel, "Scheduling resources in multi-user, heterogeneous, computing environments with SmartNet," *7th IEEE Heterogeneous Computing Workshop (HCW '98)*, Mar. 1998, pp. 184–199.

[6] A. Ghafoor and J. Yang, "Distributed heterogeneous supercomputing management system," *IEEE Computer*, Vol. 26, No. 6, June 1993, pp. 78–86.

[7] D. A. Hensgen, T. Kidd, D. St. John, M. C. Schnaidt, H. J. Siegel, T. D. Braun, M. Maheswaran, S. Ali, J.-K. Kim, C. Irvine, T. Levin, R. F. Freund, M. Kussow, M. Godfrey, A. Duman, P. Carff, S. Kidd, V. Prasanna, P. Bhat, and A. Alhusaini, "An overview of MSHN: The Management System for Heterogeneous Networks," *8th IEEE Heterogeneous Computing Workshop (HCW '99)*, Apr. 1999, pp. 184–198.

[8] M. A. Iverson, F. Özgüner, and G. J. Follen, "Statistical prediction of task execution times through analytic benchmarking for scheduling in a heterogeneous environment," *8th IEEE Heterogeneous Computing Workshop (HCW '99)*, Apr. 1999, pp. 99–111.

[9] R. Jain, *The Art of Computer Systems Performance Analysis*, John Wiley & Sons, Inc., New York, NY, 1991.

[10] M. Kafil and I. Ahmad, "Optimal task assignment in heterogeneous distributed computing systems," *IEEE Concurrency*, Vol. 6, No. 3, July-Sep. 1998, pp. 42–51.

[11] L. L. Lapin, *Probability and Statistics for Modern Engineering*, Second Edition, Waveland Press, Inc., Prospect Heights, IL, 1998.

[12] N. Lopez-Benitez and J.-Y. Hyon, "Simulation of task graph systems in heterogeneous computing environments," *8th IEEE Heterogeneous Computing Workshop (HCW '99)*, Apr. 1999, pp. 112–124.

[13] M. Maheswaran, S. Ali, H. J. Siegel, D. Hensgen, and R. F. Freund, "Dynamic mapping of a class of independent tasks onto heterogeneous computing systems,"*Journal of Parallel and Distributed Computing*, Special Issue on Software Support for Distributed Computing, Vol. 59, No. 2, pp. 107–131, Nov. 1999.

[14] M. Maheswaran, T. D. Braun, and H. J. Siegel, "Heterogeneous distributed computing," in *Encyclopedia of Electrical and Electronics Engineering, Vol. 8*, J. G. Webster, ed., John Wiley, New York, NY, 1999, pp. 679–690.

[15] A. Papoulis, *Probability, Random Variables, and Stochastic Processes*, McGraw-Hill, New York, NY, 1984.

[16] H. Singh and A. Youssef, "Mapping and scheduling heterogeneous task graphs using genetic algorithms," *5th IEEE Heterogeneous Computing Workshop (HCW '96)*, Apr. 1996, pp. 86–97.

[17] H. Topcuoglu, S. Hariri, and M.-Y. Wu, "Task scheduling algorithms for heterogeneous processors," *8th IEEE Heterogeneous Computing Workshop (HCW '99)*, Apr. 1999, pp. 3–14.

[18] J. Yang, I. Ahmad, and A. Ghafoor, "Estimation of execution times on heterogeneous supercomputer architecture," *1993 International Conference on Parallel Processing (ICPP '93)*, Vol. I, Aug. 1993, pp. 219–225.

Biographies

Shoukat Ali is pursuing a PhD degree from the School of Electrical and Computer Engineering at Purdue University, where he is currently a Research Assistant. His main research topic is dynamic mapping of meta-tasks in heterogeneous computing systems. He has held teaching positions at Aitchison College and Keynesian Institute of Management and Sciences, both in Lahore, Pakistan. He was also a Teaching Assistant at Purdue. Shoukat received his MS degree in electrical and electronic engineering from Purdue University in 1999, and his BS degree in electrical and electronic engineering from the University of Engineering and Technology, Lahore, Pakistan in 1996. His research interests include computer architecture, parallel computing, and heterogeneous computing.

Howard Jay Siegel is a Professor in the School of Electrical and Computer Engineering at Purdue University. He is a Fellow of the IEEE and a Fellow of the ACM. He received BS degrees in both electrical engineering and management from MIT, and the MA, MSE, and PhD degrees from the Department of Electrical Engineering and Computer Science at Princeton University. Prof. Siegel has coauthored over 250 technical papers, has coedited seven volumes, and wrote the book *Interconnection Networks for Large-Scale Parallel Processing*. He was a Coeditor-in-Chief of the *Journal of Parallel and Distributed Computing*, and was on the Editorial Boards of the *IEEE Transactions on Parallel and Distributed Systems* and the *IEEE Transactions on Computers*. He was Program Chair/Co-Chair of three conferences, General Chair/Co-Chair of four conferences, and Chair/Co-Chair of four workshops. He is an international keynote speaker and tutorial lecturer, and a consultant for government and industry.

Muthucumaru Maheswaran is an Assistant Professor in the Department of Computer Science at the University

of Manitoba, Canada. In 1990, he received a BSc degree in electrical and electronic engineering from the University of Peradeniya, Sri Lanka. He received an MSEE degree in 1994 and a PhD degree in 1998, both from the School of Electrical and Computer Engineering at Purdue University. He held a Fulbright scholarship during his tenure as an MSEE student at Purdue University. His research interests include computer architecture, distributed computing, heterogeneous computing, Internet and world wide web systems, metacomputing, mobile programs, network computing, parallel computing, resource management systems for metacomputing, and scientific computing. He has authored or coauthored 15 technical papers in these and related areas. He is a member of the Eta Kappa Nu honorary society.

Debra Hensgen is a member of the Research and Evaluation Team at OpenTV in Mountain View, California. OpenTV produces middleware for set-top boxes in support of interactive television. She received her PhD in the area of Distributed Operating Systems from the University of Kentucky. Prior to moving to private industry, as an Associate Professor in the systems area, she worked with students and colleagues to design and develop tools and systems for resource management, network re-routing algorithms and systems that preserve quality of service guarantees, and visualization tools for performance debugging of parallel and distributed systems. She has published numerous papers concerning her contributions to the Concurra toolkit for automatically generating safe, efficient concurrent code, the Graze parallel processing performance debugger, the SAAM path information base, and the SmartNet and MSHN Resource Management Systems.

Sahra Ali is pursuing a PhD degree at the School of Electrical and Computer Engineering at Purdue University, where she is currently a Research Assistant. Her main research topic is the modeling of reliability in software intensive communication networks. She has also been working as a software developer for Cisco Systems since 1997. She was previously a Teaching Assistant and lab coordinator at Purdue. Sahra received her MS degree in electrical engineering from Purdue University in 1998, and her BS degree in electrical and electronic engineering from Sharif University of Technology, Tehran, Iran in 1995. Her research interests include testing, reliability and availability of communication networks, as well as multimedia.

Distributed Quasi-Monte Carlo Methods in a Heterogeneous Environment

E. deDoncker, R. Zanny, M. Ciobanu and Y. Guan
Department of Computer Science
Western Michigan University
Kalamazoo, Michigan 49008, USA
{elise,rrzanny}@cs.wmich.edu

Abstract

We present an asynchronous Quasi-Monte Carlo (QMC) algorithm for numerical integration tailored for heterogeneous environments. QMC techniques are better suited for high dimensions than adaptive methods and have generally better convergence properties than classical Monte Carlo methods.

The algorithm focuses on the asynchronous computation of randomized lattice (Korobov) rules. Whereas the individual rules disallow realistic error estimates, randomization provides a tool for giving confidence intervals for the magnitude of the error. The algorithm generates a sequence of stochastic families, using an increasing number of points, for the purpose of automatic termination.

In the algorithm, each each randomized rule constitutes a single unit of work; a work assignment consists of a set of work units. Static and dynamic load balancing strategies are explored to keep the processors busy performing useful work while gradually calculating higher-level families needed to reach the desired accuracy. We present results in the context of a performance model for parallel programs executing in a heterogeneous environment.

1. Introduction

Sampling techniques for numerically solving integration problems are well established, and are particularly useful when solving problems of high dimensions. While synchronous parallel implementations of these techniques appear to be straightforward on tightly coupled parallel architectures, various factors push for an asynchronous solution in heterogeneous, coarse-grained, network of workstations (NOW) architectures. We focus on the design and implementation of asynchronous sampling techniques on these architectures, and the unique difficulties in doing so. This work is based on the sequential implementation by Genz [12].

The next section presents background information; Section 3 introduces Quasi-Monte Carlo techniques, and Section 4 discusses the algorithm. A performance model is outlined in Section 5 and test results are given in Section 6.

2. Background

The goal is to calculate an approximate answer Q to the multivariate integral $I = \int_{\mathcal{D}} f(\mathbf{x})d\mathbf{x}$, and an error bound E_a such that $|I - Q| \leq E_a \leq \varepsilon = \max\{\varepsilon_a, \varepsilon_r|I|\}$, where ε_a and ε_r specify absolute and relative error tolerances, respectively. The integration domain $\mathcal{D}$ is a d-dimensional hyper-rectangular region, though without loss of generality we assume that it is the d-dimensional unit hypercube $\mathcal{H}_d$.

When d is small (say, $d \leq 10$), adaptive partitioning methods, (APM) generally work well for finding highly accurate solutions. They continually divide $\mathcal{D}$ into smaller subregions, evaluating each with a quadrature rule, until an answer Q of the desired accuracy ε is reached. We have done extensive work on the parallelization of adaptive methods (see, e.g., [6]), leading to the parallel software package PARINT1.0 [7].

However, when d is large, the dimensional effect in the required number of evaluation points grows to an unacceptable level [4]. Indeed, if a particular 1-dimensional problem requires s subdivisions and one considers a d-dimensional problem of the same degree of difficulty in each dimension, then one would expect to need about s^d subdivisions for the d-dimensional problem. Sampling techniques such as Monte Carlo (MC) and Quasi-Monte Carlo (QMC) methods are used for high dimensions.

A previous version of PARINT [8] allowed for multiple integration problems to be solved in parallel by dividing the processors into groups, with each group solving a single integration problem by a parallel APM. In this version, the groups could also have applied QMC methods depending

0-7695-0556-2/00 $10.00 © 2000 IEEE

on the type of problem. This hierarchical, two-level parallelization is well-suited for problems where sets of high-dimensional integrals need to be calculated, such as in the computational finance problem treated in [17]. The hierarchical system has a natural inclination towards a heterogeneous implementation. The ability to easily add the QMC technique to the hierarchical version tightens our focus on finding a heterogeneous implementation of QMC methods.

Automatic QMC techniques require a sequence of approximations over all of $\mathcal{H}_d$ until an adequate answer is reached. Each rule is a mean of function values and can be obtained from a number of partial sums. Each of these can be calculated independently, so they form a natural starting point for the parallelization of the task.

To calculate the approximations in sequence using parallel processes and a synchronous accumulation of the contributions from all processes to each element of the sequence would require a synchronization point for each sequence element. On a heterogeneous network this would result in a large communication cost. Furthermore, on a network there is often a significant lag between the creation of the first and the last of the parallel processes, so that processes which started earlier would have to wait for the later ones. This suggests an asynchronous approach for updating the global results.

3. Quasi-Monte Carlo methods

Basic QMC methods evaluate the integrand function at a set of points calculated deterministically, as opposed to MC which evaluates the integrand at random points. QMC methods are classified according to the type of point set used. Lattice rules have been found particularly useful for mid-range dimensions (say, $10 \leq d \leq 20$). Equidistributed point sequences have been used effectively in higher dimensions as well, including Richtmyer rules and low discrepancy rules such as Sobol's LP_τ sequences.

We use the lattice (Korobov) and Richtmyer rules from [12] for low-to-moderate and for high dimensions, respectively. Their convergence properties are generally better than the $\mathcal{O}(1/\sqrt{N})$ rate of classic MC, where N is the number of points used.

Let $I = K_N + E_N$ be the d-variate integral of f over $\mathcal{H}_d$ with

$$K_N = \frac{1}{N}\sum_{i=1}^{N} f(\{\frac{i}{N}\mathbf{v}\}), \tag{1}$$

where $\{x\}$ = the fractional part of x and $\mathbf{v}$ is a predetermined generator vector with integer coefficients. The error for a sequence of lattice rules (1) for increasing $N = N_0, N_1, \ldots$, satisfies

$$E_N = \mathcal{O}(\frac{(\log N)^{k\gamma^*}}{N^k}),$$

where γ^* is known as the index of the rules [4], for $f \in \mathcal{E}^k$ $(k > 1)$, which is the class of all functions f, periodic with period 1 in each variable and for which the Fourier coefficients $c_{\mathbf{m}}$ satisfy

$$|c_{\mathbf{m}}| \leq C r_{\mathbf{m}}^{-k}$$

for all $\mathbf{m} \neq 0$, a constant $C > 0$ and $r_{\mathbf{m}} = \prod_{j=1}^{d} \max\{1, |m_j|\}$.

In view of the periodicity requirement, a periodizing transformation is applied to the original integral.

Two drawbacks are that good lattice rules are hard to obtain in high dimensions and that the error is hard to estimate.

Richtmyer rules satisfy $E_N = \mathcal{O}(N^{-1})$ for $f \in \mathcal{E}^k$, $k > 1$. We have $I = R_N + E_N$ where

$$R_N = \frac{1}{N}\sum_{i=1}^{N} f(\{i\theta_1\}, \ldots, \{i\theta_d\})$$

and $\theta_1, \theta_2, \ldots, \theta_d$ are d irrational numbers such that $1, \theta_1, \theta_2, \ldots, \theta_d$ satisfy $\lambda_0 + \lambda_1\theta_1 + \cdots + \lambda_d\theta_d \neq 0$ for rational λ coefficients not all zero [4].

Genz [12] uses $\theta_i = \sqrt{\pi_i}$ where π_i is the i-th prime, and applies the Richtmyer sequence for dimensions > 20.

4. Algorithm

Cranley and Patterson [3] randomize (1) to obtain a stochastic family of rules. Let

$$K_N(\boldsymbol{\beta}) = \frac{1}{N}\sum_{i=1}^{N} f(\{\frac{i}{N}\mathbf{v} + \boldsymbol{\beta}\}), \tag{2}$$

where $\boldsymbol{\beta}$ is a uniformly distributed random vector. Using a random sample set of size q,

$$\bar{K}_N = \frac{1}{q}\sum_{j=1}^{q} K_N(\boldsymbol{\beta}_j) \tag{3}$$

preserves the integration properties of (1) and allows for a standard error estimation by $\hat{\sigma}_q^2 = E_N = \frac{1}{q(q-1)}\sum_{j=1}^{q}(K_N(\boldsymbol{\beta}_j) - \bar{K}_N)^2$.

Our algorithm must calculate K_N values for successively larger values of N until either an answer is found to the user-specified accuracy or the function count limit is reached. As these $K_{N_1}, K_{N_2}, \ldots$, values are calculated, the overall result is calculated as the weighted sum $\bar{Q} = (\sum_i \frac{\bar{K}_{N_i}}{E_{N_i}})/(\sum_i \frac{1}{E_{N_i}})$ (where $\bar{K}_{N_i}$ is weighted with the inverse of its corresponding squared standard error) and, correspondingly, $\bar{E} = 1.0/(\sum_i \frac{1}{E_{N_i}})$. The randomized lattice rules in 3 can be written as

$$\kappa_{ij} = K_{N_i}(\boldsymbol{\beta}_j), \quad j = 1, \ldots, J_i \leq q, \tag{4}$$

where a single κ_{ij} represents a single unit of work in our algorithm. The κ_{ij} values are calculated by the worker processes and applied (asynchronously) to update the overall result $\bar{K}_{N_i}$ and error by the controller. It is thus not necessary that $J_i = q$ in the asynchronous computation. In our implementation, the controller can optionally also function as a worker.

To alleviate a potential erratic generation of the κ_{ij} table, a successful distributed algorithm will attempt to gradually calculate higher-level rules as needed to reach the desired accuracy, while keeping all processors busy performing useful work. Particularly on a heterogeneous network of workstations, in which the various processors may be operating at different speeds and which may have different communication latencies between processors, the asynchronous algorithm must perform dynamic load balancing to keep the processors busy.

Our algorithm fits within the paradigm of scheduling tasks within a distributed system [11]. An important characteristic of this problem is that we do not know in advance when termination will occur, as it depends upon either reaching the user's accuracy or the function count limit. There are no strict *precedences* among the tasks (a task being the calculation of a κ_{ij} value), but the algorithm will generally perform minimal work when a full K_{N_i} value is known before beginning to calculate $K_{N_{i+1}}$.

Initially, all workers independently and *statically* assign themselves an initial task for a low-level rule and report their answer to the controller via an update message. After that, the controller *dynamically* assigns tasks to the workers via work assignment messages as the updates are received.

A work request is for the calculation of one or more κ_{ij} values. Since higher level rules require more function evaluations, a reasonable approach is to apportion the work into pieces of similar size based on the number of function points required to complete the work. A single work request may therefore consist of a set of κ_{ij} for multiple values of j and i. However, we have found that after several rows have been calculated, and for the computationally intensive problems with which we are experimenting, it is enough to assign κ_{ij} cells to workers one at a time, as the computation of each cell can require a significant amount of work.

Note that this method of task allocation can be considered a FIFO work assignment [11], as the statically defined sequence of rules (for a given q value in (3)) forms an implicit queue of tasks to be completed, and the set of work assignments for a worker forms its own FIFO queue. Once a task has been assigned to a worker, it is not transferred.

The heterogeneity of the target parallel platform is handled automatically, in that the faster workers will finish work more often, and therefore are assigned more work than the slower processors.

5. Performance on a heterogeneous network

In this section we will discuss a model (of [2]) to assess algorithm efficiency on a heterogeneous network.

Let the network be designated by ν. The total work, W, is assumed constant. The work is split up over p processors, processor i being responsible for the part W_i of the total work, i.e., $\sum_{i=1}^{p} W_i = W$. Since we will assume a particular ordering on the processors of the network, we refer to the $p-$processor portion of the network by ν_p.

The *speed* of a processor (for the application) is expressed as work performed per unit of time, i.e, the speed σ_i of processor i is $\sigma_i = W_i/T_i$, where T_i is the sequential time for the execution of W_i on processor i. Note that the time T_i is proportional to the number of units of work W_i.

The total speed of the network is $\sum_{i=1}^{p} \sigma_i$ for the corresponding partitioning of the work. The *relative network speed* $R_\rho(\nu_p)$ with respect to reference processor ρ is then defined as

$$R_\rho(\nu_p) = \frac{1}{\sigma_\rho} \sum_{i=1}^{p} \sigma_i,$$

and has the meaning of the number of processors equivalent to the reference processor, which would together have the same total speed as the network ν_p. Furthermore, $R_\rho(\nu_p)$ provides a bound for the speedup which can be obtained on the network.

Denoting the sequential time on reference processor ρ by $T = \frac{W}{\sigma_\rho}$, the speedup of the network with respect to the reference processor is given by

$$S_\rho(\nu) = \frac{T}{T(\nu_p)} = \frac{W}{\sigma_\rho T(\nu_p)},$$

and $S_\rho(\nu_p) \leq R_\rho(\nu_p)$. Hence, the efficiency of the network,

$$E(\nu_p) = \frac{S_\rho(\nu_p)}{R_\rho(\nu_p)} \leq 1.$$

Note that it is customary to use the fastest processor as the reference processor in defining the speedup [16, 18]. Furthermore, the above model is invalid when there are different types of work (which may take different times per unit of work on different processors). The study in [16] shows how this contributes to allowing superlinear speedup, for their decomposition of a global climate model on a heterogeneous, distributed system.

We remark that Colombet [2] extends his model by splitting up the work into its different types, $W = \sum_{j=1}^{J} W^j$, so that a cost (time) τ_{ij} is incurred per unit work of type j on processor i. Denoting the number of units of type j work on processor i by w_{ij}, the parallel time can be minimized by solving for $\min_{\mathbf{w}} T(\nu_p) = \min_{\mathbf{w}} \max_{i=1}^{p} (\sum_{j=1}^{J} \tau_{ij} w_{ij})$. While imposing further that $T(\nu_p) = T_i$, $1 \leq i \leq p$, the

above expression becomes $T_{//} = min_{\mathbf{w}}(\sum_{j=1}^{J} \tau_{pj} w_{pj})$. Solving the optimization problem for $w_{ij} > 0$ and denoting the optimal solution elements by w_{ij}^*, the work assigned to processor i by $W_i^* = \sum_{j=1}^{J} w_{ij}^*$ and corresponding speed by $\sigma_i = W_i^*/T_{//}$, a bound for the speedup is again obtained by $S_\rho(\nu_p) \leq R_\rho(\nu_p)$.

6. Experimental results

6.1. Problem class

Preliminary results were reported in [9] using static load balancing and Korobov rules for the calculation of the multivariate normal distribution function $|\Sigma|^{-\frac{1}{2}}(2\pi)^{-\frac{d}{2}} \int_{\mathbf{a}}^{\mathbf{b}} e^{-\frac{1}{2}\mathbf{x}^T\Sigma^{-1}\mathbf{x}} d\mathbf{x}$, where Σ is an $d \times d$ symmetric positive definite covariance matrix and one or more of the integration limits may be $+\infty$ or $-\infty$. Good accuracy and speedup were obtained on our LAN of Ultra-10 Sparcstations and on the IBM-SP machine at Argonne National Laboratory.

In [10] we reported timing results using static load balancing on a homogeneous network, for a class of problems based on an integral $I(f)$ from computational finance [1, 15], where $f(\mathbf{x})$ has a Gaussian weight and a factor of the form $g(\mathbf{x}) =$

$$C \sum_{k=1}^{d} \frac{((1 - w_k(\mathbf{x})) + w_k(\mathbf{x}) c_k) \prod_{j=1}^{k-1} (1 - w_j(\mathbf{x}))}{\prod_{j=0}^{k-1} (1 + i_j(\mathbf{x}))}, \quad (5)$$

where $i_j(\mathbf{x}) = i_0 K_0^j e^{\tilde{\sigma}(x_1 + \ldots + x_j)}$, $K_0 = e^{-\tilde{\sigma}^2/2}$, $w_k(\mathbf{x}) = K_1 + K_2 \tan^{-1}(K_3 i_k(\mathbf{x}) + K_4)$, $c_k = \sum_{j=0}^{n-k} (1+i_0)^{-j}$, and the domain of integration is $\Re^d$. The integral represents the current value of a security backed by mortgages of length (d) months with fixed monthly interest rate i_0.

In order to map the infinite domain to the $d-$dimensional unit hypercube we performed the transformation $z_k = \Phi(x_k), \quad k = 1, \ldots, d$, where Φ_k is the univariate normal distribution function. Note that the transformation absorbs the Gaussian weight. The resulting integral is $\int_{\mathcal{H}_d} f(\mathbf{z}) d\mathbf{z}$ with

$$f(\mathbf{z}) = g(\Phi^{-1}(z_1), \ldots, \Phi^{-1}(z_d)).$$

Using the constants $C = 1$, $i_0 = .007$, $\tilde{\sigma} = .02$ and $K_1 = .01$, $K_2 = -.005$, $K_3 = 10$, $K_4 = .5$ yields Caflish and Morokoff's "nearly linear" problem.

6.2. Experimental environment

We ran our tests on a network ν of Sparc workstations, using 10 Ultra-Sparc 10's, 1 Sparc 20, and 5 Sparc 5's. We took an Ultra 10 as our reference machine ρ, and always considered orderings of the machines in decreasing speed.

Table 1. Table of machines' relative performances

Machine	Time	σ/σ_ρ	σ_ρ/σ
Ultra Sparc 10	123.69	1.00	1.00
Sparc 20	228.20	1.84	0.54
Sparc 5	571.91	4.62	0.22

To derive speeds relative to ρ, we timed the sequential solving of a reference problem on each type of machine (thus ensuring a benchmark containing a similar mix of instructions as the actual algorithm tested). The problem was the financial problem in 50 dimensions to 10000 function evaluations. Table 1 shows the results: the time (in seconds) for each machine to solve the reference problem, the speed $\sigma = W/T$ (using a reference workload W of 1.0), the relative speed σ/σ_ρ, and the inverse of the relative speed (σ_ρ/σ).

The ratio for the Sparc 5 is 4.62, meaning that an Ultra 10 has the same performance as 4.62 Sparc 5's, or, inversely, that adding a Sparc 5 to a given set of machines adds the power of about one-fifth of an Ultra 10.

We present results for the financial problem solved to 60 dimensions. The function count limit was set to 30000 evaluations, with q (samples per lattice rule) set to 10. The error tolerance was set low so that termination occurred by reaching the function count limit.

6.3. Speedup and efficiency results

The speedup $S_\rho(\nu_p)$ and ideal speedup $R_\rho(\nu_p)$ was calculated for $1 \leq p \leq 16$ as per Section 5. The graph in Figure 1 shows the speedup results. As the processors were ordered from fastest to slowest, the reduced slope in the ideal speedup curve represents the reduced expectations from the slower workstations. Figure 2 shows the same results in an efficiency graph.

From the efficiency graph, one can see that we achieve an efficiency of about .70, and retain that efficiency through about 13 workstations. At that point, we are using 10 fast workstations, 1 medium, and 2 slow. Beyond that point we begin to lose efficiency.

We believe that this loss in efficiency is due to the slower communication to and from the slower workstations. The relative speed σ_i/σ_ρ of workstation i only takes into account the relative computational speed. If a workstation has slower communication hardware and software, then this workstation will work even slower than its relative speed can account for. As the experimental results model does not take this into account, it appears as a loss of efficiency. In these models, any form of communication, at any speed, is

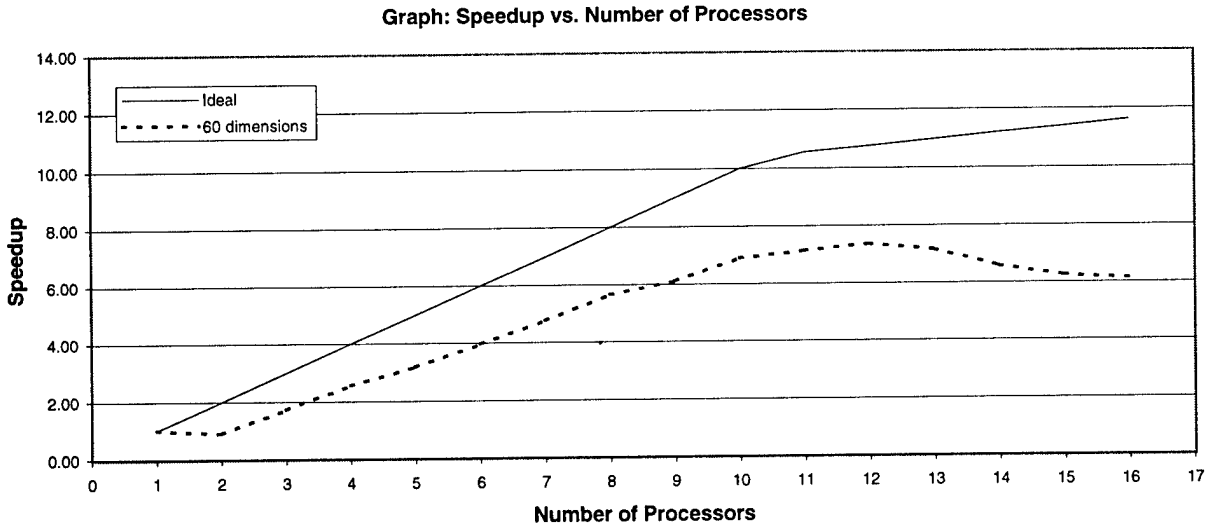


Figure 1. Speedup results

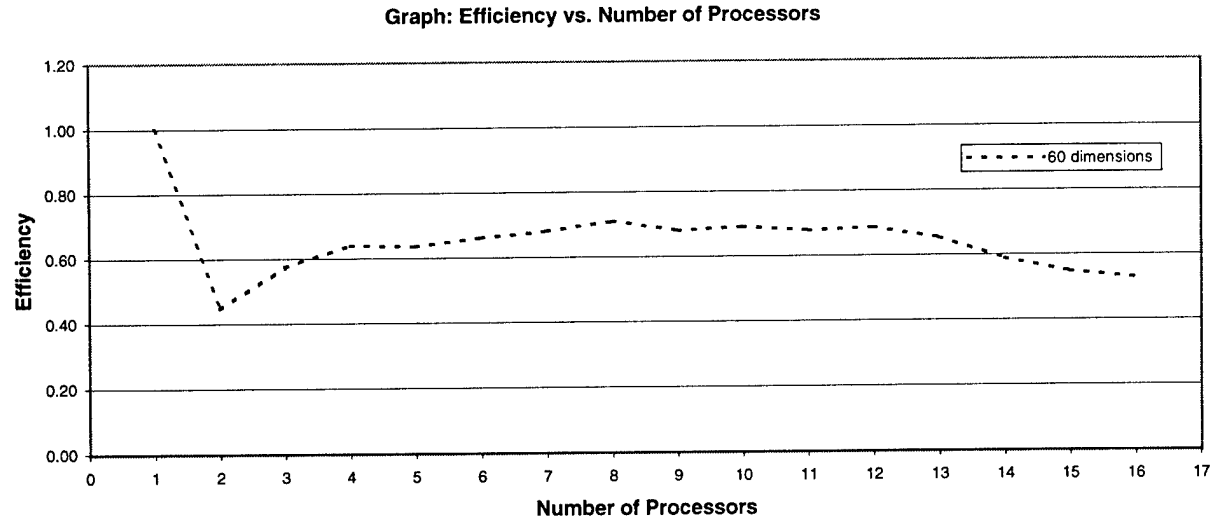


Figure 2. Efficiency results

taken to be a loss of efficiency, as the comparison is always to a sequential implementation that does no communication.

Note the downward dip in efficiency and speedup at $p = 2$. We determined that having the controller participate in the work resulted in problems. With this option on, the controller would begin to calculate a κ_{ij} value when there were no updates to receive. When a task was completed, the controller would again look for updates. This worked fine in the lower rules, but as the rules became more expensive to calculate, the controller could work for extended periods of time without checking for any updates. The result was *starvation loss* (i.e., a loss in efficiency), where the workers would be waiting idle for work, and, *breaking loss*, where the update that would allow the algorithm to terminate would be waiting on an incoming message queue while the controller was busy.

The fact that the controller is not working is the cause of the downward dip in the graphs at $p = 2$. A hybrid approach is probably best, allowing for the controller to do smaller amounts of work at a time, especially for small values of p, and allowing it to devote more time, as necessary, to controlling the workers for larger values of p.

Breaking loss also forms a loss of efficiency, regardless of the issue of the controller working. As higher level rules are calculated, each rule requires more and more points. If execution terminates due to reaching the function count limit, then the final function count will be generally higher than the limit; the number of excess function values will be the amount required to finish the last κ_{ij}. As the tasks require more points, this amount increases.

Also note that our use of a non-dedicated network was detrimental to the overall efficiency.

6.4. Comparisons with adaptive partitioning

As mentioned in Section 1, adaptive partitioning techniques suffer exponentially as the number of dimensions increases. For a given function of moderate (say, 5 to 15) dimensions, it is interesting to consider whether APM or QMC techniques perform better. Table 2 shows some results comparing these techniques for the financial problem at 10 dimensions, for varying amounts of requested accuracy. The ε_d values referenced in the table correspond roughly to the number of digits of accuracy requested. The function count limit was set to 3 million.

The table indicates that neither method has to do much work initially. The QMC technique is able to handle tighter accuracies before going over the function limit. The APM blows up as soon as any partitioning is required. Note that given the large number of points required per cubature rule application (1265 points at 10 dimensions for our rules from [13, 14]), 3 million function evaluations corresponds to only about 2300 rule evaluations.

Table 2. Table of number of function evaluations and time (in seconds), vs. requested accuracy, for APM and QMC

ε_d	APM		QMC	
	Time	Fcn Evals	Time	Fcn Evals
1	0.35	1265	0.21	620
2	0.35	1265	0.21	620
3	0.35	1265	0.21	620
4	0.36	1265	0.21	620
5		> 3M	0.21	620
6			0.22	620
7			2.83	8740
8			58.66	181000
9			133.41	413000
10				> 3M

7. Conclusions and future work

We focused on the asynchronous computation of a sequence of stochastic lattice rule families using dynamic load balancing, and presented test results on a heterogeneous network.

Further theoretical work and experimentation is needed, e.g., regarding the number of entries needed in a family for satisfactory error estimation. We also need to assess the quality of the weighted average of the sequence of results and error estimates. We are investigating the applicability of LP_τ sequences to deal with some classes of singular behavior [5]. Furthermore, we intend to experiment with modifications of the scheduling strategy.

Finally, we are considering the incorporation of the asynchronous QMC methods as a significant addition to ParInt [7].

References

[1] R. Caflisch and W. Morokoff. Quasi-Monte Carlo computation of a finance problem. In *Proceedings of the Workshop on Quasi-Monte Carlo Methods and their Applications*, pages 15–30. Statistics Research and Consultancy Unit, Hong Kong Baptist Univeristy, 1996.

[2] L. Colombet. *Parallélisation d'applications pour des réseaux de processeurs homogènes ou héterogènes.* PhD thesis, l'Institut National Polytechnique de Grenoble, 1994.

[3] R. Cranley and T. N. L. Patterson. Randomization of number theoretic methods for multiple integration. *SIAM J. Numer. Anal.*, 13:904–914, 1976.

[4] P. J. Davis and P. Rabinowitz. *Methods of Numerical Integration.* Academic Press, New York, 1984.

[5] E. deDoncker and A. Genz. Parallel computation of multivariate normal probabilities. *Computing Science and Statistics*, 30, 1999.

[6] E. deDoncker, A. Gupta, J. Ball, P. Ealy, and A. Genz. ParInt: A software package for parallel integration. In 10^{th} *ACM International Conference on Supercomputing*, pages 149–156. Kluwer Academic Publishers, 1996.

[7] E. deDoncker, A. Gupta, A. Genz, and R. Zanny. PARINT web site. At http://www.cs.wmich.edu/parint.

[8] E. deDoncker, A. Gupta, and R. Zanny. Large scale parallel numerical integration. *Journal of Computational and Applied Mathematics*, 112:29–44, 1999.

[9] E. deDoncker, A. Genz, and M. Ciobanu. Parallel computation of multivariate normal probabilities. *Computing Science and Statistics*, 30, 1999.

[10] E. deDoncker, R. Zanny, M. Ciobanu, and Y. Guan. Asynchronous quasi monte-carlo methods. In *Proceedings of the High Performance Computing Symposium 2000*. To appear.

[11] H. El-Rewini, T. G. Lewis, and H. A. Ali. *Task Scheduling in Parallel and Distributed Systems*. Prentice Hall Series in Innovative Technology. Prentice Hall, 1994.

[12] A. Genz. MVNDST: Software for the numerical computation of multivariate normal probabilities, 1998. At http://www.sci.wsu.edu/math/faculty/genz/homepage.

[13] A. Genz and A. Malik. An adaptive algorithm for numerical integration over an n-dimensional rectangular region. *Journal of Computational and Applied Mathematics*, 6:295–302, 1980.

[14] A. Genz and A. Malik. An imbedded family of multidimensional integration rules. *SIAM J. Numer. Anal.*, 20:580–588, 1983.

[15] A. Genz and J. Monahan. A stochastic algorithm for high dimensional integrals over unbounded regions with Gaussian weight. Available from website at http://www.sci.wsu.edu/math/faculty/genz/homepage.

[16] C. R. Mechoso, J. D. Farrara, and J. A. Spahr. Achieving superlinear speedup on a heterogeneous, distributed system. *IEEE Parallel and Distributed Technology*, 2(2):57–61, 1994.

[17] S. H. Paskov and J. F. Traub. Faster valuation of financial derivatives. *J. Portfolio Management*, 22(1):113–120, 1995.

[18] X. Zhang and Y. Yan. Modeling and characterizing parallel computing performance on heterogeneous networks of workstations. In *Proceedings of the Seventh IEEE Symposium on Parallel and Distributed Processing*, pages 25–34, 1995.

Elise deDoncker is a professor of computer science at Western Michigan University. Her research interests include parallel and distributed algorithms/ computing, scientific computations, numerical analysis and quantum computing. She received her doctorate in Mathematics from the Katholieke Universiteit Leuven in Belgium.

Rodger Zanny is a doctoral student at Western Michigan University. His research interests include numerical integration, adaptive partitioning algorithms, search algorithms, and more generally, distributed and parallel computing. He received his B.S. in Computer Science in 1991 from Cornell University and his M.S. from WMU in 1999.

Manuel Ciobanu graduated from the Department of Computer Science at Western Michigan University. His interests include parallel and distributed computing, and weather modeling. He works for Reliant Energy Services (Houston, TX) as a Quantitative Analyst.

Yuqiang Guan is a doctoral student in computer science at the University of Texas at Austin. His research interests include the theory of computation, parallel/ scientific computing, algorithms, quantum computing, data mining and system modeling.

SESSION 3-B
SCHEDULING I

Chair: A. Somani, *Iowa State University, USA*

Scheduling Multi-Component Applications in Heterogeneous Wide-Area Networks

Jon B. Weissman

Department of Computer Science and Engineering
University of Minnesota
(jon@cs.umn.edu)

Abstract

In this paper we present a scalable scheduling heuristic for several common classes of multi-component applications (meta-applications). We consider this scheduling problem in a wide-area heterogeneous computing environment, or metasystem. The heterogeneity and scale of the computing environment and the heterogeneity of the application make this a challenging problem. We have studied the performance of the heuristic in simulation and the results are encouraging. Completion times for three common classes of meta-applications were within 10-20% of optimal on average with a worst-case variance of 60%. The results suggest that effective scheduling of meta-applications is possible, if sufficient application and system resource cost information is provided[1].

1.0 Introduction

Metacomputing is the seamless application of wide-area distributed computing resources to user applications. A number of research groups are building low-level metacomputing infrastructure [2][4]. What distinguishes metacomputing applications from other wide-area distributed applications is the objective of high-performance. In fact, it is the promise of performance greater than can be achieved using single site resources that makes metacomputing most attractive in solving complex scientific problems. The applications most suitable for metacomputing are often very heterogeneous in structure [6]. For instance, such applications may include remote databases or servers, remote instruments, remote supercomputers, VR devices, and humans-in-the-loop. In addition to hardware heterogeneity, the underlying networks that connect the different sites may also exhibit performance heterogeneity in both latency and bandwidth. The challenges inherent in heterogeneous computing are well known [3].

Exploiting the performance potential in metacomputing environments requires effective application scheduling: the selection and allocation of resources to the application. This problem is particularly challenging due to the heterogeneous nature of both the resources (machines and networks) and the application itself. In this paper, we consider the scheduling of *meta-applications*; applications consisting of multiple schedulable components across multiple sites with the goal of reduced completion time. Previous work showed that selecting the best single site for single component parallel applications can be done efficiently [8]. The scheduling of multiple interacting components across multiple sites is a complex problem especially when the network capacity is assumed to vary between sites. Most of the metacomputing scheduling work assumes either communication between components can be ignored, or the application will be confined to run in a single site, or the number of sites and components are small enough to make a brute-force scheduling algorithm feasible. In contrast, we present a scalable scheduling heuristic that has achieved excellent results (typically within 10% of optimal on average) in a detailed simulation study of three common classes of meta-applications.

2.0 Meta-Application Model

The underlying network contains a collection of sites connected by wide-area networks (Figure 1). The network sites each offer an amount of one or more resources (cycles, memory, disk, or other specialized resources) and have a point-to-point bandwidth to each site (bw_{i-j} that may be different). In this paper, we characterize the site resources and network capabilities solely in terms of their delivered performance to applications as in [1].

Meta-applications consist of a set of distinct application *components* that may communicate and interact

1. This work was partially funded by NSF ACR-9996418

0-7695-0556-2/00 $10.00 © 2000 IEEE

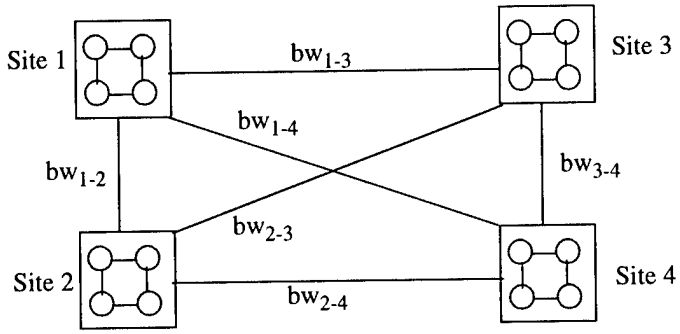


Figure 1: Network Model.

over the course of the application. Components may be schedulable computations, remote servers or databases, remote instruments, humans-in-the-loop, etc. (Figure 2). Computational components themselves may be sequential or parallel computations. Some components are fixed such as a remote database server and do not require scheduling, but may impact the scheduling of other components. For example, the placement of component B may be influenced by the amount of data transmitted by a database to B. If a great deal of data is moved, then a high-speed link between B and the database may be desired. It is also possible that other components are fixed due to scheduling constraints, such as a given program component must run in a particular site.

We consider meta-applications in which the inter-component communication pattern is statically known (Figure 2) and divide meta-applications into three categories — *concurrent*, *parallel*, and *pipeline* (Figure 3). Concurrent is the classic meta-application in which a set of components each running in a single site are executing concurrently and exchanging data. Examples include global climate modelling which often integrates several large-scale coupled computational models [5]. A *parallel* application is a special case of *concurrent* in which a component might benefit by distribution across multiple sites[2]. This normally applies to large-scale parallel applications in which a task (or subcomponent) may be replicated an arbitrary number of times with minimal task interaction relative to task computation. Very-coarse grain SPMD computations or highly compute-intensive applications such as large parallel simulations, RSA factoring, are examples in this category. Finally, pipeline applications consist of a number of components connected in a chain-like fashion. Examples include certain multi-disciplinary optimization problems in which the outputs of one program are often

2. This is not to be confused with a component that happens to be a parallel computation, but would not benefit by multi-site distribution.

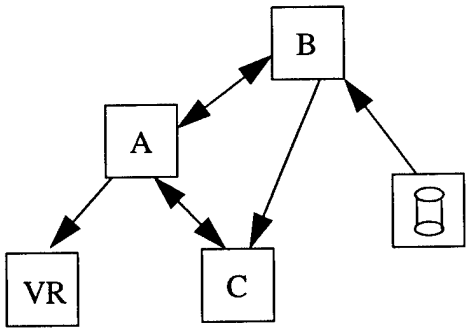
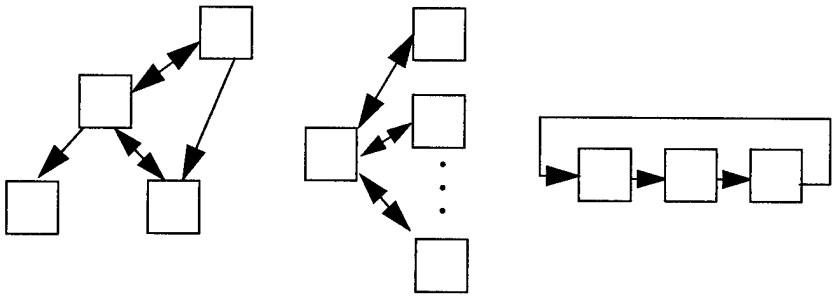


Figure 2: A meta-application consisting of five components. Three computations (A, B, C), a database server and a VR server. The presence of an arc indicates data communication.



a) concurrent b) parallel c) pipeline

Figure 3: Meta-application classes.

fed into another [7]. In addition, applications may be hybrids of the above classes. For example, the application may be pipelined, but one of the components may be parallel. In this work we consider concurrent and pipeline applications with components confined to a single site. Hybrid and parallel applications are the subject of future work. Scheduling meta-applications to minimize application completion time requires a cost model to evaluate the potential set of candidate schedules. These cost functions require information about the application. For each application component C_i, we assume the following information:

comp_amt [C_i] -- the amount of computation in # of instructions to be executed, C_i schedulable

comm_amt [C_i, C_j] -- the amount of communication (in # of bytes transmitted) between C_i, C_j

The following cost functions are constructed using this information and system resource information (S is a site, N is the number of components, n is the number of schedulable components, and m is the number of sites):

comp [C_i, S_k] for all ($i \le n$), ($k \le m$)

comm [C_i, C_j, S_k, S_l] for all ($i, j \le N$), ($k, l \le m$)

comm_total [C_i, S_k] for all ($i \le n$), ($k \le m$)

init_time [C_i, S_k] for all ($i \le n$), ($k \le m$)

The function *comp* gives the computation time for component C_i running in site S_k given the current state of the resources S_k is willing to provide to the application assuming no other component is scheduled there. If other components are also scheduled in S_k, then the *comp* function for these components may be degraded to reflect the sharing of S_k's resources.

The function *comm* gives the communication time spent by C_i in communication with C_j when C_i and C_j are run in S_k and S_l respectively. It has the value 0, if there is no communication. If multiple components are sharing a link, then the *comm* function for these components may be degraded to reflect the sharing of the link. The total communication cost for a component *comm_total*, is a function of the *comm* values associated with its links. In the simplest case it would be the sum of these costs, but in other situations some link communication might be overlapped. The communication costs will vary for different component assignments due to the heterogeneity of the underlying network. For example, some sites might be vBNS-connected, but others might be limited to standard Internet connections. The function *init_time* includes any start-up overhead which could include queue time for S_k's resources, time to transmit component binaries from the initiating site if they are not already located at S_k, etc. For simplicity, we will assume it is 0 and that $n = N$ in the remainder of the paper.

Numerous research groups, including ours, are addressing the problem of producing such cost functions using application and system resource information [1][9]. Here, we focus on the problem of making scheduling decisions given these cost functions for three classes of meta-applications: pipelines in which a computation stage cannot begin until the prior stage has completed, and concurrent applications in which all component computations and inter-component communication are either overlapped or sequential. The meta-application completion time T_{CT} can be defined in terms of the component cost functions (where i, j, k, l range over the values above):

(1) T_{CT} [pipeline] = Σ *comp* [C_i, S_k]+ *comm_total* [C_i, C_j, S_k, S_l]

(2) T_{CT} [concurrent, overlapped] = max {*comp* [C_i, S_k], *comm_total* [C_i, C_j, S_k, S_l]}

(3) T_{CT} [concurrent] = max {*comp* [C_i, S_k]+ *comm_total* [C_i, C_j, S_k, S_l]}

Other cost functions are possible depending on the application. For example, pipeline computation and communication could be overlapped, or *some* of the components within the application may have overlapped computation and inter-component communication. These cases are the subject of future work.

2.1 Scheduling Heuristic

The scheduling problem is to determine an assignment of schedulable components to site resources that minimizes T_{CT}. Unfortunately, this scheduling problem is NP-complete. An exhaustive search is not feasible since if there are *n* schedulable components and *m* sites, then there are m^n possible schedules if components can be co-located and single components cannot span multiple sites. We presume that while *n* may be small in practice (likely less than 10), *m* may grow as metacomputing environments achieve large-scale deployment. In practice, we may only want to consider a subset of sites or we may limit *m* to include sites that contain resources specifically requested by the application.

The general steps of the scheduling process are the following:

1: determine candidate sites; for each site
  2: a) collect available resources from the site
  b) compute cost function estimates for each component from the site
3: run scheduling heuristic to search for best component/site assignment

Two scheduling heuristics have been developed, one suitable for compute-intensive meta-applications and the other for data-intensive meta-applications (a newly emerging class of meta-applications). In the former class, component-site scheduling is most critical, while for the latter, link scheduling (i.e. the link capacity is considered first) becomes important.

TABLE 1. Simulation Parameters

name	**value range/units**	**comments**
num sites	3 .. 10	this size covers today's testbeds
comp_rate	[1, 10000] MIPS	covers weak to powerful sites
intra_link_rate	[500, 1000] KBps -- fixed value	~ ethernet speed
inter_link_rate	[50, 100], [100, 200], [500, 10000] KBps	slow Internet up to vBNS-like
link_variance	1, 2, 5, 10	reflects ↑ network heterogeneity
affinity	1, 2, 5, 10	reflects ↑ component/site affinity
appl_type	concurrent w/wo overlap, pipeline	
num_components	3 .. 8	8 should cover most meta-apps
comp_amt	[10000, 100000], [100000, 100000], [1000000, 10000000] MInstructions	comp_ and comm_amt ranges cover medium-coarse grain apps
comm_amt	[1, 10000], [1, 100000], [1, 100000] Bytes	
col_degrade	[0, 1] -- real interval	0 = no degradation, 1 = linear

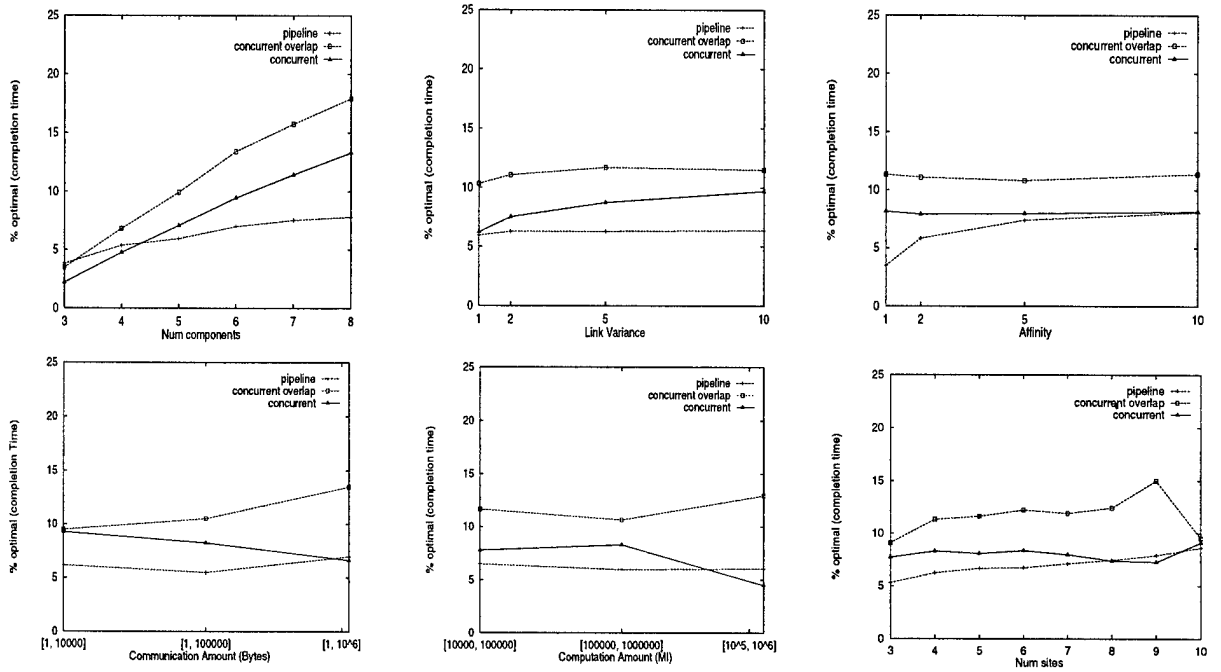


Figure 4: Performance of scheduling heuristic

Below we present the compute-intensive heuristic.

This heuristic considers components in decreasing order of computational weight:

1. order application components by decreasing *comp_amt* value
2. init PLACEMENT[C_i]= -1 for all components C_i
3. for each component C_i
   4. for each site S_k
   5. compute T_{CT} with PLACEMENT[C_i]=S_k, given PLACEMENT[C_j], j<i, unchanged
   (use Eq. 1-3 to compute T_{CT} based on application type)
   6. remember best T_{CT} and best associated site S_{best}
   7. end for
   8. PLACEMENT[C_i]=S_{best}
9. end for

This is a greedy algorithm with complexity $O(mn^2)$. There are *mn* schedules and *n* steps to evaluate T_{CT} (*n* links per component). We consider this heuristic to be scalable as *n* will be small in practice. It finds a single best candidate. The computation of T_{CT} in step 5 is based on an assignment of components up to the current component. The *comp* and *comm* terms for unassigned components are set to 0 in this calculation.

3.0 Initial Results

We have run the scheduling heuristic over a large set of simulated metacomputing environments and meta-applications, and measured the performance of the heuristic with respect to the optimal schedule. The simulated metacomputing environment consists of a number of interconnected sites that provide a particular computation and communicate rate. The communication rate between the different sites is also specified. A link variance parameter is used to vary to inter-site communication performance allowing us to simulate a truly uneven network.

Meta-applications consists of a type and a number of components. For each component we have an amount of computation and an amount of communication to all other components. These parameters are varied to allow us to simulate a wide range of application granularities. We also provide an affinity parameter which is used to bias particular components to particular sites. From this parameter, we generate an affinity value for each component site pair. This allows us to simulate environments with differing degrees of heterogeneity. This

parameter is used to adjust the *comp* value. Finally, when components are collocated to particular sites, we use a parameter to degrade the sites computing power to all hosted components. This value ranges from no effect (e.g. perhaps the site has ample resources for all components) to a degradation linear in the number of hosted components. Currently, we do not degrade inter-site link performance in the event that a link is shared since we do not yet have empirical data for this parameter. For each schedule to be evaluated, the simulator can easily construct *comp*, *comm_total*, and T_{CT} from these parameter values. The function *comm_total* is defined to be the sum of the individual link communication costs for a component.

Realistic values are chosen for the parameter ranges (Table 1). For example, the intra-site link rate is based on a typical ethernet LAN, while the inter-site rate ranges from Internet WAN speeds up to reported vBNS rates. For parameters that are specified as ranges, a random value uniformly distributed over the range is generated for each simulation run.

The results that suggest that the heuristic performs very effectively over the simulated parameter ranges for the three application classes. A simulation study of over 800,000 distinct environment and application instances was performed. For each run, we execute the heuristic and search for the optimal schedule. We use several comparison metrics: how close the heuristic is to the optimal on average and the maximum variance from optimal. The first set of results are depicted in Figure 4. The heuristic performs best on the application classes in the following order: pipeline, concurrent, and concurrent-overlap. This order reflects increasing sensitivity of the overall completion time to the scheduling of a single component. Since the completion time of pipelines are a sum of all components, the heuristic can afford to schedule a few components sub-optimally. For the other classes, suboptimal scheduling of a single component can have a larger impact on the completion time. However, the heuristic performs quite well for all application classes. When the number of components is ≤ 5, the heuristic is within 10% of optimal on average. In general, it is within 20% on average. The heuristic also appears to be insensitive to the heterogeneity of the network environment; performance is fairly flat for changes in link_variance and affinity parameters. The heuristic also performs well as the amount of computation and communication varies. Pipeline applications are insensitive to these parameters, while the other application classes exhibit greater sensitivity. The heuristic is also sensitive to the number of sites in the environment, but exhibits slow degradation as the number of sites increases. The second set of results indicate that the worst-case variance from optimal is within 60% for all parameter ranges, and typically within 30-40% (Table 2).

Table 2. Maximum variance for heuristic. Shown for each parameter value and each application class.

Parameter	Max Variance for appl classes [% drift from optimal]: (pipeline, concurrent overlap, concurrent)
num sites	3: (21.7, 36.5, 26.8), 4: (24.5, 41.9, 28.2), 5: (25.7, 42.7, 28.5), 6: (25.4, 45.0, 30.0), 7: (25.6, 43.4, 28.3), 8: (27.1, 47.5, 25.5), 9: (28.4, 53.5, 28.5), 10: (32.4, 42.6, 28.6)
num_components	3: (19.2, 19.4, 12.2), 4: (23.3, 32.9, 22.6), 5: (23.7, 40.4, 27.4), 6: (25.8, 49.1, 32.8), 7: (27.0, 50.7, 25.6), 8: (27.0, 57.1, 38.0)
comp_amt	r1: (25.8, 44.5, 27.9), r2: (22.8, 38.3, 28.3), r3: (27.8, 40.5, 47.1)
comm_amt	r1: (21.7, 36.5, 3.1), r2: (21.3, 35.2, 25.8), r3: (29.7, 52.6, 25.4)
link_variance	1:(25.0, 40.9, 23.4), 2: (25.5, 41.9, 27.0), 5: (23.5, 42.1, 29.9), 10: (23.1, 40.8, 32.2)
affinity	1:(14.2, 42.1, 28.2), 2: (21.0, 41.5, 27.8), 5: (29.1, 40.2, 28.0), 10: (32.7, 42.0, 28.5)

4.0 Summary

The scalability of scheduling and resource management strategies will become an increasingly important problem as Grids are scaled up. We presented a scalable heuristic that has performed extremely well in a simulation study of synthetic meta-applications and metacomputing environments. Completion times for three common classes of meta-applications were within 10-20% of optimal on average with a worst-case variance of 60%. The results suggest that effective scheduling of meta-applications is possible, if sufficient application and system resource cost information is provided to the system. Future work includes experimental validation of the algorithms on a live system. We are also investigating the problem of multiple job scheduling and the interplay between multiple meta-applications and single-resource jobs.

5.0 References

[1] F. Berman and R. Wolski, "Scheduling From the Perspective of the Application," *Fifth International Symposium on High Performance Distributed Computing*, 1996.

[2] I. Foster and C. Kesselman, "Globus: A Metacomputing Infrastructure Toolkit," *International Journal of Supercomputing Applications*, 11(2), 1997.

[3] F. Freund and H.J. Siegel, "Heterogeneous Processing," *IEEE Computer*, 1993.

[4] A.S. Grimshaw and W. A. Wulf, "The Legion Vision of a Worldwide Virtual Computer," *Communications of the ACM*, Vol. 40(1), 1997.

[5] C.R. Mechoso et al, "Running a Climate Model in a Heterogeneous Distributed Computer Environment," *Proceedings of the Third IEEE International Symposium on High Performance Distributed Computing*, 1994.

[6] G.J. McRae, "How Application Domains Define Requirements for the Grid," *Communications of the ACM*, Vol. 40(11), 1997.

[7] Multidisciplinary Optimization Home Page, *http://akao.larc.nasa.gov/dfc/mdo/moo.html.*

[8] J.B. Weissman, "Gallop: The Benefits of Wide-Area Computing for Parallel Processing," *Journal of Parallel and Distributed Computing*, Vol 54(2), November 1998.

[9] J.B. Weissman and A.S. Grimshaw, "A Framework for Partitioning Parallel Computations in Heterogeneous Environments," *Concurrency: Practice and Experience*, Vol. 7(5), pp. 455-478, 1995.

Biography

Jon B. Weissman received the B.S. degree from Carnegie-Mellon University in 1984, and the M.S. and Ph.D. degrees from the University of Virginia in 1989, 1995. He joined the University of Minnesota as an Assistant Professor of Computer Science in Fall 1999. He was an active member of the Mentat and Legion research groups while at the University of Virginia. His current research interests are in distributed systems, high performance computing, and metacomputing.

Application-Aware Scheduling of a Magnetohydrodynamics Application in the Legion Metasystem

Holly Dail* Graziano Obertelli*

Francine Berman* Rich Wolski† Andrew Grimshaw‡

* Computer Science and Engineering Department
University of California, San Diego
[hdail, graziano, berman] @cs.ucsd.edu

† Department of Computer Science
University of Tennessee
rich@cs.utk.edu

‡ Department of Computer Science
University of Virginia
grimshaw@virginia.edu

Abstract

Computational Grids have become an important and popular computing platform for both scientific and commercial distributed computing communities. However, users of such systems typically find achievement of application execution performance remains challenging. Although Grid infrastructures such as Legion and Globus provide basic resource selection functionality, work allocation functionality, and scheduling mechanisms, applications must interpret system performance information in terms of their own requirements in order to develop performance-efficient schedules.

We describe a new high-performance scheduler that incorporates dynamic system information, application requirements, and a detailed performance model in order to create performance efficient schedules. While the scheduler is designed to provide improved performance for a magneto hydrodynamics simulation in the Legion Computational Grid infrastructure, the design is generalizable to other systems and other data-parallel, iterative codes. We describe the adaptive performance model, resource selection strategies, and scheduling policies employed by the scheduler. We demonstrate the improvement in application performance achieved by the scheduler in dedicated and shared Legion environments.

This research was supported in part by DARPA Contract#N66001-97-C-8531, DoD Modernization Contract 9720733-00, and NSF/NPACI Grant ASC-9619020

1. Introduction

Computational Grids [7] are rapidly becoming an important and popular computing platform for both scientific and commercial distributed computing communities. Grids integrate independently administered machines, storage systems, databases, networks, and scientific instruments with the goal of providing greater delivered application performance than can be obtained from any single site. There are many critical research challenges in the development of Computational Grids as an effective computing platform. For users, both performance and programmability of the underlying infrastructure are essential to the successful implementation of applications in Grid environments.

The **Legion** Computational Grid infrastructure [11] provides a sophisticated object-oriented programming environment that promotes application programmability by enabling transparent access to Grid resources. Legion provides basic resource selection, work allocation, and scheduling mechanisms. In order to achieve desired performance levels, applications (or their users) must interpret system performance information in terms of requirements specific to the target application. **App**lication **L**evel **S**cheduling (**AppLeS**) [3] is an established methodology for developing adaptive, distributed programs that execute in dynamically changing and heterogeneous execution settings. The ultimate goal of this work is to draw upon the AppLeS and Legion Computational Grid research efforts to design an adaptive application scheduler for regular iterative stencil codes in Legion environments.

We consider a general class of regular, data-parallel stencil codes which require repeated applications of relatively

0-7695-0556-2/00 $10.00 © 2000 IEEE

constant-time operations. Many of these codes have the following structure:

```
Initialization

Loop over an n-dimensional mesh

Finalization
```

in which the basic activity of the loop is a stencil based computation. In other words the data items in the n-dimensional mesh are updated based on the values of their nearest neighbors in the mesh. Such codes are common in scientific computing and include parallel implementations of matrix operations as well as routines found in packages such as ScaLAPACK [18].

In this paper we focus on the development of an adaptive strategy for scheduling a regular, data-parallel stencil code called PMHD3D on the Legion Grid infrastructure. The primary contributions of this paper are:

- We describe an adaptive performance model for PMHD3D and demonstrate its ability to predict application performance in initial experiments. The performance model represents the application's requirements for computation, communication, overhead, and memory, and could easily be extended to serve more generally as a framework for regular iterative stencil codes in Grid environments.
- We couple the PMHD3D performance model with resource selection strategies, schedule selection policies, and deployment software to form an AppLeS scheduler for PMHD3D.
- In order to satisfy the requirements of the PMHD3D performance model we implement and utilize a new *memory sensor* as part of the Network Weather Service (NWS)[22]. The sensor collects measurements and produces forecasts of the amount of free memory available on a processor.
- We demonstrate the ability of the AppLeS methodology to provide enhanced performance for the PMHD3D application, using the Legion software infrastructure as a platform for high-performance application execution.

In the next section we discuss the structure of the target application and the environment that we used as a test-bed. In Section 3, we discuss the AppLeS we have designed for PMHD3D and provide a generalizable performance model. Section 4 provides experimental results and demonstrates performance improvements we achieved via AppLeS using Legion. In Sections 5 and 6 we review related work and investigate possible new directions, respectively.

2. Research Components: AppLeS, NWS, PMHD3D and Legion

In order to build a high-performance scheduler for PMHD3D we leveraged application characteristics, dynamic resource information from NWS, the AppLeS methodology, and the Legion system infrastructure. In this section we explain each of these components in detail.

2.1. AppLeS

The AppLeS project focuses on the development of a methodology and software for achieving application performance via adaptive scheduling [1]. For individual applications, an *AppLeS* is an agent that integrates with the application and uses dynamic and application-specific information to develop and deploy a customized adaptive application schedule. For structurally similar classes of applications, an *AppLeS template* provides a "pluggable" framework which comprises a class-specific performance model, scheduling model, and deployment module. An application from the class can be instantiated within the template to form a performance-oriented self-scheduling application targeted to the underlying Grid resources.

AppLeS schedulers often rely on available tools in order to deploy the schedule or to gather information on resources or environment. AppLeS commonly depends on the Network Weather Service (NWS) (see Section 2.4) to provide dynamic predictions of resource load and availability. Together, AppLeS and the Network Weather Service can be used to adapt application performance to the deliverable capacities of Grid resources at execution time. In this project AppLeS uses Legion to execute a schedule and the Internet Backplane Protocol (IBP) [13] to effectively cache the data coming from NWS.

2.2. PMHD3D

The target application for this work, PMHD3D [12, 15], is a magnetohydrodynamics simulation developed at the University of Virginia Department of Astronomy by John F. Hawley and ported to Legion by Greg Lindhal. The code is an MPI FORTRAN stencil-based application and shares many characteristics with other stencil codes. The code is structured as a three-dimensional mesh of data, upon which the same computation is iteratively performed on each point using data from its neighbors. PMHD3D alternates between CPU-intensive computation and communication (between "slab" neighbors and for barrier synchronizations).

At startup PMHD3D reads a configuration file that specifies the problem size and the target number of processors. Since the other two dimensions are fixed in PMHD3D's three-dimensional mesh, we refer to the height of the mesh

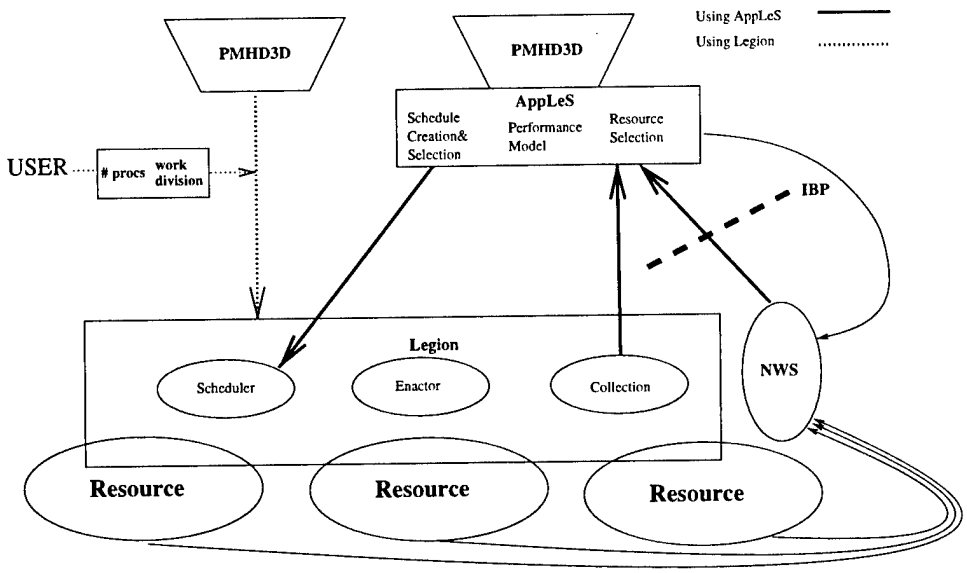


Figure 1. PMHD3D run-time scenarios with and without AppLeS.

as the *problem size*. In order to allocate work among processors in the computation the mesh is divided into horizontal *slabs* such that each processor receives a slab. For load balancing purposes each processor can be assigned a different amount of work (by dividing the work into slabs of varying height). The AppLeS scheduler determines the optimal height of each slab depending on the raw speed of the processor and on NWS forecasts of CPU load, the amount of free memory, and network conditions. AppLeS is dynamic in the sense that the data used by the scheduler is computed and collected just before execution, but once the schedule is created and implemented, the execution currently proceeds without interaction with the AppLeS.

2.3. Legion

Legion, a project at the University of Virginia, is designed to provide users with a transparent, secure, and reliable interface to resources in a wide-area system, both at the programming interface level as well as at the end-user level [9, 14]. Both the programmer and the end-user have coherent and seamless access to all the resources and services managed by Legion. Legion addresses challenging issues in Computational Grid research such as parallelism, fault-tolerance, security, autonomy, heterogeneity, legacy code management, resource management, and access transparency.

Legion provides mechanisms and facilities, leaving to the programmer the implementation of the policies to be enforced for a particular task. Following this idea, scheduling in Legion is flexible and can be tailored to suit applications with different requirements. The main Legion components involved in scheduling are the *collection*, the *enactor*, the *scheduler*, and the *hosts* which will execute the schedule [5]. The collection provides information about the available resources and the scheduler selects the resources to be used in a schedule. The schedule is then given to the enactor, which contacts the host objects involved in the schedule and attempts to execute the application. This scheme provides scheduling flexibility; for example, in case of host failures, the enactor can ask the scheduler for a new schedule and continue despite the failure, the collection can return subsets of the resources depending on the user and/or the application, or the hosts can refuse to serve a specific user.

Legion currently provides default implementations of all the objects described herein. Moreover, new objects can be developed and used rather than the default ones. Note that the PMHD3D AppLeS is developed "on top" of Legion, and uses default Legion objects. We would expect the performance improvement for such a code to conservatively bound from below that which would be achievable if the AppLeS were structured as a Legion object. We plan to eventually develop the AppLeS described here as a Legion scheduling object for a class of regular, iterative, data-parallel applications.

2.4. Network Weather Service

The Network Weather Service [17, 22] is a distributed system that periodically monitors and dynamically forecasts the performance various network and computational

resources can deliver. NWS is composed of sensors, memories and forecasters. *Sensors* measure the availability of the resource, for example CPU availability, and then record the measurement in a NWS *memory*. In response to a query, the NWS software will return a time series of measurements from any activated sensor in the system. This time series can then be passed to the NWS *forecaster* which predicts the future availability of the resource. The forecaster tests a variety of predictors and returns the result and expected error of the most accurate predictor. To obtain better performance for PMHD3D we developed a **memory sensor** that measures the available free memory of a machine. The sensor has been extended and is now part of NWS.

2.5. Interactions Among System Components

PMHD3D can directly access Legion's scheduling facilities or can use AppLeS to obtain a more performance-efficient schedule. Figure 1 shows the interactions among components in each of these scenarios. The dotted line represents the scheduling of a PMHD3D run without AppLeS facilities: the user supplies the number of processors, the processor list, and the associated problem size per processor and the rest of the scheduling process is supplied by a default scheduler within the Legion infrastructure.

When the application uses AppLeS for scheduling, the interactions among components can instead be represented by the solid lines in Figure 1. In this case the user supplies only the problem size of interest. AppLeS collects the list of available resources from the environment (via the Legion collection object or, in our case, via the Legion context space), and then queries NWS to obtain updated performance and availability predictions for the available resources. As the figure shows, AppLeS collects the NWS predictions as an IBP client: the predictions are pushed into the IBP server by a separate process.

AppLeS then creates a performance-promoting adaptive schedule and asks the Legion scheduler to execute it. The schedule is adaptive because AppLeS assigns a different amount of work to each processor depending on their predicted performance. As is suggested by the figure, the PMHD3D AppLeS is built on top of Legion facilities. A future goal is to integrate the AppLeS as an alternative scheduler in Legion for the class of regular, data-parallel, stencil applications.

3. The PMHD3D AppLeS

The general AppLeS approach is to create good schedules for an application by incorporating application specific characteristics, system characteristics, and dynamic resource performance data in scheduling decisions. The PMHD3D AppLeS draws upon the general AppLeS methodology [3] and the experience gained building an AppLeS for a structurally similar Jacobi-2D application [2].

Conceptually, the PMHD3D AppLeS can be decomposed into three components:

- a **performance model** that accurately represents application performance within the Computational Grid environment;
- a **resource selection strategy** that identifies potentially performance-efficient candidate resource sets from those that are available at run time;
- a **schedule creation and selection strategy** that creates a good schedule for each of the various candidate resource sets and then selects the most performance-efficient schedule.

The overall strategy and organization of the scheduler will be discussed here but the details of each component are reserved for the following sections.

An accurate performance model (Section 3.1) is fundamental for the development of good schedules. The performance model is used in two important ways, the first of which is to guide the creation of schedules for specific resource sets. For example, load balancing is a necessary condition developing an efficient schedule but is difficult or impossible to achieve without an estimate of the relative costs of computation on various resources. An accurate performance model is also necessary for selection of the highest performance schedule from a set of candidate schedules.

The resource selection strategy (Section 3.2) produces several orderings of available resources based on different concepts of "desirability" of resources to PMHD3D. Our definitions of desirability incorporate Legion resource discovery results, dynamic resource availability from NWS, dynamic performance forecasts from NWS, and application-specific performance data for each resource. Once complete, the ordered lists of resources are passed on to the schedule creation and selection component of the AppLeS.

The schedule creation step (Section 3.3) takes the proposed resource lists and creates a good schedule for each based on the constraints the system and application impose. System constraints are characteristics such as available memory of the resources while the application constraints are characteristics such as the amount of memory required for the application to remain in main memory. Once all schedules have been created the performance model is used to select the highest performance schedule (the one in which the execution time is expected to be the lowest).

The decomposition of the scheduling process into these disjoint steps provides an overly simplistic view of the interactions between steps. In reality the scheduling process

```
1  Rset = getResourceSet()                          \\ Available resources obtained from Legion
2  NWS_data = NWS(Rset)                             \\ NWS forecasts of resource performance
3  C = getScheduleConstraints()                     \\ Obtain scheduling constraints for simplex
4  for (balance = {0, 0.5, 1})                      \\ Select for CPU power, connectivity, both
5    S = sort(Rset, balance, maxP)                  \\ Returns list of hosts sorted by desirability
6    for (n = 2..maxP)                              \\ Searching for correct number of processors
7       sched = findSched(n, S, NWS_data, C)        \\ Use simplex to find schedule on S using C
8       while (sched is not found)                  \\ Simplex was unsolvable with S and C
9         "Schedule constraints are too restrictive"
10         relaxConstraints (C)                     \\ More schedule flexibility, more possible error
11         sched = findSched(n, S, NWS_data, C)     \\ Try to find schedule again
12      endwhile                                    \\ Found a feasible schedule
13      if (cost(sched) < best)                     \\ If best one so far keep it, else throw away
14         best = sched
15      endif
16   endfor
17 endfor
18 run(best)                                        \\ Best schedule found, run it
```

Figure 2. PMHD3D AppLeS pseudo-code.

requires more complicated interactions. To accurately represent the true interaction of the scheduling components we present a pseudo-code version of the PMHD3D AppLeS strategy in Figure 2. The steps shown in Figure 2 will become clearer in the following sections.

3.1. Performance Model

The goal of the performance model is to accurately predict the execution time of PMHD3D. Since the run-time may vary somewhat from processor to processor, we take the maximum run-time of any processor involved in the computation as the overall run-time. During every iteration each processor computes on its slab of data, communicates with its neighbors, and synchronizes with all other processors.

Formally, the running time for processor i is given by:

$$T_i = Comp_i + Comm_i + Over_i$$

where $Comp_i$, $Comm_i$ and $Over_i$ are the predicted computation time, the predicted communication time, and the estimated overhead for P_i, respectively.

Computation time is directly related to the units of work assigned to a processor (in other words the height of the slab) and to the speed of that processor. The computation time for P_i is:

$$Comp_i = \frac{x_i * BM_i}{Avail_i}$$

where x_i is the amount of work allocated to processor P_i (dynamically determined by the scheduling process), BM_i is a benchmark for the application-specific speed of P_i's processor configuration, and $Avail_i$ is a forecast of the CPU load on processor P_i (obtained from dynamic NWS forecasts). To obtain the benchmarks, we run PMHD3D on dedicated machines with various problem sizes and variable number of hosts. Execution times were proportional to problem size and are given in terms of seconds per point on each platform.

Communication time is modeled as the time required for transferring data to neighboring processors across the available network. This represents communication for all iterations and accounts for both the time to establish a connection and the time to transfer the messages. To simplify the communication model, we have not attempted to directly predict synchronization time or the time a processor waits for a communication partner. We hope instead to capture the *effect* of these communication costs in our estimate of overhead costs, which we discuss shortly. Communication time is then:

$$Comm_i = MB/(b_{i,i+1} + b_{i,i-1}) + M * (l_{i,i+1} + l_{i,i-1})$$

where MB is the total megabytes transfered, M is the number of messages transfered, and b_{ij} and l_{ij} are predictions of available bandwidth and latency from P_i to P_j, respectively. Predictions of available bandwidth and latency between pairs of processors are obtained from dynamic NWS forecasts. To provide an estimate of the number of messages transferred (M) and the megabytes transferred (MB) we examined post-execution program performance reports provided by Legion. For a variety of problem sizes and resource set sizes the number of megabytes transferred varied by less than 5% so we used an average value for all runs. Data transfer does not significantly vary with problem size because the problem size affects only the height of the grid while the decomposition is performed horizontally. Data transfer costs also do not vary with number of processors because each processor must communicate with only its neighbors, regardless of the total number of processors. Although the number of messages transferred varied more

significantly from run-to-run we also used an average value for this variable. This approximation did not adversely affect our scheduling ability in the environments we tested; in cases where communication costs are more severe a model could be developed to approximate the expected number of messages transferred.

The **overhead** factor $Over_i$ is included in the performance model to capture application and system behavior that cannot be accounted for by a simple communication/computation model. For example, a processor will likely spend time synchronizing with other processors, waiting for neighbor processors for data communication, and waiting for system delays. System overheads are associated with specifics of the hardware and Legion infrastructure such as the time required to resolve the physical location of a data object needed by the application. The overhead for PMHD3D can be estimated by:

$$Over_i = 16 - 1.5 * probSize/1000 + 0.094P^2$$

where P is the number of processors involved in the computation and $probSize$ is the height of the PMHD3D mesh.

$Over_i$ was estimated empirically using data from 106 individual application executions with problem sizes varying from 1000 to 6000 and with resource set sizes varying between 4 and 26. To determine the effect of the number of processors on overhead runs, runs were grouped by problem size and the corresponding execution times plotted against number of processors. For each set of runs performed with the same problem size, a quadratic fit was performed on the difference between the actual execution time and the predicted execution time (without the overhead factor). The quadratic factor varied between 0.090 and 0.096 with a mean of 0.094 (standard deviation of 0.0022). To determine the effect of problem size on overhead we used the same runs but did a linear datafit on the predicted/actual execution time difference with problem size.

3.2. Resource Selection

Resource selection is the process of selecting a set of target resources (processors in this case) that will be performance-efficient. Finding the optimal set of resources requires comparing all possible schedules on all possible subsets of the resource pool - clearly an inefficient process as the resource pool becomes large. Instead, we create several ordered lists of resources by employing a heuristic to sort candidate resources in terms of several definitions of *resource desirability*. Resource desirability is based on how resource characteristics such as computational speed and network connectivity will affect the performance of PMHD3D.

The resource selection process begins by querying Legion to discover the available set of resources. Effective evaluation of the desirability of each resource requires application-specific performance information as well as dynamic resource performance information. As of this writing, Legion collection objects report available resources and their static configurations but do not provide up-to-date dynamic information on availability, load, or connectivity. Accordingly, the list of available resources reported by Legion is used to query NWS for dynamic forecasts of resource availability, CPU load, and free memory for each host and of latency and bandwidth between all pairs of hosts. To obtain the computational cost per unit of the PMHD3D grid on each type of resource we used the benchmarking method described in Section 3.1.

Once the available resource lists and the dynamic system characteristics are collected, the list can be ordered in terms of desirability. We use three definitions of desirability of a resource: desirability based on connectivity, desirability based on computational power, and desirability based equally on the two characteristics. *Connectivity* is approximated by computing the latency and bandwidth between the resource in question and all other resources in the resource pool: as a metric we calculate the amount of time (seconds) it would take for the resource in question to exchange a packet of size 1 byte to and from every other host. *Computational power* is measured by the time (seconds) it would take the host to compute 1 point for 1 iteration based on the NWS predictions and the benchmarks we discussed earlier. The *balanced* strategy orders the resources based on an average of computational power and connectivity.

The resource set is sorted into 3 resource lists using the 3 notions of resource desirability. We then create subsets of the lists by selecting the n most desirable hosts from each list where $n = 2...maxP$ and n is even. We select multiple subsets from each list because it is often impossible to know the optimal number of hosts a priori. Once the subsets have been created the resulting group of proposed resource sets are passed on to the schedule creation step described in the next section. Although the approach described here is not guaranteed to find the optimal resource set, the methodology provides a scalable and performance-efficient approach to resource selection.

3.3. Schedule Creation and Selection

For each of the proposed resource sets, a schedule is developed. Essentially, schedule development on a given resource set for PMHD3D reduces to finding a work allocation that provides good time balancing. As in Section 3.1 work allocation is represented by x_i and is the height of the slab given to processor P_i.

One of the most important characteristics for any solution to this problem is time balancing: all processors should finish at the same time. Using the notation from Section 3.1,

$T_i = T_{i+1}$, $i \in \{1 \ldots (n-1)\}$ and, since all of the work must be allocated, we also have $\sum_i x_i = probSize$. Taken together we have n equations in n unknowns and the problem can be solved with a basic linear solver. This approach was successful for the Jacobi-2D AppLeS [2] but is not powerful enough to incorporate several additional constraints required to develop good schedules for PMHD3D.

One of the important constraints for PMHD3D performance is the amount of memory available for the application. There is a limit to the size of problem that can be placed on a machine because if the computation spills out of memory, performance can drop by two orders of magnitude. To quantify this constraint a benchmark for application memory usage must be obtained by observing memory usage for varying problem sizes on each type of resource. Formally, this constraint becomes:

$$BMmem_i * x_i < MemAvail_i$$

where $MemAvail_i$ is the available memory for processor i (provided by the NWS memory sensor) and $BMmem_i$ is the memory benchmark (megabytes/unit) recorded for processor i's architecture.

We formalize the work allocation constraints as a *Linear Programming* problem (from now on simply LP), solvable with the simplex method [6]. In short, LP solves the problem of finding an extreme (maximum or minimum) of a function $f(x_1, x_2, \ldots, x_n)$ where the unknowns have to satisfy a set of constraints $g(x_1, x_2, \ldots, x_n) \geq b$ and both the *objective function* and the constraints are linear. The simplex is a well-known method used to solve LP problems. The simplex formulation requires that constraints are expressed in *standard form*; that is the constraints must be expressed as equalities and each variable is assigned a *non-negativity* sign restriction. There is a simple procedure that can be used to transform LP problems into a standard form equivalent.

We modified the time balancing equations to provide some flexibility for the constraints specification: expected execution time for any processor in the computation must fall within a small percentage of the expected total running time. This flexibility is beneficial, especially as additional constraints such as memory limits are incorporated into the problem formulation. The constraints are initially very rigid but can be relaxed in cases where no solution can be found given the initial constraints. The time balancing equations and the application memory requirements form the application constraints on which the simplex has to operate. The simplex formulation also requires specification of an *objective function* where the goal of the solver is to maximize the objective function while satisfying the simplex constraints. We use $\sum_i x_i$ as the objective function and search for a solution where all work is allocated.

For each of the proposed resource sets the simplex is used to create the best schedule possible for that resource set. We use a library [16] which provides a fast and easy to use implementation of the simplex. There are several benefits of using linear programming and the simplex method to create a good schedule:

- Linear programming is well known and commonly used so that fast and reliable algorithms are readily available.
- Once the constraints are formalized as a linear programming problem, adding additional constraints is trivial. For example, the FORTRAN compiler used to compile PMHD3D enforced a limit on the maximum size of arrays, therefore limiting the maximum units of work that could be allocated to any processor. This constraint was easily added to the problem formalization.
- The linear programming problem can be extended to give integer solutions, although the problem then becomes much more difficult. Currently the solver computes real values for work allocation and we redistribute the fractional work portions. In some problems a linear solution may be required for additional accuracy.
- In the case that a solution cannot be found, the simplex method provides important feedback. For this application, the simplex could not find a solution if the constraints were too restrictive. In this case the simplex is reiterated with successively relaxed constraints until a solution can be reached.

Once the proposed schedules are identified, schedule selection is surprisingly simple. The performance model is used to evaluate the expected execution time of each proposed schedule, and the schedule with the lowest estimated execution time is selected and implemented.

4. Results

The PMHD3D AppLeS has been implemented and we present results to investigate the usefulness of the methodology. The goals of these experiments were to:

- Evaluate the accuracy of our performance prediction model.
- Evaluate the ability of the PMHD3D AppLeS to promote application performance in a multi-user Legion environment.

The previous sections stressed the importance of the performance model for effective scheduling. In Section 4.2 we explain in detail results demonstrating the accuracy of the

performance model. In Section 4.3 we present evidence that the scheduling methodology and implementation are effective in practice. Before discussing these results we first outline our experimental design.

4.1. Experimental Design

To evaluate the PMHD3D AppLeS, we conducted experiments on the University of Virginia Centurion Cluster, a large cluster of machines maintained by the Legion team (see [4] for more information on the cluster). The Centurion Cluster is continuously upgraded for new Legion version releases; during the 3-month period of the experiments, we used Legion versions 1.5 through 1.6.1. The cluster itself is composed of 128 Alphas and 128 Dual-Pentium II PCs; 12 fast Ethernet switches and a gigaswitch connect the whole cluster. Although we employed both Alphas and Pentiums during the development and initial testing process, we had multiple difficulties with Alpha Linux kernel instabilities and a faulty network driver which made our data for the Alphas machines unreliable. The results presented here are based only on the 400 MHz Dual Pentium II machines. We didn't employ the second processor on the Dual Pentium: therefore when we talk about host or machine we consider the machines to be uniprocessors. It is worth noting that many users only use one processor per node so that even a computationally intensive user will not affect CPU availability as much as might be expected. However, the two processors on each Dual Pentium machine utilize the same memory, sometimes leading to performance degradation due to overloaded memory systems. Inclusion of memory constraints in the performance model helped the AppLeS scheduler avoid overloaded memory systems.

We restricted our experiments to 34 machines for practical reasons: the dynamic information collected from NWS includes a large amount of data, even for a relatively small cluster. Limiting the resource pool did not impact investigations of application performance or schedule efficiency because, as will become clear, the parallelism available in PMHD3D for the problem sizes studied here is well below the 34 machine limit. As explained in Section 2.5 we used an IBP server running at all times at UCSD, while AppLeS acted as an IBP client retrieving the forecasts. This setup allowed us to obtain updated predictions for a large number of resources in a reasonable amount of time. On average it took less than 4 seconds to retrieve the data, with a minimum of 2.5 seconds and a maximum of 8.5 seconds.

To test the performance of PMHD3D under a variety of conditions, experiments were typically performed with maximum resource set sizes (from now on called *resource pool* or simply *pool*) of $4, 6...26$ and problem sizes of $1000, 2000...6000$. Problem size is the height of the data grid used by PMHD3D. The *pool* is the maximum number of machines the scheduler is allowed to employ. We test varying pool sizes to simulate conditions under which a user may be limited to a certain number of resources by cost or access considerations. Although our overall resource pool contains 34 machines in total, the maximum pool size we simulate is only 26. This choice was practical: we frequently found unavailable or inaccessible machines in our overall resource pool and so were never able to access all 34 machines at one time. Note also that the scheduler may determine that utilizing the entire pool is not the most performance efficient choice. In this case the pool is larger than the number of *target resources*.

The experiments presented in Section 4.2 were conducted under unloaded conditions while those presented in Section 4.3 were conducted under loaded conditions. The ambient load present during most of our loaded runs consisted of heavy use of some machines and light use of others. In order to investigate application performance we report performance results based on application execution time. However, there is a cost associated with using AppLeS to develop a schedule. We analyzed 43 runs in detail and the dominant scheduling cost is associated with querying the Legion Collection and the Legion context space. The time required to access NWS and IBP is on average less than 4 seconds. Once the system and performance information has been collected, the AppLeS required on average roughly 1 second to order the resources, create schedules, and select the best schedule.

4.2. Performance Model Validation

The performance model is the basis for determining a good work allocation and, more importantly, provides the basis for selecting a final schedule among those that have been considered. We tested model accuracy for a variety of problem sizes and target resource sets (see Figure 3). For the 62 runs shown in this figure the model accurately predicts execution time within *1.5%*, on average. The performance model consistently achieved this level of accuracy for other runs taken under similar conditions. Notice that as the problem size becomes larger, the smallest pool that we test also increases (i.e. the smallest pool for a problem size of 2000 is of size 4 while for a problem size of 6000 it is 12). This experimental setup was required by a limit in the g77 FORTRAN compiler we employed: no more than 507 work units could be allocated to any one processor during the computation.

Figure 3 demonstrates the importance of selecting an appropriate number of target resources for PMHD3D. For example, for a problem size of 1000 the minimal execution time is achieved when the application is run on 10 processors. If fewer processors are used, the amount of work per processor is high and the overall execution time is higher.

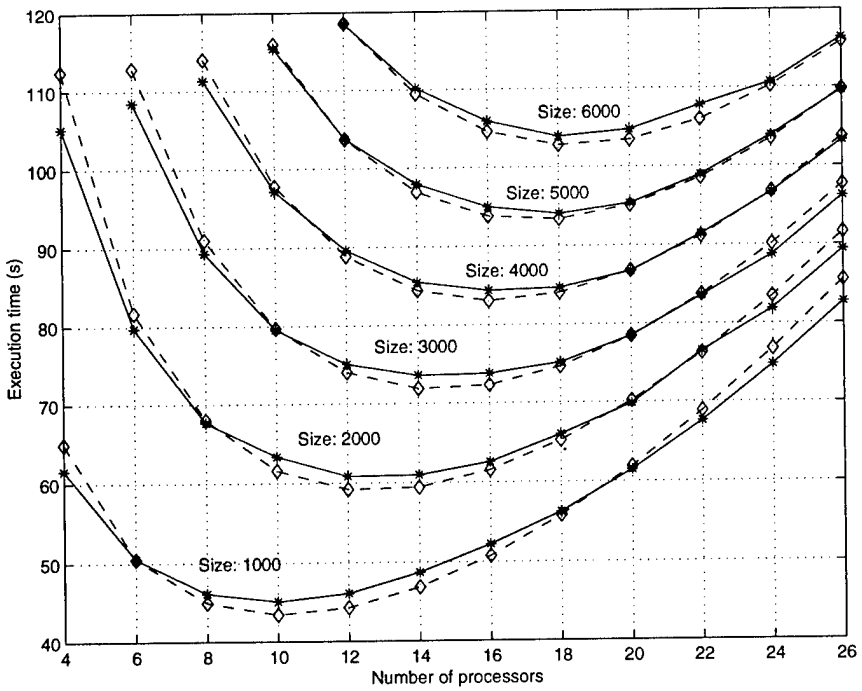


Figure 3. Model predictions (*dashed lines*) and observed execution time (*solid lines*) for a variety of problem sizes and pool sizes.

Table 1. Number of resources to target for various problem sizes under unloaded conditions. Optimal is the best choice, range indicates close to optimal choices.

Size	1000	2000	3000	4000	5000	6000
Hosts	10	12	14	16	18	18
Range	8-12	12-14	14-16	14-18	16-18	18-20

If more processors are used, the added communication and system overheads cannot be offset by the advantage of the additional computational power. Significantly, the performance model accurately tracks the *knee* (i.e. inflection point) in the curve and is thus capable of predicting the correct number of target resources, at least under these conditions. We report the optimal number of target resources for all problem sizes tested in Table 1. As will be obvious in Section 4.3, the optimal number of processors may vary with resource performance and dynamic system conditions as well as with problem size.

Figure 4 demonstrates the scheduling advantage of accurately predicting the correct number of processors to target. In these experiments the PMHD3D AppLeS was allowed to select any number of processors up to the maximum pool size. The PMHD3D AppLeS selects the maximum number of resources for each resource pool up to and including a size of 18. For resource pools of size 20 and larger the optimal number of hosts is 18 and the PMHD3D AppLeS correctly selects only 18 hosts.

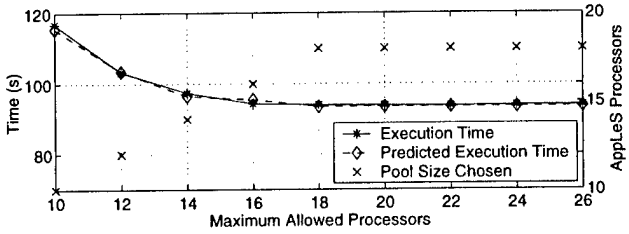


Figure 4. PMHD3D AppLeS predicted and actual execution times for a problem size of 5000.

4.3. Performance Results

Once we verified that the performance model is accurate in a predictable environment (i.e. where resources are dedicated), we turned our attention to considering the

performance of the AppLeS in a more dynamic, unpredictable, multi-user environment. We begin by investigating the ability of PMHD3D AppLeS to compare available resources and select *desirable* hosts (computationally fast, well-connected, or both). To provide a comparison point we test the performance of another available scheduler, namely the default Legion scheduler. We conducted experiments in *runs*, namely back-to-back PMHD3D executions using the same resource pool and the same problem size but utilizing the PMHD3D AppLeS scheduler first and the default Legion scheduler second.

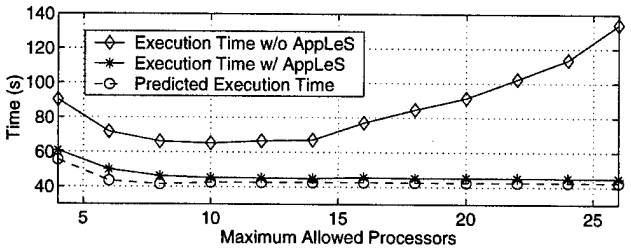


Figure 5. PMHD3D performance attained with and without the AppLeS scheduler for a problem size of 1000.

In Fig. 5 we show a series of runs comparing the two schedulers for a problem size of 1000. Clearly, the PMHD3D AppLeS provides a performance advantage for all resource set sizes tested. However, it is notable that the two execution time curves follow the same trend only when the resource pool is in the range of 4-12 hosts. When more resources are added to the pool the execution time achieved with the PMHD3D AppLeS remains constant while the default Legion scheduler execution time diverges. The default Legion scheduler allocates *all* available resources, a less than optimal strategy for PMHD3D. In Table 2 we report the typical number of processors selected by AppLeS for different problem sizes and resource set sizes.

For pool sizes of $4 - 12$ performance achieved via the PMHD3D AppLeS is consistently $20 - 25$ seconds lower than that achieved via the default scheduler. In this range of pool sizes, the PMHD3D AppLeS selects the maximum number of hosts available and so uses the *same number* of resources as the default Legion scheduler. The performance advantage is achieved by selecting "desirable" resources, i.e. resources that are computationally fast and/or well-connected. Figure 6 illustrates the load of all available machines just before scheduling occurred for the 18-processor run shown in Figure 5. Clearly, the PMHD3D AppLeS selects lightly loaded hosts (i.e. those hosts with high availability) while the default scheduler selects several loaded hosts. It is the load on these selected machines that causes

Table 2. Hosts chosen by PMHD3D AppLeS. The Legion default scheduler always selects the maximum number of hosts.

	Problem Size				
Max Hosts	1000	2000	4000	5000	6000
4	4	4			
6	6	6			
8	8	8	8		
10	10	10	10	10	
12	10	12	12	12	12
14	10	12	14	14	14
16	10	12	14	16	16
18	10	12	16	16	18
20	10	12	14	18	20
22	10	12	14	18	20
24	10	14	14	18	18
26	10	14	14	18	18

a performance disadvantage for the default scheduler. In a more heterogeneous network environment the connectivity of the hosts would also play an important role in host selection and resulting performance.

We obtained 83 runs comparing the default Legion scheduler to the PMHD3D AppLeS for a variety of problem sizes (1000-6000) and pool sizes (4-26). Figure 7 shows a histogram of the percent improvement the PMHD3D AppLeS achieved over the default Legion scheduler for the 83 runs (the average improvement was 30%).

Note that in a few runs there was little or no advantage to using the PMHD3D AppLeS. In these cases the processors were essentially idle and the pool size was below the optimal number so that the schedulers selected the same number of processors. In one run the PMHD3D AppLeS-determined schedule was considerably slower than that determined by the default Legion scheduler. In this case the scheduler created a schedule based on incorrect system information: NWS forecasts of CPU availability were unable to a predict a sudden change in load on several machines and the resulting schedule was poorly load balanced.

The Legion default scheduler was designed to provide general scheduling services, not the specialized services we include in the PMHD3D AppLeS. It is therefore not surprising that the AppLeS is better able to promote application performance. In fact, the PMHD3D AppLeS could be developed as a Legion object for scheduling regular, iterative, data-parallel computations, and this is a focus of future work. Using the PMHD3D AppLeS and the Legion default scheduling strategy as extremes, we wanted to explore a third alternative for scheduling – that of what a "smart user" might do: In a typical user scenario for a cluster of machines a user will have access to a large number of machines and will typically do a back-of-the-envelope static

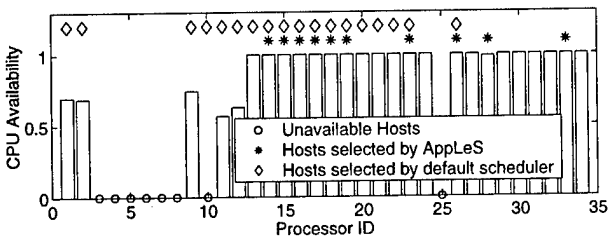


Figure 6. A snapshot of CPU availability taken during scheduling for the 18-processor run shown in Figure 5.

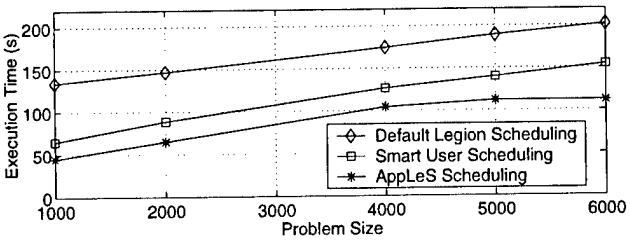


Figure 8. Performance obtained by three schedulers when each was given access to at most 26 processors.

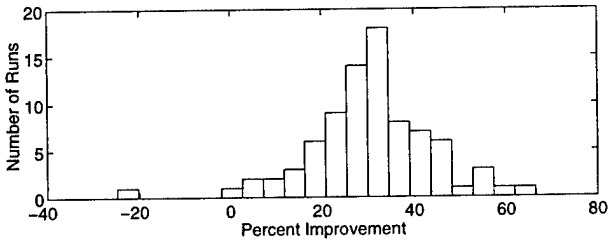


Figure 7. Range of performance improvement obtained by PMHD3D AppLeS.

calculation to determine an appropriate number of target resources given the granularity of the application. Although a user may correctly determine the number of hosts to target, accurate information on resource load and availability will be difficult or impossible to obtain and interpret prior to or at compile-time.

To simulate this user scenario, we developed a third scheduling method called the **smart user**. The *smart user* selects an appropriate number of hosts but does not select hosts based on desirability. Experiments were performed for problem sizes ranging from 1000 to 6000 with a pool size of 26 hosts. Figure 8 shows the performance obtained by the PMHD3D AppLeS, the default Legion scheduler, and that obtained by the *smart user*. In these experiments, the PMHD3D AppLeS provides a significant performance advantage over both alternatives.

5. Related Work

The PMHD3D AppLeS is an adaptation and extension of previous work targeting the structurally similar Jacobi-2D application ([2],[3]). Jacobi-2D is a data-parallel, stencil-based iterative code, as is PMHD3D. Both applications allow non-uniform work distribution, however Jacobi-2D employs strip decomposition (using strip widths) for its 2-dimensional grid while PMHD3D employs slab decomposition (using slab height) for its 3-dimensional grid. While the applications are structurally similar, PMHD3D required tighter constraints on memory availability and a more complex performance model. Additionally, PMHD3D was targeted for a much larger resource set (34 machines vs. 8). The availability of a larger resource pool for this work motivated the introduction of the quadratic overhead term in the PMHD3D performance model. Previous AppLeS work has not included the additional overhead of using extra machines in scheduling decisions.

As part of our previous work, we developed an AppLeS for Complib and the Mentat distributed programming environment. Complib implements a genetic sequencing algorithm for libraries of sequences. It is particularly difficult to schedule because of its highly data dependent execution profile. The implementation of Complib we chose was for Mentat [8] which is an early prototype of the Legion Grid software infrastructure. By combining a fixed initial distribution strategy (based on a combination of application characteristics and NWS forecasts) with a shared work-queue distribution strategy, the Complib AppLeS was able to achieve large performance improvements in different Grid settings [20]. In addition to AppLeS for Legion/Mentat applications, we have developed AppLeS for a variety of Grid infrastructures and applications [19, 21, 7].

In [10], the authors describe a scheduler targeting data parallel "stencil" applications that use the Mentat programming system. They specifically examine Gaussian elimination using a master/slave work-distribution methodology. While it is difficult to compare the performance of each system, their approach differs from AppLeS in that it requires more extensive modification of the application and it does not incorporate dynamic information.

6. New Directions

An ultimate goal is to offer the PMHD3D AppLeS agent within the Legion framework as a default scheduler for iterative, regular, stencil-based distributed applications. In particular, the scheduler's performance model is flexible enough to incorporate the requirements and constraints of other stencil applications and the characteristics of other platforms. To use this model for other appropriate applications, good predictions of megabytes transferred, number of messages initiated, overhead factor, benchmarks for program CPU and memory utilization over the different target architectures, as well as access to dynamic system information from NWS or a similar system would be required. Once obtained, these characteristics are used as inputs to the model without changing the model structure.

Portability and heterogeneity are also important. The AppLeS itself is written in C and Perl and has been compiled successfully and executed on various architectures and systems (Pentium, Alpha, Linux and Solaris). Initial results indicate that the scheduler can be used effectively on different target environments without changes to the structure of the performance model. For example, we used *mpich* on a local cluster for initial development and debugging. The schedule worked well with only the previously described changes in model input parameters.

Acknowledgements

The authors would like to express their gratitude to the reviewers for their comments and suggestions. The insight and focus provided by their comments improved the paper greatly. We thank the NWS team and Jim Hayes in particular for sharing their NWS expertise with us. We also thank the Legion team and Norman Francis Beekwilder in particular for sharing their Legion expertise with us.

References

[1] Application-Level Scheduling. `http://apples.ucsd.edu`.

[2] F. Berman and R. Wolski. Scheduling from the perspective of the application. In *Proceedings of High-Performance Distributed Computing Conference*, 1996.

[3] F. Berman, R. Wolski, S. Figueira, J. Schopf, and G. Shao. Application level scheduling on distributed heterogeneous networks. In *Proceedings of Supercomputing*, 1996.

[4] Virginia Centurion Cluster. `http://legion.virginia.edu/centurion/facts`.

[5] S. J. Chapin, D. Katramatos, J. Karpovich, and A. Grimshaw. Resource management in Legion. In *Journal of Future Generation Computing Systems*, volume 15, 1999. page583-594 vol 15.

[6] D. Dantzig. Programming of interdependent activities, ii, mathematical model. *Activity Analysis of Production and Allocation*, July-October 1949.

[7] I. Foster and C. Kesselman, editors. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers, July 1998.

[8] A. Grimshaw. Easy-to-use object-oriented parallel programming with mentat. *IEEE Computer*, May 1993.

[9] A. Grimshaw, A. Ferrari, F. Knabe, and M. Humphrey. Wide-area computing: Resource sharing on a large scale. In *IEEE Computer 32(5)*, volume 32(5), May 1999. page 29-37.

[10] A. Grimshaw, J. Weissman, E. West, and E. J. Loyot. Metasystems: an approach combining parallel processing and heterogeneous distributed computer systems. *Journal of Parallel and Distributed Computing*, June 1994.

[11] A. Grimshaw and W. Wulf. Legion–a view from 50,000 feet. In *Proceedings of High-Performance Distributed Computing Conference*, 1996.

[12] John Hawley. `http://www.astro.virginia.edu/~jh8h`.

[13] Internet Backplane Protocol. `http://www.cs.utk.edu/~elwasif/IBP`.

[14] Legion. `http://legion.virginia.edu`.

[15] G. Lindhal. Private communication.

[16] W. Naylor. wnlib. ftp://ftp.rahul.net/pub/spiketech/softlib/wnlib/wnlib.tar.Z.

[17] Network Weather Service. `http://nws.npaci.edu/`.

[18] ScaLAPACK. `http://www.netlib.org/scalapack/scalapack_home.html`.

[19] S. Smallen, W. Cirne, J. Frey, F. Berman, R. Wolski, M.-H. Su, C. Kesselman, S. Young, and M. Ellisman. Combining workstations and supercomputers to support grid applications: The parallel tomography experience. *Heterogeneous Computing Workshop*, May 2000. To appear.

[20] N. Spring and R. Wolski. Application level scheduling: Gene sequence library comparison. In *Proceedings of ACM International Conference on Supercomputing 1998*, July 1998.

[21] A. Su, F. Berman, R. Wolski, and M. M. Strout. Using AppLeS to schedule simple SARA on the computational grid. *International Journal of High Performance Computing Applications*, 13(3):253–262, 1999.

[22] R. Wolski. Dynamically forecasting network performance using the network weather service. *Cluster Computing*, 1998.

Holly Dail is currently a M.S. student in the Department of Computer Science and Engineering at the University of California San Diego. She received a B.S. in Physics and a B.S. in Oceanography from the University of Washington in 1996. Her current research interests focus on achieving application performance in Computational Grid environments.

Graziano Obertelli is currently an Analyst Programmer in the Department of Computer Science and Engineering

at the University of California, San Diego. He received his Laurea in Computer Science at Univerità degli Studi, Milano.

Francine Berman is a Professor of Computer Science and Engineering at the University of California, San Diego. She is also a Senior Fellow at the San Diego Supercomputer Center, Fellow of the ACM, and founder of the Parallel Computation Laboratory at UCSD. Her research interests over the last two decades have focused on parallel and distributed computation, and in particular the areas of programming environments, tools, and models that support high-performance computing. She received her B.A. from the University of California, Los Angeles, her M.S. and Ph.D. from the University of Washington.

Rich Wolski is an Assistant Professor in the Department of Computer Science at the University Tennessee, Knoxville and a partner in the National Partnership for Advanced Computational Infrastructure. His research interests include parallel and distributed computing, on-line performance analysis techniques and software, compiler runtime system, and dynamic scheduling. He received his B.S. from the California Polytechnic University, San Luis Obispo and his M.S. and Ph.D. from the University of California at Davis/Livermore Campus.

Andrew S. Grimshaw is an Associate Professor of Computer Science and director of the Institute of Parallel Computation at the University of Virginia. His research interests include high-performance parallel computing, heterogeneous parallel computing, compilers for parallel systems, operating systems, and high-performance parallel I/O. He is the chief designer and architect of Mentat and Legion. Grimshaw received his M.S. and Ph.D. from the University of Illinois at Urbana-Champaign in 1986 and 1988 respectively.

Fast and Effective Task Scheduling in Heterogeneous Systems

Andrei Rădulescu Arjan J.C. van Gemund
Faculty of Information Technology and Systems
Delft University of Technology
P.O.Box 5031, 2600 GA Delft, The Netherlands
{A.Radulescu,A.J.C.vanGemund}@its.tudelft.nl

Abstract

Recently, we presented two very low-cost approaches to compile-time list scheduling where the tasks' priorities are computed statically or dynamically, respectively. For homogeneous systems, these two algorithms, called FCP and FLB, have shown to yield a performance equivalent to other much more costly algorithms such as MCP and ETF. In this paper we present modified versions of FCP and FLB targeted to heterogeneous systems. We show that the modified versions yield a good overall performance, which is generally comparable to algorithms specifically designed for heterogeneous systems, such as HEFT or ERT. There are a few cases, mainly for irregular problems and large processor speed variance, where FCP and FLB's performance drops down to 32% and 63%, respectively. Considering the good overall performance and their very low cost however, FCP and FLB are interesting options for scheduling very large problems on heterogeneous systems.

Keywords: compile-time task scheduling, list scheduling, low-cost, heterogeneous systems

1 Introduction

Heterogeneous systems have recently become widely used as a cheap way of obtaining a parallel system. Clusters of workstations connected by high-speed networks, or simply the Internet are common examples of heterogeneous systems. However, in order to obtain high-performance from such a system, both compile-time and runtime support is necessary, in which scheduling the application to the parallel system is a crucial factor. The problem, known as task scheduling, has been shown to be NP-complete [3].

The general problem of task scheduling has been extensively studied, mainly for homogeneous systems. Various heuristics have been proposed, including list algorithms [4, 11, 12, 13, 20], multi-step algorithms [14, 15, 22], duplication based algorithms [7, 2, 1], genetic algorithms [18], algorithms using local search [21], bin packing [19], or graph decomposition [6]. Within all these approaches, list scheduling has been shown to have a good cost-performance trade-off, as considering its low cost, the performance is still very good [8, 13, 12]. The low-cost is a key issue for large problems, in which even a $O(V^2)$ algorithm, where V is the number of tasks, may have a prohibitive cost.

Task scheduling has also been studied in the specific context of heterogeneous systems ([5, 9, 10, 16, 17]). It has been shown that minimizing the tasks' completion time throughout the schedule is preferable to minimizing the tasks' start time [10, 17]. With respect to list scheduling algorithms, one can note that most of them can be easily modified to meet the task's completion time minimization criterion, and thus obtain good performance also in the heterogeneous case (e.g., HEFT [17] and ERT [9] are the versions using the tasks' completion time as the task priority of MCP [20] and ETF [4], respectively). However, two very low-cost list scheduling algorithms that we

0-7695-0556-2/00 $10.00 © 2000 IEEE

proposed recently, namely FCP (Fast Critical Path) [13] and FLB (Fast Load Balancing) [12], cannot be modified in such an easy way without sacrificing their competitively low cost.

In this paper we present the modifications required to obtain a good performance from FCP and FLB in heterogeneous systems. We show that the modified versions of FCP and FLB yield a good overall performance, which is generally comparable to algorithms specifically designed for heterogeneous systems, such as HEFT (Heterogeneous Earliest-Finish-Time) [17] and ERT (Earliest Task First) [9]. There are a few cases, mainly for irregular problems and wide processor speed ranges, in which FCP and FLB's performance drops down to 32% and 63%, respectively. Considering their very low cost and reasonably good performance, we believe that FCP and FLB are interesting options for task scheduling in heterogeneous systems, especially for large problems where scheduling time would otherwise be prohibitive.

This paper is organized as follows: The next two sections briefly describe the scheduling problem, and the FCP and FLB algorithms, respectively. In Section 4 we study their performance for heterogeneous systems. Section 5 concludes the paper.

2 Preliminaries

The task scheduling algorithm input is a directed acyclic graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, that models a parallel program, where $\mathcal{V}$ is a set of V nodes and $\mathcal{E}$ is a set of E edges. A node in the DAG represents a task, containing instructions that execute sequentially without preemption. Each task is assumed to have a *computation cost*. The edges correspond to task dependencies (communication messages or precedence constraints) and have a *communication cost*. The *communication-to-computation ratio* (CCR) of a parallel program is defined as the ratio between its average communication and computation costs. If two tasks are scheduled to the same processor, the communication cost between them is assumed to be zero. The task graph *width* (W) is defined as the maximum number of tasks that are not connected through a path.

A task with no input edges is called an *entry* task, while a task with no output edges is called an *exit* task. The task's *bottom level* is defined as the longest path from the current task to any exit task, where the path length is the sum of the computation and communication costs of the tasks and edges belonging to the path. A task is said to be *ready* if all its parents have finished their execution. Note that at any given time the number of ready tasks never exceeds W. A task can start its execution only after all its messages have been received.

As a distributed system we assume a set $\mathcal{P}$ of P processors connected in a clique topology in which interprocessor communication is assumed to perform without contention. The processors' computing speeds differ and are represented as fractions of the slowest processor speed. We assume that the task execution time is proportional with the speed of the processor it is executed on, and consists of the computation cost multiplied by the processor speed.

In our algorithms, an important concept is that of the *enabling processor* of a ready task t, $EP(t)$, which is the processor from which the last message arrives. Given a partial schedule and a ready task t, the task is said to be of *type EP* if its last message arrival time is greater than the ready time of its enabling processor and of *type non-EP* otherwise. Thus, an EP type task starts the earliest on its enabling processor.

3 The Algorithms

List scheduling algorithms use two approaches to schedule tasks. The first category is the *static* list scheduling algorithms (e.g., MCP [20], DPS [11], HEFT [17], FCP [13]) that schedule the tasks in the order of their previously computed priorities. A task is usually scheduled on the processor that gives the earliest start time for the given task. Thus, at each scheduling step, first the task is selected and afterwards its destination processor.

The second approach is *dynamic* list scheduling (e.g. ETF [4], ERT [9], FLB [12]). In this case, the tasks do not have a precomputed priority. At each scheduling step, each ready task is tentatively scheduled to each processor, and the best <task, processor> pair is selected (e.g., the ready task that starts the earliest on the processor where this earliest start time is obtained for ETF, or the ready task that finishes the earliest on the processor where this earliest finish time is obtained for ERT). Thus, at each step both the task and its destination processor are

selected at the same time.

Both static and dynamic approaches of list scheduling have their advantages and drawbacks in terms of the schedule quality they produce. Static approaches are more suited for communication-intensive and irregular problems, where selecting important tasks first is more crucial. Dynamic approaches are more suited for computation-intensive applications with a high degree of parallelism, because these algorithms focus on obtaining a good processor utilization.

FCP (Fast Critical Path) [13] and FLB (Fast Load Balancing) [12] significantly reduce the cost of the static and dynamic list scheduling approaches, respectively. In the next two sections, we describe both algorithms and we outline the differences between them and previous list scheduling algorithms.

3.1 FCP

Static list scheduling algorithms have three important steps: (a) *task priorities computation*, that takes at least $O(E + V)$ time, since the whole task graph has to be traversed, (b) *task selection* according to their priorities, that takes $O(V \log W)$ time, and (c) processor selection, that selects the "best" processor for the previously selected task, usually the processor where the current task starts/finishes the earliest. Processor selection takes $O((E + V)P)$ time, since each task is tentatively scheduled to each processor. Thus, the highest complexity steps are the task and processor selection steps, which determine the $O(V \log (W)+(E+V)P)$ time complexity of the static list scheduling algorithms

In FCP, the processor selection complexity is significantly reduced by restricting the choice for the destination processor from all processors to only *two* processors: (a) the task's enabling processor, or (b) the processor which becomes idle the earliest. In [13] we prove that the start time of a given task is minimized by selecting one of these two destination processors. The proof is based on the fact that the start time of a task t on a candidate processor p is defined as the maximum between (a) the time the last message to t arrives, and (b) the time p becomes idle. As the above-mentioned processors minimize the two components of the task's start time, respectively, it follows that one of the two processors minimizes the task's start time. Consequently, the algorithm's performance is not affected, while the time complexity is drastically reduced from $O((E+V)P)$ to $O(V \log (P) + E)$.

The task selection complexity can be reduced by maintaining only a *constant size* sorted list of ready tasks. Thus, we sort as many tasks as they fit in the fixed size sorted list, while the others are stored in an unsorted FIFO list which has an $O(1)$ access time. The time complexity of sorting tasks using a list of size H decreases to $O(V \log H)$ as all the tasks are enqueued and dequeued in the sorted list only once. We have found that for FCP, which uses bottom level as task priority, a size of P is required to achieve a performance comparable to the original list scheduling algorithm (see Section 4). A sorted list size of P results in a task sorting complexity of $O(V \log P)$.

Using the described techniques for task sorting and processor selection the total time complexity of FCP ($O(V \log (P) + E)$) is clearly a significant improvement over the time complexity of typical list scheduling approaches with statically computed priority.

3.2 FLB

In FLB, at each iteration of the algorithm, the ready task that can start the earliest is scheduled to the processor on which that start time is achieved. Note that FLB uses the same task selection criterion as in ETF. In contrast to ETF however, the preferred task and its destination processor are identified in $O(\log (W) + \log (P))$ time instead of $O(WP)$.

To select the earliest starting task, pairs of a ready task and the processor on which the task starts the earliest need to be considered. As shown earlier, in order to obtain the earliest start time of a ready task on a partial schedule, the given task must be scheduled either (a) to the task's enabling processor, or (b) to the processor becoming idle the earliest.

Given a partial schedule, there are only two pairs task-processor that can achieve the minimum start time for a task: (a) the EP type task t with the minimum estimated start time $EST(t, EP(t))$ on its enabling processor, and (b) the non-EP type task t' with the minimum last message arrival time $LMT(t')$ on the processor becoming idle the earliest. The first case minimizes the earliest start time of the EP type tasks, while the second case minimizes the earliest start time of the non-EP type tasks. If in both cases

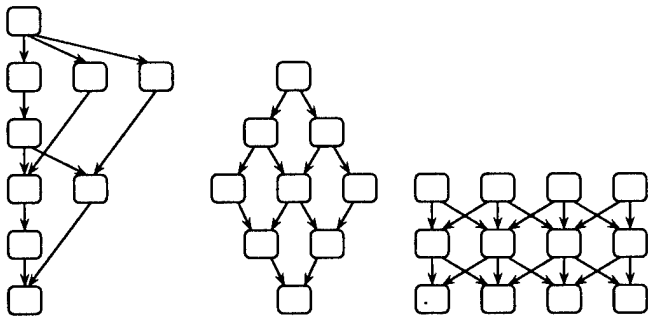


Figure 1: Miniature task graphs

the same earliest start time is obtained, the non-EP type task is preferred, because the communication caused by the messages sent from the task's predecessors are already overlapped with the previous computation. Considering the two cases discussed above guarantees that the ready task with the earliest start time will be identified. A formal proof is given in [12].

To reduce the complexity even further, the same scheme as in FCP can be used. Instead of maintaining all EP and non-EP tasks sorted, only a fixed number of tasks are stored sorted, while the other are stored in FIFO order. The FLB's complexity is reduced to $O(V \log(P) + E)$, while the performance is maintained at a level comparable to using the fully sorted task lists (see Section 4).

3.3 The Modifications

As mentioned earlier, task scheduling algorithms for heterogeneous systems perform better when they sort tasks by their finish time rather than start time. The reason is that sorting by finish time implicitly takes into consideration processor speeds. However, in order to maintain their very low complexity, FCP and FLB must sort the tasks according to their start time. As a consequence, the processor speed is not considered when scheduling a non-EP task, but only the time the processors becomes idle.

To overcome this deficiency, we change the priority criterion for processors for both FCP and FLB. Instead of using the time the processor becomes idle the earliest as a priority, we now use the *sum* of the processor idle time and the mean task execution time. Using this priority scheme, we are now able to incorporate the processor speed when selecting the processor for a non-EP task. This is a raw approximation of finding the processor where a non-EP type task finishes the earliest.

In FLB, we also modify the task priority for the EP-type tasks. The EP-type tasks are sorted by their finish time on their enabling processor instead of their start time.

Finally, for both FCP and FLB, we change the final choice between the two candidate tasks, by selecting the task finishing the earliest instead of the task starting the earliest.

Note, that all these modifications of FCP and FLB do not involve any extra cost compared to the original versions. As a consequence, the cost of both FCP and FLB is maintained at the same very low level.

4 Performance Results

The FCP and FLB algorithms are compared with ERT (Earliest Task First) [9] and HEFT (Heterogeneous Earliest-Finish-Time) [17]. ERT ($O(W(E+V)P)$) and HEFT ($O(V \log W + (E+V)P)$) are well-known and have been shown to obtain competitive results in heterogeneous systems [9, 17].

For both FCP and FLB we used two versions. The first version uses *fully* sorted task lists. For this first version, FCP and FLB have exactly the same scheduling criteria as MCP and ETF, respectively. The second version uses

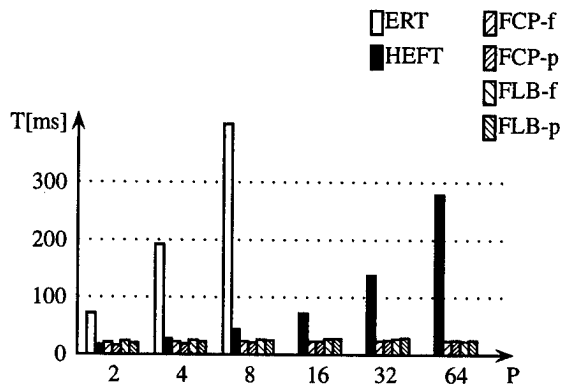


Figure 2: Cost comparison

partially sorted priority lists of size P. We call the first version of the algorithms FCP-f and FLB-f, and the second FCP-p and FLB-p, respectively.

We consider task graphs representing various types of parallel algorithms. The selected problems are *LU decomposition* ("LU"), *Laplace equation solver* ("Laplace") and a *stencil algorithm* ("Stencil"). For each of these problems, we adjusted the problem size to obtain task graphs of about 2000 nodes. For each problem, we varied the task graph granularities, by varying the communication-to-computation ratio (CCR). The values used for CCR are 0.2 and 5.0. For each problem and each CCR value, we generated 5 graphs with random execution times and communication delays (i.i.d. uniform distribution with unit coefficient of variation), the results being the average over the 5 graphs (in view of the low overall variance, 5 samples are sufficient). Miniature task graphs samples of each type are shown in Figure 1.

We schedule the task graphs on 2, 4, 8, 16 and 32 processors. For each P, we use 10 heterogeneous configurations in which the processors' speed are uniformly distributed over the following intervals: $[8, 12]$, $[6, 14]$ and $[4, 16]$. Thus, the total number of test configurations is 3 (problems) $\times$ 2 (CCR) $\times$ 5 (sample graphs) $\times$ 5 (processor ranges) $\times$ 10 (processor configurations) $\times$ 3 (processor intervals) $= 5500$.

4.1 Running Times

In Fig. 2 the average running time of the algorithms is shown in CPU seconds as measured on a Pentium Pro/300MHz PC with 64Mb RAM running Linux 2.0.32. ERT is the most costly among the compared algorithms. Its cost increases from 72 ms for 2 processors up to 11 s for 64 processors (we do not include ERT's running times for $P \geq 16$ in Figure 2 due to their too much higher values). HEFT's cost also increases with the number of processors, but it is significantly lower. For $P = 2$, it runs for 17 ms, while for $P = 64$, the running time is 279 ms.

Both versions of the FCP and FLB have considerably lower running times. FCP-p's running time is the lowest, varying from 16 ms for $P = 2$ to 25 ms for $P = 64$. FCP-f varies from 21 ms for $P = 2$ to 24 ms for $P = 64$. One can note that for larger number of processors both versions of FCP have the same running times. The reason is that the ready tasks fit in the sorted part of the FCP-f's priority list.

FLB has a slightly higher cost compared to FCP, because of the more complicated task and processor selection schemes. The running times vary around 26 ms and 24 ms for FLB-f and FLB-p, respectively. Their running times do not vary significantly with the number of processors. One can note that for larger number of processors, FCP and FLB's running times tend to become similar.

4.2 Scheduling Performance

In this section we study how the FCP and FLB algorithms perform. We first compare FCP and FLB's performance to ERT and HEFT's performance, with respect to granularity, problem type and processor heterogeneity. Next, we show the speedups achieved by FCP and FLB.

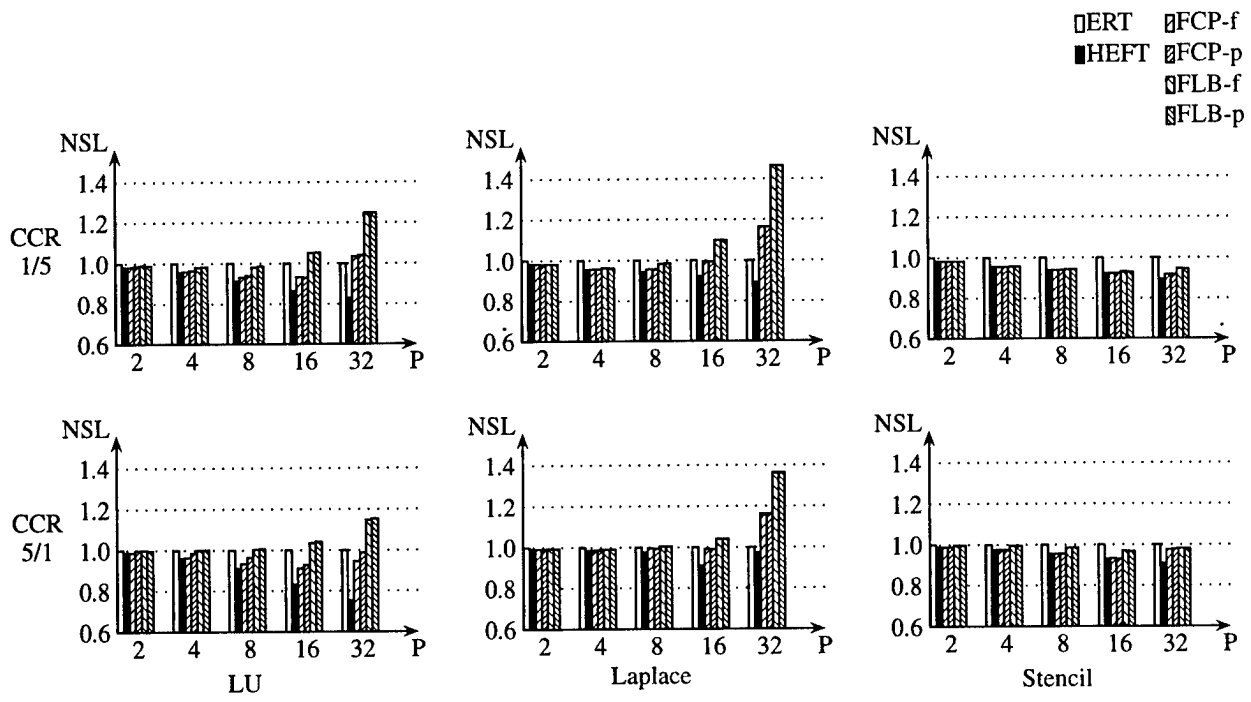


Figure 3: Performance comparison with respect to the problem

For performance comparison, we use the *normalized schedule length* (NSL), defined as the ratio between the schedule length of the given algorithm and the schedule length of ERT.

In Figure 3 we study the algorithms' performance with respect to the problem type by comparing the schedule lengths averaged over the three processor speed intervals. One can note that for both FCP and FLB, the partial versions obtain performance similar to the full versions. Therefore we will further refer only to the partial versions of FCP and FLB.

One can note that the overall performance of FCP is comparable to ERT's performance, although at a much lower cost. For problems involving a large number of fork and join tasks, such as LU and Laplace, for a large number of processors ERT performs better, up to 16% for both coarse and fine-grain cases (Laplace, $P = 32$). For all the other cases (i.e., for regular problems, such as Stencil, or for small number of processors) FCP performs equal or better compared to ERT, up to 8% (Stencil, $P = 32$) and 7% (LU, $P = 16$) for coarse and fine-grain problems, respectively.

Compared to HEFT, FCP is outperformed for problems involving a large number of fork and join tasks, such as LU and Laplace, for a large number of processors, with up to 27% (Laplace, $P = 32$) and 23% (LU, $P = 32$) for coarse and fine-grain cases, respectively. However, in all the other cases (i.e., for regular problems, such as Stencil, or for small number of processors) FCP performs comparable to HEFT.

FLB's performance is generally worse, being outperformed by ERT, HEFT and FCP by up to 46%, 57%, and 30% (all for coarse-grain Laplace, $P = 32\%$), respectively. However, even for FLB, the performance becomes comparable to the other three algorithms for regular problems, such as Stencil, or small number of processors.

In Figure 4 we study the influence of the heterogeneity to the performance. The results are averaged over the LU, Laplace and Stencil problems. Again, both FCP and FLB obtain similar performance for the full and partial versions.

Again, the overall performance of FCP is comparable

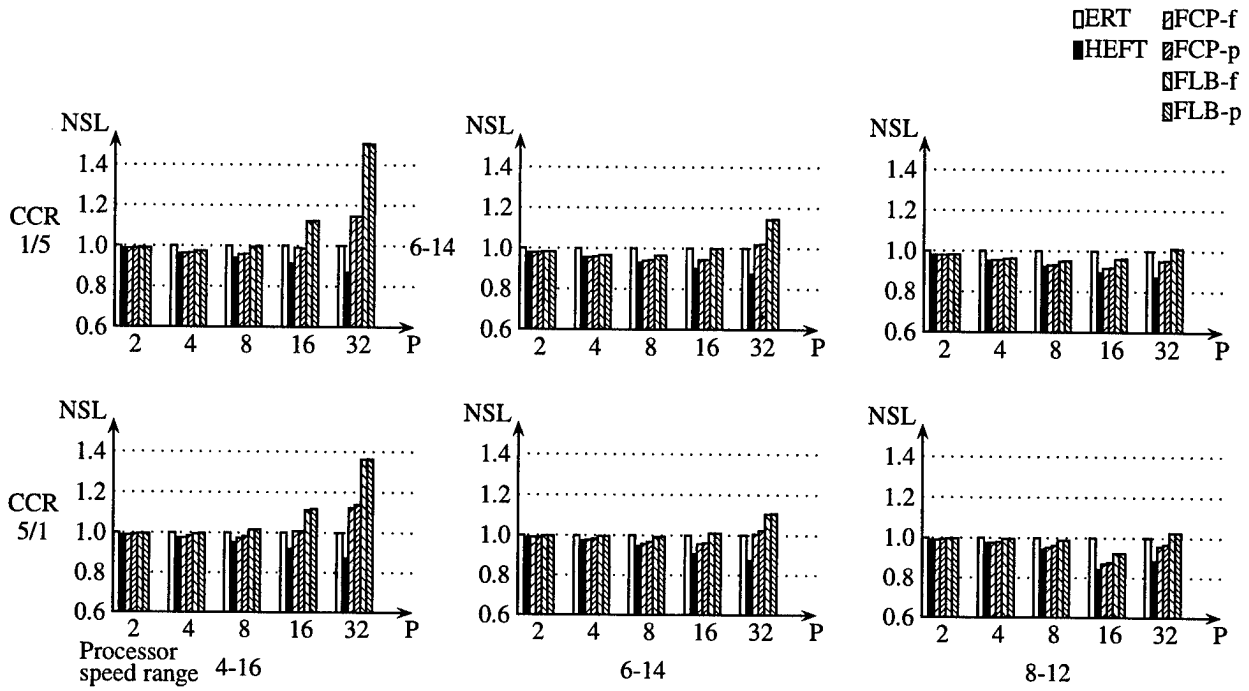


Figure 4: Performance comparison with respect to heterogeneity

to ERT's performance. For a large processor speed variance (i.e., $4-16$) and for a large number of processors ERT performs better, up to 15% and 12% for coarse and fine-grain cases ($4-16$ processor speed range, $P=32$), respectively. For all the other cases (i.e., small processor speed variance, or for small number of processors) FCP performs equal or even better compared to ERT, up to 8% and 12% (both for Stencil, $P=16$) for coarse and fine-grain problems, respectively.

Compared to HEFT, FCP is also outperformed for a large processor speed variance and for a large number of processors, with up to 28% and 26% (both for $4-16$ processor speed range, $P=32$) for coarse and fine-grain cases, respectively. However, for small processor speed variance, or for small number of processors, FCP's performance tends to become comparable to HEFT.

FLB's performance is generally worse, being outperformed by ERT, HEFT and FCP with up to 50%, 63%, and 35% (all for $4-16$ processor speed range, coarse-grain problems, $P=32\%$), respectively. However, even for FLB, the performance becomes comparable to the other three algorithms for regular problems, such as Stencil, or small number of processors.

One can note that for heterogeneous systems, the versions using fully and partially sorted priority lists perform comparable for both FCP and FLB. Similar to homogeneous systems, a partially sorted list of size P yields competitive results, while the scheduling complexity becomes extremely low: $O(V \log(P)+E)$.

Figures 5 and 6 show the speedups achieved for the FCP and FLB algorithms respectively. Although FCP performs better, the two algorithms perform similar with respect to problem type, granularity and processor speed range. For Stencil the speedup is almost linear. However, for LU and Laplace the speedup starts leveling off for more than 32 processors. The reason is that LU and Laplace have a large number of fork and join nodes, and as a consequence a limited parallelism, while Stencil is a regular problem with a large and constant parallelism. Also, one can note that for a large processor speed

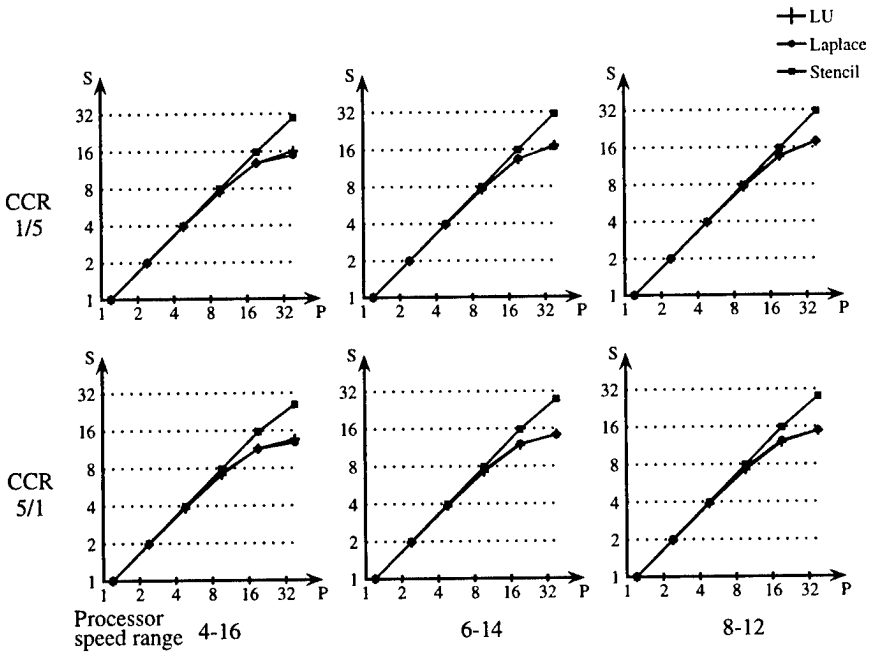


Figure 5: FCP-p Speedup

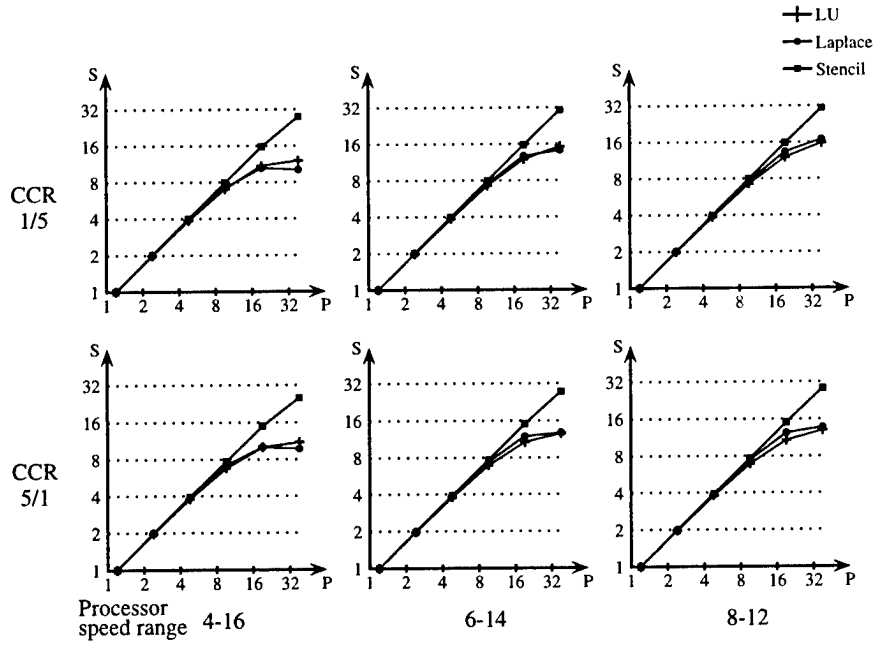


Figure 6: FLB-p Speedup

variance (i.e., $4 - 16$) and a large number of processors ($P = 32$) the speedup is lower compared to a small processor speed variance. Also, for fine-grain problems, the speedup is lower for a large number of processors. In both cases the reason is that there are not enough tasks to fully utilize the existing processors, and, as FCP and FLB are not specifically designed for heterogeneous processors, they do not always select the faster processors first.

5 Conclusion

In this paper we investigate the performance of the low-cost static list scheduling algorithm FCP and dynamic list scheduling algorithm FLB, modified to schedule applications for heterogeneous systems. We show that making minimal modifications that do not affect their very low cost, FCP and FLB still obtain good performance in heterogeneous systems, at a cost that is considerably below typical scheduling algorithms for heterogeneous systems.

We show that the performance of the modified versions of FCP and FLB is generally comparable to algorithms specifically designed for heterogeneous systems, such as HEFT and ERT. There are only a few cases, mainly for irregular problems and large processor speed variance, where FCP and FLB's performance drops down to 32% and 63%, respectively.

Considering the overall performance and their very low cost compared to the other algorithms, we believe FCP and FLB to be interesting compile-time candidates for heterogeneous systems, especially considering the large problem sizes that are used in practice.

References

[1] I. Ahmad and Y.-K. Kwok. A new approach to scheduling parallel programs using task duplication. In *Proc. Int'l Conf. on Parallel Processing*, 1994.

[2] Y. C. Chung and S. Ranka. Application and performance analysis of a compile-time optimization approach for list scheduling algorithms on distributed-memory multiprocessors. In *Proc. Supercomputing*, 1992.

[3] R. L. Graham. Bounds on multiprocessing timing anomalies. *SIAM Journal on Applied Mathematics*, 17(2):416–429, Mar. 1969.

[4] J.-J. Hwang, Y.-C. Chow, F. D. Anger, and C.-Y. Lee. Scheduling precedence graphs in systems with interprocessor communication times. *SIAM Journal on Computing*, 18:244–257, Apr. 1989.

[5] M. Kafil and I. Ahmad. Optimal task assignment in heterogeneous computing systems. In *Proc. Heterogeneous Computing Workshop*, 1997.

[6] A. A. Khan, C. L. McCreary, and M. S. Jones. A comparison of multiprocessor scheduling heuristics. In *Proc. Int'l Conf. on Parallel Processing*, 1994.

[7] B. Kruatrachue and T. G. Lewis. Grain size determination for parallel processing. *IEEE Software*, pages 23–32, Jan. 1988.

[8] Y.-K. Kwok and I. Ahmad. Benchmarking the task graph scheduling algorithms. In *Proc. Int'l Parallel Processing Symp. / Symp. on Parallel and Distributed Processing*, 1998.

[9] C.-Y. Lee, J.-J. Hwang, Y.-C. Chow, and F. D. Anger. Multiprocessor scheduling with interprocessor communication delays. *Operations Research Letters*, 7:141–147, June 1988.

[10] M. Maheswaran and H. J. Siegel. A dynamic matching and scheduling algorithm for heterogeneous computing systems. In *Proc. Heterogeneous Computing Workshop*, 1998.

[11] G.-L. Park, B. Shirazi, J. Marquis, and H. Choo. Decisive path scheduling: A new list scheduling method. In *Proc. Int'l Conf. on Parallel Processing*, 1997.

[12] A. Rădulescu and A. J. C. van Gemund. FLB: Fast load balancing for distributed-memory machines. In *Proc. Int'l Conf. on Parallel Processing*, 1999.

[13] A. Rădulescu and A. J. C. van Gemund. On the complexity of list scheduling algorithms for distributed-memory systems. In *Proc. ACM Int'l Conf. on Supercomputing*, 1999.

[14] A. Rădulescu, A. J. C. van Gemund, and H.-X. Lin. LLB: A fast and effective scheduling algorithm for distributed-memory systems. In *Proc. Int'l Parallel Processing Symp. / Symp. on Parallel and Distributed Processing*, pages 525–530, 1999.

[15] V. Sarkar. *Partitioning and Scheduling Parallel Programs for Execution on Multiprocessors*. PhD thesis, MIT, 1989.

[16] M. Tan, H. J. Siegel, J. K. Antonio, and Y. A. Li. Minimizing the application execution time through scheduling of subtasks and communication traffic in a heterogeneous computing system. *IEEE Trans. on Parallel and Distributed Systems*, 8(8):857–871, Aug. 1997.

[17] H. Topcuoglu, S. Hariri, and M.-Y. Wu. Task scheduling algorithms for heterogeneous processors. In *Proc. Heterogeneous Computing Workshop*, 1999.

[18] L. Wang, H. J. Siegel, V. P. Roychowdhury, and A. A. Maciejewski. Task matching and scheduling in heterogeneous computing environments using a genetic-algorithm-based approach. *Journal of Parallel and Distributed Computing*, 47:8–22, 1997.

[19] C. M. Woodside and G. G. Monforton. Fast allocation of processes in distributed and parallel systems. *IEEE Trans. on Parallel and Distributed Systems*, 4(2):164–174, Feb. 1993.

[20] M.-Y. Wu and D. D. Gajski. Hypertool: A programming aid for message-passing systems. *IEEE Trans. on Parallel and Distributed Systems*, 1(7):330–343, July 1990.

[21] M.-Y. Wu, W. Shu, and J. Gu. Local search for dag scheduling and task assignment. In *Proc. Int'l Conf. on Parallel Processing*, 1997.

[22] T. Yang and A. Gerasoulis. Pyrros: Static task scheduling and code generation for message passing multiprocessors. In *Proc. ACM Int'l Conf. on Supercomputing*, 1992.

Biographies

Andrei Rădulescu received a MSc degree in Computer Science in 1995 from "Politehnica" University of Bucharest. Between 1995 and 1997 he was a teaching assistant at the "Politehnica" University of Bucharest. Since 1997, he is a PhD student at the Department of Information Technology and Systems of Delft University of Technology. His research interests are in multiprocessor scheduling, software support for parallel computing and parallel and distributed systems programming,

Arjan J.C. van Gemund received a BSc in Physics in 1981, a MSc degree (cum laude) in Computer Science in 1989, and a PhD (cum laude) in 1996, all from Delft University of Technology. In 1981 he joined the R & D organization of a Dutch multinational company as an Electrical Engineer and Systems Programmer. Between 1989 and 1992 he joined the Dutch TNO research organization as a Research Scientist specialized in the field of high-performance computing. Since 1992, he works at the Department of Information Technology and Systems of Delft University of Technology, currently as Associate Professor. His research interests are in the area of parallel and distributed systems programming, scheduling, and performance modeling.

SESSION 4-A
GRID APPLICATIONS

Chair: I. Pramanick, *Sun Microsystems, USA*

Combining Workstations and Supercomputers to Support Grid Applications: The Parallel Tomography Experience

Shava Smallen* Walfredo Cirne* Jaime Frey¶ Francine Berman* Rich Wolski†
Mei-Hui Su§ Carl Kesselman§ Steve Young‡ Mark Ellisman‡

* Computer Science and Engineering Department
University of California, San Diego
[ssmallen, walfredo, berman] @cs.ucsd.edu

¶ Department of Computer Science
University of Wisconsin
jfrey@cs.wisc.edu

† Department of Computer Science
University of Tennessee
rich@cs.utk.edu

§ Information Sciences Institute
University of Southern California
[mei, carl] @isi.edu

‡ National Center for Microscopy and Imaging Research
University of California, San Diego
mark@ncmir.ucsd.edu

Abstract

Computational Grids are becoming an increasingly important and powerful platform for the execution of large-scale, resource-intensive applications. However, it remains a challenge for applications to tap into the potential of Grid resources in order to achieve performance. In this paper, we illustrate how work queue applications can leverage Grids to achieve performance through ***coallocation****. We describe our experiences developing a scheduling strategy for a production tomography application targeted to Grids that contain both workstations and parallel supercomputers.*

Our strategy uses dynamic information exported by a supercomputer's batch scheduler to simultaneously schedule tasks on workstations ***and*** *immediately available supercomputer nodes. This strategy is of great practical interest because it combines resources available to the typical research lab: time-shared workstations and CPU time in remote space-shared supercomputers. We show that this strategy improves the performance of the tomography application compared to traditional scheduling strategies, which target the application to either type of resource alone.*

This research was supported in part by NSF grants ASC-9701333 and ASC-9318180, DoD Modernization contract 9720733-00, NPACI/NSF award ASC-9619020 and Cooperative Agreement ANI-9807479, NIH/NCR grants RR04050 and RR08605, CAPES grant DBE2428/95-4, and DARPA/ITO under contract #N66001-97-C-8531.

1. Introduction

The aggregation of heterogenous resources into a Computational Grid [9] provides a powerful platform for the execution of large-scale resource-intensive applications. The simultaneous use of heterogeneous resources can greatly improve the performance of many applications, and permits researchers to run applications at the very large problem sizes critical to the discovery of new results. Although we are gaining considerable experience in the development of infrastructures which integrate distributed, heterogeneous resources, we have less experience developing applications which can leverage the distributed resources of the Grid to improve performance.

One application which has profited from leveraging the processing power of the Computational Grid is the Parallel Tomography (GTOMO) application being used in production at the National Center for Microscopy and Imaging Research (NCMIR). GTOMO is an embarrassingly-parallel application implemented with a work queue scheduling strategy. It uses Globus [10] services to perform a 3-D reconstruction from a series of images produced by NCMIR's electron microscope. As is the case with many laboratories, NCMIR owns a limited number of workstations (which are used as desktop machines and as a platform for parallel processing) and has access to supercomputer time. *In this*

0-7695-0556-2/00 $10.00 © 2000 IEEE

paper, we describe a coallocation strategy for using both supercomputers and interactive workstation clusters to improve the execution performance of GTOMO within the context of a typical lab environment.

The scheduling strategy for GTOMO works at the application-level to target the application to both interactive workstation clusters and supercomputers. In an interactive workstation cluster, typically a *time-shared* computational platform, jobs begin execution immediately but share the CPU and network with other competing processes. In contrast, job submissions to a supercomputer, typically a *space-shared* computational platform, must wait in a batch queue until the desired number of the machine's processors become available for dedicated use. The time an application spends waiting in the queue impacts its *turnaround time*, the time elapsed from the submission of the application by the user until all of the results are available. Because the queue wait time can be quite lengthy [20], an application's turnaround time can be relatively large compared to its execution time.[1] Furthermore, the queue wait times make it difficult to use supercomputers and workstations concurrently, a strategy that could increase the processing power available to an application. Our strategy avoids unpredictable queue time delays by adaptively submitting requests to the supercomputer that can start running immediately.

The adaptive scheduler developed for GTOMO is framed as an AppLeS [2]. An AppLeS application scheduler integrates with the target application to develop a schedule for deploying the application in a shared, dynamic Grid environment [3, 23, 22]. The scheduler makes predictions of the performance the application may experience on prospective resources at execution time. Using these predictions, a potentially performance-efficient schedule for the application is identified and deployed. We developed a simple and effective coallocation strategy for the GTOMO AppLeS which targets both supercomputers and interactive workstations. Our experiments show that the GTOMO AppLeS coallocation strategy improves the turnaround time of the application over strategies which target either interactive workstations alone or a parallel supercomputer alone. We believe that the GTOMO AppLeS coallocation strategy will be effective for other work queue applications as well.

The next section provides a brief description of GTOMO. Section 3 describes our coallocation strategy for scheduling GTOMO over workstations and supercomputers. Section 4 presents the results of comparing our strategy against other scheduling alternatives. Section 5 discusses related work. Section 6 concludes the paper and discusses future work.

[1] In practice, queue times may range from seconds to days.

2. GTOMO Structure

Tomography allows for the reconstruction of the 3-D structure of an object based on 2-D projections through it taken at different angles. Electron microscopy is a classical use for tomography. Biological specimens on the cellular and sub-cellular level are viewed with an electron microscope and their images are recorded at a number of different angles. These images are then aligned and reconstructed into 3-D volumes using analytic and iterative tomographic techniques [18].

Reconstructing a typically sized volume using a simple algorithm (filtered back-projection) currently takes several hours on a workstation. NCMIR researchers have been interested in increasing the computation speed of the reconstruction for two reasons. First, they want to make use of more elaborate tomographic algorithms, which produce more refined 3-D volumes. These algorithms are more computationally intensive than the algorithms currently used. Second, NCMIR is interested in *on-line* tomography where the volume is rendered while the biologist is still collecting data on the microscope. This provides immediate feedback about the specimen being viewed and thus may prompt the researcher to change the experiment as a whole, or just some parameters of it (e.g., orientation and/or number of projections). For this to be useful, a rough reconstruction would have to finish in 5 to 10 minutes. No single processor can achieve this presently, which led the NCMIR researchers to explore parallelism.

The tomography application is highly amenable to parallelism. Because specimens are only rotated about a single axis as images are acquired, for any slice orthogonal to the axis of rotation, all information for that slice falls onto a single line on each of the projections (see Figure 1). More importantly, any such slice can be reconstructed independently of projection information for the rest of the volume. This makes the reconstruction embarrassingly parallel. Therefore, the the tomographic reconstruction can naturally be implemented as a *work queue*. In our implementation, we use Globus services to support efficient application execution within a heterogeneous, distributed environment.

The structure of GTOMO is depicted in Figure 2. There are four types of application processes: *driver, reader, writer*, and *ptomo*. The driver controls the work queue: it assigns one work unit or *slice* to a free ptomo until no more slices remain. The driver is invoked by the user and starts up the other processes. The reader and writer are I/O processes and hence have direct access to the user file system. The reader reads input files off the disk and sends them to the ptomos for processing. The writer receives output files from ptomos and writes them to disk. Note that the reader and writer enable GTOMO to run across different file system domains. The ptomo receives input files from a

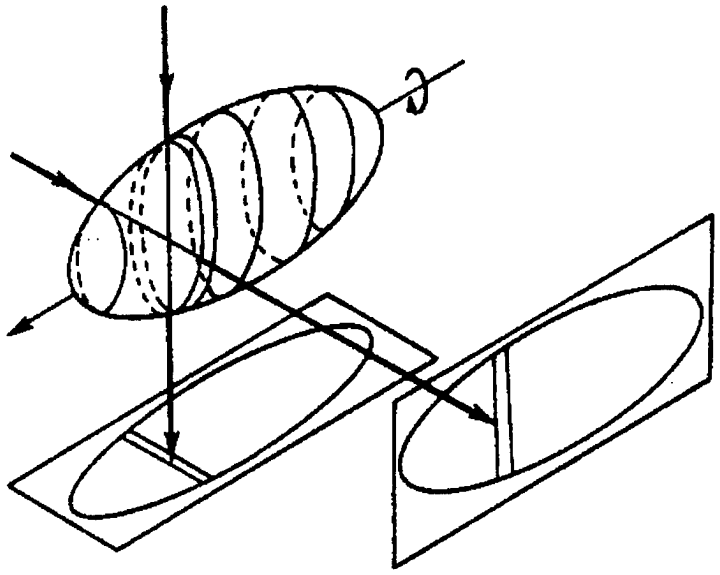


Figure 1. Projection geometry relating to a single-axis tilting experiment (from [12])

reader, does all the computational work, and sends output to a writer. In this study, we use one reader, one writer, and any number of ptomos. Due to the multi-threaded nature of Globus' Nexus communications library, one reader can service I/O requests for many ptomos simultaneously, and the same applies to the writer.

3. Scheduling GTOMO

Generally speaking, the set of potential resources available to GTOMO consists of workstations $w_1, ..., w_\omega$ and supercomputers $s_1, ..., s_\sigma$. A request to run a process p on workstation w causes p to start immediately, but p timeshares w with other processes. To use a supercomputer s, one has to specify how many processors n will execute copies of p and for how long t. The n copies of p do not necessarily start immediately; they might wait in the queue for an indeterminate amount of time until n nodes become available for t seconds. However, supercomputer processes run over dedicated resources once they are acquired.

Scheduling a GTOMO job consists of (i) choosing the requests to send to both supercomputers and workstations, and (ii) assigning work for the ptomos. For (ii), we use the work queue strategy shown in Figure 2 that assigns work on demand. For the first, we have to determine performance-efficient values of n and t for each available supercomputer s. Our goal is to select n and t in a way that minimizes GTOMO's turnaround time. Note that difficulty in predicting supercomputer queue wait times make it difficult to find an optimal n and t [8, 20, 14]. We avoid the queue time prediction problem by using supercomputer nodes that are immediately available. Therefore, we minimize the turnaround time of GTOMO by scheduling its execution at once on workstations and any immediately available supercomputer nodes.

We assume that the supercomputer scheduler can provide us with the maximum values of n and t for which execution can begin immediately. In our implementation, this information is supplied by the showbf command provided with the Maui Scheduler [15], a scheduler available for the IBM SP2. The showbf command returns a set of *backfill windows*, $b_1, ..., b_\theta$. Each $b_i = (n, t)$ where n nodes are available for immediate execution for the next t seconds.

The GTOMO AppLeS scheduler uses the following algorithm to schedule the ptomos:

```
for i = 1 to σ:
  b = showbf(s_i);
  for j = 1 to θ:
    start b_j.n ptomos on s_i for time b_j.t
for i = 1 to ω:
  start ptomo on w_i
```

Therefore, if backfill windows are available on any of the supercomputers, the job will be coallocated on those idle supercomputer nodes and workstations. If no backfill windows are returned by any of the supercomputers, the job

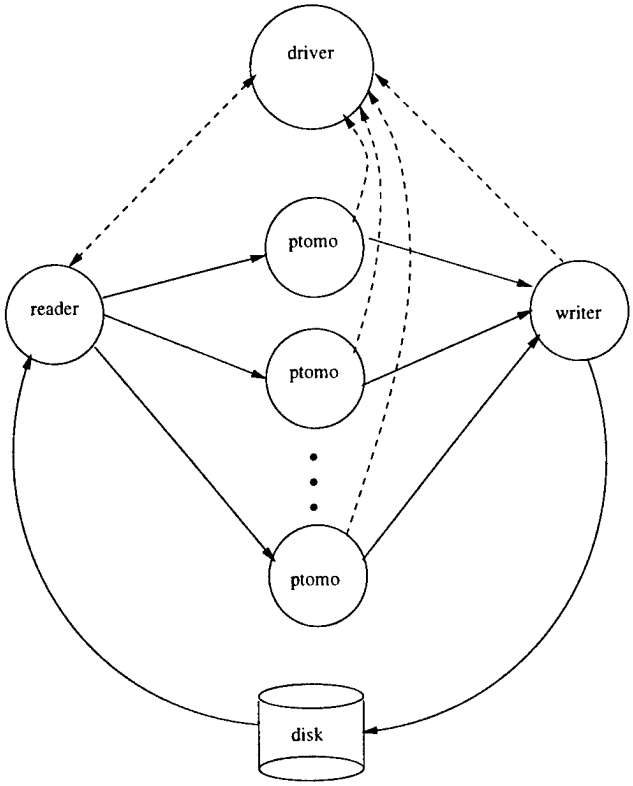


Figure 2. Application components of GTOMO. Solid lines represent transfer of input and output. Dotted lines denote control connections

will run only on workstations. The reader is scheduled on the machine where the input data is located and the writer is scheduled on the machine where the output data will be placed.

Note that the nodes immediately available in the SP2 may not be available for the full duration of the application. Therefore, the GTOMO AppLeS scheduler has to cope with ptomo processes that detach themselves from the application before execution has completed. We have added a fault recovery mechanism to GTOMO, which enables us to treat this problem as a ptomo failure. Whenever a ptomo fails, the slice it was processing is returned to the work queue. We can use such a simple scheme because processing a slice has no side effects. The advantage of reducing this problem to fault recovery is, of course, that it also covers real faults.

4. Experimental Results

We denote the GTOMO AppLeS scheduling strategy as *SP2Immed/WS* since it adaptively combines both the immediately available SP2 nodes and workstations. In order to ascertain how this strategy performs, we compared it against other possible scheduling strategies: using workstations only (*WS*), using only the nodes that are immediately available in the SP2 (*SP2Immed*), and requesting a predetermined number of nodes in the SP2 and probably waiting for them in the queue (*SP2Queue*). WS and SP2Queue respectively are the standard ways to use a cluster of workstations and a parallel supercomputer.

We ran experiments on a cluster of 7 workstations available in the Parallel Computation Laboratory (PCL) at U.C. San Diego and on the San Diego Supercomputer Center's SP2, one of the supercomputers available to NCMIR scientists. The PCL workstation cluster includes one 200 MHz UltraSPARC 2, a 110 MHz Sparc 5, a 85 MHz Sparc 5, and four 400 MHz Pentium IIs. The workstations are connected by a mixture of 10 and 100 Mbit/s ethernet subnets. The SP has 128 thin node POWER2 processors running at 160 MHz where processor pairs are interconnected by a 110 MB/s bi-directional network [21]. Other users were present on all resources during the experiments. Our dataset consisted of 300 slices; each input slice was 238 KB and the output slice was 1.2 MB.

We note that it is problematic to design experiments which compare multiple scheduling strategies under the same load and queue conditions for multi-user production environments. In such environments, the load and availability of resources change over time, so reproducibility of the

same ambient load conditions is generally not an option. In contrast, it is possible to achieve reproducibility using simulation, but it may be difficult to represent dynamic load variation in complex heterogeneous systems authentically.

For our experiments, we performed sets of runs of SP2Immed, SP2Immed/WS, WS, and SP2Queue back-to-back [2] hoping that experiments within the same set would enjoy roughly similar load conditions. Moreover, we monitored the number of free nodes in the SP2 and used this information to discard execution sets in which the nodes available to SP2Immed/WS and SP2Immed differed by more than two. In this case, we considered the load conditions for strategies in the same set to be different. This was the case in 37 (out of 100) experiment sets. We therefore ended up with 63 valid experiment sets.

There are two other details to note in the design of our experiments. First, when we use only the resources immediately available in the SP2, it might be that there are no nodes available to execute the application. In this case, we did not run the set of experiments until the necessary resources became available. This happened 9 times out of 63 attempts. Notice that by excluding this retry time, we present optimistic turnaround times for the SP2Immed method. A user using this method would have experienced longer delays.

Second, we needed to decide on n and t when we used the SP2 in the traditional way (i.e., SP2Queue). Note that the determination of the best n requires an accurate queue time prediction. Since such predictions are not available, we rotated among values of n likely to be used by GTOMO users: 8, 16, and 32 nodes. We then executed benchmarks on the SP2 to determine the average processing time of one slice, t_b. This enabled us to conservatively determine t given n (a conservative estimate is needed because a job is killed when its execution exceeds t) using the following:

$$t = 2 \times \frac{t_b \times number\ of\ slices}{n}$$

The results of the 63 experimental sets are partitioned into three groups using the number of nodes requested for SP2Queue: SP2Queue(8), SP2Queue(16), and SP2Queue(32). Figure 3 shows the results of the experiment sets in which SP2Queue used 8 nodes, Figure 4 shows the results for 16 nodes, and Figure 5 shows the results for 32 nodes. Figures 3-5(a) depict the turnaround times of the different strategies (WS, SP2Immed/WS, SP2Immed, and SP2Queue). Each set of bars in the figure depicts a set of four executions, one under each of the four strategies. Since several of the SP2Queue turnaround times did not fit on the graphs, Table 1 displays the turnaround times for the SP2Queue runs. The number of nodes SP2Immed and SP2Immed/WS acquired in each set of experiments is shown in Figures 3-5(b).

We see that the SP2Immed/WS strategy yielded the best performance in all cases except one (Figure 4, run 8). Further study indicated contention on the reader and writer due to the selection of too many ptomos (in this experiment set, we received the highest number of immediately available SP2 nodes). A future scheduler improvement would be to model the contention and incorporate it into the GTOMO AppLeS.

We also assess the variability of each strategy using the coefficient of variance, c_v, which measures the amount of variance relative to the mean [7]. It is defined as follows:

$$c_v = \frac{standard\ deviation}{mean}$$

The SP2Immed/WS strategy exhibited the lowest c_v in all groups of experiments. Table 2 shows the mean, coefficient of variance, minimum, and maximum values for each strategy in each group of experiments. Table 2(a) shows the results of the experiment sets where SP2Queue used 8 nodes, Table 2(b) shows the results for 16 nodes, and Table 2(c) shows the results for 32 nodes. As expected, SP2Queue's c_v is quite large due to the unpredictable wait times in the queue. While its turnaround time was sometimes close to SP2Immed/WS (544s for SP2Queue vs. 601s for SP2Immed/WS for the one time it beat SP2Immed/WS), its worst time was more than two orders of magnitude greater than SP2Immed/WS (88,323s for SP2Queue vs. 601s for SP2Immed/WS). Also, we note that the SP2Immed strategy had a high c_v in the SP2Queue(8) results due to the variability of number of nodes acquired. This variability was amortized in the SP2Immed/WS strategy because of the relatively low c_v of WS.

5. Related Work

The GTOMO code is also used in the Computed Microtomography (CMT) Project at Argonne National Laboratory (ANL) [26, 27]. In contrast to NCMIR, projections are collected from a x-ray source at the Advanced Photon Source (APS) located at ANL. Their work has focused on on-line tomography where data is collected at APS, transferred to a 128 node SGI Origin 2000 for processing, and then transferred back to the user for visualization. Currently, they are able to deliver a reconstructed image to the user within minutes after data acquisition has completed. The CMT and NCMIR versions of GTOMO are currently being integrated as part of the NPACI Telescience Alpha Project [25].

Application scheduling for Grids is a recent and very active area. Existing work has focused primarily on resource discovery and scheduling [4, 17, 10, 28] and coallocation

[2] We used a 5 minute interval between experiments to ensure that the Maui scheduler had time to update its availability information.

run	8 nodes	16 nodes	32 nodes
1	685.0235	591.2153	1293.9811
2	759.2754	581.2684	532.6011
3	698.2693	20483.5053	536.9106
4	3077.8569	723.0844	541.6972
5	701.3900	17268.0273	658.2000
6	708.8977	2097.6415	565.0041
7	691.5886	579.8207	27480.0918
8	1805.0212	543.7824	9868.9585
9	6163.5884	581.0620	2267.3595
10	687.5382	615.1581	614.1135
11	689.0494	793.9383	17735.0314
12	682.3540	9193.4053	39286.8783
13	700.8448	742.4905	34642.5641
14	4625.1468	2164.4710	1120.0531
15	1260.1495	621.3329	88322.6571
17	3249.5812	574.9321	1809.4320
18	710.8228	593.0303	664.4993
19	718.0481	1203.5411	6815.6301
20	721.3216	575.2712	607.1331
21	719.9383	33715.8070	29675.0315
22	707.4284	580.8794	
23	717.6290		

Table 1. Turnaround times for SP2Queue

strategy	mean	c_v	min	max
SP2Immed	1946.68	2.10	504.81	19694.46
SP2Immed/WS	437.99	0.17	346.71	554.50
WS	775.98	0.19	596.26	1133.44
SP2Queue	1430.94	1.05	682.35	6163.59

(a) SP2Queue(8) results

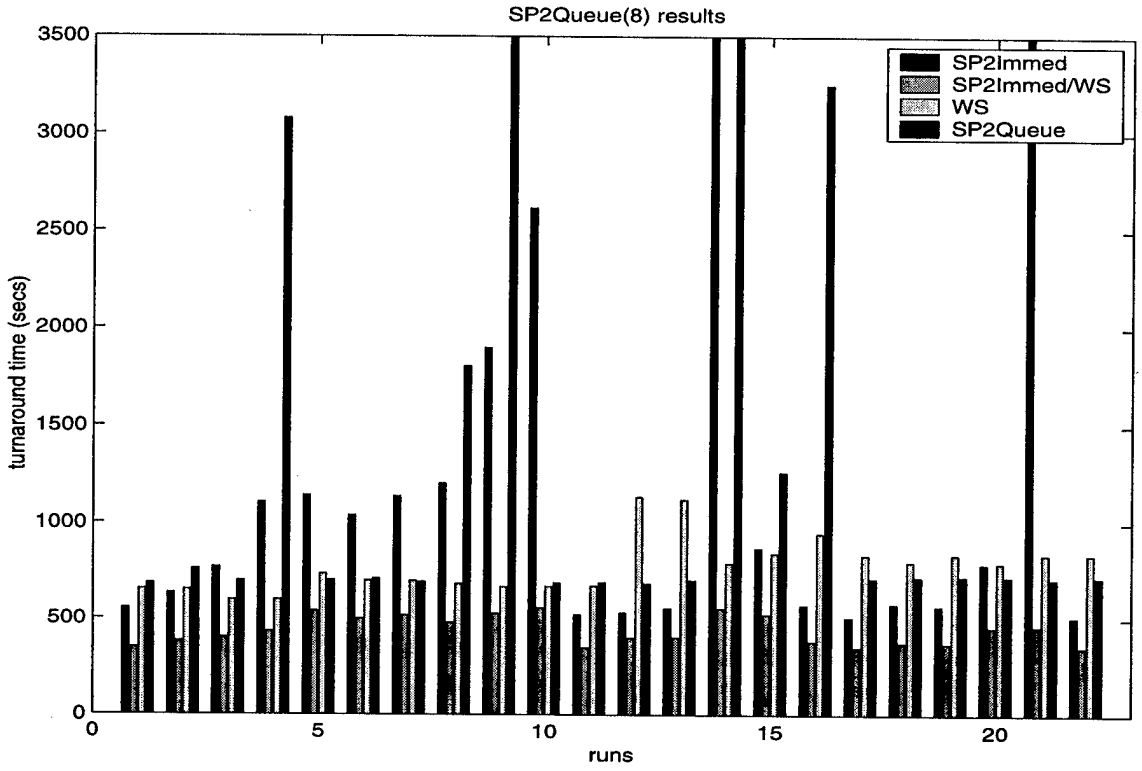
strategy	mean	c_v	min	max
SP2Immed	660.58	0.28	502.75	1105.61
SP2Immed/WS	402.31	0.17	342.03	600.98
WS	777.53	0.19	588.76	1127.03
SP2Queue	4515.41	1.94	543.78	33715.81

(b) SP2Queue(16) results

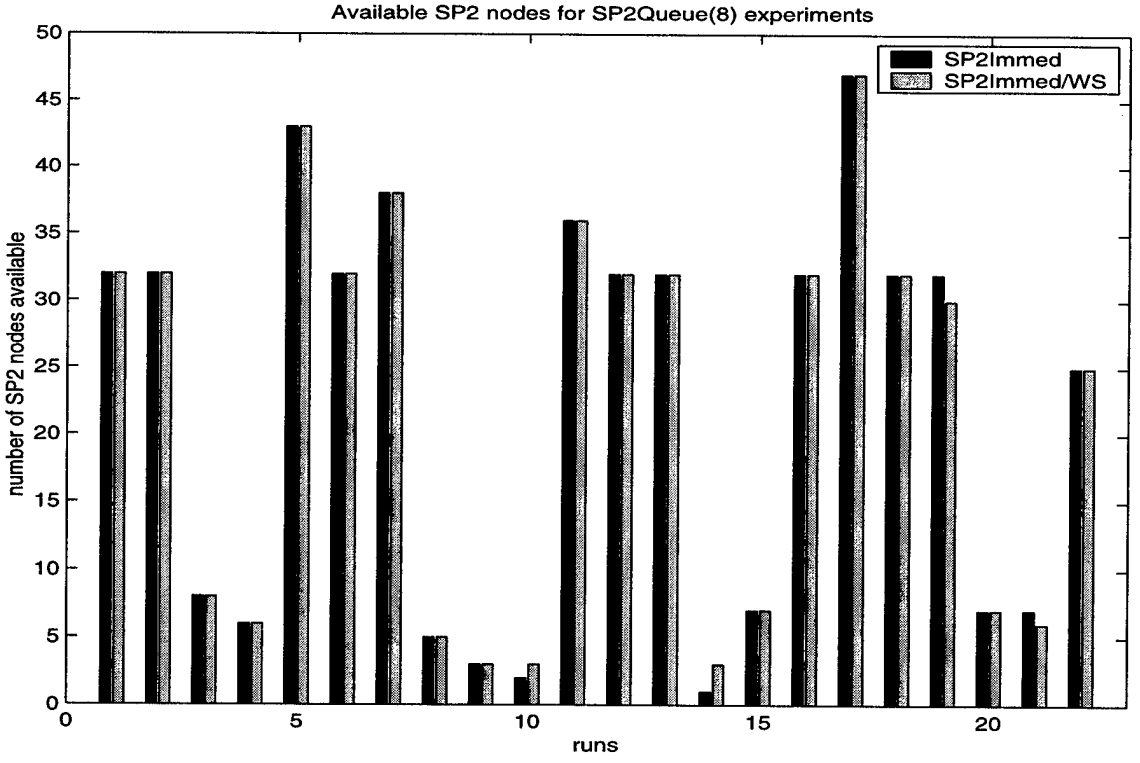
strategy	mean	c_v	min	max
SP2Immed	659.30	0.33	500.24	1262.72
SP2Immed/WS	397.37	0.12	342.70	519.13
WS	789.69	0.20	587.68	1128.41
SP2Queue	13251.89	1.66	532.60	88322.66

(c) SP2Queue(32) results

Table 2. Summary results of experiments

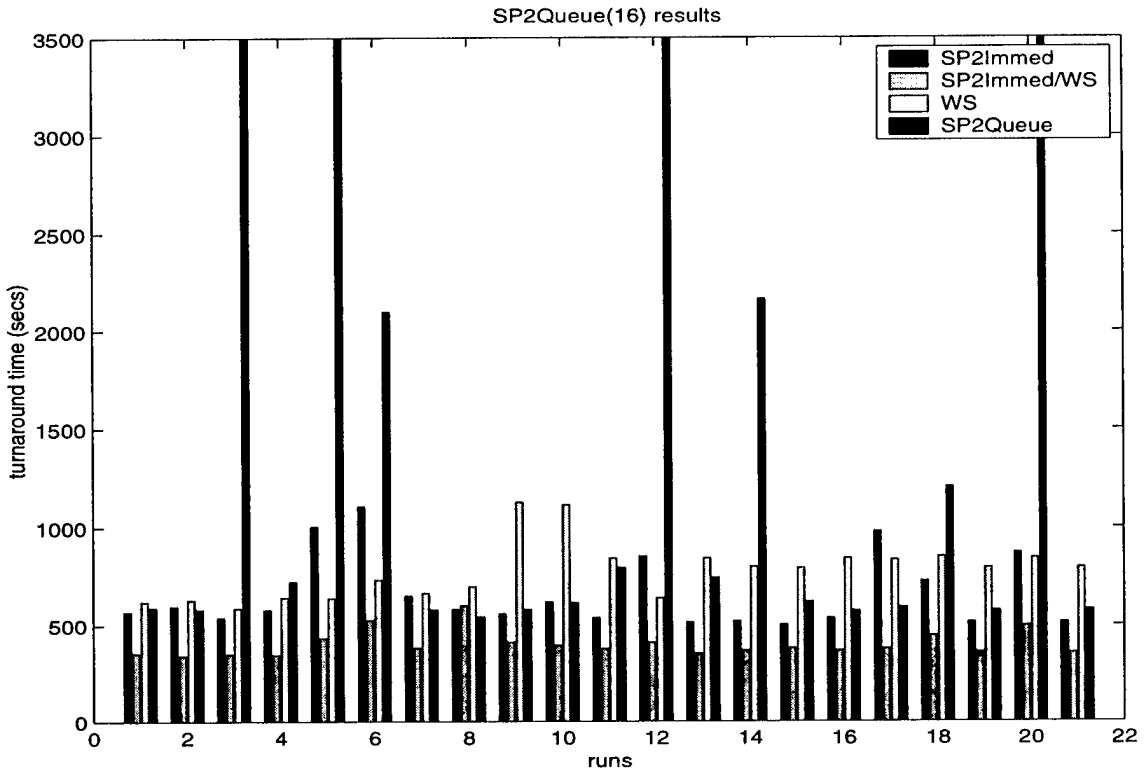


(a)

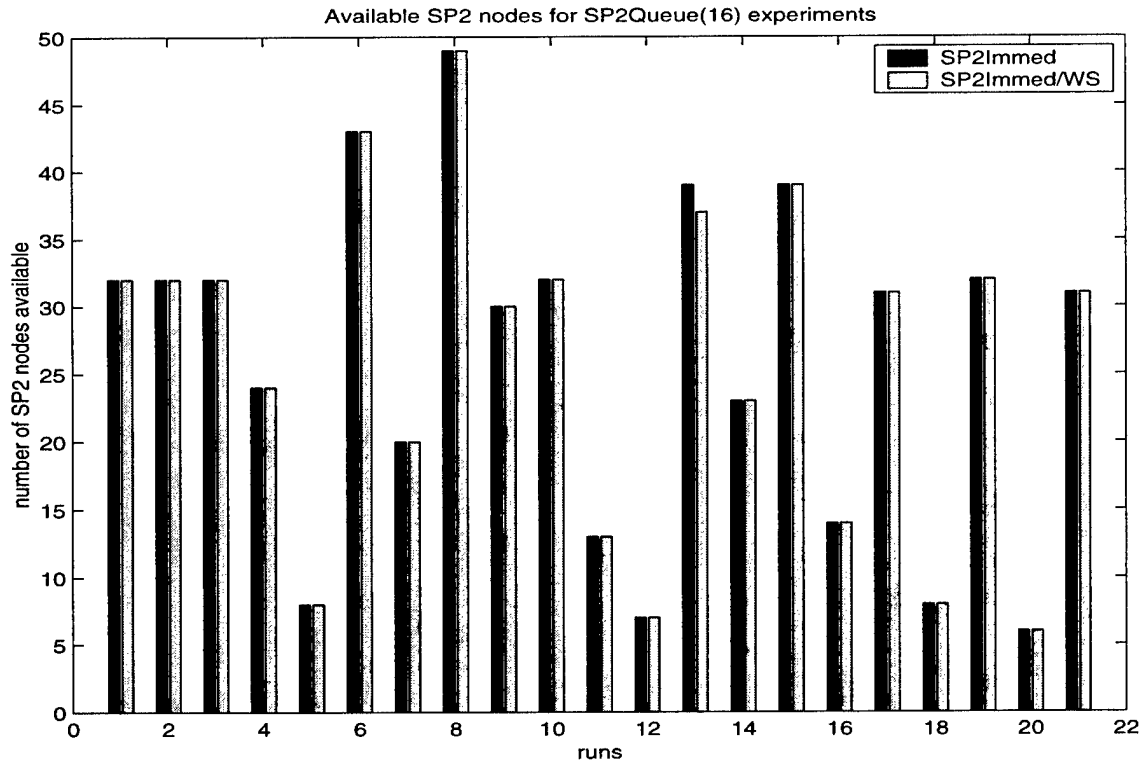


(b)

Figure 3. Experiment results when 8 nodes were requested for SP2Queue. (a) Turnaround time. (SP2Immed values too large to fit on graph: run 14 - 5067s, run 21 - 19694s) (b) Number of SP2 nodes used by SP2Immed/WS and SP2Immed.

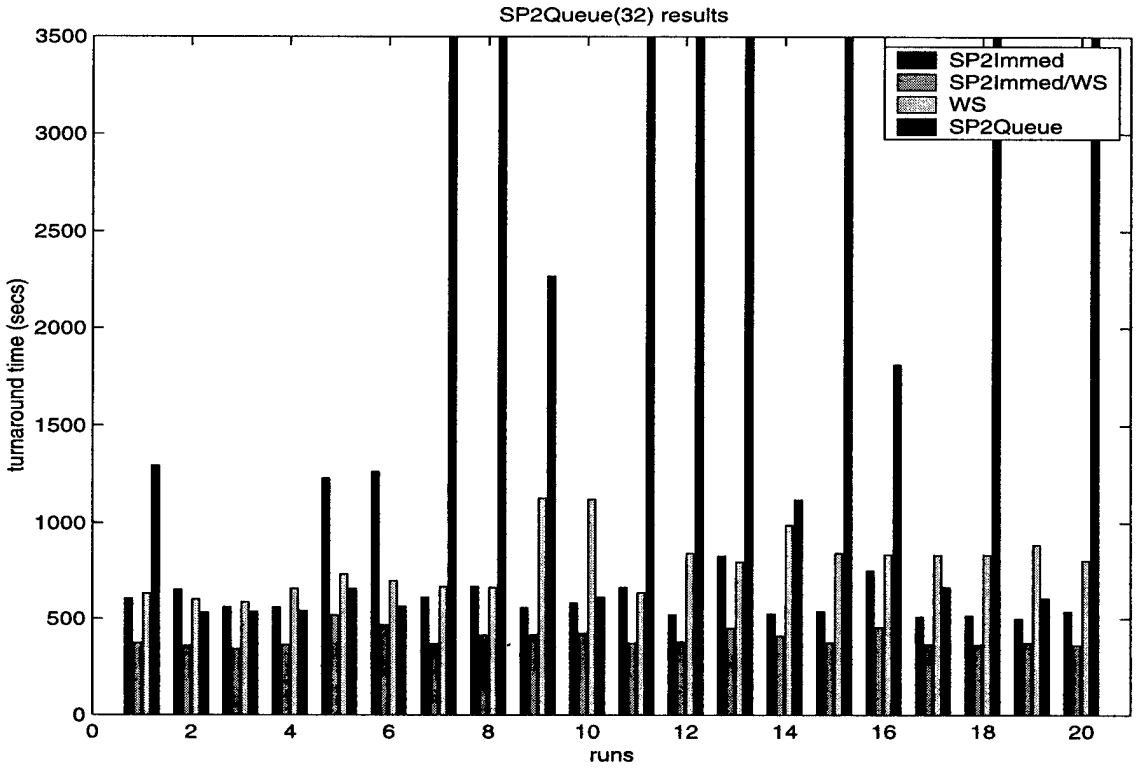


(a)

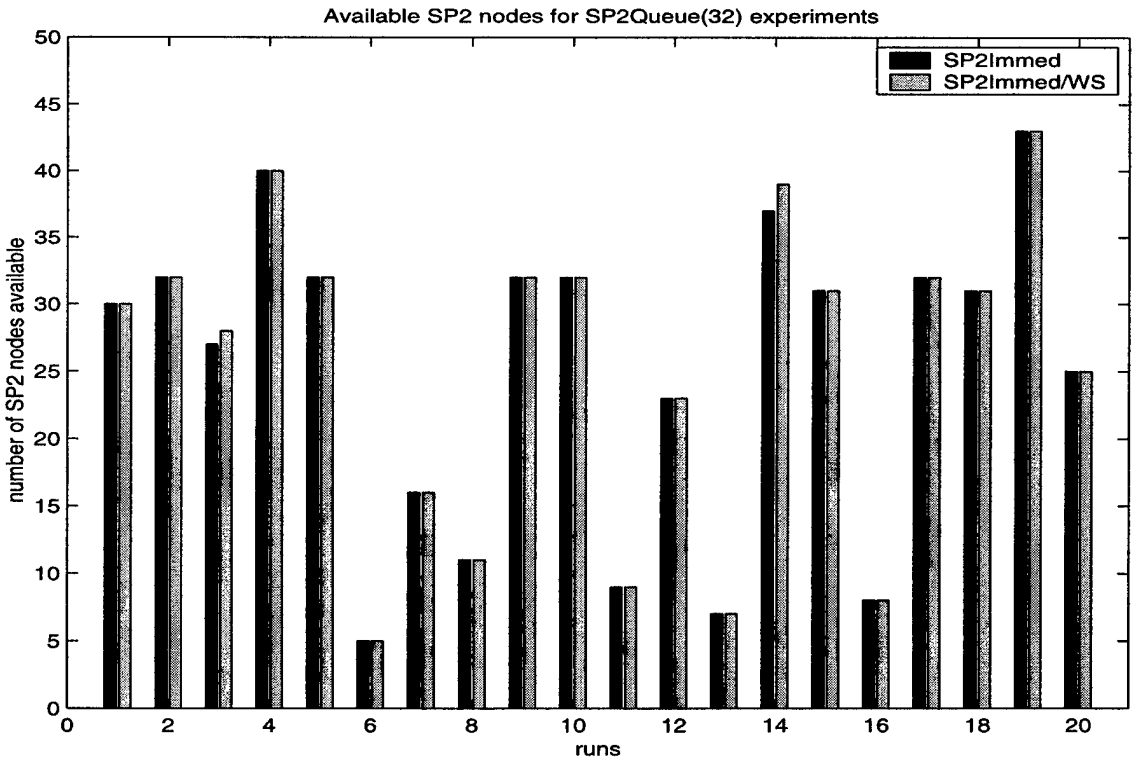


(b)

Figure 4. Experiment results when 16 nodes were requested for SP2Queue. (a) Turnaround time. (b) Number of SP2 nodes used by SP2Immed/WS and SP2Immed.



(a)



(b)

Figure 5. Experiment results when 32 nodes were requested for SP2Queue. (a) Turnaround time. (b) Number of SP2 nodes used by SP2Immed/WS and SP2Immed.

among workstations [3, 1, 19, 22, 23, 9]. The work reported herein extends the target domain for GTOMO by targeting both parallel supercomputers and interactive resources simultaneously.

6. Discussion and Conclusions

In this work, we show how to combine workstations and supercomputers to run GTOMO, a work queue application used in production at NCMIR. Our solution automatically selects all resources immediately available across the system. We leverage the Maui Scheduler to obtain information on immediately available SP2 nodes. This strategy has the advantage of not requiring predictions of how long requests wait in the supercomputer queue. Our experimental results show that the GTOMO AppLeS scheduling strategy consistently outperforms three other strategies that can be used for scheduling in an typical laboratory setting where researchers have access to a local cluster of workstations and supercomputer time.

We have learned three interesting lessons about Computational Grids in general as a result of this effort. First, the interface exported by the resource scheduler has great impact on application schedulers. In fact, we can implement our strategy in a very straightforward manner thanks to the Maui Scheduler's showbf command. On the other hand, the Maui Scheduler (as with other supercomputer schedulers, for that matter) precluded us from trying something more sophisticated due to the difficulty in predicting queue times for supercomputer requests. Emerging efforts such as S^3 [6], GARA [11], and more generally, the Grid Forum Scheduling Working Group [13] are working to change this.

Second, evaluating solutions for real applications running over production environments has proven to be difficult due to the impossibility of reproducing the system load and queue conditions for comparison runs. Others have encountered the same problem. Indeed, a simulation environment specifically targeted toward Grids such as the Bricks project [24], the MicroGrid [16], or the work described in [5] would be very useful.

Third, fault tolerance is likely to be even more important in Grid computing than it is in parallel computing. For our solution in particular, fault recovery was a natural way to deal with the time expiration of SP2 requests. In general, using autonomous and distributed resources increases the chance that some component of the application will fail.

The GTOMO AppLeS scheduler has been incorporated with the production version of GTOMO at NCMIR and is used daily be researchers. Current work involves extending the applicability of the scheduler to additional resources and different scenarios of the application.

Dedication and Acknowledgements

This paper is dedicated to Steve Young who was a cornerstone of this work. Steve was a respected scientist and treasured friend and we will miss him greatly.

We are grateful to the NPACI partnership and SDSC researchers for their assistance with this work. We benefited greatly from discussions with the AppLeS team, the Globus team, and our colleagues at NCMIR.

References

[1] D. Andersen, T. Yang, O. Ibarra, and O. Egecioglu. Adaptive partitioning and scheduling for enhancing WWW application performance. *Journal of Parallel and Distributed Computing*, 49:57–85, Feb 1998.

[2] AppLeS webpage at `http://apples.ucsd.edu`.

[3] F. Berman, R. Wolski, S. Figueira, J. Schopf, and G. Shao. Application level scheduling on distributed heterogeneous networks. In *Proceedings of Supercomputing 1996*, 1996.

[4] H. Casanova and J. Dongarra. Netsolve: A network server for solving computational science problems. In *Proceedings of Supercomputing 1996*, 1996.

[5] H. Casanova, A. Legrand, D. Zagorodnov, and F. Berman. Using Simulation to Evaluate Scheduling Heuristics for a Class of Applications in Grid Environments. Technical Report RR1999-46, École Normale Supérieure de Lyon, LIP, 1999.

[6] W. Cirne and F. Berman. Application scheduling over supercomputers: A proposal. Technical Report UCSD-CS99-631, University of California, San Diego, October 1999.

[7] J. L. Devore. *Probability and Statistics for Engineering and the Sciences*. Duxbury Press, 1995.

[8] A. Downey. Using queue time predictions for processor allocation. In *3rd Workshop on Job Scheduling Strategies for Parallel Processing, in conjunction with IPPS'97*, 1997.

[9] I. Foster and C. Kesselman, editors. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers, July 1998.

[10] I. Foster and C. Kesselman. The Globus Project: A Status Report. In *Proc. IPPS/SPDP '98 Heterogeneous Computing Workshop*, 1998.

[11] I. Foster, C. Kesselman, C. Lee, B. Lindell, K. Nahrstedt, and A. Roy. Distributed resource management architecture that supports advance reservations and co-allocation. In *Intl Workshop on Quality of Service*, 1999.

[12] J. Frank and M. Radermacher. Three-dimensional reconstruction of nonperiodic macromolecular assemblies from electron micrographs. In J. K. Koehler, editor, *Advanced Techniques in Biological Electon Microscopy III*. Springer-Verlag, 1986.

[13] Grid Forum webpage at `http://www.gridforum.org/`.

[14] R. Gibbons. A historical application profiler for use by parallel schedulers. In *3rd Workshop on Job Scheduling Strategies for Parallel Processing, in conjunction with IPPS'97*, 1997.

[15] Maui Scheduler webpage at `http://www.mhpcc.edu/maui`.

[16] MicroGrid webpage at `http://www-csag.ucsd.edu/projects/grid/microgrid.html`.

[17] H. Nakada, M. Sato, and S. Sekiguchi. Design and implementations of ninf: towards a global computing infrastructure. *Future Generation Computing Systems*, 1999. Metacomputing Issue.

[18] G. Perkins, C. Renken, S. Young, S. Lamont, M. Martone, S. Lindsey, T. Frey, and M. Ellisman. Electron tomography of large multicomponent biological structures. *J. Struct.Biol.*, 120:219–227, 1997.

[19] G. Shao, R. Wolski, and F. Berman. Predicting the cost of redistribution in scheduling. In *Proceedings of the 8th SIAM Conference on Parallel Processing for Scientific Computing*, 1997.

[20] W. Smith, V. Taylor, and I. Foster. Using run-time predictors to estimate queue wait times and improve scheduler performance. In *5th Workshop on Job Scheduling Strategies for Parallel Processing, in conjunction with IPPS'99*, 1999.

[21] NPACI's SP webpage at `http://www.npaci.edu/SP`.

[22] N. Spring and R. Wolski. Application level scheduling of gene sequence comparison on metacomputers. *12th ACM International Conference on Supercomputing*, July 1998.

[23] A. Su, F. Berman, R. Wolski, and M. M. Strout. Using AppLeS to schedule a distributed visualization tool on the computational grid. *International Journal of Supercomputer and High-Performance Applications*, 1999.

[24] A. Takefusa, S. Matsuoka, H. Nakada, K. Aida, and U. Nagashima. Overview of a performance evaluation system for global computing scheduling algorithm. To appear in the 8th HPDC conference, 1999.

[25] Telescience webpage at `http://www.npaci.edu/Alpha/Telescience/`.

[26] G. von Laszewski, M.-H. Su, J. Insley, I. Foster, J. Bresnahan, C. Kesselman, M. Thiebaux, M. Rivers, S. Wang, B. Tieman, and I. McNulty. Real-time analysis, visualization, and steering of tomography experiments at photon sources. *Ninth SIAM Conference on Parallel Processing for Scientific Computing*, Apr 1999.

[27] Y. Wang, F. D. Carlo, I. Foster, J. Insley, C. Kesselman, P. Lane, G. von Laszewski, D. Mancini, I. McNulty, M.-H. Su, and B. Tieman. A quasi-realtime x-ray microtomography system at the Advanced Photon Source. *Proceedings of SPIE99*, 3772, 1999.

[28] J. Weissman. Gallop: The benefits of wide-area computing for parallel processing. *Journal of Parallel and Distributed Computing*, 54, Nov 1998.

Shava Smallen received a B.S. and is currently pursuing a M.S. at the Department of Computer Science and Engineering at the University of California, San Diego. Her current research interest is in scheduling on Computational Grids.

Walfredo Cirne is a Ph.D. student at the University of California, San Diego and an assistant professor at Brazil's Universidade Federal da Paraiba (currently on leave). He has previously worked on Machine Learning and Network Security. Now, his research efforts concentrate on scheduling on Computational Grids. In particular, he is investigating how application schedulers can use resources controlled by space-shared supercomputers. He received his B.S. and M.S from the Universidade Federal da Paraiba.

Jaime Frey is currently pursuing a Ph.D. at the Department of Computer Science at the University of Wisconsin. His current research interests include scheduling in distributed systems. He received his B.S. from the University of California, San Diego.

Francine Berman is a Professor of Computer Science and Engineering at the University of California, San Diego. She is also a Senior Fellow at the San Diego Supercomputer Center, Fellow of the ACM, and founder of the Parallel Computation Laboratory at UCSD. Her research interests over the last two decades have focused on parallel and distributed computation, and in particular the areas of programming environments, tools, and models that support high-performance computing. She received her B.A. from the University of California, Los Angeles, her M.S. and Ph.D. from the University of Washington.

Rich Wolski is an Assistant Professor in the Department of Computer Science at the University of Tennessee and a partner in the National Partnership for Advanced Computational Infrastructure. His research interests include parallel and distributed computing, on-line performance analysis techniques and software, compiler runtime system, and dynamic scheduling. He received his B.S. from the California Polytechnic University, San Luis Obispo and his M.S. and Ph.D. from the University of California at Davis/Livermore Campus.

Mei-Hui Su is a programmer at the Information Sciences Institute located at the University of Southern California. Her current research interest is in distributed parallel computing. She received her B.S. from the University of California, Berkeley.

Carl Kesselman leads a research group at the Information Sciences Institute located at the University of Southern California. His research focus has been in the development of new methods, tools, and programming environments for large-scale, high-performance computer systems. He received his B.S. from the University of Buffalo, his M.S. from the University of Southern California, and his Ph.D. from the University of California, Los Angeles.

Steve J. Young was a Specialist in Psychiatry and an Associate Director of the National Center for Microscopy and Imaging Research (NCMIR). Until his untimely

passing on January 9th, 1999, he led the technology development team at NCMIR, contributing to the work presented here and nearly all projects at the Center. His research interests ranged from the neurophysiology of perception to the design of software to improve our ability to accurately represent complex relationships between biological systems. He was a unique man with remarkable talents. He was also a friend and colleague who shared his extensive knowledge freely enhancing the lives of all around him, including the authors of this paper. He earned his Undergraduate Degree at Berkeley and Ph.D. at the University of California, Los Angeles. He became a Professor of Physiological Psychology at the University of Colorado at Boulder before moving to UCSD where together with Ellisman he established the NCMIR.

Mark H. Ellisman is a Professor of Neurosciences and Bioengineering at the University of California, San Diego and Director of the National Center for Microscopy and Imaging Resource and the Center for Research on Biological Structure at UCSD. He leads the Neuroscience Thrust for the National Partnership for Advanced Computational Infrastructure and is a Senior Fellow at the San Diego Supercomputer Center. His research interests over the last three decades have focused on structure and function of the nervous system including the development of advanced imaging instruments and computational approaches for the refinement of data about and visualization of multiscale biological complexes. He received his B.A. from the University of California, Berkeley and M.A. and Ph.D from the University of Colorado at Boulder.

Cluster Performance and the Implications for Distributed, Heterogeneous Grid Performance

Craig Lee[1] Cheryl DeMatteis[1] James Stepanek[1] Johnson Wang[2]

[1] *Computer Systems Research Department, M1-102*
[2] *Fluid Mechanics Department, M4-965*
The Aerospace Corporation, P.O. Box 92957, El Segundo, CA 90009-2957
{lee|cdematt|stepanek}@aero.org
Johnson.C.Wang@notes.aero.org

Abstract

This paper examines the issues surrounding efficient execution in heterogeneous grid environments. The performance of a Linux cluster and a parallel supercomputer is initially compared using both benchmarks and an application. With an understanding of how benchmark and application performance is affected by processor and interconnect speed, a comparison is made with the bandwidth and latencies available in a grid testbed. Of significant concern is the fact that the available communication bandwidth and latencies have a dynamic range of 3 to 4 orders of magnitude while processor speeds have a range of about one half order of magnitude. Also, while both processor speed and network bandwidth are increasing very rapidly, simple propagation delay will become more significant in the network latencies seen by many grid applications. That is to say, the pipes in a grid will be getting fatter but not commensurately shorter. How are we to effectively utilize such an infrastructure? Clearly an attractive approach is to require sufficient concurrency in the application such that a coarse-grain, data-driven model of execution can be used to hide latencies while hopefully keeping context switching overheads low. If the "spatial component" of an application is understood, then runtime systems could also apply established techniques like caching, compression, estimation and speculative pre-fetching. Ideally this low-level performance management should be encapsulated in an easy-to-use abstraction.

1 Introduction

Cluster computing has been gaining wide acceptance over single-machine, massively parallel computing due to its undeniable cost-effectiveness for suitable applications [4]. Since clusters are built from commodity hardware, however, they typically have slightly slower processors and lower communication bandwidths than "big iron" machines. Hence, suitability in this context means simply that either (1) an application must be more tolerant of higher communication costs, or (2) the user's "mission requirements" are lenient enough to accept the lower performance at a much lower dollar cost.

The increasing potential of *grid computing*, however, means that users and applications will be faced with environments that have an even greater heterogeneity of communication abilities. [8]. While this potential includes the flexible harnessing of resources on a scale not previously considered for individual applications, it also means that achieving efficient use of those resources will be harder than ever. This paper endeavors not to present any solutions to this problem but to quantitatively demonstrate the bounds of the problem as motivation for exploring candidate programming and execution models that can effectively operate in a grid environment.

We will do this by comparing the performance of a parallel machine, a cluster, and a grid testbed by several means. These are specifically the Cray T3E [13], a Pentium cluster with fast ethernet, and the Globus GUSTO testbed [7], which are representative of their respective classes. (Different examples of each class could be used but the fundamental relationships between them would not be altered.) The Cray T3E used here is the T3E-1200 at the CEWES Major Shared Resource Center in Vicksburg, Mississippi. It has 512 DEC Alpha 21164 processors clocked at 600 MHz. with 128 MB of memory per processor. Its 3D torus dedicated interconnect is capable of 650 MB/sec. (theoretical peak) in both directions. The cluster used here is at the Aerospace Corporation. It has fourteen Intel Pentium

0-7695-0556-2/00 $10.00 © 2000 IEEE

II processors clocked at 400 MHz. running Red Hat Linux with 192 MB of memory per processor. They are connected by 100 Mbit/sec. fast ethernet through a Baystack 450-24T switched ethernet hub. The Globus GUSTO testbed is distributed across many adminstative sites and includes a large variety of machines. The current configuration of GUSTO can always be examined by using the Metacomputing Directory Service (MDS) Browser on the Globus web site (www.globus.org).

These "machines" (including the grid) will be compared using parallel benchmarks, a parallel application, and a distributed performance monitoring tool. We will look at the relative processing speeds and communication speeds. We then discuss the implications of achieving efficiency in an increasingly heterogeneous computing infrastructure.

2 A Benchmark Comparison

To compare the performance of a Linux cluster and a parallel supercomputer, we use the NAS Parallel Benchmarks [12]. These benchmarks were developed by the Numerical Aerodynamics Simulation (NAS) group at NASA Ames with the goal of being able to make more reliable quantitative performance comparisons among parallel machines. These benchmarks consist of numerical kernels for a wide range of computing problems. Rather than a single "micro-benchmark" that may exercise only one aspect of a machine, these benchmarks were chosen to exercise all aspects of a machine, individually and in combination. Specifically these benchmarks exercise communication, integer computation and floating-point computation.

For brevity and conciseness, we only need to present the results of two benchmarks that illustrate the major difference between these two platforms. For each benchmark, the *per processor performance* is plotted as a function of the number of nodes. These two benchmarks involve integer computation, so the measurement metric is millions of operations per second per node: Mop/sec/node. This allows the relative performance *and* scalability on each platform to be shown in one graph. For each benchmark, there are also three *classes*, A, B, and C, that correlate to three different problem sizes, with A being the smallest and C being the largest. Hence, for each benchmark graph, there are three curves (one for each class) for both platforms. For consistency and ease of comparison, the same point symbol is used for each platform. The same line style is used for each benchmark class.

Figure 1 shows the Random Number Generation benchmark. This is an "embarrassingly parallel" benchmark since the parallel tasks (generating random numbers) are completely independent, i.e., after the tasks are started, there is absolutely no communication or synchronization between nodes. As expected, both platforms show good scaling (flat curves). The Alpha processors, however, are approximately 4x faster than the Pentium IIs.

Figure 2 shows the Integer Sorting benchmark. Integer sorting is not a computationally complex task since it primarily requires the comparison of integers. It can, however, require massive amounts of communication as data values are relocated to their sorted positions. Here we see that the T3E exhibits not only faster processing but also much better communication scaling. For the cluster, the per-node performance falls off dramatically as the number of nodes increases.

These two benchmarks dramatically illustrate the performance differences between parallel and distributed computations that are compute-bound versus communication-bound. In the sorting benchmark, the communication bandwidth is clearly dominating the overall performance. In terms of relative performance and scalability, the other NAS Parallel Benchmarks fall inbetween these two extremes.

3 An Application Comparison

In this section, we use an application to compare the performance of these two platforms. That application is ALSINS (Aerospace Launch Systems Implicit Navier-Stokes), a computational fluid dynamics (CFD) code developed by the Fluid Mechanics Department at Aerospace and used to investigate flow fields of the Delta-II and Titan-IV launch vehicles [16, 15].

CFD works by discretizing the space around a physical object into "cells" and computing the flux of material between cells by solving the Navier-Stokes equations for a sequence of time steps until the solution has converged to a final state. CFD is typically parallelized by decomposing the discretized spatial domain and assigning different blocks to different processors. The algorithm has an iterative structure consisting of (1) exchanging neighbor data, (2) computing the minimum time step among all blocks, and (3) computing the flux for the current time step. For ALSINS, this is implemented using MPI.

With this basic structure, there are two hard synchronizations per iteration: exchanging neighbor data and the minimum time reduction. Aside from potential synchronization delays, the minimum time reduction is a very quick operation since it only involves finding the minimum of a single floating-point time step value across all nodes. The time required for communication and the local flux computation, however, depends on the data block size allocated to each node. Note that it is possible to improve efficiency by overlapping communication and computation for a given iteration. The rate of convergence for the solution depends on the geometry of the test case and can be on the order of 10^5 iterations. The speed at which iterations can be computed depends on the total size of the discretized space and

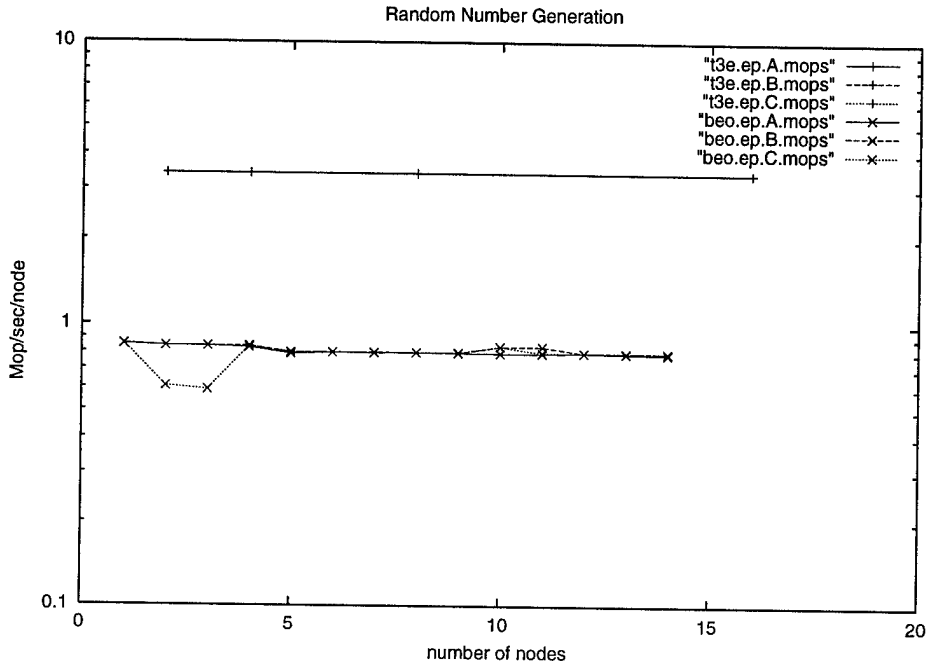


Figure 1. *The Random Number Generation Benchmark.*

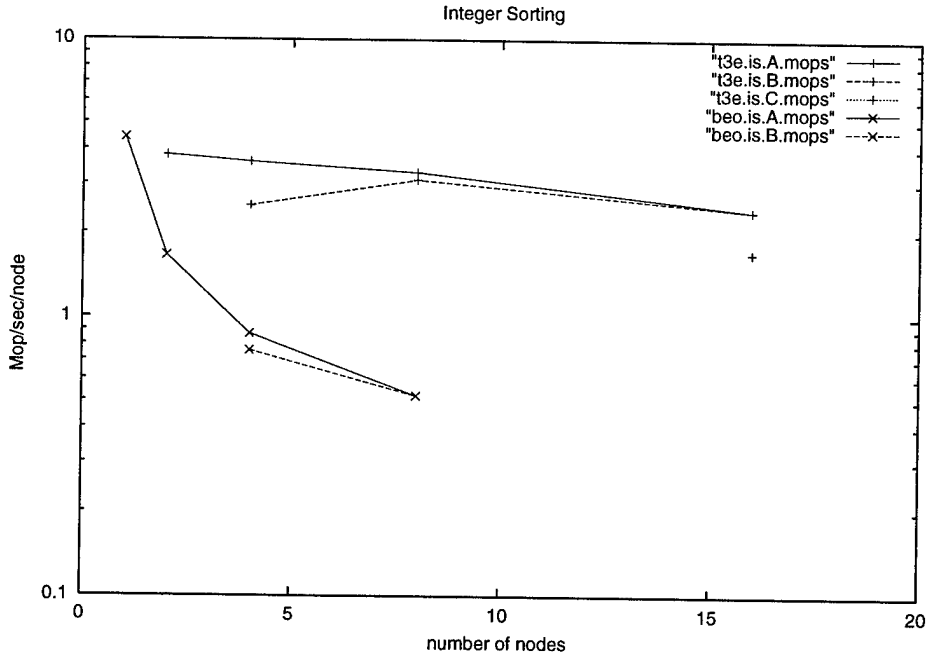


Figure 2. *The Integer Sorting Benchmark.*

the number and speed of the processing nodes used on the problem.

The test case computed using ALSINS is the flow field around the base of a Centaur launch vehicle with both engines running with exhaust plumes. Figure 3 shows a flow field computation done on the Pentium cluster.

ALSINS performance was measured and analyzed using NetLogger [14], a tool developed at Lawrence Berkeley Lab for analyzing distributed systems. NetLogger logs timestamped, application-defined events either locally or to a remote logging daemon. The NetLogger visualization tool, *nlv*, can subsequently display these events as grouped, color-coded sets of events, called *lifelines*, over time. Other events or statistics associated with a scalar values, such as cpu load, can be also be displayed as *loadlines*.

ALSINS with the Centaur Double Nozzle test case was run on both platforms in two versions using overlapped and non-overlapped communication. The NetLogger visualization display for two representative iterations of the overlapped code version on the Pentium cluster is shown in Figure 4. This shows the color-coded, per-iteration lifelines for each node. Each lifeline consists of six events tags: REDUCE_TAU, START_COMM, START_SOLVER, SOLVER_DONE, COMM_DONE, and COPIES_DONE. Loadlines for the utilization (number of processors actively engaged in communication or computation) and the efficiency (utilization over the duration of the computation) are also shown.

These results show us that per iteration, ALSINS is $\approx$2.9x faster on the T3E than on the cluster (4.75 seconds vs. 1.63 seconds). Part of this difference is due to the faster processors on the T3E and also a memory subsystem augmented with stream buffers. Of this iteration time, however, there is $\approx$19% idle time due to the load imbalance. Hence, there is an efficiency of $\approx$81% where processors are busy doing communication or computation. On the cluster, communication takes $\approx$16% of the iteration time. On the T3E, communication takes $\approx$2%.

4 A Bandwidth μbenchmark Comparison

It is certainly not news that communication bandwidth plays a direct role in determining parallel application performance. But in the scope of emerging computational infrastructures, however, what is the depth of the communication hierarchy? What is the range of impact that communication infrastructures can have, will have, on distributed, parallel applications? We examine this question in two parts. First, we do a simple MPI bandwidth test between the Pentium cluster and the T3E. Second, we compare these results with a histogram of host pair bandwidths on the Globus GUSTO testbed [7].

The MPI bandwidth test program we used tests a variety of communication patterns with differing number of nodes and different data volumes (message sizes). We ran this program on both the Pentium cluster and the T3E. For brevity and conciseness, we present only the most relevant data in Table 1. In this particular test, bidirectional communication occurs among all nodes simultaneously for two to eight nodes. This means that every node is sending and receiving a 1 MB message from all other nodes at the same time to stress the limits of performance.

The cluster is theoretically capable of 100 Mbit/sec. or 12.5 MB/sec. For two nodes, a bidirectional bandwidth of over 10 MB/sec., or 80 Mbit/sec., is achieved. This is the expected end-to-end result since overhead in the message-passing process, e.g., buffer copying and device driver scheduling, etc., means that an application will always see less bandwidth than the physical medium is "clocked" at; in this case, fast ethernet. Note, however, that as more nodes are added to the test, the realized bandwidth sinks to about 4 MB/sec. This indicates that contention for resources is occurring somewhere. (While the hub is technically non-blocking, it may still have a backplane that is becoming saturated as the aggregate bandwidth demand increases.) For the T3E, we see that two nodes are capable of over 300 MB/sec. For eight nodes, the average bidirectional bandwidth is still over 200 MB/sec. This means that the dedicated communication hardware on the T3E is 30x to 50x faster than fast ethernet in a cluster. (This might lead one to conclude that much of a large machine's cost is in its dedicated interconnect.)

How do these bandwidths compare with that typically available in a grid environment? To answer this question, we made use of the Gloperf network performance data that is periodically uploaded into the Globus Metacomputing Directory Service (MDS) [6]. The MDS is based on the Lightweight Directory Access Protocol (LDAP) and provides an information naming scheme and repository for all manner of grid computing information, e.g., available hosts, number of nodes, current load, network interfaces, gatekeeper contact information, etc. It is also used to record bandwidth and latency data periodically measured between host pairs by Gloperf.

Gloperf [10] is a simple tool that is automatically deployed on each Globus host. At Globus boot-time, the Gloperf daemon will register itself in the MDS and then query the MDS for all other Gloperf daemons. The daemon will then make periodic bandwidth and latency tests with all other daemons and store the results in the MDS. (The initial implementation of Gloperf did measurements between all pairs which does, of course, result in non-scalable behavior. The latest implementation uses a simple group scheme to produce hierarchies of measurements.)

The actual Gloperf measurement mechanism is borrowed from netperf. Gloperf is configured to perform a

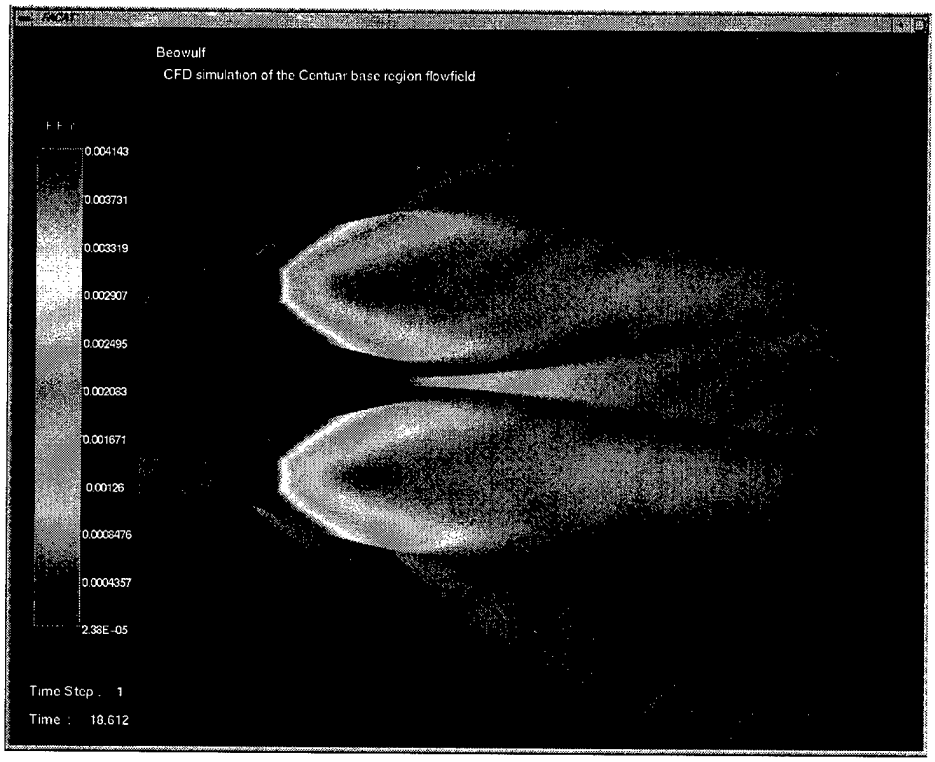


Figure 3. *The Centaur Double Nozzle Test Case Computed on a Pentium Cluster.*

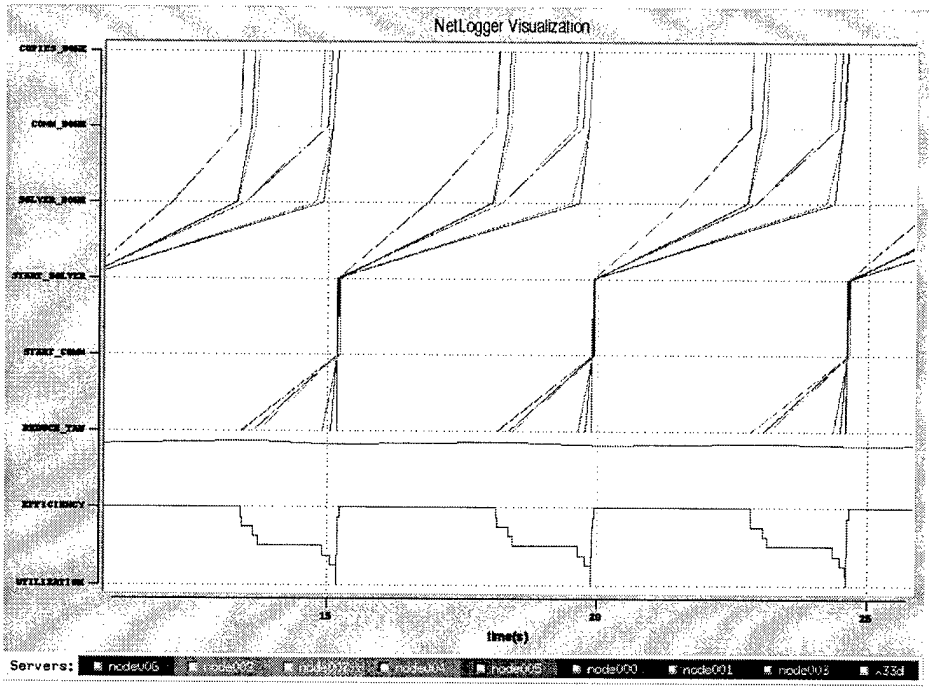


Figure 4. *ALSINS with overlapped communication/computation on the Pentium Cluster.*

Cluster Bandwidth, MB/sec.

Avg.	Node 0	Node 1	Node 2	Node 3	Node 4	Node 5	Node 6	Node 7
10.348	10.379	10.317						
8.076	10.225	6.957	7.047					
8.730	10.028	7.561	9.910	7.421				
7.869	9.979	7.941	6.675	8.135	6.616			
6.472	6.689	6.216	7.035	6.207	6.354	6.333		
6.778	8.355	7.276	6.483	5.868	7.194	6.445	5.822	
4.489	6.072	4.907	4.948	4.426	4.167	3.942	3.727	3.726

T3E Bandwidth, MB/sec.

Avg.	Node 0	Node 1	Node 2	Node 3	Node 4	Node 5	Node 6	Node 7
319.252	319.101	319.402						
253.973	253.161	253.259	255.498					
224.496	218.161	211.452	241.411	226.958				
194.584	194.335	190.279	219.954	183.446	184.905			
203.172	215.007	181.744	242.082	184.683	213.967	181.548		
222.200	231.971	193.002	282.610	198.681	238.679	217.750	192.702	
211.705	212.216	191.008	236.182	200.079	235.900	222.200	205.023	191.024

Table 1. Bandwidth tests for T3E and Cluster.

10-second, TCP "packet-blasting" test and measure the data volume sent. Gloperf also measures the number round-trips that can be made in a 10-second period. The sequence of hosts and test types (bandwidth and latency) are randomized in order. Since Gloperf does untuned TCP testing from the user-level, it essentially observes the same end-to-end performance that an application would see.

To extract the Gloperf data from the MDS, a simple program would periodically snapshot the MDS Gloperf data into log files. Scripts were then used to extract just the bandwidth and latency data and eliminate duplicates. Figure 5 shows histograms of Gloperf bandwidth and latency measurements on GUSTO beginning in August and continuing through October, 1999. This represents 18615 unique measurements between 3405 unique host pairs over 138 unique hosts; mostly in North America but including a few in Europe, Asia, and Australia. Note that these histograms employ log-sized bins that make the mode of the distributions much more evident. While it was not uncommon to observe bandwidths as high as 96 Mbits/sec., the median bandwidth for this distribution is 2.2 Mbits/sec. and the 90th percentile is 15.1 Mbits/sec. While the latency distribution has a much narrower (rhinokurtotic) mode, common latencies span three orders of magnitude. Here, the median is 57.35 msec. and 90% of the latencies are above 5.5 msec.

It is clear that in a grid environment that can include clusters and "big iron" machines, there can be a 3 to 4 order of magnitude dynamic range in the bandwidth and latencies available to an application.

5 Discussion and Implications

What are the implications of these observations for heterogeneous cluster performance and grid performance? The argument can be made that there is a much greater dynamic range in the the available communication bandwidths than there is in processor speeds; 3 to 4 orders of magnitude versus one half order of magnitude. We note that since the graphs in Figure 5 represent capacity that is shared among other non-Globus traffic, one could argue that in terms of a shared resource, processors could exhibit the same range of available cycles. A counter-argument, however, is that one typically has greater control over the compute resources rather than the network; even if one does not have complete control of the processors, the processors may be batch scheduled. Regardless of such arguments, in many cases there will be a significant differential between the available processor speeds and network speeds.

What is the implication of this processor-network differential? We note that processor speeds have been increasing according to Moore's Law (doubling every 18 months). Memory bandwidth, however, has been increasing much more slowly, by some estimates as little as 7% per year. To cope with this processor-memory differential, hardware designers have had use increasingly larger caches and to employ numerous techniques to overlap operations to hide latency, such as speculative execution, prefetching, and hardware multithreading. This has also motivated the research in Processing-In-Memory (PIM) architectures [9]

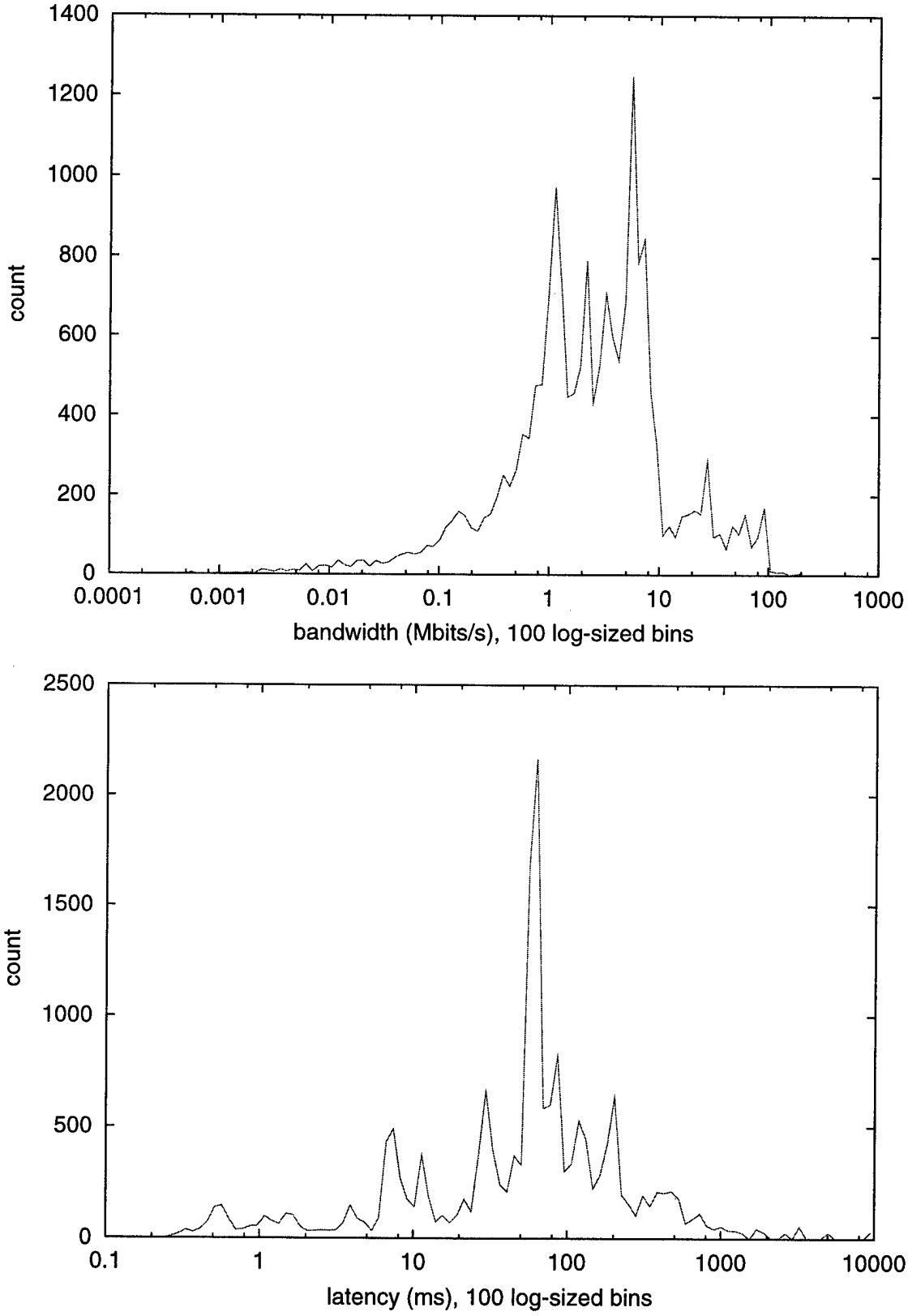


Figure 5. *Globus testbed bandwidth and latency distributions.*

where much higher bandwidths between memory and the processing units can be realized.

Fortunately network bandwidths seem to be increasing at least as fast as Moore's Law, if not faster, since around 1994 (A.W. – After Web). Unfortunately this improvement in bandwidth will not affect the speed of light. A significant part of latencies present in a grid is simply propagation delay. As an example, the end-to-end, application-level message-passing latency between Los Angeles and Chicago can already be as much as 33% propagation delay. Clearly this limits the reduction in latencies that are physically possible in a grid computing environment. Hence, in ten years time, we might expect the bandwidth distribution in Figure 5 to move to the right by an order of magnitude. While the possible relative reduction in latency will depend on the geographic separation among the compute resources, it is safe to say that many latencies in the latency distribution will not decrease as much. The bottom-line is that pipes will get fatter but not commensurately shorter.

How exactly will this relative change in bandwidths, latencies and processing speeds affect application performance? Work done by Martin, et al., is relevant to this question [11]. They examined the effect of latency, overhead and bandwidth on cluster performance. For this work, latency is defined as the end-to-end delay in sending a message from its source to its destination. Overhead is defined as the time that a processor is engaged in message transmission or receipt during which it cannot do anything else. Bandwidth is inversely defined in terms of the "gap" between consecutive message sends, i.e., messages per unit time.

For a set of applications on an UltraSPARC cluster using Myrinet A set of applications were run on an UltraSPARC cluster using Myrinet where the LANai processor on the Myricom network interface card was used to emulate a range of latencies, bandwidths and overhead. The applications in this experimental context were much more sensitive to overhead than to bandwidth or latency. For these applications, one is forced to conclude that since the additional per message overhead was unavoidable, it directly affected the application's running time, while at least part of the additional latency was naturally overlapped or hidden by the structure of the application. It was also shown that the applications had relatively modest bandwidth requirements compared to the dedicated network's capacity. Most applications did not slow-down significantly until the bandwidth was effectively reduced to approximately 12% of its normal capacity. Indeed, even the increased latencies in this cluster were well below those found in a grid and the reduced bandwidths were above the 90th percentile. In a general grid environment, these results would be different.

In the light of these considerations, the next question to ask is "How tightly coupled do distributed, heterogeneous grid applications need to be or can be?" Clearly not all applications are tightly coupled or need to be, in the sense that a CFD code is tightly coupled. Applications that connect unique resources, such as X-ray sources, with visualization devices, such as CAVEs, typically rely on a functional decomposition that is more tolerant of the dynamic range of bandwidth within a machine and between machines. Nonetheless, all distributed applications will run better with faster networks. This is not news. In the context of the World Wide Web, most people probably feel that downloads are too slow. In part, the notion of *quality of service* is to provide a "floor" to the performance that a user receives from a shared resource, e.g., a network.

The opinion is also held that *flexibility* is actually more important for grid applications than performance management. For a large class of applications, this will be true. The grid is being designed to make it as easy as possible to compose disparate resources such as specialized databases, unique instruments, and embedded systems. For another large class of applications, however, the grid holds the promise of applying very large amounts of aggregate compute power to very large problems that is not economically feasible any other way. Hence, what can be done to manage performance across these bandwidths and latencies?

Cluster computing can, again, be used as a point of departure. Several projects have been reported that deal with programming clusters of SMPs, or *clumps*, where the heterogeneity of in-memory communication vs. network communication is the central issue. The SIMPLE model [2], for example, provides a simple set of collective operations that are handled by different modules for intra-node and inter-node communication. KeLP and its Data Mover [5, 1] take a different approach. KeLP defines a set of meta-data abstractions, such as `Region`, `Map`, `FloorPlan` and `MotionPlan`, that capture the geometry of block-structured decomposition and the resulting data dependencies in parallel execution. The current Data Mover implementation uses a private MPI communicator and asynchronous point-to-point messages to actually move the data.

An important issue for communication libraries or runtime systems that support higher-level semantics, however, is that of *irregular* communication; communication that does not follow a regular, geometric pattern and may be dynamic and not known until run-time. This issue has been faced by the High Performance Fortran (HPF) community for some time. This has given rise to an *inspector-executor paradigm* where an inspector routine does a run-time analysis of array accesses for communication and derives a *communication schedule* that is then used by the executor routine to actually perform the communication. Since the inspector routine can be very time-consuming, there has been work done on minimizing its overhead and reusing any schedules produced [3].

For some applications, it will be best to use a program-

ming model that does not hide the heterogeneity of the underlying resources and requires the application builder to hand-code the application to the resources. There are, of course, great benefits in not having to hand-code applications to tolerate bandwidths and latencies. For these situations, dealing with a heterogeneous infrastructure means addressing the fundamental problems of (1) data locality and (2) scheduling, where scheduling in this context means both communication and execution scheduling which are, in fact, interdependent. A clearly attractive approach is to require sufficient concurrency in the application such that a coarse-grain, data-driven model of execution can be used. The initial challenge is to hide latency with concurrency while keeping context switching overheads low. The next challenge is to encapsulate this low-level performance management into an easy-to-use package or component. If this is possible, then other established techniques such as caching, compression, estimation and speculative pre-fetching, could also be used.

Finally we note that applications tend to have their own, natural "problem architecture" and some, by their very nature, are more tightly coupled than others. As soon as a distributed implementation is considered, it imparts a three-dimensional or spatial "density distribution" to the computation. This density and the available bandwidth and latency become part of the algorithmic complexity governing performance. Some applications will have unavoidable spatial constraints that will be best addressed by recasting the problem and its solution in a more loosely coupled fashion. The Barnes-Hut algorithm, for example, solves the N-Body problem in less than $O(n^2)$ complexity by representing space with an octree such that from any given body, groups of far away bodies can be represented as a point source.

The challenge for grids and heterogeneous computing, however, is to minimize the class of applications that have to be recast by developing systems and runtimes that understand the "spatial component" of an application and can act accordingly to provide the best overall performance with the available communication resources. This is one of the goals of the Grid Forum's Advanced Programming Models Working Group (www.gridforum.org). The need for such "spatial component" management will only increase as systems like long-latency satellite networks and low-power mobile networks come online with high-performance compute systems such as hardware multithreaded processors that tolerate deep memory hierarchies.

References

[1] S. Baden and S. Fink. The Data Mover: A machine-independent abstraction for managing customized data motion. *LCPC*, August 1999.

[2] D. Bader and J. JáJá. SIMPLE: A methodology for programming high performance algorithms on clusters of symmetric multiprocessors. Technical report, University of Maryland, Department of Computer Science, and The University of Maryland Institute for Advanced Computer Studies, 1997. Tech report, CS-TR-3798, UMIACS-TR-97-48.

[3] S. Benkner, P. Mehrotra, J. V. Rosendale, and H. Zima. High-level management of communication schedules in HPF-like languages. *International Conference on Supercomputing*, 1998.

[4] CESDIS. The Beowulf Project. Technical report, NASA, 1999. `http://www.beowulf.org`.

[5] S. Fink and S. Baden. Runtime support for multi-tier programming of block-structured applications on smp clusters. *International Scientific Computing in Object-Oriented Parallel Environments Conference (ISCOPE '97)*, December 1997. Available at www-cse.ucsd.edu/groups/hpcl/scg/kelp/pubs.html.

[6] S. Fitzgerald, I. Foster, C. Kesselman, G. von Laszewski, W. Smith, and S. Tuecke. A directory service for configuring high-performance distributed computations. In *Proceedings 6th IEEE Symp. on High Performance Distributed Computing*, pages 365–375, 1997.

[7] I. Foster and C. Kesselman. Globus: A metacomputing infrastructure toolkit. *Intl. J. Supercomputing Applications*, 11(2):115–128, 1997.

[8] I. Foster and C. Kesselman. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, 1998.

[9] M. Hall et al. Mapping irregular applications to DIVA, a PIM-based data-intensive architecture. *Supercomputing '99*, 1999.

[10] C. Lee, R. Wolski, J. Stepanek, C. Kesselman, and I. Foster. A network performance tool for grid environments. *Supercomputing '99*, 1999.

[11] R. Martin, A. Vahdat, D. Culler, and T. Anderson. Effects of communication latency, overhead, and bandwidth in a cluster architecture. *24th ISCA*, June 1997.

[12] NAS. The NAS Parallel Benchmarks. Technical report, NASA, 1999. `http://www.nas.nasa.gov/Software/NPB/index.html`.

[13] SGI. The Cray T3E Homepage. Technical report, SGI, 1999. `http://www.sgi.com/t3e`.

[14] B. L. Tierney. NetLogger: A methodology for monitoring and analysis of distributed systems. Technical report, Lawrence Berkeley Laboratory, 1999. `http://www-didc.lbl.gov/NetLogger`.

[15] J. Wang and S. Taylor. *Industrial Strength Parallel Computing*, chapter 10: An architecture-independent Navier-Stokes Code. Morgan Kaufman, Inc., 1999.

[16] J. Wang and G. Widhopf. An efficient finite volume TVD scheme for steady state solution of the 3-D compressible Euler/Navier-Stokes equations. In *AIAA paper 90-1523*, June 1990.

A Debugger for Computational Grid Applications*

Robert Hood Gabriele Jost

MRJ Technology Solutions—Numerical Aerospace Simulation Division
NASA Ames Research Center
{rhood,gjost}@nas.nasa.gov

Abstract

The Portable Parallel/Distributed Debugger project at the NASA Ames Research Center has built a debugger for applications running on heterogeneous computational grids. It employs a client-server architecture to simplify the implementation, and its user interface has been designed to provide process control and state examination functions on computations with a large number of processes. The debugger can find processes participating in distributed computations even when those processes were not created under debugger control. In addition to working in a computational grid environment, these techniques also work on other distributed memory jobs such as those initiated by mpirun.

1. Introduction

While tools for debugging computationally intensive programs have improved substantially in the last few years [5] [22], there are two areas where further improvement is needed. First, existing tools do not cope well with applications running on heterogeneous computing platforms. Second, they do not provide sufficiently abstract and scalable operations for examining and controlling execution.

This combination of inadequacies is particularly felt by programmers building applications to run on large-scale computational grids, such as NASA's Information Power Grid (IPG) [12]. The IPG is based on the Globus toolkit [7] and can give an application access to a variety of computing resources across the country. Debugging such a computation using existing techniques is at best very tedious. In the worst case, it may not be possible.

In order to provide a reasonable debugging system for computational grid computations, the Portable Parallel/ Distributed Debugger (*p2d2*) project in the Numerical Aerospace Simulation (NAS) Division of the NASA Ames Research Center has taken their existing debugger and extended its capabilities. The original goals of the project in 1994 were to build a debugger that was both portable across a variety of target machines and whose user interface scaled to be able to debug a large number of processes. (At that time we interpreted that to mean at least 256 processes.) The result of that initial effort was a debugger [10] that ran on a variety of Unix-based machines and could be used on both MPI [15] and PVM [21] applications.

In this paper we report on the effort to enhance that debugger to work in a computational grid environment. We begin with a discussion of how *p2d2*'s architecture accommodates the debugging of heterogeneous computations. In section 3 we look at user interface features that enhance scalability. Following that we discuss how *p2d2* meets the requirements imposed by a computational grid environment. In section 5 we examine how heterogeneity affects the user interface components.

2. An architecture to support heterogeneity

Debuggers, even serial ones, are inherently nonportable. Their basic task is to take a user request at the source level, map it to the machine level where it can be performed, and then map the result back to the source level. To accomplish this they rely on information and services from a variety of sources. For example:

- the compiler provides source line and symbol mapping data,
- the operating system provides services for starting and stopping processes, and
- the computer architecture defines a trap instruction that can be used for implementing breakpoints.

The most successful portable debugger, *gdb* from the Free Software Foundation [6], defines abstract interfaces for many low-level functions in a debugger, such as reading values in another process's address space. The *gdb* source distribution includes machine specific implementations for those functions, and at compile time it determines which code needs to be present to build a debugger for a given platform. One problem that gdb does not attempt to solve is that of debugging heterogeneous computations,

*This work was supported through NASA contract NAS 2-14303.

The screen dumps in this paper have been modified from their normal screen appearance in order to aid reproducibility. The modifications include color changes as well as a resizing of some components from their defaults.

0-7695-0556-2/00 $10.00 © 2000 IEEE

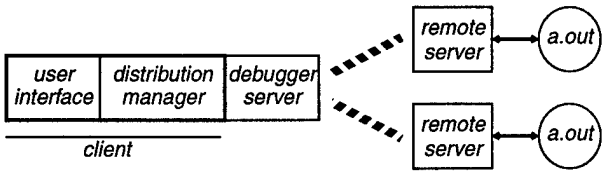


FIGURE 1. The client-server architecture of *p2d2*.

where these portability issues must be solved in a way that enables different target platforms to be available at the same time.

In the *p2d2* project we wanted to address the portability issues at a higher level of abstraction than *gdb* did. We used a client-server architecture to isolate the platform-dependent code in a *debugger server* (see Figure 1). The server defines a collection of C++ objects that would exist in a debugging session, such as `Process` and `Stack`. The client consists of those parts of the debugger that deal with the distributed nature of the target computation and with the user. It can be implemented in a highly portable fashion. For example, if the client has a `Process *p`, it could resume execution in it by invoking the operation

```
p->Continue().
```

The object collection is discussed in detail in a previous paper [11].

In the initial version of *p2d2* we decided to build a debugger server based on *gdb*. The main reason for this was that our debugger would be easily portable to any platform where a *gdb* implementation existed—thus saving us a huge amount of implementation effort. In the *gdb*-based implementation of *p2d2* (see Figure 2) the remote server of Figure 1 is replaced with an instance of *gdb*. The debugger server is then an implementation of the C++ objects that uses *gdb* commands to perform any requested debugger server requests. In effect, it maps operations on the C++ objects into *gdb* commands and then maps the *gdb* response back to the object level.

To continue the example above, if the client invokes a continue operation on a process with "`p->Continue()`" the server sends a "`cont`" request to the *gdb* controlling that process and marks the process as "running". When that process hits a breakpoint, *gdb* reports it to the debugger server which analyzes the reason for stopping and updates its own picture of the process state. In doing so, it may send other requests to *gdb*, such as "`where`" to find out what the runtime stack looks like. When it has completed its picture of the process state, it notifies the client that the process has stopped.

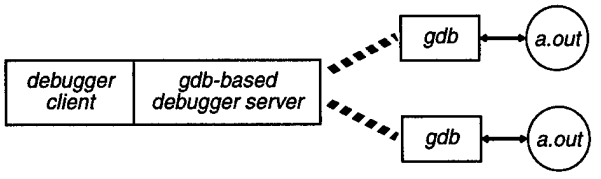


FIGURE 2. The *gdb*-based implementation of *p2d2*.

3. User interface basics

From a user's perspective, a debugger has two primary functions:

- *state examination*, where the user can scrutinize expression values, source code, run-time stack, and other components of the current computational state; and
- *process control*, where the user is permitted to start execution of the target computation and to describe circumstances under which it should stop.

The challenge in a multiprocess debugger is to provide these functions in a way that scales well to a large number of processes. In particular, the challenge for state examination is to provide both an abstract, top-level view of the computation as well as information about a single process that has the same level of detail that a serial debugger would have. The challenge for process control is to provide a way to propagate a single process control request, such as ***Continue***, to a collection of processes, thereby relieving the user from the burden of directing processes individually.

To address the state examination challenge, *p2d2* defines three zooming levels, providing a varying degree of abstraction versus detail.

- A top-level view, called the *process grid*, provides a programmable display showing a few bits of information about each of the processes in the computation.
- An intermediate-level view provides a line of text summarizing the state of each process in a user-selected set called the *focus group*.
- A low-level view provides full information about a single, user-selected *focus process*.

The selection of the focus group and the focus process are done in the process grid display. When the user changes one of the selections, the display is updated to reflect the information about the new focus. For example, if the user changes the focus process, then all state examination displays that are focus-process-sensitive, such as the source and stack displays, will be updated.

To address the process control challenge, *p2d2* uses the notion of a *control set*, which is the collection of target processes that are subject to process control requests. The user has a variety of ways of setting membership in the control set. The current membership of the control set is indicated in the process grid display. When a process control operation such as setting a breakpoint or continuing execution is requested, it is forwarded to all processes in the control set.

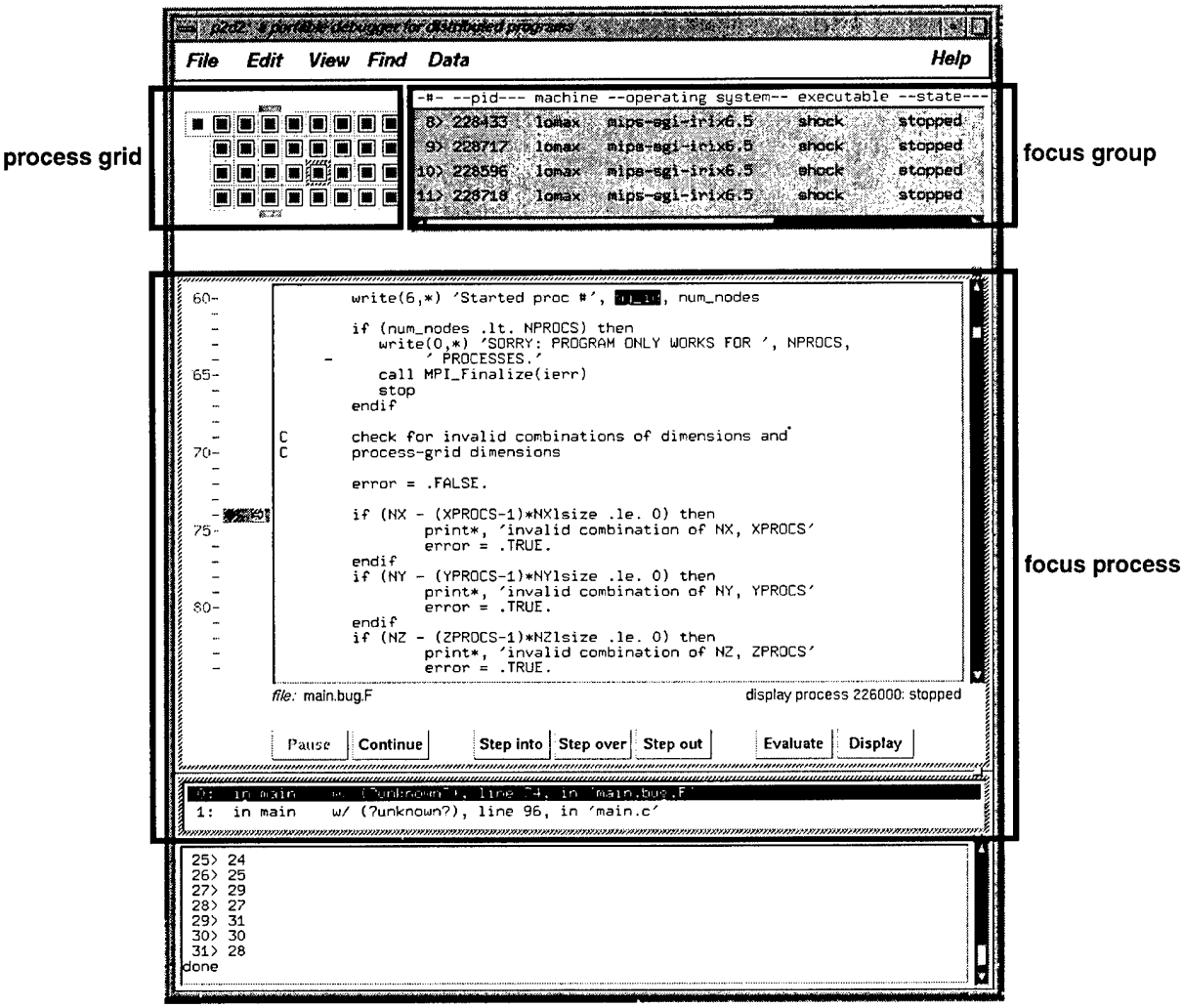


FIGURE 3. The main window of *p2d2*.

The main window of *p2d2* is shown in Figure 3. In that example, a Globus computation of 32 processes is being debugged. The process grid shows all of the processes in the computation, plus the *globusrun* process that initiated the job (but does not participate in the computation). The focus group displays one line of text for each process in the selected column of the process grid. In the figure, the user has selected the fourth of the nine columns. The focus process part of the display resembles a serial debugger on the single process selected in the process grid (the one in row 3, column 6).

Perhaps the most novel feature of the *p2d2* user interface is the programmability of the process grid described earlier. This feature permits a quick scan of a large number of processes to isolate a process behaving in an unexpected manner. Such a process is a good candidate for closer scrutiny as the focus process.

This customization is achieved by having the user specify a list of predicates that should be tested in each process and how a process should be depicted in the process grid if the predicate is true about it. For example, in the default view of the display, running processes are represented by green squares and stopped processes by red ones.

P2d2 defines a collection of conditions that can be tested. These include:

- the process is running,
- the process is on some machine M,
- an expression E evaluates to true in the process, and
- the process is stopped in user function F_1 and is calling non-user function F_2.

The customization feature is illustrated in Figure 4. In that example, the user suspected that a computation was deadlocked. She paused all of the processes and then requested that the process grid show where each process was stopped.

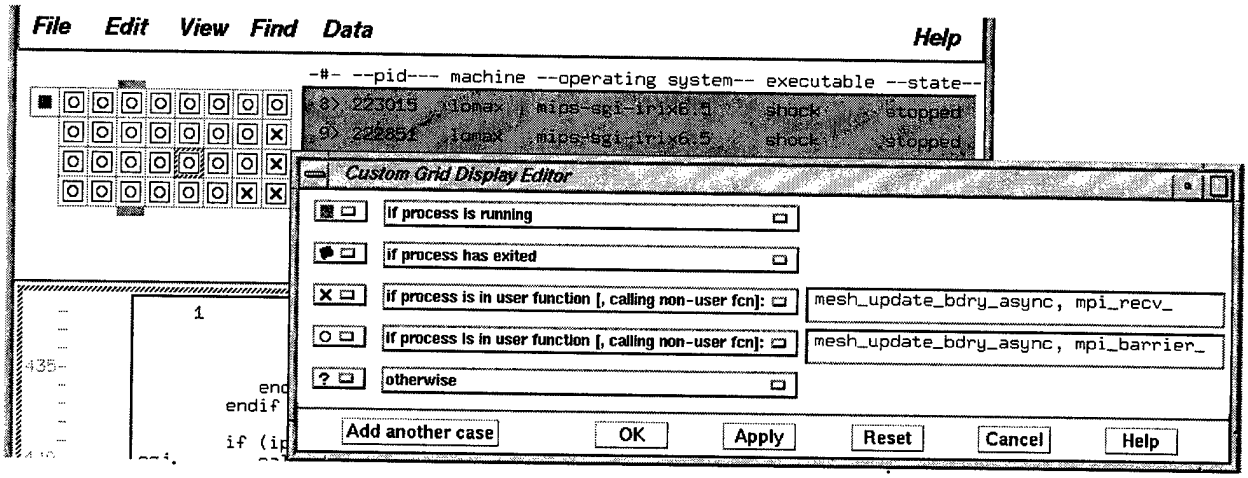


FIGURE 4. Customizing the process grid display.

The debugger constructed the customization shown in the "***Custom Grid Display Editor***" window. It shows running processes with a green square—only the *globusrun* startup process is in this category. Stopped processes are depicted with an "x" if they are in `mesh_update_bdry_asynch` and calling `mpi_recv`; they are depicted with a "o" if they are in that same function but calling `mpi_barrier`. This feature gives the user a quick way to find out what each process is doing. In particular, in this example, the user was able to focus in on the four processes doing an `mpi_recv`, and find a communication pattern error.

In addition to providing tools for abstracting state across a collection of process, *p2d2* also provides various means to examine specific data values in a computation. Scalar expressions can be evaluated on each process in the control set with the result being displayed in the output window (the bottom pane in Figure 3). As an alternative, a scalar data viewer allows the persistent display of scalar values for up to 4 focus processes (Figure 5). Data values there are updated each time a breakpoint is hit or when the focus is shifted to another process.

Array data can be examined using the *p2d2* array viewer. It displays the array in either textual or graphical mode for each of the focus processes. Figure 6 shows an example of the data displayed as text. As in the scalar viewer, the values are updated when the program reaches a breakpoint or when the focus is shifted to another process.

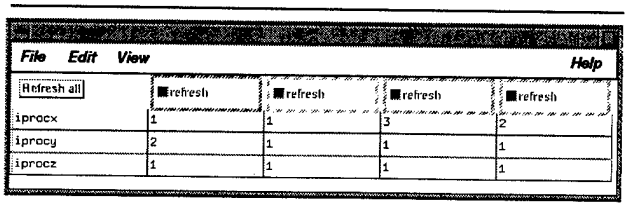


FIGURE 5. Comparing data across processes.

4. Handling grid-based computations

In addition to state examination and process control features, a successful debugger will need to automate the task of finding and controlling all of the processes participating in a distributed computation. The user should not be required to filter through lists of processes running on a large number of machines in order to determine which of them belongs to a job.

As with serial debuggers there are two cases to consider in acquiring initial control over processes to be debugged:

1. the computation was initiated from the debugger when the user invoked the ***Run*** command, and
2. the user initiated the computation outside of the debugger and then requested that the debugger "attach" to it.

In order to handle case 1, the debugger needs to resolve a conflict with the process starting mechanism (e.g., *mpirun*, *globusrun*, *pvmrun*) that initiates the distributed computation. The conflict comes about because both the debugger and the process starter want to control the actual `fork()` and `exec()` that start the individual processes. A customary way to resolve this conflict is for the process starting

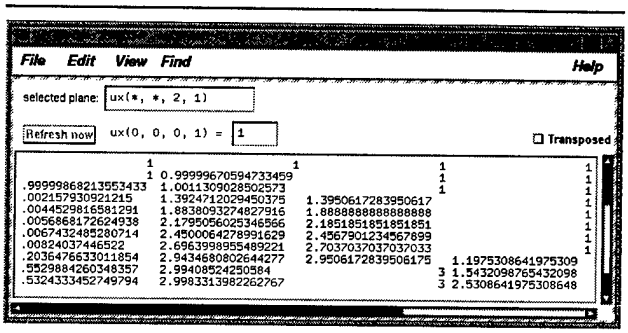


FIGURE 6. Array data viewed as text (compare with graphical view in Figure 11).

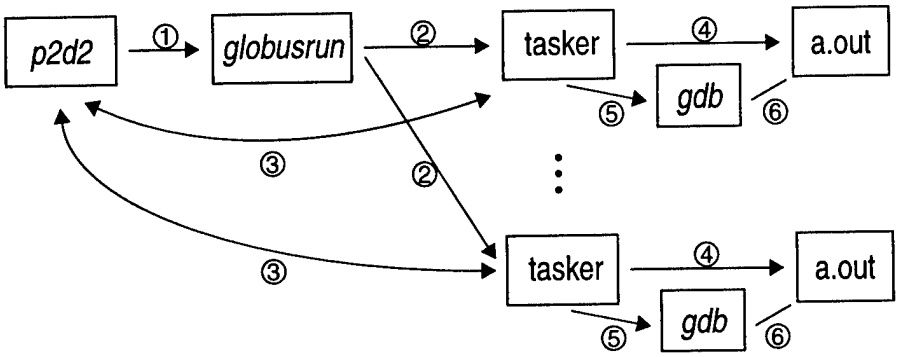


FIGURE 7. After the user requests a *globusrun*.

mechanism to allow a user-supplied proxy program (sometimes called a *tasker*) to perform the `fork` and `exec`. Both *pvmrun* and *globusrun* permit the debugger to gain control over process creation in this way.

To debug Globus jobs, *p2d2* uses the tasking mechanism provided by *globusrun*. If the debugger is going to be used to initiate a Globus job, the user must include the clause

```
(paradyn="P2D2_HOST P2D2_PORT p2d2 \
          /u/p2d2/bin/gdbserver")
```

in the RSL script to be handed off to *globusrun*. This indicates that `/u/p2d2/bin/gdbserver` should be used as a tasker. When the user requests a ***Run***, the following sequence of events happens. It is depicted in Figure 7.

1. *P2d2* invokes *globusrun*, changing the `P2D2_HOST` and `P2D2_PORT` strings in the RSL script to the machine name on which *p2d2* is running and the number of a tasker contact port that it created.
2. When *globusrun* starts the tasker, it passes it the machine name and port number that *p2d2* wrote in the RSL script.
3. The tasker and *p2d2* then establish a socket.
4. The tasker starts the target executable and reports the target's pid on the socket. The target sleeps.
5. *P2d2* asks the tasker to start *gdb* and to forward an `attach` request to it.
6. *Gdb* attaches to the target to take control.

In order to handle case 2 above, where the user requests that the debugger attach to an existing computation, the debugger needs:

- a list of the processes that are participating in a computation, and
- a mechanism for gaining control over them.

If a tasking mechanism exists, it can be used to meet these needs. For example, if *p2d2* is to be used to attach to an existing Globus job, the job must have been started with the "`paradyn`" option described previously. Then the following steps (illustrated in Figure 8) take place.

1. *Globusrun* creates a tasker process for each target process in the computation.
2. The resulting tasker processes will each create a port.
3. The port contact information from all taskers is combined in a single file in the file system.
4. When the user starts up *p2d2* and asks for it to attach to the Globus processes, the debugger will retrieve the tasker port contact information in the file.
5. *P2d2* will then establish sockets with the taskers.
6. The debugger will then ask the tasker to start up a *gdb* and pass an `attach` request to it.
7. *Gdb* will then attach to the target process.

Storing the tasker contact information in the file system can be problematic. The machine where *p2d2* runs may not mount the same file system that the taskers do. In fact, the

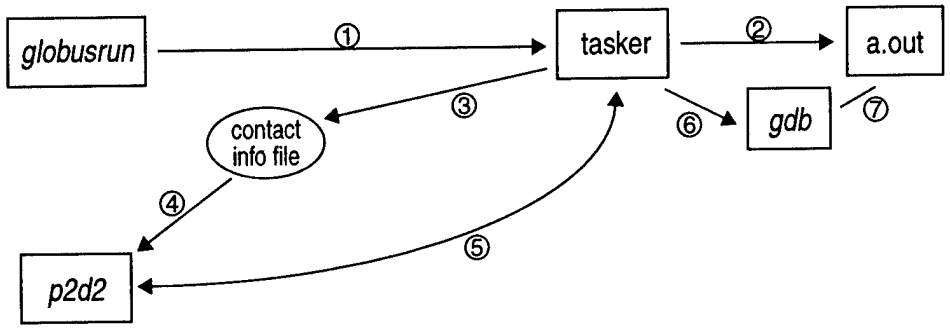


FIGURE 8. When *p2d2* attaches to an existing Globus process.

taskers themselves may not share a common file system. Under Globus, the right way for the taskers to get the contact information to *p2d2* is to use the Metacomputing Directory Service (MDS). We are currently modifying our tasker to use that approach.

In our discussions so far, we have relied on a tasking mechanism at process startup. Unfortunately the initial version of MPI does not have such a feature, because process creation was not part of the standard. To handle MPI jobs when there is no tasking mechanism, *p2d2* uses *rsh* to run a copy of *gdb* on the machine where the target process exists. There are two remaining needs:

- a list of pairs [*machine*, *pid*] for each process in the job, and
- a way to keep a newly started MPI process from executing code.

The second condition allows us to handle debugger-initiated runs in an identical manner to run initiated outside of the debugger. To handle a ***Run*** request in this scenario, *p2d2* invokes *mpirun*, which starts the processes on the remote machine. If we have a way to keep the newly started MPI process from making progress, we can simply attach to it as we do for runs initiated outside of the debugger.

We can address both of the needs above by using the profiling mechanism of MPI and providing a specialized version of `MPI_Init()`. The `MPI_Init` used by *p2d2* does the following.

- It calls `PMPI_Init()`, to do the normal initialization for MPI.
- The process with rank 0 gets the machine name and process ID for all processes. It writes that data in the file system.
- If the process was initiated from the debugger, it goes into an infinite sleep loop.

When the debugger attaches, it establishes any necessary breakpoints, terminates the sleep loop, and then continues execution.

There are two minor limitations in the version of `MPI_Init` used by *p2d2*:

- it is not possible to debug the code that executes before `MPI_Init` called, and
- the user must link the application with *p2d2*'s version of `MPI_Init`.

The latter condition could lead to a conflict if other libraries want to use the profiling mechanism of MPI.

While these limitations exist, in practice they restrict *p2d2*'s capabilities very little. Furthermore, we are hopeful that an *mpirun* based on the process control operations in MPI-2 [15] will provide a tasking mechanism that will eliminate the restrictions altogether.

5. Heterogeneity and the user interface

In adapting *p2d2* to work in a heterogeneous computational grid environment, we found two areas that needed more work:

- displaying what kind of machine and operating system a process was running on, and
- providing abstract, consistent views of data across heterogeneous processors.

The first problem was relatively easy to solve. *P2d2* extracts system type information from its debugger servers and then displays it in two different ways, as shown in Figure 9. First, it puts system information in the focus group display. Second, it defines a predicate "process is on operating system *S*" so system information can be displayed in the process grid. In the example shown in Figure 9, the grid view is programmed so that processes running on IRIX are depicted as

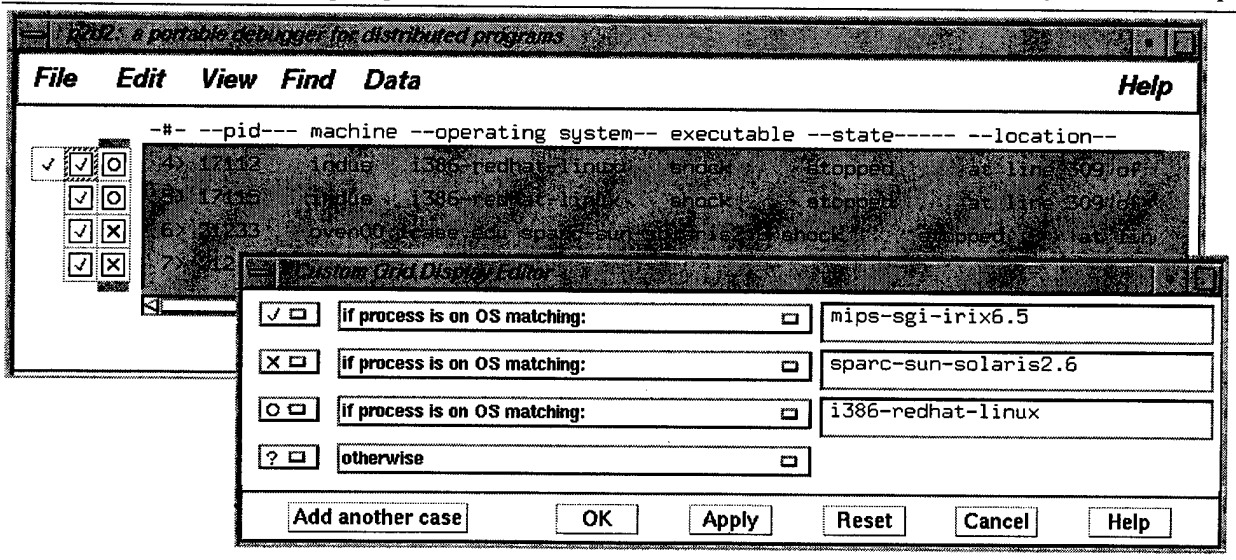


FIGURE 9. Support for heterogeneity in the process grid.

a "√", processes running on Solaris are indicated with an "x", and processes running under Linux show a "o". This results in the process grid view as shown.

To address the second problem, that of providing consistent, abstract representations of program state across heterogeneous processes, we needed to make the existing state examination tools more robust and to provide some new ones as well. One of the problems we ran into when first looking at heterogeneous computations concerned providing automatic assistance for comparing the value of an expression in processes on different architectures. Data representation was not an issue because *gdb* provides the expression's value as text. Instead the issue was that of finding where in the process the evaluation should take place.

Expression evaluation in *p2d2* has always tried to make sure that the user compares apples to apples. That is, when evaluating an expression on more than one process, the debugger attempts to use the same context in each of the processes doing the evaluation. So, if a variable is being evaluated in 2 processes, the evaluation will take place in stack frames that are "similar". What this means is that the debugger needs to compare the runtime stacks of the non-focus processes in order to determine which frame best corresponds to the selected frame in the focus process. In a homogeneous environment this is not too difficult. The problem we needed to address in a heterogeneous environment was that the runtime stacks looked somewhat different. In particular, function names often changed slightly. We addressed the problem by mapping function names to a canonical form. Then stack comparison could be handled as in the homogeneous case.

In order to increase the abstraction level of our data displays, we wanted to address the issue of displaying data from arrays that are conceptually distributed across multiple processes. Thus, *p2d2*'s array viewer provides a mechanism to give the user a global picture of a distributed array. The local data contributions from each of the participating processes are gathered and assembled into a global picture. When gathering the data from different machine architectures we had to take into account inconsistencies of *gdb* across different compilers. An example is the "`whatis`" command. For a Fortran array declared as `real a(10,5)` on a Linux platform using *g77* this results in `type = real*4 (10,5)`. On a SGI Origin using the MIPSPro compiler it results in `type = real*4 (5,10)`. In this case, *p2d2* addresses the differences by reversing dimension lists on the SGI's.

Figure 10 shows a global display of a 2-dimensional slice of the 4 dimensional array `ux` at a breakpoint. The array `ux` is distributed across 8 processes: 4 SGI Origins, 2 processes on a Sun Solaris platform, and 2 processes on a Linux PC cluster. The array elements that reside on the focus process are highlighted. To make comparison simpler, Figure 11 shows the local contribution from the focus process.

In order to assemble the local contributions of a distributed array into a global picture, information about how the data is distributed is required. If the program has been parallelized without the use of parallelization support tools, *p2d2* will prompt the user to provide distribution information *via* a dialog box (Figure 12). At the moment only simple, structured distribution types are supported.

In cases where the program has been parallelized using a parallelization support tool, it is often possible to retrieve such information through the tool that has been used. Currently *p2d2* supports the CAPTools [3] parallelization tool, which was developed at the University of Greenwich. CAPTools generates parallel code from a serial program by performing extensive dependence analysis, logically partitioning the data, and inserting calls to communication routines. The analysis results gathered during this process are stored in a data base, which is then probed by the debugger to retrieve the required distribution information without user intervention. Some of the information stored in the CAPTools database is symbolic and has to be evalu-

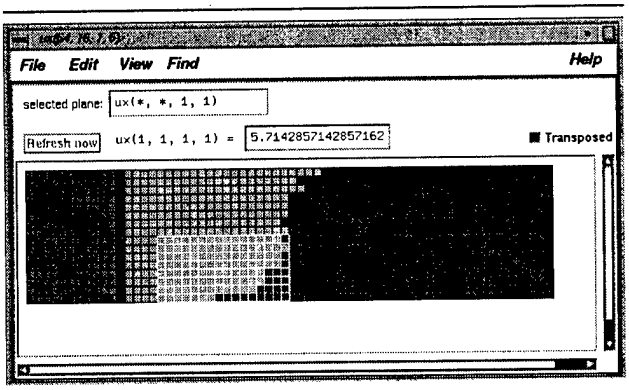


FIGURE 10. A global view of distributed data.

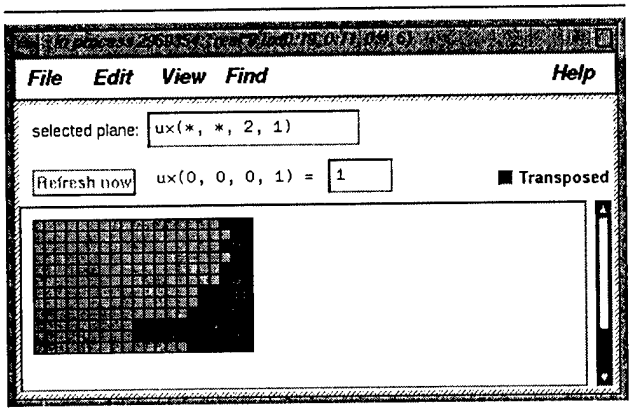


FIGURE 11. Local array data.

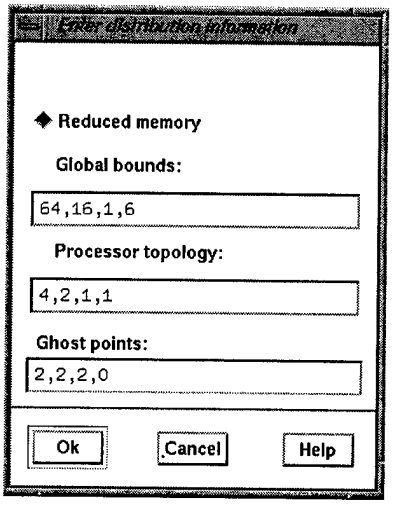


FIGURE 12. Specifying a distribution.

ated by *p2d2* for each processor at run time. For example the upper and lower loop bounds, which determine the effectively used area in a local array, are stored symbolically. These bounds vary with the number of processors and are potentially different for each processor.

6. Related work

There are two commercially available distributed debuggers of note. *TotalView* [5], from Etnus, is a third party debugger that runs on a number of high performance computing platforms. It is currently not capable of debugging heterogeneous computations. Furthermore, while it can debug thousands of processes and threads, the user interactions are at a fairly low level. *Prism* [22], from Sun Microsystems is derived from the Thinking Machines product of the same name. It is not portable to systems other than Sun. While its user interface led the way in scalability, it too, could be more abstract.

SGI's *Jessie* [19] is a freely available, cross platform development environment that provides a debugger based on *gdb* and a performance analysis tool based on *gprof*. Like *p2d2*, Jessie is aimed at providing portability. When it comes to debugging programs consisting of multiple processes, Jessie is limited to what *gdb* supports. That means while it is possible to invoke several instances of *gdb* to debug multiple processes on different machines, to our knowledge Jessie, at this time, does not provide means to control them in a convenient, scalable way.

Guard is a debugger developed at Griffith University, Australia [1][2]. It provides the ability to debug programs in a distributed and heterogeneous environment by allowing control of execution of separate programs on different machines. Like *p2d2*, it uses a client-server paradigm to provide portability. A *gdb*-based debugger server runs on each of the machines to control the processes. The debugger servers communicate with the client via RPC. Guard provides a command language for user interaction that contains commands like "compare" and "assert" to compare values between programs that are running on different machines, and were possibly written in different languages. It also allows the comparison between parallel and sequential versions of a program by providing language constructs that enable the user to map a serial data structure onto the equivalent parallel version.

The Distributed Array Query and Visualization (*DAQV*) project [9] aims to provide a solution for the problem of exposing distributed data structures to external tools. The original work started as a Parallel Tools Consortium [16] project and focused on HPF as a target language. Information about the distributed array could be obtained via the HPF compiler. In the second phase of the project (*DAQV-II*), Fortran 90 and MPI became the primary implementation targets [8]. As in *p2d2*, DAQV-II requests array distribution information from the user if it can not be obtained otherwise.

The *SPiDER* debugging system [20] for HPF programs uses the *GDDT* (Graphical Data Distribution Tool) [13] for the display of distributed arrays. It doesn't appear to support viewing arrays distributed across a heterogeneous collection of machines.

7. Project status and future work

The current *p2d2* system has been demonstrated on several target architectures and has been used to debug both MPI and PVM applications. After the recent work to accommodate Globus computations, it has been successfully used to control 128 processes running on 3 different SGI Origins on the IPG. It has also been used on heterogeneous computations running under Globus (see Figure 9).

At the time of writing this paper, we have requested permission from NASA to distribute *p2d2* under an Open Source copyright [17]. Those desiring up to date information about the status of that distribution are requested to consult the *p2d2* web site [18].

In the near future, we will start using the Metacomputing Directory Service (MDS) in Globus to record information about jobs started outside the debugger. This will enable us to attach to Globus computations without relying on the target systems sharing a file system with the debugger host.

Further in the future we may adapt *p2d2* to work with Legion [14] and Condor [4] if there is sufficient user demand. We also plan to enhance *p2d2* to find differences between serial and distributed versions of the same code. This could be particularly useful when computer-aided par-

allelization tools such as CAPTools are used to perform the transformation.

8. Conclusions

In this paper we have described a debugger for heterogeneous, distributed programs. We found that a client-server model greatly simplifies the implementation of a debugger. The debugger's user interface has been designed to provide a simple collective mechanism for process control, as well as multiple levels of zooming for state examination. These features facilitate the debugging of a computation containing a large number of processes. We also described several approaches for finding processes participating in a distributed computation and how those techniques could be used in a computational grid environment.

Acknowledgements

The authors would like to thank Michael Frumkin and Warren Smith of NAS for their comments on this paper. Warren also provided valuable help regarding our use of the Globus system. Henry Jin of NAS and Steve Johnson of the University of Greenwich helped with retrieving data distribution information from the CAPTools database. Ravi Samtaney from the California Institute of Technology provided a parallel version of the RM3d code for the solution of Euler's equations in three dimensions, which ran in a heterogeneous fashion under Globus.

References

[1] Abramson, D., Sosic, R., and Watson, G. "Implementation Techniques for a Parallel Relative Debugger." *Proceedings of PACT'96*, Boston, October 1996.

[2] Abramson, D. and Watson, G. "Relative Debugging for Parallel Systems." *Proceedings of PCW 97*, September 25-26, 1997, Canberra, Australia.

[3] Computer Aided Parallelization Tools (CAPTools). `http://captools.gre.ac.uk/`.

[4] The Condor Project. `http://www.cs.wisc.edu/condor/`.

[5] Etnus, Inc. The TotalView Multiprocess Debugger. `http://www.etnus.com/products/totalview/`.

[6] The Free Software Foundation. `http://www.fsf.org/`.

[7] The Globus Project. `http://www.globus.org/`.

[8] Hackstadt, S. T., Harrop, C. W., and Malony, A. D. "A Framework for Interacting with Distributed Programs and Data." *Proceedings of the Seventh IEEE International Symposium on High Performance Distributed Computing (HPDC-7)*, Chicago, IL, July 28-31, 1998, pp. 206-214.

[9] Hackstadt, S. T. and Malony, A. D. "Distributed Array Query and Visualization for High Performance Fortran." *Proceedings of Euro-Par '96*, Lyon, France, August 1996.

[10] Hood, R. "The *p2d2* Project: Building a Portable Distributed Debugger." *Proceedings of the SIGMETRICS Symposium on Parallel and Distributed Tools*, May 1996.

[11] Hood, R. and Cheng, D. "Accommodating heterogeneity in a debugger—a client-server approach." *Proceedings of the Twenty-eighth Annual Hawaii International Conference on System Sciences*, Jan. 1995. [Also published in an extended form as a chapter in *Tools and Environments for Parallel and Distributed Systems*, Kluwer Academic Publishers, Norwell, MA.]

[12] The Information Power Grid Project Plan. `http://www.nas.nasa.gov/~wej/IPG/`.

[13] Koppler R., Grabner S., and Volkert, J. "Visualization of Distributed Data Structures for HPF-like Languages." *Scientific Programming*, special issue: *High Performance Fortran Comes of Age*, Vol. 6, No. 1, pp. 115-126, spring 1997.

[14] The Legion Project. `http://legion.virginia.edu/`.

[15] Message Passing Interface. `http://www-unix.mcs.anl.gov/mpi/`.

[16] The Parallel Tools Consortium. `http://www.ptools.org`.

[17] Open Source. `http://www.opensource.org`.

[18] The *p2d2* Project. `http://www.nas.nasa.gov/Tools/p2d2`.

[19] SGI, Inc. The Jessie Cross Platform Integrated Development Environment. `http://oss.sgi.com/projects/jessie/`.

[20] The SPiDER debugging system. `http://www.par.univie.ac.at/~sowa/spider`

[21] Sunderam, V. "PVM: A Framework for Parallel Distributed Computing." *Concurrency: Practice and Experience* 2(4):315-339, 1990.

[22] Thinking Machines Corporation. *Prism User's Guide*. Thinking Machines Corporation, Cambridge, MA, Dec. 1991; also: `http://www.sun.com/servers/hpc/software/configuration.html#prism`.

Biographical sketches

Robert Hood joined the faculty of Rice University in 1982. He participated in the R^n and ParaScope research projects, concentrating on debugging issues. After ten years at Rice, he took a position with Kubota Pacific Computer. In 1993 he joined the contract staff of the Numerical Aerodynamic Simulation (NAS) division at the NASA Ames Research Center and has been there since. Robert Hood's professional interests are in debuggers for parallel and computationally intensive programs, programming environments, and advanced compilation systems. Most recently, he has been leading the effort to build *p2d2*, a portable, scalable debugger.

Gabriele Jost received her doctorate in Numerical and Applied Mathematics from the University of Goettingen, Germany, in 1986. In 1987 she joined the Suprenum project, a German effort to develop and market a vector-parallel supercomputer. She has been working in the high performance computing industry with vendors and research institutes ever since. Her experience includes compiler development; performance optimization of scientific and engineering applications for cache-based, vector, and parallel computer architectures; and tools and programming environments for high performance computing. Since November of 1998 Gabriele Jost has been a member of the NAS Parallel Tools Team, where she works on the design and implementation of *p2d2*.

SESSION 4-B
RESOURCE MANAGEMENT

Chair: P. Stelling, *The Aerospace Corporation, USA*

A Framework for Mapping with Resource Co-Allocation in Heterogeneous Computing Systems

Ammar H. Alhusaini * and Viktor K. Prasanna*
Department of EE-Systems, EEB 200C
University of Southern California
Los Angeles, CA 90089-2562
Ph: (213) 740-4483
{ammar + prasanna}@usc.edu

C.S. Raghavendra
The Aerospace Corporation
P. O. Box 29257
Los Angeles, CA 90009
Ph: (310) 336-1686
raghu@aero.org

Abstract

It is often the case in Heterogeneous Computing (HC) systems that an application requires multiple resources of different types to be allocated simultaneously. In general, this problem is the resource co-allocation problem. In this paper, we develop a general framework for mapping a collection of applications with resource co-allocation requirements. In our framework, application tasks have two types of constraints to be satisfied: precedence constraints and resource sharing constraints. We use a graph theoretic framework to capture these constraints. A Directed Acyclic Graph is used to represent precedence constraints of tasks within an application and a Compatibility Graph is used to represent resource sharing constraints among tasks of applications. Both these graphs are used to find maximal independent sets of tasks that can be executed concurrently.

The objective of the mapping is to minimize the overall schedule length for a given set of applications. We develop heuristic algorithms to solve the mapping problem with resource co-allocation constraints. We also provide a two-phase algorithm that can be used for run-time adaptation. We conducted extensive simulation experiments to evaluate the performance of our heuristic algorithms. Simulation results for our algorithms show a performance improvement of 10% to 30% over a baseline algorithm of list scheduling which considers only the precedence constraints and allocates tasks from the resulting order. This paper demonstrates the importance of considering the co-allocation requirements when mapping applications in heterogeneous computing environments including grid environments.

*Supported by the DARPA/ITO Quorum Program through the Naval Postgraduate School under subcontract number N62271-97-M-0931.

1. Introduction

In *Heterogeneous Computing* (HC) systems [8, 13, 20, 25, 26], a diverse set of resources are used in a coordinated and effective way to solve computationally challenging applications. Such systems are also called *metacomputing* systems [29] or *computational grids* [10]. In general, such HC systems have compute resources with different capabilities, input/output devices, data repositories, and other resources all interconnected by heterogeneous local and wide area networks. A major challenge in using HC systems is to effectively use all the available resources.

Mapping applications in HC system is a well researched problem in the literature. The mapping problem is defined as the problem of assigning application tasks to suitable resources (matching problem) and ordering task executions in time (scheduling problem) to optimize a specific objective function. Many algorithms exist for mapping applications in HC systems (for a detailed classification see [4]). For applications consisting of several tasks and represented by Directed Acyclic Graphs (DAGs), many static and dynamic mapping algorithms have been proposed. Dynamic algorithms include the Hybrid Remapper [23], the Generational algorithm [12], as well as others [1, 18, 21]. Several static algorithms for mapping application DAGs in HC systems are described in [19, 24, 27, 32]. Most of the previous algorithms focus on compute resources only.

In our earlier work [2], we developed a unified resource scheduling framework for HC systems which supports multiple resource requirements, advance reservation, and data replication. Each application was assumed to consist of several tasks and was represented by a DAG. A task's input data can be data items from its predecessors and/or data sets from data repositories. Input data sets can be accessed from one or more data repositories. Sources of input data and the execution times of the tasks on various machines along with

0-7695-0556-2/00 $10.00 © 2000 IEEE

their availability were considered simultaneously to minimize the overall completion time. Although we considered multiple resource requirements in [2], tasks were not required to access different types of resources simultaneously.

In this paper, we consider the problem of mapping a set of applications in a HC system where application tasks require *concurrent access* to multiple resources of different types. In general, this problem is the *resource co-allocation* problem. For example, an interactive data analysis application may require simultaneous access to a storage system holding a copy of the data, a supercomputer for analysis, network elements for data transfer, and a display device for interaction [11]. For such applications, co-allocation of all required resources is necessary. A special case of this problem where a single application requires concurrent access to a set of resources in a computational grid has been considered in [5].

In this paper, we develop a general framework for mapping with resource co-allocation in HC systems. The framework defines the system and application models and formulates the co-allocation problem. Two graphs are used to represent applications: a Directed Acyclic Graph (DAG) and a *Compatibility Graph* (defined in Section 3.4). DAG representation is given initially and stay unchanged throughout the mapping process while the compatibility graph is updated during the mapping process. In classical mapping problems, only DAGs are used to represent the precedence constraints among tasks. In this paper, the co-allocation requirements add another type of constraint among the tasks: the resource sharing constraint which is captured in the compatibility graph. Tasks that share one or more resources cannot be executed concurrently due to the resource sharing constraints even if they have no precedence constraints among them. Known mapping algorithms for the classical DAG scheduling problem cannot be directly used for the above problem since they only consider the precedence constraints. In this paper, we develop heuristic algorithms that can be used with different allocation techniques to efficiently solve the co-allocation problem defined by our framework.

In our approach, multiple DAGs of different applications are combined into a single DAG. All tasks that have satisfied the precedence constraints are ready for allocation provided they have no resource sharing constraints. Using the compatibility graph, we will select tasks that can be executed concurrently. This is achieved by finding maximal independent sets in the compatibility graph.

Our research is part of the MSHN project [16], which is a collaborative effort between DoD (Naval Postgraduate School), academia (NPS, USC, Purdue University), and industry (NOEMIX). MSHN (Management System for Heterogeneous Networks) is designing and implementing a Resource Management System (RMS) for distributed heterogeneous and shared environments. MSHN assumes heterogeneity in resources, processes, and QoS requirements. Processes may have different priorities, deadlines, and compute characteristics. The goal is to schedule shared resources among individual applications so that their Quality of Service (QoS) requirements are satisfied. MSHN supports adaptive applications that can exist in several different versions. These versions may differ in the precision of computation or input data, and therefore have different values to a user. Unlike other HC and grid projects, MSHN seeks to determine how to meet QoS requirements of multiple application simultaneously.

The rest of this paper is organized as follows. In next section we give the definition of the co-allocation problem and summarize some related work. The problem framework is defined in Section 3. In Section 4, we give the outline of our approach to solve the co-allocation problem using our framework. Experimental results are given in Section 5. Finally, Section 6 gives the conclusions and future research directions.

2. The Co-Allocation Problem

The co-allocation problem can be defined as the problem of simultaneously allocating multiple resources of different types to applications in order to meet specific performance requirements. The need of co-allocation is a common characteristic of applications running in HC environments (as well as computational grids). For example, an application may require a data repository, a HPC platform, multiple display devices, and communication links all to be allocated simultaneously.

A version of resource co-allocation has been introduced in the high-performance distributed computing community by the Globus project [5]. The co-allocation problem is defined as the provision of allocation, configuration, and monitoring/control functions for the resource ensemble required by a single application [5]. The Globus tool-kit provides a flexible set of co-allocation mechanisms that can be used to construct application-specific co-allocation strategies. Only compute resources are considered in the Globus project at this time, to synchronize the start of complex applications at multiple sites.

The notion of co-allocation was also considered in the Legion project [22]. In the Legion project, an *Enactor* provides a mechanism to co-allocate compute and storage resources (hosts and vaults) to a single application. The co-allocation mechanism is based on advance resource reservation.

In [5] and [22], the focus is on implementation issues of the co-allocation process. Algorithms for efficient mapping with co-allocation requirements are not considered. Also, both the above projects focus on executing a single application. The problem becomes challenging when a number

of applications share resources.

In this paper, we study the co-allocation problem in the context of mapping a set of applications where each application is represented by a DAG. We consider conflicts among tasks caused by precedence constraints as well as due to resource sharing. The objective is to minimize the overall schedule length for a set of applications. One of our main contributions in this paper is the formulation of the mapping problem in the presence of co-allocation requirements for multiple applications. To the best of our knowledge, this work is the first step towards a general framework for mapping applications with resource co-allocation in HC systems.

3. The Framework

3.1 System Model

We consider a heterogeneous computing system with m compute resources (machines), $M = \{m_1, m_2, \ldots, m_m\}$, and a set of r resources, $R = \{r_1, r_2, \ldots, r_r\}$. Compute resources can be HPC platforms, workstations, personal computers, etc. Resource $r_k \in R$ can be a data repository, an input/output device, etc. We assume that only one task can use any resource (compute and non-compute resource) at any given time. Resources are interconnected by heterogeneous communication links. Communication costs are given by two matrices: MM_comm and RM_comm, where MM_comm gives the communication cost for transferring a byte between machines and RM_comm gives the communication cost for transferring a byte between the resources in R and the machines.

We assume that an estimate of the computation time of a given task t_i on machine m_j is available at compile-time. These estimated computation times are given in an *Estimated Computation Time* (ECT) matrix. Thus, $ECT(t_i, m_j)$ gives the estimated computation time for task t_i on machine m_j. If task t_i cannot be executed on machine m_j, then $ECT(t_i, m_j)$ is set to infinity.

$MA(m_j)$ gives the earliest time when machine m_j is available and $RA(r_k)$ gives the earliest time when resource r_k is available. As the mapping proceeds, the earliest time when a resource (m_j or r_k) is available is calculated as the finish time of the last task assigned to this resource.

3.2. Application Model

In this HC system, a set of N applications, $A = \{A_1, \ldots, A_N\}$, compete for system resources. Each submitted application consists of several tasks and is modeled by a DAG, where the nodes represent computational requirements and the edges represent both precedence constraints and communication requirements. Figure 1 shows

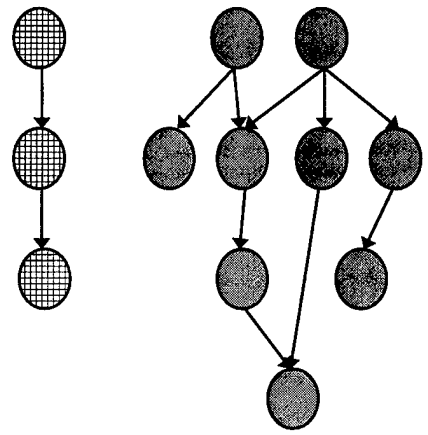


Figure 1. An example of two application DAGs

an example of two application DAGs. We assume that the whole set of applications to be mapped is known *apriori* (static applications). The problem is to execute these N applications as efficiently as possible. Our approach is to combine all submitted application DAGs into a single DAG, $G = (T, E)$, where T represents the set of tasks to be executed from all applications, $T = \{t_1, t_2, \ldots, t_n\}$, and E represents the data dependencies and communication between tasks. Edge e_{ij} indicates that there is communication from task t_i to t_j and its weight denotes the amount of communication. G is constructed by connecting the root nodes (tasks) of all applications to a hypothetical zero-cost entry node with zero-weight edges.

We assume that each task t_i needs *concurrent access* to a set of resources: one compute resource m_j and a number of additional resources as specified by the set $R(t_i)$, where $R(t_i) \subseteq R$. The amount of data to be transferred between t_i and r_k, where $r_k \in R(t_i)$, is given by $DATA(t_i, r_k)$. A task t_i cannot start execution until all its required resources are available to the task. All required resources will be allocated to the task during its execution. These resources will be available after the task completes its execution. We assume that all required resources are acquired at the same time (atomic transaction).

We say that task t_i and task t_j are *incompatible* if and only if $R(t_i) \cap R(t_j) \neq \phi$. Incompatible tasks cannot be executed concurrently even if they have no precedence constrains among them. Therefore, in our framework, tasks may be unable to run concurrently for either of the following reasons: (1) precedence constraints, or (2) resource sharing constraints.

The execution time of task t_i on machine m_j, $Exec(t_i, m_j)$, depends on the computation time of t_i on m_j and data transfer times between m_j and all resources which t_i needs to access during its execution. For example, for systems that assume computation and communication cannot be overlapped, $Exec(t_i, m_j)$ can be defined as

$$Exec(t_i, m_j) = ECT(t_i, m_j) + \sum_{r_k \in R(t_i)} (DATA(t_i, r_k) \times RM\_comm(r_k, m_j))$$

where the last term gives the total time to transfer any required data between machine m_j and every resource $r_k \in R(t_i)$. $Exec(t_i, m_j)$ can also be defined in different ways to consider the overlapping between computation and communication as well as other communication models.

The average execution time of task t_i is defined as

$$\overline{Exec(t_i)} = \sum_{j=1}^{m} Exec(t_i, m_j)/m$$

$ST(t_i, m_j)$ and $FT(t_i, m_j)$ are the earliest *start time* and the earliest *finish time* of task t_i on machine m_j, respectively if t_i were to be mapped on m_j. $ST(t_i, m_j)$ is defined as

$$ST(t_i, m_j) = \max\{MA(m_j), Data\_Pred(t_i, m_j)\}$$

where $Data\_Pred(t_i, m_j)$ is the time when task t_i receives all the needed data from all tasks in its predecessor set, $Pre(t_i)$, if t_i is mapped onto machine m_j. $FT(t_i, m_j)$ is defined as

$$FT(t_i, m_j) = ST(t_i, m_j) + Exec(t_i, m_j)$$

3.3. Mapping Objective

The objective function in our framework is to determine an assignment (matching) of tasks to compute resources and schedule their executions based on all resource requirements such that the overall schedule length (or *makespan*) of all submitted applications is minimized while satisfying all

1. Application-specified precedence constraints and
2. Implied resource sharing constraints.

Thus, we can define our objective function as

$$Minimize\ \{\max_{i=1}^{N} [Finish\ Time(A_i)]\ \},$$

where $Finish\ Time(A_i)$ is the completion time of application A_i. Note that the resource sharing constraint is a dynamic constraint - it depends on tasks ready to be allocated and their resource requirements.

Task	Resource Requirements
V_1	r_1, r_2
V_2	r_2, r_3
V_3	r_3, r_5
V_4	r_1, r_4
V_5	r_4, r_5, r_6
V_6	r_6

Table 1. An example showing 6 tasks and their resource requirements

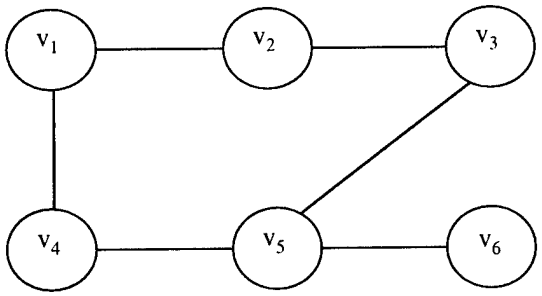


Figure 2. The compatibility graph for the tasks shown in Table 1

Task	Execution Time
V_1	5
V_2	6
V_3	2
V_4	4
V_5	1
V_6	3

Table 2. Execution times for the tasks in Figure 2

3.4. Compatibility Graph

To capture the implied resource sharing constraints among tasks that may belong to the same or different applications, we use the *compatibility graph*, $g = (V, E)$, where vertex v_i denotes task t_i and edge e_{ij} exists if and only if t_i and t_j are incompatible. Recall, task t_i and task t_j are incompatible if and only if $R(t_i) \cap R(t_j) \neq \phi$. An *independent set* [6] is a set of vertices of g such that no two vertices of the set are adjacent. An independent set is called a *maximal independent set* if there is no other independent set of g that contains it. A maximal independent set with the largest number of vertices among all maximal independent sets is called a *maximum independent* set [6]. The maximum independent set problem is NP-complete [15]. In our model, a maximal independent set of g represents a maximal set of tasks that can be executed concurrently if there is no precedence constraints among them.

As an example, consider a set of 6 independent tasks. Each task needs concurrent access to a set of resources as specified in Table 1. The compatibility graph g for this example is shown in Figure 2. The maximal independent sets of g are $\{V_1,V_5\}$, $\{V_2,V_5\}$, $\{V_1,V_3,V_6\}$, $\{V_2,V_4,V_6\}$, and $\{V_3,V_4,V_6\}$. The last three sets are maximum independent sets.

4. Our Solution

In classical DAG scheduling problem, application DAGs are partitioned onto levels such that each level contains independent tasks, i.e., there are no data dependencies among the tasks in the same level. Therefore, all tasks in the same level can be executed concurrently. In our framework, incompatible tasks in the same level cannot be executed concurrently due to resource sharing constraints. Therefore, mapping algorithms for the classical DAG scheduling problem (ex. [1, 30, 18, 23, 9, 31, 32]) cannot be directly used for our problem.

In this section, we develop a static co-allocation algorithm using the framework defined in Section 3. The algorithm can be used with different maximal independent set selection strategies and different allocation heuristics to solve the mapping problem with co-allocation requirements. Several strategies for selecting maximal independent sets and several allocation heuristics are given in this section. Also, we provide a two-phase algorithm that performs run-time adaptation.

4.1. The Co-Allocation Algorithm

Pseudo code for our co-allocation algorithm is shown in Figure 3. Given a set of applications and resource requirements of tasks, we first find tasks that have satisfied precedence constraints and then select maximal independent sets among these for allocation. The compatibility graph is used to find maximal independent sets. Since the maximum independent set problem is NP-complete [15], our approach for selecting a maximal independent set is based on first choosing a critical node v_c, and then finding a maximal independent set that contains v_c. Different strategies for selecting critical nodes are given in Section 4.2.

To ensure precedence constrains are satisfied, we combine all submitted applications into a single DAG, G, by using zero-weight edges to connect the root nodes (tasks) of all applications to a hypothetical zero-cost entry node. Then we partition the combined DAG onto l levels such that level 0 contains the entry node and level 1 contains all tasks that do not have any predecessors in the submitted DAGs. All tasks in level l have no successors. For each task t_i in level k, all of its predecessors are in levels 0 to $k-1$, and at least one of them in level k-1.

Let RDY be the set of tasks that have no precedence constraints among them and that are ready for allocation. A task is ready for allocation if for each predecessor all required resources have been allocated. Let W be the set of ready tasks that are waiting for allocation, and $ALLOCATED$ be the set of allocated tasks. After executing the algorithm, the list $SCHEDULE$ will give the resulting scheduling order of tasks and the variable $length$ will give the resulting schedule length.

In steps 1 and 2 of the algorithm, we combine all submitted applications into a single DAG, G, and partition G into l levels. Then the algorithm proceeds level-by-level as follows. For each level l of G, we construct the compatibility graph g for all tasks in this level (step 6). g is used to find maximal independent sets of tasks that can be executed concurrently. The first maximal independent set of tasks to be allocated is selected in steps 7-8 where a critical node v_c is chosen in step 7 and a maximal independent set that contains v_c is determined in step 8.

In step 10, all tasks in the selected maximal independent set are allocated to their required resources. For the allocation, we first find the scheduling order of the tasks. Several heuristics are given in Section 4.3. Then, we use this scheduling order to assign a compute resource m_j to each task t_i in order to minimize its finish time $FT(t_i, m_j)$. Availability times ($MA(m_j)$ and $RA(r_k)$) of all resources required by task t_i are updated based on $FT(t_i, m_j)$.

In steps 12-16, a new set of maximal independent tasks among all waiting tasks is selected to be allocated at the next allocation event. The next allocation event is calculated as the earliest finish time, $FT(t_i, m_j)$, among all allocated tasks. An allocated task v_x with the earliest finish time is identified in step 12 and then removed from the $ALLOCATED$ set. Initially (step 14), the set of candidate tasks that can be allocated next ,C, contains all waiting

Inputs: application DAGs, estimated computation and communication costs, and resource requirements of tasks
Outputs: the scheduling order of tasks ($SCHEDULE$) and the schedule length (makespan) ($lenght$) based on the given inputs

Begin
1. Combine all submitted application DAGs into a single DAG (G)
2. Do level partitioning of G /* tasks in each level have no precedence constraints */
3. Let $SCHEDULE = \phi$ and $lenght = 0$
4. For level 1 to l do
5. Initialize W to include all tasks in the current level and let $ALLOCATED = \phi$
6. Construct the compatibility graph g for all tasks in the current level
7. Pick a critical node v_c from W /* several strategies can be used for critical node selection*/
8. Find a maximal independent set of tasks S from W such that $v_c \in S$ /* g is used to find the maximal independent set */
9. While W is not empty do
10. Allocate all tasks in S to their required resources by doing the following two steps:
10a. Find the scheduling order of the tasks and add them to $SCHEDULE$ /* different heuristics can be used */
10b. For each task t_i in S (in the scheduling order) do
- Assign a compute resource m_j to task t_i in order to minimize its finish time $FT(t_i, m_j)$
- Update $MA(m_j)$ and $RA(r_k)$, $\forall r_k \in R(t_i)$, based on $FT(t_i, m_j)$
- If ($FT(t_i, m_j) > lenght$) then $lenght$=$FT(t_i, m_j)$
11. Add all tasks in S to $ALLOCATED$ and remove them from W
12. Let v_x be the allocated task with the lowest finish time
13. Remove v_x from $ALLOCATED$
14. Let $C = W$, where C is the set of candidate tasks that can be allocated next
15. Remove all tasks from C that are incompatible with any allocated task
16. If ($C \neq \phi$)
16a. Pick a critical node v_c from C such that v_c is adjacent to v_x in g
16b. Find a maximal independent set of tasks S from C such that $v_c \in S$
17. End (while)
18. End (for)
End

Figure 3. The co-allocation algorithm

tasks. The candidate set, C, is updated in step 15 by removing all tasks that are incompatible with any allocated task. Then g is used to find a maximal independent set of tasks from C. The algorithm repeats steps 10-16 until all tasks in this level have been allocated.

4.2. Maximal Independent Set Selection

Since the maximum independent set problem is NP-complete [15], we use a heuristic approach for selecting maximal independent sets. Our approach is based on first selecting a critical node v_c, and then finding a maximal independent set that contains v_c. Critical nodes need to be selected carefully.

The length of the schedule is influenced by the selection of maximal independent sets and by the order in which these sets are considered for scheduling. This is shown in the following example using the compatibility graph in Figure 2. For this example, for the sake of simplicity, we assume that all the resource requirements of tasks (compute and non-compute resources) are pre-specified. Therefore, the execution times of all tasks are known *apriori*. These times are shown in Table 2. Example schedules are given in Figures 4-6. These schedules have different schedule lengths. The optimal length of the schedule is 11 time units. This is achieved by schedules 2 and 3. In schedules 1 and 2, two different maximum independent sets were selected to be scheduled first. Schedule 1 has a length of 13 time units, while schedule 2 has the optimal length. This clearly shows the importance of the order in which the maximal independent sets are considered for scheduling. From schedule 3, we can also see that it is not always efficient to select a maximum independent set to be scheduled first. Schedule 3, which starts with a maximal independent set $\{V_2,V_5\}$ (not a maximum independent set), has the optimal length while schedule 1, which starts with a maximum set $\{V_3,V_4,V_6\}$, has a non-optimal length of 13 time units.

The idea behind our approach for selecting a maximal independent set S is to select a *critical* vertex v_c and add it to S which is initially empty. Then we attempt to enlarge S by traversing g. Different strategies can be used for selecting critical vertices. In the following we describe some of these strategies.

S1 Highest average execution time. In this strategy, we give priority for tasks that need more time for execution since they can be critical tasks. In HC systems, tasks have different execution times on different machines. Therefore, we use the average execution time $\overline{Exec(t_i)}$ as the selection criterion.

S2 Highest degree. The node out-degree in a DAG has been used in many list scheduling heuristics as a priority function. The out-degree of a node t_i gives the number of tasks that have precedence constraints with t_i. The idea is to advance the execution of tasks with high out-degree. Thus, many tasks can be ready for mapping once high out-degree tasks complete execution. In our framework, the out-degree of node t_i in the combined DAG G (which represents task t_i) does not reflect all dependencies between t_i and other tasks since G only captures the precedence constraints. Resource sharing constraints should also be considered. Therefore, we define the degree of task t_i as the sum of its out-degree in G and its degree in g. This number gives a better indication about the number of tasks that can be ready for mapping once t_i completes its execution, either because those tasks have a precedence or resource sharing dependencies with t_i.

S3 Critical path nodes. A Critical Path (CP) in a DAG is a path from an entry node to an exit node with the largest completion time. We use average execution times and average communication costs to find the critical path. In some situations, the average execution time or the degree of a task t_i cannot reflect how important for other tasks that t_i finishes execution as soon as possible. The successors of t_i may not be critical tasks and advancing their execution may not improve the schedule length. For these reasons, selecting critical path nodes from G as critical tasks can be a good strategy. In this paper, we implement two variations of this strategy:

- **S3.1** In this version, the task that is on the critical path is selected as a critical task. If there is no such task among the current set of candidate tasks, the task with the highest average execution time is selected as a critical task.
- **S3.2** This version is similar to S3.1 except for the case when there is no critical path node among the current set of candidate task. In this case, the task with the highest degree is selected as a critical task.

S4 Maximum weighted clique. In [3], a similar approach to the compatibility graph has been used for scheduling independent tasks. Each task in [3] requires simultaneous access to a set of pre-specified processors. All resource (processor) requirements and all execution times were assumed to be known *apriori*. It has been shown in [3] that the weight of the maximum weighted clique in the *constraint graph* (compatibility graph) is a lower bound on the optimum makespan, where the weight of each node is its execution time. In our previous example, notice that the weight of the maximum weighted clique (V_1 and V_2) is 11 and the optimal schedule length is 11. Also notice that any se-

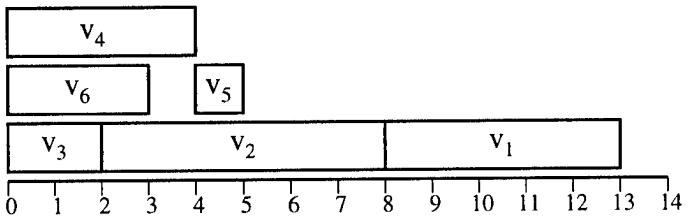


Figure 4. Schedule 1

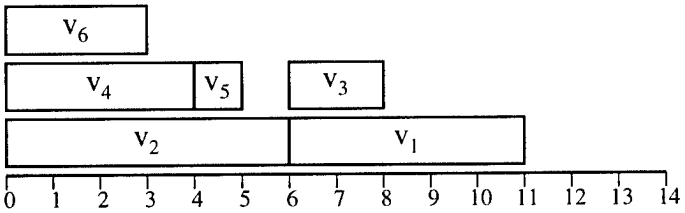


Figure 5. Schedule 2

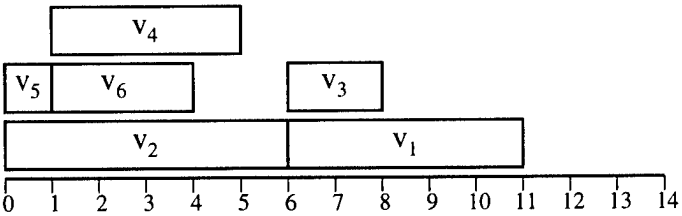


Figure 6. Schedule 3

lected maximal independent set should contain a task that belongs to the maximum weighted clique in order to achieve the optimal schedule length. Inspired by this observation, we can use nodes in the maximum weighted clique as candidates for selecting critical tasks. In our approach, we use the average execution times as the node weights. It is obvious that in our model, maximum weighted clique cannot guarantee the optimal solution but it could be a good heuristic for selecting maximal independent sets.

4.3. Allocation Heuristics

After selecting a maximal independent set of tasks, careful allocation of these tasks to compute resources (machines) is required to achieve our objective. Different heuristic can be used for allocating tasks of the selected maximal independent set S to compute resources. In the following we describe some of our allocation heuristics. The idea behind our heuristics is to advance the execution of tasks that may be critical in order to minimize the overall schedule length.

1. **Highest Average-Execution-Time First (HAETF).** In this heuristic, the average execution time is used as a priority function to place tasks in a list. All tasks are placed in a list in the order of non-increasing average execution times. Using this order, each task is allocated to the required resources such that its finish time is minimized.

2. **Maximum Finish-Time First (MAX).** For each task, we calculate the best finish time that can be achieved. Then we select the task with the maximum best finish time among all tasks. The task is allocated its required resources such that its finish time is minimized. We repeat until all tasks are allocated.

3. **Minimum Finish-Time First (MIN).** This heuristic is similar to the Maximum Finish-Time First (MAX) heuristic except that we select the task with the minimum finish time instead of selecting the task with the maximum finish time.

4. **Highest Degree First (HDF).** In this heuristic, all tasks are placed in a list in descending order according to their degrees (ties are broken arbitrarily). Then, tasks are allocated one-by-one to the required resources such that the finish time for each task is minimized.

4.4. Two-Phase Algorithm

We propose a two-phase algorithm for run-time adaptation using our static co-allocation algorithm. The two-phase algorithm can be used for the problem of mapping with resource co-allocation as defined in our framework as follows.

Phase 1: Compile-time mapping. At this phase, the co-allocation algorithm described in Section 4.1 is used to obtain an ordered list of tasks. The order of tasks in the list is based on their scheduling order as produced by our co-allocation algorithm. The list is obtained by satisfying all precedence and resource sharing constraints with the objective of minimizing the overall schedule length. Estimated computation and communication times are used to calculate the schedule length.

Phase 2: Run-time Adaptation. Run-time adaptation can be useful for the cases when actual execution times differ from the estimated execution times. One way to consider this is to scan through the ordered list obtained in phase 1 once a task completes its execution in order to find all tasks that can be executed at this time and make local reordering. The scanning can be done through a window of tasks with specific size k, where $k > 0$.

4.5. Implementation Issues

The focus of this paper is the mapping problem with resource co-allocation requirements in HC systems. The implementation details for the co-allocation process are outside the scope of this paper. A good discussion of implementation issue can be found in [5]. In the following for the sake of completeness, we briefly state our assumptions regarding the co-allocation implementation.

We assume that a task t_i cannot start execution until all its required resources are available. These resources will be acquired at the same time. Once a task t_i completes its execution, all its allocated resources will be released and will be available for other tasks. We assume that any allocation request for any resource will be granted as long as this resource is available. In this paper, we do not consider the cases of resource failures that can occur in the HC and Grid environments.

5. Performance Evaluation

A simulator was implemented to evaluate the performance of our co-allocation algorithms and the proposed selection strategies and allocation heuristics discussed in Section 4. In this section, we explain our simulation procedure and give experimental results.

5.1. Simulation Procedure

To define the HC system, numbers of machines and resources are given to the simulator as inputs. Communica-

tion costs among all resources are selected randomly from a uniform distribution with a mean equal to *ave_comm*. The communication costs are source and destination dependent.

The workload consists of randomly generated DAGs. Random DAGs are generated as follows: The number of tasks in the graph, *no_tasks*, maximum out-degree of a node, *max_outdegree*, average computation cost of a node, *ave_comp*, and average message size to be transferred among tasks, *ave_msg_size*, are given as inputs. First, the computation time of each task on every compute resource is randomly selected from a uniform distribution with a mean equal to *avg_comp*. Starting with the first task, the number of children (out-degree) is randomly selected between 1 and *max_outdegree*. Then, children are randomly selected for this task. The weight of each edge in the DAG is randomly selected from a uniform distribution with a mean equal to *ave_msg_size*. Resource requirements for each task are randomly selected from available resources. The amount of data to be transferred to/from each resource in the resource requirements set is randomly selected from a uniform distribution with a mean equal *ave_data_size*. The sizes of random DAGs range from 50 to 250 tasks with increments of 50.

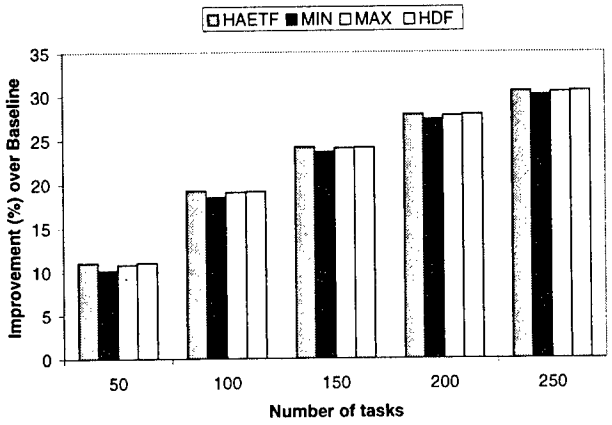


Figure 7. Performance of the allocation heuristics when using selection strategy $S1$

5.2. Baseline Algorithm

Many mapping algorithms exist in the literature for mapping DAGs in HC systems. None of these algorithms consider the co-allocation problem we define in this paper. Therefore, we will use a simple list scheduling algorithm as a baseline algorithm to evaluate our co-allocation algorithm. The baseline algorithm is a fast static algorithm for mapping DAGs in HC environments. It partitions the tasks in the DAG into levels using an algorithm similar to the level partitioning algorithm described in Section 4.1. Then all the tasks are ordered such that the tasks in level k come before the tasks in level $k+1$. The tasks in the same level are sorted in descending order based on the average execution time of each task (ties are broken arbitrarily). The tasks are considered for mapping in this order. A task is mapped to the required resources such that its finish time is minimized.

The Baseline algorithm is similar to our algorithms in the sense that all algorithms proceed level-by-level. In the Baseline algorithm, the scheduling order of tasks at the same [illegible] tasks. On the

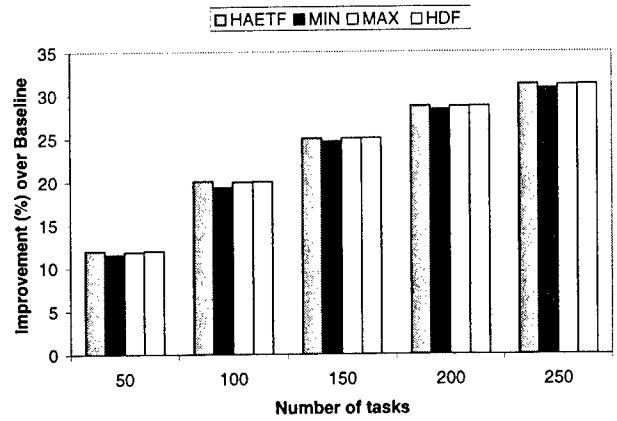


Figure 8. Performance of the allocation heuristics when using selection strategy $S2$

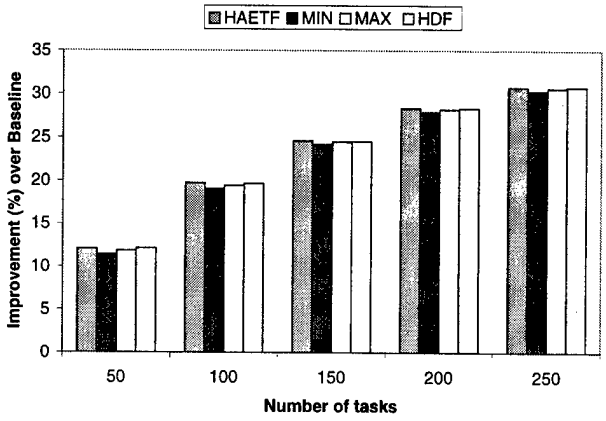


Figure 9. Performance of the allocation heuristics when using selection strategy $S3.1$

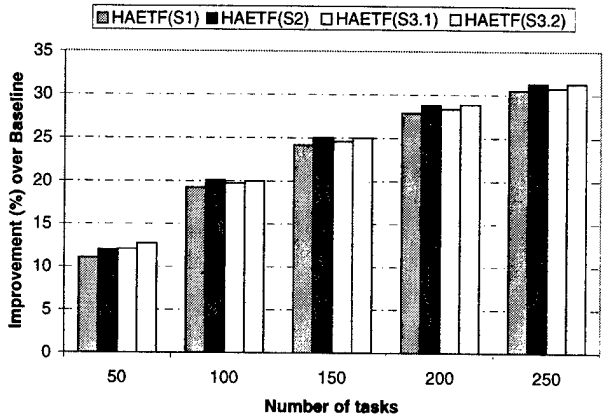


Figure 11. Performance of the selection strategies with $HAETF$ allocation heuristic

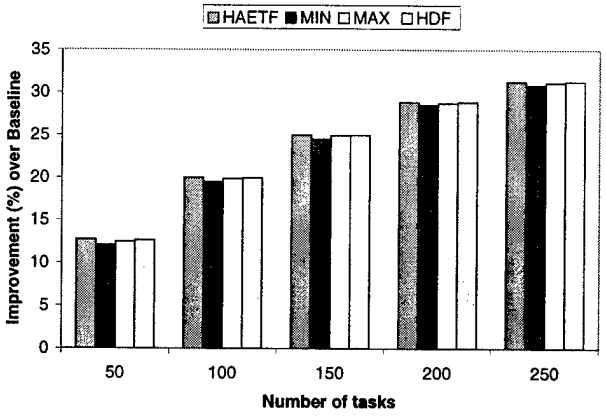


Figure 10. Performance of the allocation heuristics when using selection strategy $S3.2$

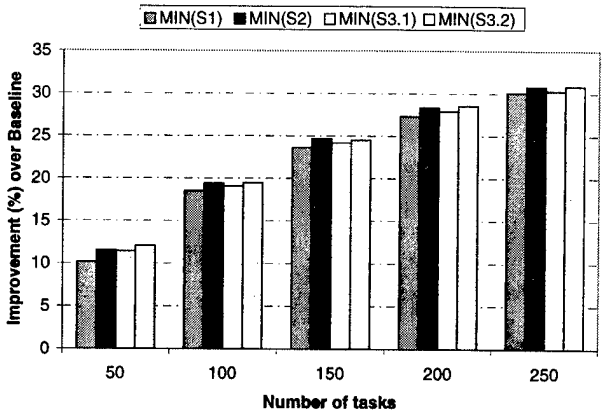


Figure 12. Performance of the selection strategies with MIN allocation heuristic

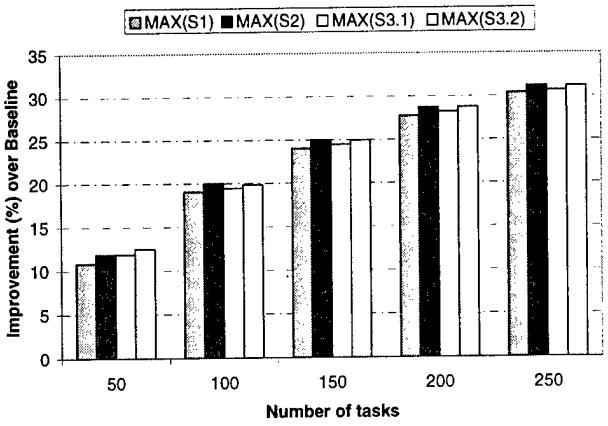


Figure 13. Performance of the selection strategies with MAX allocation heuristic

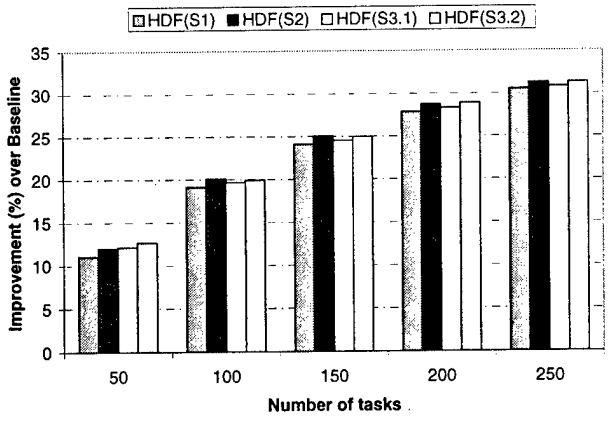


Figure 14. Performance of the selection strategies with HDF allocation heuristic

5.3. Experimental Results

Our experimental results are given in Figures 7- 14. The total number of tasks were varied from 50 to 250 with increments of 50. Each point in the figures is an average of 400 runs with different random DAGs. Random DAGs were generated with *max_outdegree*={2,3,4,5}, *ave_comp*=50, *ave_msg_size*=50K byte, and *ave_data_size*=300K byte.

Figures 7- 10 show the performance results of our allocation heuristics compared to the Baseline algorithm when using different maximal independent set selection strategies. The improvement of our heuristics over the Baseline increases as the total number of tasks increases. This shows the importance of considering co-allocation requirements in mapping algorithms. Generally, our allocation heuristics have relatively the same performance.

The performance results of maximal independent sets selection strategies when using different allocation heuristics are given in Figures 11- 14. As in the previous set of results, the improvement over Baseline algorithm increases as total number of tasks increases. Also, the selection strategies have relatively same performance.

In our simulation study, we found that the number of machines and the number of resources did not have a significant impact on the performance of allocation heuristics and selection strategies.

6. Conclusions and Future Work

This paper proposes a novel framework for the problem of mapping applications with resource co-allocation in HC systems. We formulated the co-allocation problem and developed several algorithms for solving this problem using a graph theoretic approach. Our simulation results show the importance of considering the co-allocation requirements during mapping decisions.

In solving our co-allocation problem, we need to find maximal independent sets among tasks competing for system resources. Although we considered many heuristics, they all seem to perform equally well indicating that a simple heuristic will suffice (even though one can create pathological examples for each heuristic).

In our future work, we plan to expand our framework to consider concurrent usage of multiple compute resources and *advance resource reservations*. With advance reservation, system resources can be reserved in advance for specific time intervals. Therefore, resource availability must be expressed as a list of available time slots and mapping algorithms should be insertion-based algorithms. To co-allocate a set of resources in this case, efficient algorithms are needed to find the best time slot when all resources are available for the required duration. In this paper, our algorithms are

non insertion-based since the earliest available time for a resource r_i is the finish time of the last task assigned to r_i.

References

[1] I. Ahmad and Y. Kwok, "On parallelizing the multiprocessor scheduling problem," *IEEE Trans. on Parallel and Distributed Systems,* 10(4):414-432, Aprill 1999.

[2] A. Alhusaini, V. Prasanna, and C.S. Raghavendra, "A unified resource scheduling framework for heterogeneous computing environments,' ' *8th Heterogeneous Computing Workshop (HCW' 99),* pp. 156-165, April 12, 1999.

[3] L. Bianco, P. Dell'Olmo, and M. Grazia Sperenza, "Nonpreemptive scheduling of independent tasks with prespecified processor allocations," *Naval Research Logistics,* 41:959-971, 1994.

[4] T. Braun, H.J. Siegel, N. Beck, L. Boloni, M. Maheswaran, A. Reuther, J. Robertson, M. Theys, and B. Yao, "A Taxonomy for describing matching and scheduling heuristics for mixed-machines heterogeneous computing systems," *Workshop on Advances in Parallel and Distributed Systems (APADS),* West Lafayette, IN, Oct. 1998.

[5] K. Czajkowski, I. Foster, and C. Kesselman, "Resource coallocation in computational grids," *7th IEEE Symposium on High Performance Distributed Computing,* pp. 219-228, 1999.

[6] N. Christofides, *Graph theory: An algorithmic approach,* Academic Press, 1975.

[7] Legion Web Page. *http://legion.virginia.edu*

[8] M. Eshagian (ed.), *Heterogeneous computing,* Norwood, MA: Artech House, 1996.

[9] M. Eshagian and Y.C. Wu, "Mapping heterogeneous task graphs onto heterogeneous system graphs," *6th Heterogeneous Computing Workshop (HCW' 97),* pp. 147-160, 1999.

[10] I. Foster and C. Kesselman, ed., *The Grid: blueprint for new computing infrastructure,* Morgan Kaufmann Publishers, San Francisco, CA, 1999.

[11] I. Foster, C. Kesselman, C. Lee, B. Lindell, K. Nahrstedt, and A. Roy, "A distributed resource management architecture that support advance reservations a nd co-allocation," *Intl. Workshop on Quality of Service,* 1999.

[12] R. Freund, B. Carter, D. Watson, E. Keith, and F. Mirabile, "Generational scheduling for heterogeneous computing systems," *Int'l Conf. Parallel and Distributed Processing Techniques and Applications (PD PTA '96),* pp. 769-778, Aug. 1996.

[13] R. Freund and H. J. Siegel, "Guest editors' introduction:Heterogeneous processing" *IEEE Computer,* 26(6):13-17, June 1993.

[14] Globus Web Page. *http://www.globus.org.*

[15] M. Gary and D. Johnson, *Computers and intractability: A guide to the theory of NP-completeness,* W.H. Freeman and Company, San Francisco, CA, 1979.

[16] D. Hensgen, T. Kidd, D. St.John, M. Schnaidt, H.J. Siegel, T. Braun, M. Maheswaran, S. Ali, J. Kim, C. Irvine, T. Levin, R. Freund, M. Kussow, M. Godfrey, A. Duman, P. Carff, S. Kidd, V. Prasanna, P. Bhat, and A. Alhusaini, "An overview of MSHN: the Management System for Heterogeneous Networks," *8th Heterogeneous Computing Workshop (HCW' 99),* pp. 184-198, April 12, 1999.

[17] O. Ibarra and C. Kim, "Heuristic algorithms for scheduling independent tasks on non identical processors. " *Journal of The ACM,* 24(2):280-289, April 1977.

[18] M. Iverson and F. Ozguner, "Dynamic, competitive scheduling of multiple DAGs in a distributed heterogeneous en vironment," *7th Heterogeneous Computing Workshop (HCW' 98),* pp. 70-78, March 1998.

[19] M. Iverson, F. Ozguner, and G. J. Follen, "Parallelizing existing applications in a distributed heterogeneous environment," *4th Heterogeneous Computing Workshop (HCW' 95),* pp. 93-100, Apr. 1995.

[20] A. Khokhar, V. K. Prasanna, M. Shaaban, and C. L. Wang, "Heterogeneous computing: challenges and opportunities," *IEEE Computer,* 26(6):18-27, June 1993.

[21] C. Leangsuksun, J. Potter, and S. Scott, "Dynamic task mapping algorithms for a distributed heterogeneous computing environm ent," *4th Heterogeneous Computing Workshop (HCW' 95),* pp. 30-34, Apr. 1995.

[22] Legion Web Page. *http://legion.virginia.edu.*

[23] M. Maheswaran and H. J. Siegel, "A Dynamic matching and scheduling algorithm for heterogeneous computing systems," *7th Heterogeneous Computing Workshop (HCW '98),* pp. 57-69, March 1998.

[24] P. Shroff, D. W. Watson, N. S. Flann, and R. F. Freund, "Genetic simulated annealing for scheduling data-dependent tasks in heterogeneous e nvironment," *5th Heterogeneous Computing Workshop (HCW' 96),* pp. 98-117, Apr. 1996.

[25] H.J. Siegel, J. Antonio, R. Metzger, M. Tan, and Y. Li, " Heterogeneous computing," in *Parallel and distributed computing handbook,* A.Y. Zomaya (ed.), McGraw-Hill, New york, 1996, pp.725-761.

[26] H.J. Siegel, M. Mahesewaran, and T. Braun, "Heterogeneous distributed computing," in *Encyclopedia of electrical and electronics engineering,* J. Webster (ed.), John Wiley & Sons, New York, to appear.

[27] G. C. Sih and E. A. Lee, "A Compile-time scheduling heuristic for interconnection-constrained heterogeneous processor architectures," *IEEE Trans.on Parallel and Distributed Systems,* 4(2):175-187, Feb. 1993.

[28] H. Singh and A. Youssef, "Mapping and scheduling heterogeneous task graphs using genetic algorithms," *5th Heterogeneous Computing Workshop (HCW' 96),* pp. 86-97 , April 15-16, 1996.

[29] L. Smarr and C. E. Catlett, "Metacomputing," *Communications of the ACM,* 35(6):45-52, June 1994.

[30] H. Topcuoglu, S. Hariri, and M. Wu, "Task scheduling algorithms for heterogeneous processors," *8th Heterogeneous Computing Workshop (HCW' 99),* pp. 3-14, April 12, 1999.

[31] R. Venkataramana and N. Ranganathan, "Multiple cost optimization for task assignment in heterogeneous computing systems using learning automata," *8th Heterogeneous Computing Workshop (HCW' 99),* pp. 137-145, April 12, 1999.

[32] L. Wang, H.J. Siegel, V. Roychowdhury, and A. Maciejewski, "Task Matching and Scheduling in Heterogeneous Computing Environments Using a Genet ic-Algorithm-Based Approach," *Journal of Parallel and Distributed Computing,* 47(1):8-22, Nov. 1997.

Biographies

Ammar Alhusaini is a Ph.D. candidate in the Department of Electrical Engineering - Systems at the University of Southern California, Los Angeles, California, USA. His main research interest is task scheduling in heterogeneous environments. Mr. Alhusaini received a B.S. degree in computer engineering from Kuwait University in 1993 and M.S. degree in computer engineering from the University of Southern California in 1996. Mr. Alhusaini is a member of IEEE, IEEE Computer Society, and ACM.

Viktor K. Prasanna (V.K. Prasanna Kumar) is a Professor in the Department of Electrical Engineering - Systems, University of Southern California, Los Angeles. He received his B.S. in Electronics Engineering from the Bangalore University and his M.S. from the School of Automation, Indian Institute of Science. He obtained his Ph.D. in Computer Science from Pennsylvania State University in 1983. His research interests include parallel computation, computer architecture, VLSI computations, and high performance computing for signal and image processing, and vision. Dr. Prasanna has published extensively and consulted for industries in the above areas. He is widely known for his pioneering work in reconfigurable architectures and for his contributions in high performance computing for signal and image processing and image understanding. He has served on the organizing committees of several international meetings in VLSI computations, parallel computation, and high performance computing. He also serves on the editorial boards of the Journal of Parallel and Distributed Computing and IEEE Transactions on Computers. He has the founding chair of the IEEE Computer Society Technical Committee on Parallel Processing. He is a Fellow of the IEEE.

Cauligi Raghavendra is a Senior Engineering Specialist in the Computer Science Research Department at the Aerospace Corporation. He received the Ph.D degree in Computer Science from University of California at Los Angeles in 1982. From September 1982 to December 1991 he was on the faculty of Electrical Engineering-Systems Department at University of Southern California, Los Angeles. From January 1992 to July 1997 he was the Boeing Centennial Chair Professor of Computer Engineering at the School of Electrical Engineering and Computer Science at the Washington State University in Pullman. He received the Presidential Young Investigator Award in 1985 and became an IEEE Fellow in 1997. He is a subject area editor for the Journal of Parallel and Distributed Computing, Editor-in-Chief for Special issues in a new journal called Cluster Computing, Baltzer Science Publishers, and is a program committee member for several networks related international conferences.

Heterogeneous Resource Management for Dynamic Real-Time Systems

Eui-Nam Huh, Lonnie R. Welch
Department of Electrical Engineering and Computer Science, Ohio University
{ehuhlwelch@ace.cs.ohiou.edu}

Behrooz A. Shirazi, Charles D. Cavanaugh
Department of Computer Science Engineering, The University of Texas at Arlington
{shirzailcavan@cs.uta.edu}

Abstract

Dynamic real-time systems face many resource management problems. This paper addresses the following problems: (1) dynamic resource allocation to provide QoS objectives, (2) heterogeneous resources, and (3) non-intrusive accurate monitoring of QoS, resource availability, and resource needs. This paper describes the techniques of resource manager (RM) handling above problems to support QoS of dynamic distributed real-time systems. The contributions of this paper to solve these problems are as follows: unification of dynamic resource requirements among heterogeneous hosts, control of resources in heterogeneous environments, feasibility analysis, and dynamic load balancing/sharing. Our heuristic allocation scheme not only allows higher workloads than random, round robin, least load by 257%, 142%, and 36.4%, respectively, but also improves QoS better than random, round robin, and least load 38.6%, 28.5%, and 31.6%, respectively.

1. Introduction

This paper describes techniques for managing heterogeneous host resources to support QoS of dynamic distributed real-time systems. Our approach is based on the dynamic path paradigm. A path-based real-time subsystem (see [1], [2]) typically consists of a detection & assessment path, an action initiation path, and an action guidance path. The paths interact with the environment by evaluating streams of data from sensors, and by causing actuators to respond (in a timely manner) to events detected during evaluation of sensor data streams.

An overview of our approach for RM is shown in Figure 1. The "s/w spec" is used to describe QoS

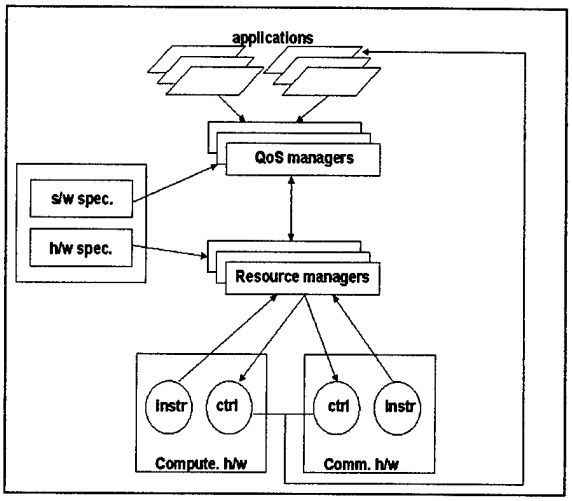


Figure 1. Resource management architecture

requirements. The "h/w spec" also defines information about the hosts and networks such as speed, OS type, the number of CPUs, benchmark rate, bandwidth, and interconnected equipment. The "QoS managers" collect QoS metrics, compare to s/w spec, and request resources, if QoS violations occur. The "resource manager" is the brain, which makes allocation decisions to achieve QoS objectives.

This paper focuses on the resource management component, and discusses a new technique for dynamic feasibility analysis on heterogeneous resource platforms. Most previous work in distributed real-time systems assumed that all system behaviors follow a statically known pattern (see [3][4]). When applying the previous work to some applications (such as shipboard AAW systems [1][5]), problems arise with respect to scalability of analysis and modeling techniques; furthermore, it is sometimes impossible to

0-7695-0556-2/00 $10.00 © 2000 IEEE

Table 1. System resource model

SYMBOL	NOTATION
a_{ij}	name of application j in path i
$tl = tl(c, P_i)$	workload or tactical load at cycle c in path i
$CUP_{obs}(a_{ij}, tl, H_k)$	the CPU user-percentage of host H_k for the application a_j in path i at work load tl,
$CUP_{uni}(a_{ij}, tl, H_k)$	the unified CPU user-percentage of host H_k for the application a_j in path i at work load tl
$CUP(H_i,t)$	the CPU user-percentage of host H_i at time t
$CIP(H_i,t)$	the idle-percentage of host H_i at time t
$MEM_{obs}(a_{ij}, tl, H_k)$	the memory usage of application a_j in path i on host H_k at work load tl
$FAM(H_i,t)$	the free-available-memory of host H_i at time t
$X_{obs}(a_{ij}, tl, H_k)$	the execution time of application a_j in path i on host H_k at work load tl
$P_{obs}(a_{ij}, tl, H_k)$	the period of application a_j in path i on host H_k at work load tl
$CCR(H_i)$	CPU clock rate in MHz at host H_i
$SPECint95(H_i)$	the fixed point operation performance of SPEC CPU95 of host H_i
$SPECfp95(H_i)$	the floating point operation performance of SPEC CPU95 of host H_i
$SPEC\_RATE(H_i)$	the relative SPECCPU95 rating of host H_i
$Threshold\_CPU(H_j)$	the CPU threshold
$Threshold\_MEM(H_j)$	the memory threshold

obtain some of the parameters required by the models.

In contrast, DeSiDeRaTa RM(see [6][2]) allows the modeling of systems that work in environments that have unknown scenarios (such as battle environments) (see [7]); the dynamic path paradigm is based on obtainable parameters, since it evolved from the study of existing computer systems; and the large granularity of the path makes it more scalable than task allocation approaches.

The new contributions of this paper are as follows: (1) unification of dynamic resource requirements among heterogeneous hosts, (2) control of resources in heterogeneous environments, (3) feasibility analysis, and (4) dynamic load balancing/sharing.

Section 2 shows the feasibility analysis and laxity based RM approach with system model. Section 3 shows the results of experiments. Finally, Section 4 is the summary and conclusion.

2. Laxity Based RM

In this section, the resource management approach is explained. Basic steps of dynamic resource management are follows: (step 1) Resource Requirement , (step 2) Resource Discovery, (step 3) Resource Unification, (step 4) Feasibility Analysis, and (step 5) Optimization. These steps are explained in detail in the remainder of this section. First, a mathematical model, which is used in the detailed explanation, is presented.

Table 1 shows the system resource model. a_{ij} and tl represent application and workload of an application, respectively. Indices starting with CUP stand for CPU usage. CIP is CPU idle percentage of a host. MEM and FAM relate to memory usage. X and P are the execution time and period of an application a_{ij}, respectively. CCR stands for CPU clock rate of a host. The SPEC CPU95 host benchmark consists of SPECint95 and SPECfp95 that show relative performance of fixed and floating point operations in a system. SPEC_RATE is overall relative system rank Indices, Threshold, are

certain amount of resource to tolerate different amount of resource requirement.

The steps taken by RM are now explained in detail. The resource requirement step works as follows. QM detects QoS violation by monitoring QoS of a path and each application and requests additional resources based on decisions. When a significant amount of workload is observed, QM analyzes the latency of each application. If a_{ij} uses more resources than others, or the latency of a_{ij} is higher than minimum QoS slack, then QM triggers request of additional resources with another copy of the application. This is called "scale-up" decision. When workload is not changed, but QoS violation occurs, QM triggers migration of a_{ij} running on the overloaded host. It is called "move" decision.

Therefore, different resource requirements should be measured according to decisions. Hence, for "move" decision, RM measures dynamic resource requirement of CPU for the violated application using $CUP_{obs}(a_{ij}, tl, H_k) = X((a_{ij}, tl, H_k) / P(a_{ij}, tl, H_k)$. For the "*scale-up*" decision, the resource requirement is measured by interpolation and extrapolation from the initial profile for the new workload: tl = current tl / (replicas + 1). The resource discovery step is explained here in detail.

Monitoring of resource availability in dynamic environments has more difficulties than in static environments, because of unknown system activation. $FAM(H_k,t)$, $CIP(H_k,t)$, and $CUP(H_k,t)$ are collected for all host "k" once per second. And these resources are filtered by exponential moving average(EMA) as illustrated below for CUP:

$EMA(CUP(H_j,t)) = (1-\beta)*(CUP(H_j,t)) + \beta * EMA(CUP(H_j,t-1))$, where, $t \geq 1$, $\beta = e^{-T}$.

Each resource has various scales and capacities even in the same unit among heterogeneous platforms. In this step, resource unification method is explained in detail.

Definition: *Resource unification produces a canonical form of each resource metric.*

RM allocates and controls resources accurately, if each resource is unified. Consider $CUP_{obs}(a_{ij}, tl, H_k)$ as resource requirement. To allocate the amount of the resource, RM needs to analyze the requirement and map it to target hosts. There are two approaches, static and dynamic. The static approach uses stable system information like benchmarks, or CPU clock rate. It will decide relative amount of system resources efficiently but inaccurately for dynamic environments. The dynamic approach of predicting execution time using dynamic system information has high complexity for real-time systems, as an application uses several different resources such as I/O disk, memory, and CPU, each of which has different performance among hosts. Therefore, a static approach is selected as follows. For the unification of resources, the results of a variety of realistic SPEC CPU95 will give valuable insight into expected real performance among heterogeneous hosts.

However, no one benchmark can fully characterize overall system performance. SPEC CPU95 measures the performance of CPU, memory system, and compiler code generation by running 18 programs that are well designed to gather their throughput. The geometric mean is used to represent system overall performance compared to a reference machine, Sun-sparc-10/40Mhz. This standardized set of benchmarks (SPECint95 and SPECfp95) is adaptable to the recent generation of high-performance computing efficiently (HPC) [8]. Hence, the following formula (1) is used to *unify* CPU resource, $CUP_{uni}(a_{ij}, tl, H_T)$, onto target host, H_T from $CUP_{obs}(a_{ij}, tl, H_k)$ on source host, H_k.

$$CUP_{uni}(a_{ij}, tl, H_T) = CUP_{obs}(a_{ij}, tl, H_k) * SPEC\_RATE(H_k) / SPEC\_RATE(H_T) \quad -(1)$$

Where $\forall_j$, $SPEC\_RATE(H_i) = AVG(SPECint95(H_i) / Max(SPECint95(H_j), SPECfp95(H_i) / Max(SPECfp95(H_j)))$.

Another piece of static system information, $CCR(H_i)$ is considered but it is inapplicable to unification of resources, because a different number of CPU cycles between RISC and CISC are used, and because different VLSI technology is used, for example, Sun Ultra1-167Mhz has better performance than SPARC5-170Mhz.

Now, the feasibility analysis steps are illustrated as follows. The best-host approach (see [9]) without consideration of resource availability does not guarantee load balance. Therefore, this step distinguishes feasible hosts in terms of resource availability based on the unified resource. Furthermore, in formula (2), the *thresholds* for the load balancing process include CPU idle time and available memory; the current CPU and memory usage of the process that is to be migrated are compared against the thresholds to determine the destination host. If a host satisfies the condition of feasibility analysis in formula (2) and no faults are detected on the host, then it is a candidate host.

$$(Feasible_{CPU}(H_i,t) = CIP(H_i,t) - CUP_{uni}(a_{ij}, tl, H_i)) > Threshold\_CPU(H_i) \;\&$$
$$(Feasible_{MEM}(H_i,t) = FAM(H_i,t) - MEM_{obs}(a_{ij}, tl, H_i)) > Threshold\_MEM(H_i) \quad -(2)$$

Finally, the optimization step is explained. Optimized resources give good information to RM for efficient allocations.

Definition: *Laxity* is an available amount of unified resources after allocation of requested resources delivered from QM for the violated applications.

```
1.  QM request resources, CUP_obs(a_ij, tl, H_L), MEM_obs(a_ij, tl, H_L)
2.  Get the host list, HL, including host load indices, load metrics(LM)
3.  Calculate EMA of LM
4.  No_of_Candidate_Host = 0 ;
5.  Create Linked List of HL_CPU ;
6.  Create Linked List of HL_MEM ;
7.  For (k = first(HL(H_i, t)) ; k <= last(HL(H_i, t)))
8.      CUP_uni(a_ij, tl, H_k) = CUP_obs(a_ij, tl, H_L) * SPEC_RATE(H_L) / SPEC_RATE(H_k);
9.      Feasible_CPU(H_k, t) = CIP(H_k, t) - CUP_uni(a_ij, tl, H_k) – Threshold_CPU(H_k);
10.     Feasible_MEM(H_k, t) = FAM(H_k, t) - MEM_obs(a_ij, tl, H_k) – Threshold_MEM(H_k);
11.     If(Feasible_CPU(H_k, t) > 0) && (Feasible_MEM(H_k, t) > 0 )
12.       l_CPU(H_k,t)= Feasible_CPU(H_k, t) ) * SPEC_RATE((H_k) ;
13.       l_MEM(H_k,t)= Feasible_MEM(H_k, t) ) ;
14.       Append H_k and l_CPU(H_k,t) to HL_CPU ;
15.       Append H_k and l_MEM(H_k,t) to HL_MEM ;
16.       No_of_Canidate_Host ++ ;
17.     // end if 11
18. Loop 7;
19. Sort HL_CPU in descending order of l_CPU(H_i,t)
20. Sort HL_MEM in descending order of l_MEM(H_i,t)
21. If ( No_of_Candidate_Host == 0 ) Return(Target_Host = first(HL_CPU))
22. Target_Host = first(HL_CPU) ;
23. While(true)
24.  If((Target_Host is Alive) && (Target_Host is in top 50th percentile of HL_MEM))
25.        Return (Target_Host) ;
26. Else Target_host = next(HL_CPU) ;
27.  Loop 23 ;
```

Figure 2. Resource allocation algorithm

$Feasible_{CPU}(H_i,t)$ is the available amount of resources after allocation of a_{ij}. Unifying the $Feasible_{CPU}(H_i,t)$ gives the optimized resource availability. This optimization is an important QoS factor. Formula (3) and (4) show the Laxity of CPU, $l_{CPU}(H_i,t)$, and Laxity of memory, $l_{MEM}(H_i,t)$.

$$l_{CPU}(H_i,t) = Feasible_{CPU}(H_i,t) * SPEC\_RATE(H_i), \quad (3)$$

$$l_{MEM}(H_i,t) = Feasible_{MEM}(H_i,t) \quad (4)$$

Based on optimized resources, the resource allocation schemes, max-laxity host(Λ_{Max}) shown in formula (5), and min-laxity host(Λ_{min}) shown in formula (6) are carefully considered. Other approaches such as random(*ra*), round-robin(*rr*), and least-load($\ell\ell$) have been tested and compared with our allocation schemes of resource optimization. But the least load approach (resources are not unified) shown in formula (7) does not guarantee QoS requirement as the available resources in the supply space do not correspond to resource requirement in demand space.

$$\Lambda_{Max} = Max_i(l_{CPU}(H_i,t)) \text{ and } Top_i(l_{MEM}(H_i,t), 50) \quad (5)$$

$$\Lambda_{min} = min_i(l_{CPU}(H_i,t)) \text{ and } Bot_i(l_{MEM}(H_i,t),50) \quad (6)$$

$$\ell\ell = Max_i\,(CPI(H_i,t)*W_{cpu} + FAM(H_i,t)* W_{mem}) \quad (7)$$

where $Top_i(l_{MEM}(H_i,t), 50)$: the host "i" is in top 50th percentile in laxity of memory,

$Bot_i(l_{MEM}(H_i,t), 50)$: the host "i" is in bottom 50th percentile in laxity of memory,

$$W_{cpu}+W_{mem} = 1, \ \forall_i$$

In our approach, the other resource requirements like network bandwidth, I/O disk are applicable in a similar way. The final decision is made based on the laxity of each resource using heuristic algorithm: find a host that has maximum Λ_{CPU}, if the host is in top 50th percentile of the host list (sorted by $\Lambda_{MEM}(H_i,t)$); select the host; if not, examine the next host that has maximum $\Lambda_{CPU}(H_i,t)$. Figure 2 explains resource allocation algorithm in detail.

Instead of resource allocation, control of heterogeneous resources is an efficient way to provide quick resource management. Dynamic CPU proportion change on Linux using the Quasar scheduler [10][11]

and priority handling on NT and Solaris are implemented in our scheme.

Furthermore, for accurate allocation, the RM should consider not only load balance based on resource availability, but also a measure of system contention called *slowdown* factor. This is ongoing study, especially in the area of network load between two communication nodes.

3. Experiments

We have used DynBench[12] as an assessment tool for DeSiDeRaTa. It uses an identical scenario for experiments. The experimental system parameters and heterogeneous environment are as follows: 1 Linux Pentium 200mhz, 1 NT Pentium-III 500mhz, 2 NT Pentium-II 400mhz, 2 NT Pentium 200mhz, 2 Solaris Sparc-5 170mhz, 2 Solaris Ultra-1 167mhz, 1 SunOS on ULTRA10 300mhz, and 100Mhz Fast Ethernet.

The first experiment monitors and analyzes resource requirements corresponding to step 1 in section 2. The second experiment measures the unification approaches corresponding to step 3 in section 2. The third experiment compares different allocation schemes corresponding to step 5 in section 2. Experiment details are presented in the remainder of this section.

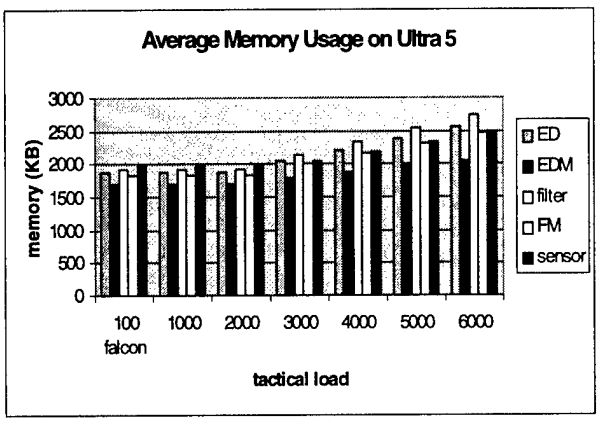
Experiment 1 shown in Figure 3 describes the measures of variance of execution time with different methods and different periods in (c), and variance of memory among hosts in (a) and (b). The three different monitoring techniques, *getrusage()(GRU)* system call, reading *process table(PT)*, and *ps(PSU)* command, are used. From the experiments (c), the variance of execution time measured by reading *PT* is high, and is dependent on the monitor cycle time as the period for accessing PT cannot exactly cover the range of process execution time. It is impossible to collect exact resource usage of a process at a particular instant of time. However, the GRU system call shows accurate process resource usage in terms of variance of execution time. The exponential moving average (EMA) of each method is used for filtering. The maximum difference of memory usage by the evaluate and decide application(ED) on two different hosts is 48Kbytes(from (a) and (b) in Figure 3). Hence, Threshold_MEM(H_k,t) and Threshold_CPU(H_k,t) are necessary components to constraint candidate host. The variances of memory requirement of applications are measured by zero.

The second experiment described in Figure 4 shows the difference between observed resource usage and unified resource estimated by SPEC_RATE and Clock_Rate(CCR). For example, the execution time is collected on Pentium-200, and we multiply the measured execution time and SPEC_RATE/CCR of the target host, PentiumIII-500. Next we experiment with the same scenario on PentiumIII-500 to observe actual execution time of the process to compare with previous estimated execution time. The difference between unified resource by CCR and the observed resource is 8% on NT, and 3.5% on Sun. The difference of unified resource by SPEC_RATE has 1% on NT, and 11% on Sun.

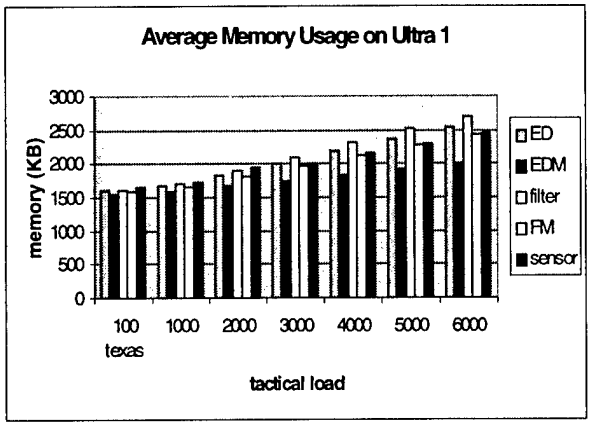
Experiment 3 proposed three measurements - QoS violation rate (QVR), QoS Sensitivity (QSS), and QoS (to compare QoS characteristics by different allocation decision algorithm as shown in Figure 5). The QVR is the number of violations within 2 minutes by increasing workload. QSS is the amount of workload to trigger second violation after the first violation. QoS is the latency of a path improved by first allocation. This experiment shows clearly that the $\ell\ell$ approach that ignores heterogeneity (proposed by Ravindran [9]) is much worse than our scheme in terms of QVR, QSS, and QoS. Our Λ_{Max} scheme improves 26.4% better in QoS, 36.4% better in QSS, and 60% less in QVR than $\ell\ell$ approach.

4. Conclusion and Future Study

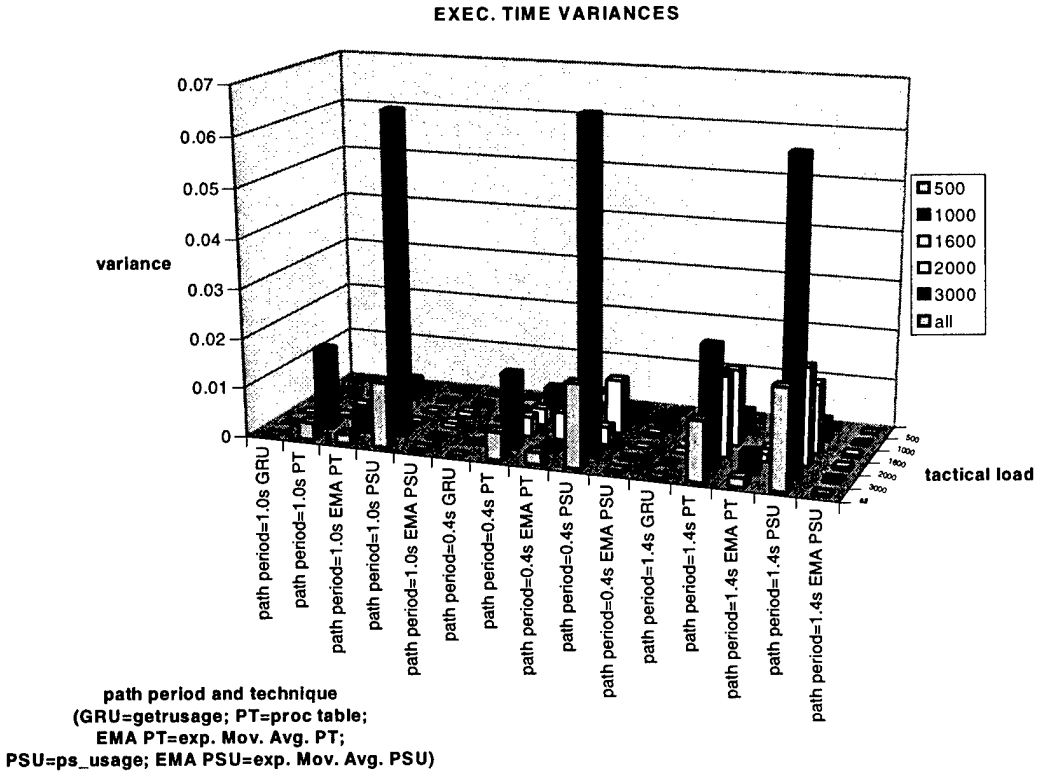
This paper presents 5 solutions of resource allocation for dynamic real-time systems. Our Λ_{Max} scheme not only allows higher workloads than *ra, rr, and* $\ell\ell$ *by* 257%, 142%, and 36.4%, respectively, but also improves QoS better than *ra*, rr, and $\ell\ell$ by 38.6% 28.5% 31.6%, respectively. Controlling heterogeneous resources using CPU proportion change and priority change is useful for the server programs. The efficiency of resource allocation in terms of QoS objectives for scalable and moveable clients is better than that of the control. Ongoing work includes finding a specific solution of the resource management for hard-real time applications, Predictive RM, Proactive RM, and QoS negotiation. Also, heterogeneous network resource monitoring and allocation, and the decision mechanism between allocation and control is an important issue in providing QoS requirements.



(a) MEM(a_{ij}, tl, Sun-Ultra-5)

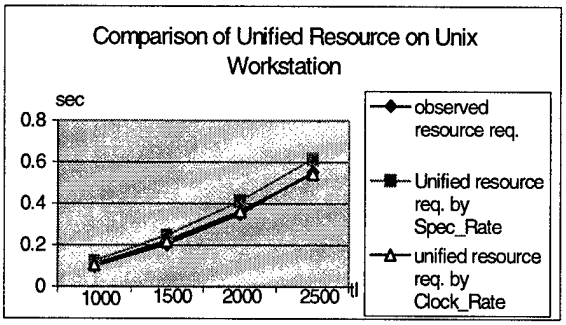


(b) MEM(a_{ij}, tl, Sun-Ultra-1)

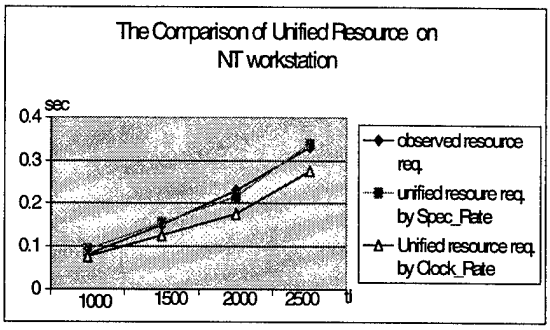


(c) Variance of execution time

Figure 3. The Dynamic measures of monitored resource requirement

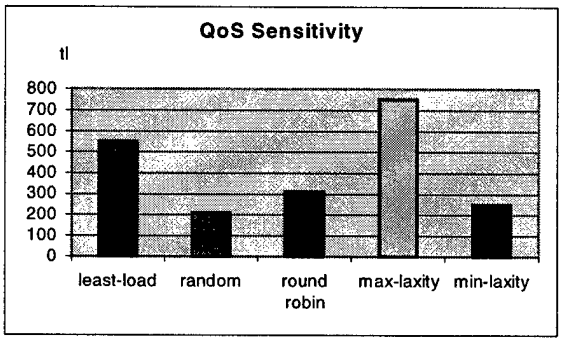


(a) Pentium (200MMX) vs Pentium-III (500Mhz)

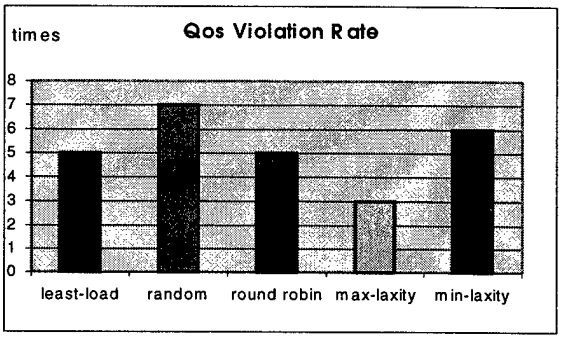


(b) Ultra-10(300Mhz) vs Ultra-1 (140Mhz)

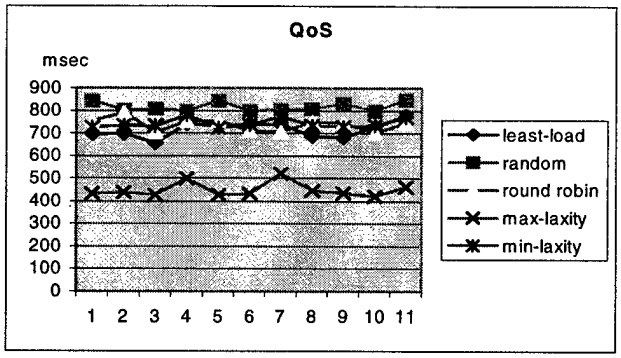
Figure 4. Resource unification by SPEC_RATE and Clock_Rate



(a) QSS



(b) QVR



(c) QoS

Figure 5. Comparison of resource management schemes

References:

[1] L. R. Welch, B. Ravindran, R. Harrison, L. Madden, M. Masters and W. Mills, "Challenges in Engineering Distributed Shipboard Control Systems", *The IEEE Real-Time Systems Symposium*, December 1996.

[2] L.R Welch, B. Ravindran, B. Shirazi and C. Bruggeman, "Specification and Analysis of Dynamic, Distributed Real-Time Systems", *in Proceedings of the 19th IEEE Real-Time Systems Symposium*, 72-81, IEEE Computer Society Press, 1998.

[3] J. Stankovic, and K. Ramamritham, *Advances in Real-Time Systems*, IEEE Computer Society Press, April 1992.

[4] S. Son, *Advances in Real-Time Systems*, Prentice Hall, 1995.

[5] High performance distributed computing, http://www.nswc.navy.mil /hiperd/index.html

[6] L.R. Welch, P V. Werme, B. Ravindran, L. A. Fontenot, M.W.Masters, D. W. Mills, and B. Shirazi, "Adaptive QoS and Resource Management Using A posteriori Workload Characterizations", *IEEE Real-time Application System*, 1999.

[7] Gary Koob, "Quorum", *Proceedings of the DARPA ITO General PI Meeting*, pages A-59 to A-87, October 1996.

[8] OSG group, *SPEC CPU95 Benchmark*, http://www.spec.org.

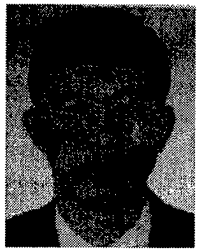
[9] Binoy Ravindran, "Modeling and Analysis of Complex, Dynamic Distributed Real Time System", *Thesis*, Computer science and engineering, The University of Texas at Arlington, 1998.

[10] Ashvin Goel, David Steere, Calton Pu, Jonathan Walpole, "Adaptive Resource Management via Modular Feedback Control", *http://www.cse.ogi.edu/DISC/ projects/ quasar/publications.html.*

[11] David C. Steere, Ashvin Goel, Joshua Gruenberg, Dylan McNamee, Calton Pu, and Jonathan Walpole, "A Feedback-driven Proportion Allocator for Real-Rate Scheduling", *Operating Systems Design and Implementation (OSDI)*, Feb 1999.

[12] L.R. Welch and B. Shirazi, "A Dynamic Real-Time Benchmark for Assessment of QoS and Resource Management Technology", *IEEE Real-time Application System*, 1999.

Biographies:

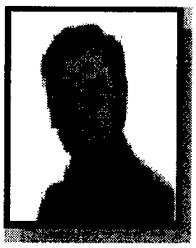


Eui-Nam Huh is currently a Ph.D. student in School of Electrical Engineering and Computer Science at Ohio University. His current research area is resource management on dynamic distributed real-time systems. He received his Master of Computer Science degree in computer science engineering from The University of Texas at Arlington in 1995 and Bachelor degree in computer science and statistics from Pusan National University in 1990. He worked for Korea Fidelity and Surety Company as a programmer (online) and network administrator for 3 years (1990-1993). He had full time faculty position at computer science department in Korean Sahmyook University for 2 years (1996-1997). He had developed a high performance distributed system for the academy service while in Korean Sahmyook University.

Lonnie R. Welch is a Professor in The School of Electrical Engineering and Computer Science (EECS) at Ohio University (OU). He received the Ph.D. degree in Computer and Information Science from the Ohio State University in 1990. Previously, he was a faculty member at the University of Texas at Arlington and at the New Jersey Institute of Technology. As Director of the Laboratory for Parallel and Distributed Real-time Systems, he conducts research in the areas of real-time systems, distributed computing, security, software/systems engineering, and dependability.

As PI of the DARPA/Quorum DeSiDeRaTa project, he has invented the dynamic path paradigm for addressing dynamic resource management for distributed mission critical real-time systems; the primary products of the DeSiDeRaTa project are middleware for distributed resource management, and a dynamic real-time benchmark suite that has characteristics similar to experimental shipboard computing systems. He has succeeded at transition and integration of DeSiDeRaTa technology into the High Performance Distributed Computing (HiPer-D) laboratory at the Naval Surface Warfare Center. Furthermore, DeSiDeRaTa technology has been selected for inclusion in the Quorum reference implementation. In addition to his work with DARPA, Dr. Welch's research has in adaptive resource management for real-time distributed systems been funded by numerous contracts from the Navy and the Army. Additionally, he was appointed as a Summer Faculty Fellow by the US Army in 1991, and by the US Navy during the summers of 1993, 1994 and 1995. During the 1995-1996 academic year, he served as a Visiting Research Scientist at the Naval Surface Warfare Center in Dahlgren, VA. Welch received an "Aegis Excellence Award" from the United States Navy in 1998 for his contributions in dynamic resource management to the High Performance Distributed Computing (HiPer-D) and SC-21 programs.

Dr. Welch is involved in many ACM and IEEE activities, and he has published over 70 articles in professional conferences and journals. He is the founder and Steering Committee Chair of the International Workshop on Parallel and Distributed Real-Time Systems. Additionally, he serves on the editorial boards of The Journal of Parallel and Distributed Computing Practices and The International Journal of Mini- and Micro-Computers, and he has been Guest Editor of several journal special issues, including ACM OOPS Messenger (issue on Object-Oriented Real-time Systems), and The Journal of Parallel and Distributed Computing (issue on Parallel and Distributed Real-time Systems).



Behrooz A. Shirazi is a Professor and Chair of the Computer Science and Engineering Department at UTA. Before joining UTA in 1990 he was on the faculty of Computer Science and Engineering at Southern Methodist University. Dr. Shirazi has conducted research in the areas of software tools, distributed real-time systems, scheduling and load balancing, and par-

allel and distributed systems over the past fifteen years. He has advised 12 PhD, 56 MS, and 40 BS students and has published over 100 papers in these areas. Dr. Shirazi's research has been sponsored by grants from NSF, DARPA, AFOSR, Texas Instruments, E-Systems, Mercury Computer Systems, and Texas ATP for a total of more than $2.5M. He has been a Guest-Editor of a special issue of the Journal of Parallel and Distributed Computing and a Track Coordinator of the HICSS'93 Conference, both on "Scheduling and Load Balancing Issues". He is the leading author of an IEEE Press book entitled "Scheduling and Load Balancing in Parallel and Distributed Systems". Dr. Shirazi is currently on the editorial board of the Journal of Parallel and Distributed Computing. He is the principal founder of the IEEE Symposium on Parallel and Distributed Processing and has served on the program committee of many international conferences. He has received numerous teaching and research awards and has served as an IEEE Distinguished Visitor (1993-96) as well as an ACM Lecturer (1993-97).

Charles D. Cavanaugh is currently a Ph.D. candidate in computer science at The University of Texas at Arlington. His current area of research is distributed real-time systems. He received his Master of Science and Bachelor of Science degrees *summa cum laude* in computer science from The University of Texas at Tyler in 1997 and 1995, respectively; and he received his Associate of Arts degree *magna cum laude* in interdisciplinary studies from Tyler Junior College in 1993. He is a member of the Tau Beta Pi Engineering Honor Society, the IEEE Computer Society, and the ACM.

A Cost/Benefit Model for Dynamic Resource Sharing

Dimitrios Katramatos
Dept. of Computer Science
University of Virginia
Charlottesville VA, 22903
dk3x@cs.virginia.edu

Deepak Saxena, Nehal Mehta, Steve J. Chapin
Dept. of Electrical Engineering and Computer Science
Syracuse University
Syracuse, NY 13244
{deepakas,nvmehtaaq}@hotmail.com, chapin@ecs.syr.edu

Abstract

The use of multicomputer clusters composed of cheap workstations connected by high-speed networks is common in modern high-performance computing. However, operating system research in such environments has lagged. Our research aims at enhancing the functionality of the operating system by providing management functions that allow dynamic resource sharing and performance prediction in a clustered environment supporting distributed shared memory and multithreading. Central to this approach is the development of a parametric cost model that can predict the performance ramifications of policy choices and allow applications and middleware to adapt to the computing environment and achieve better performance.

1 Introduction

The goal of our research is to develop and evaluate a parametrical cost/benefit model to be used as a decision-making tool for managing dynamic system resource sharing in an environment of high-performance heterogeneous clusters. Multicomputer clusters are capable of achieving computational rates equal or higher than those of conventional supercomputers. However, the performance these systems deliver to applications is usually just a fraction of their maximum capacity, even in cases of applications that can theoretically achieve much higher computation rates. Software inefficiency has to be blamed for this phenomenon. We are developing mechanisms that better share system resources and promote parallelization of tasks. These mechanisms allow applications to self-adapt to the varying availability of system resources according to their own varying resource requirements and thus run more efficiently.

The key idea in this research is the notion of cost, in terms of execution time, and the ramifications of certain operating system services (including process/thread creation, process/thread placement, and inter-process communication). This cost varies with several parameters such as application requirements, application behavior, time, system configuration, and network topology. Realistic prediction of the cost of system services and choices gives an application the ability to make its own decisions regarding the execution environment that best suits its needs.

We are therefore developing a parametric cost model for predicting the cost of certain operating system services while taking into account system characteristics and application information, as well as a set of software extensions to the operating system to support the function of this model and facilitate the dynamic sharing of system resources. The result will be an operating system with flexible resource control, able to deliver a higher percentage of underlying system performance to applications and middleware.

Section 2 of this paper presents the motivation for this work, section 3 the hardware and software environment under consideration, and section 4 the family of applications of interest. Section 5 describes our approach in more detail, while section 6 discusses related work. Section 7 gives a brief summary.

2 Motivation

In recent years there has been a new trend in high-performance computing: the use of multicomputers built from common, off-the-shelf components. These systems are capable of sustaining computation rates that rival or surpass the rates sustained by conventional supercomputers, but the demonstrated performance is just a fraction of the maximum theoretical performance. For example, the Centurion cluster of phase I achieved 3.7 sustained gigaflops on 49 CPUs during a run of an ocean modeling application [1]. This code gets just 650 megaflops on a single Cray T90 CPU. While this comparison is encouraging, Centurion of phase I had a maximum capacity of over 60 gigaflops. While it would be naive to expect to sustain a rate of computation approaching the maximum, there is certainly room to improve the performance substantially, for several cat-

0-7695-0556-2/00 $10.00 © 2000 IEEE

egories of applications. We identify software overhead as one of the primary culprits in this reduced performance issue. To overcome this overhead we have each software layer expose to higher layers information describing the behavior of the individual layer and implications that behavior has for performance. Also, the application provides information regarding its own behavior to lower software layers as hints to let them better estimate the impact on performance.

Our approach chooses to work from the bottom up, because all applications, regardless of middleware system, use the operating system. By adding a set of services to the operating system, if possible only as user-level modules for reasons of portability, we aim to allow dynamic resource sharing and also provide a resource consumer (not necessarily only an application) with information about the cost of a specified operation. For example, runtime systems will be able to get information to make service guarantees to applications, and applications will be able to self-adapt to various system configurations and varying resource availability. Load balancing environments will be able to better estimate system efficiency and better understand the effect various task mappings have on performance. In these environments dynamic granularity control will also be possible. Finally, smart compilers will be able to take into account operating system cost information during the compilation phase of an application to optimize the executable code for specific system configurations and loads. To complement this, it will also be possible to have an application loader that will take into account application supplied information and create the most suitable environment within a system's limits for running a specific application.

3 The Hardware and Software Environment

The environment in which the problem is considered is a distributed system consisting of a variety of nodes connected with networks of various speeds. We view such a system as having a physical and a logical organization.

The physical organization of the system is based on the principle of hierarchical clustering [13]. Groups of neighboring nodes form local clusters. Local clusters can be grouped together and form second level superclusters, second level supercluster groups can form third level superclusters, etc. (see figure 1). The main criterion for forming the various hierarchy echelons is the cost of communications—these echelons reflect the underlying networks. That is, echelon 0 corresponds to communications between processors of the same node (intra-node)—if nodes have more than one processor—which have hardware shared memory and thus the least cost for sharing information. Echelon 1 corresponds to intra-cluster communication, echelon 2 to cross-cluster communication between clusters of the same 2nd level supercluster, echelon 3 to cross-cluster communication between clusters of different 2nd level superclusters, etc. Another way to think of this organization is as a system map.

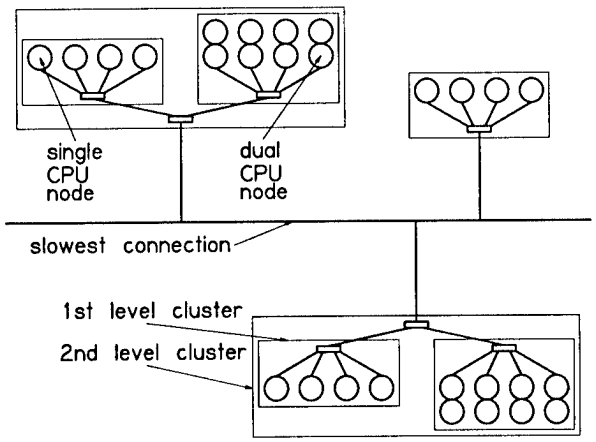


Figure 1. A clustered multicomputer - physical clustering

Over the physical clusters is laid an organization of logical clusters. The nodes of a logical cluster share information using distributed shared memory. Note the difference between physical and logical clusters: physical clusters are defined by the physical organization of a system, mainly by the nature of the interconnecting network, and are fixed; logical clusters are groups of processors that share memory using software-based distributed shared memory (DSM) and can change dynamically. Such a cluster can be a portion of a physical cluster, a whole physical cluster, or it can span several physical clusters incorporating parts or the whole of them (see figure 2)).

An important aspect of the environment is the support for threads and for distributed shared memory. The elementary computing entity here is the thread and an application includes a dynamically changing number of threads. The enhanced operating system provides a single address space for the threads of an application. This address space is visible by a number of processors within the logical cluster structure and the threads are mapped on these processors without the knowledge of the application. Thread migration is possible in this environment, as well as migration of thread groups, even whole applications, within (intra-cluster) or between logical clusters.

The choice of software DSM as the means of forming a logical cluster is a trade-off between raw performance and ease of programming. The pros and cons of the message passing and the distributed shared memory paradigm are known and there have been numerous publications in this

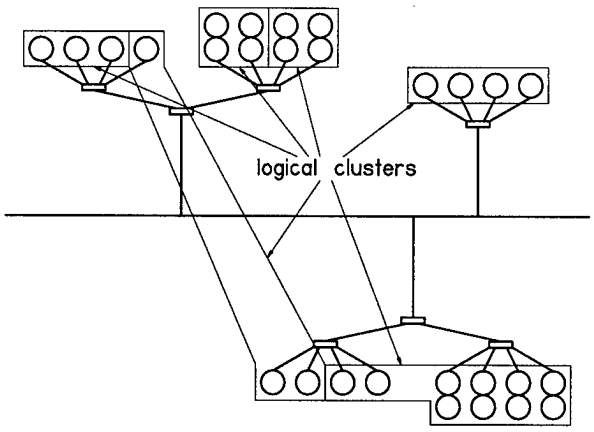


Figure 2. Logical clustering

area. We believe that there is a lot of room for improving the performance of certain categories of applications on multicomputer clusters. We expect that careful use of DSM in combination with cost/benefit prediction and dynamic adaptation to resource demand/availability will demonstrate definite performance improvements over the current figures without sacrificing usability and programming ease on the altar of performance. In any case, the idea of using a cost model to predict the cost/benefit of certain operating system operations is independent of using one paradigm or the other and can contribute in all cases to the better administration of a system's resources.

4 The Applications

The type of application of interest is a parallel application with high resource demands, such as a large scientific simulation. This type of application usually presents a dynamic change of resource requirements, that is, it presents irregularity: "data structures, communication patterns, or computation are not defined by simple, repeating structures" [4].

Our computational model mixes shared-memory programming (done with threads) with distributed-memory programming (done with a message passing environment such as MPI). Our initial work was to support Pthreads, the POSIX-standard threads library, but we are moving to support the emerging standard for multiprocessing, OpenMP, in conjunction with MPI.

As an example of this application type, consider a weather simulation, tracking the progress of a storm front as it moves across a landscape which has been mapped onto a grid. Each grid cell can be mapped onto a multithreaded MPI process. The grid cells containing the storm front will require the most computation, and as the front moves, the computational hotspots will move across the grid. To achieve good performance, we want to rebalance the workload. Historically this has been done either through data migration or computational migration.

Static thread allocation for an application with dynamic resource requirements leads either to processors sitting idle, when resource usage is overestimated, or to delay from load imbalance when usage is underestimated. Clearly, such a computation can only be performed efficiently with the combination of a load balancing technique with dynamic granularity control [3]. Using our approach, one can add extra computational power within the address space of a hotspot, thus spreading the load to more threads. In the example of storm front simulation, we can add threads to the cells containing the front (these new threads might be put on local processors, remote processors via DSM in an expanded logical cluster, or we might even choose to migrate the entire process to a larger SMP and add threads there). Once the front passes, the extra threads are no longer necessary and can be killed, freeing the occupied processors. Each multithreaded MPI process can be assigned to a logical cluster with a certain number of nodes that share memory and provide the illusion of a single address space to the threads of the process, while each thread runs on a different CPU. The MPI processes communicate with cross-cluster messages. Processors can be dynamically added to/removed from each logical cluster in response to load variations.

5 Our Approach

Our work is based on Linux, which is the de facto standard for free software cluster operating systems. This has the obvious advantage of allowing us to add code to the kernel as we deem necessary. We are building two sets of software to support our needed functionality: kernel extensions and user-level libraries. Our work to date has focused on determining the required functionality, and the line between what should go in the kernel and what should go in the user library is not yet set.

5.1 Additional Functionality

We are augmenting Linux with heterogeneous distributed shared memory and multithreading functionality. Functionality similar to the Mermaid prototype [8] is desired as well as conformation to the OpenMP standard [17] and the popular Pthreads interface.

The additional functionality that we are currently adding to the system is as follows:

1. Add a thread or a group of threads to an existing shared address space.

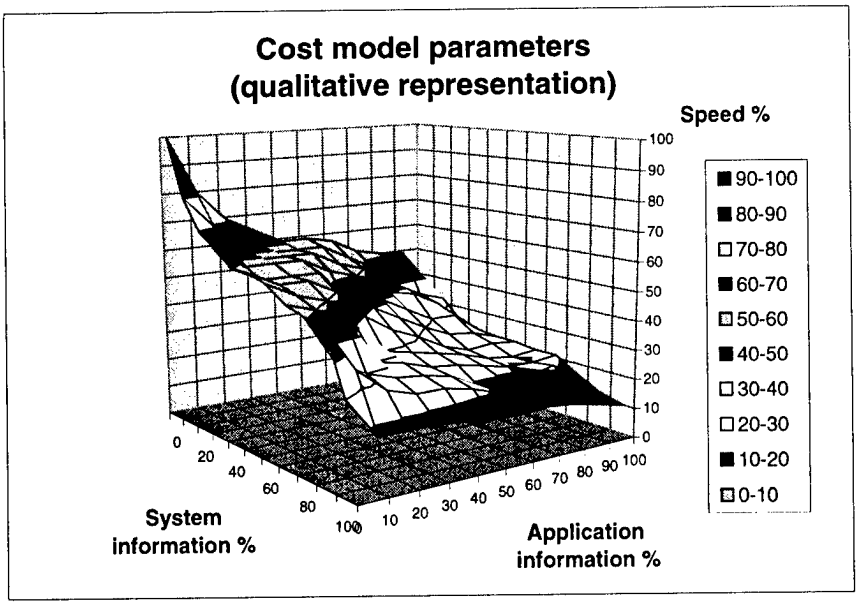


Figure 3. The main parameters of the cost model

2. Add a processor or remove a processor from a shared address space.

3. Dynamically migrate a thread from one processor to another. The target processor should automatically be added to the shared address space, if not already included, and the source processor should possibly be removed, if no other thread runs on it. Removing a processor from an address space might not always be desirable as it can cause thrashing on processor add/remove.

4. Form, modify, manage, and cancel a logical cluster of nodes. These functions will correspondingly create, modify, manage, and delete the necessary data structures that need to be maintained to support logical clusters.

5. Evaluate cost functions to estimate the cost/benefit of performing certain operations, as for example to create/cancel a cluster, add/remove a processor to/from an address space, start/kill a thread on a specific processor, and migrate a thread, which may result to the inclusion of the target processor in the shared address space and/or the removal of the source processor, as mentioned above.

5.2 The Cost/Benefit Model

The operating system extensions are mechanisms, not policies. Policy decisions will be made either in middleware layers or by the application itself, e.g. whether or not to add a thread to a running process. To assist the higher software layers in making these decisions, we have developed a cost model so that the operating system can accurately predict the ramifications of policy decisions.

Our cost model moves qualitatively along three axes: the first axis has as parameter the amount of system information taken into account; the second axis, the amount of application information; and the third, the speed of the cost estimation. Figure 3 represents a qualitative picture of the cost model behavior. The accuracy of the model is generally inversely proportional to the speed of the estimation. The speed of the model depends on the type and quantity of application and system information that must be processed. Accuracy increases and speed decreases as the volume of information increases. The cost of a cost estimation has to be low when compared with the benefit that a correct policy decision will offer. On the other hand, the estimated cost needs to be accurate enough to enable correct decisions. Clearly, this is a complex problem and there are several factors that need to be considered. Therefore, the cost model needs to be parametric and provide for various trade-offs of

accuracy for speed.

In its simplest form, the cost model views the cost of operations as set of discrete cost classes. This classification is of the least accuracy and maximum speed and it works when a fast answer is needed and no information is available from the application side. As an example, in a system with two clusters where one cluster is formed by dual-processor x86 boxes and assuming 3 cost classes, the cost of adding a thread to the second processor of a node belongs to class #0. The cost of adding a thread to a new (unused) processor in the local cluster belongs to cost class #1. The cost to migrate a thread to the neighboring cluster is of class #2, the highest. Here the accuracy of the model is the lowest as it relies only on basic system information.

In its richest form, the model can provide a set of cost information about an operation and the impact this operation will have on the performance of the application. For this purpose, it is necessary to combine information from both the system and the application. The system maintains a vector of state and cost information by directly accessing information maintained by the operating system kernels and by periodically running a set of suitable benchmarks to measure other important cost-affecting quantities, like communication latencies or node loads (cf. the Network Weather Service [26]). The application passes in a description of its behavior, e.g. memory access patterns, thread running patterns, acceptable delays, etc. Statistical information, gathered during previous runs of the application, can help identify its behavior when this behavior is not known in advance. The cost model combines the supplied information to produce a cost rating, a quantity that can be used in comparing the cost of different system operations.

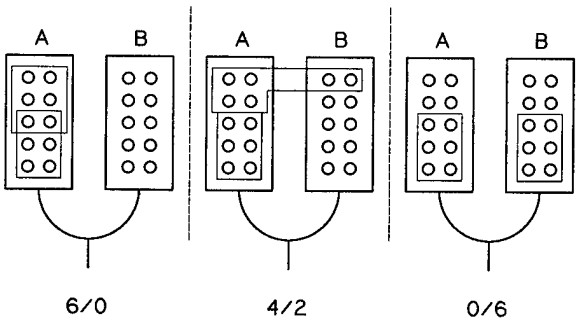


Figure 4. Three possible configurations

For example, consider two neighboring (physical) clusters, A and B, with 10 processors each, and an application running on six processors of cluster A, which form a logical cluster, with six threads, one per processor (see figure 4)). Suppose that at a certain phase during execution the application wants to spawn another group of six threads. Would it be better to use six processors of cluster A (4 free ones and two already running a thread each), use the remaining 4 processors of cluster A and two more from cluster B, or use six processors of cluster B instead? Cross-cluster communication is more expensive than intra-cluster communication so the first option seems more attractive than the second and the second more attractive than the third. However, this is not necessarily the best order. The first option suffers from the fact that two processors have to run two threads each. Also, what is of highest benefit depends on how the application behaves. If the new group of threads works in close cooperation with the initial thread group, then the first option is probably better. If the group of these six new threads performs a separate task and communicates infrequently with the initial thread group, that is, data shared between the two groups are infrequently accessed, then the third option is probably better. This is because the infrequent communication between the two thread groups keeps the communication cost low. In this case, if the second option were chosen, the frequent communication between the two threads running on cluster B with the other four on cluster A would impose a heavy increase in the communication cost.

The following is a cost rating estimation for the three thread placement options considered. Since a highly detailed estimation would be too long to present here, certain simplifying assumptions are necessary. The first assumption is that the system costs are identical for all processors. A more detailed estimation would include the cost of performing certain operations on each individual node, or type of node. A second assumption is that the costs for sending messages are estimated using the basic LogP model [16]). Consider now the following costs:

s cost to start a thread on a processor,

e cost to include a processor in a shared address space,

e_r cost to include a processor that belongs to a neighboring cluster in a shared address space,

c_i cost for sending a message across the intra-cluster network,

c_c cost for sending a message across the cross-cluster network, $c_c \geq c_i$,

Let the application supply information about itself in the form of two weight factors, w_g, and w_r. w_g expresses the percentage of accesses to variables shared between the new group of threads, and w_r, expresses the percentage of accesses to variable shared between the old and the new thread

groups. $w_g + w_r = 100\%$. The application declares that the threads of the new group communicate frequently with each other by having $w_g \gg w_r$, that is, intra-group communication is much more frequent than between groups. We'd like to compare the three basic thread group placement options, in the present case all six threads on cluster A (option 6/0), four threads on cluster A and two on B (option 4/2) or all six on cluster B (option 0/6).

The first thing to estimate is the overhead for spawning the new thread group: for the 6/0 option $6(s + \frac{4}{6}e)$, for the 4/2 option $6(s+\frac{4}{6}e+\frac{2}{6}e_r)$, and for the 0/6 option $6(s+e_r)$. Since $e_r \geq e$, the 0/6 option has more overhead than the 4/2 option, and that in turn more than the 6/0 option, which is expected.

This overhead is not enough to make a decision. Here the cost rating has to have at least two parts, a fixed part, the overhead, and a variable part, a rating for the communication and the running cost.

For estimating the communication cost rating, assume that m is the number of messages associated with the access of shared data by threads of the new group per time unit (message rate). Then the 6/0 option has a rating of $m(w_g + w_r)c_i = mc_i$. For the 4/2 option assume a split factor k to distinguish between the messages send within A and B and those that cross the boundary between them. That is, $k\%$ of messages are intra-cluster within clusters A and B, and $(1-k)\%$ cross-cluster between A and B. Then the cost rating is $m(kc_i + (1-k)c_c)$. Finally, for the 0/6 option the rating is $m(w_g c_i + w_r c_c)$ To compare the rating of 0/6 to that of 4/2 we subtract the first from the second. After the calculations we obtain $m(c_c - c_i)(w_g - k)$. Because $c_c \geq c_i$ and $w_g \gg w_r$ this quantity is positive when $w_g > k$. Having $w_g < k$ is incompatible with the assumption that the threads of the new group communicate frequently with each other and infrequently with the old group; the value of w_g is close to 100% and a value for k approaching w_g would mean that the communication between the subgroup of the two threads and that of the four threads is infrequent, in direct contrast with the above assumption. Thus, the cost rating for the 4/2 option is higher than that of the 0/6 option. In turn, the cost rating of 0/6 is higher, but only slightly, than 6/0. So far, 6/0 has the lowest overhead and cheapest communications, with 0/6 second and 4/2 last.

To make the final decision, it is necessary to estimate the cost of running for the three options. Assume a time period Δt. During this period the new thread group causes the sending of $m\Delta t$ messages. Also assume that during Δt the maximum number of thread instructions available for execution are I_t and that a processor can execute x instructions per time unit.

For cases 4/2 and 0/6, each thread will take I_t/x time. For the 6/0 case, two of the threads are running essentially at half speed $x/2$ since they run on already occupied processors. Assuming that in general we have CPU-bound threads, these two threads will take double the time of the other four threads, $\frac{2I_t}{x}$, probably delaying their work (the interference pattern of the threads is another factor to take into account for more accurate estimations). It can be concluded now that option 0/6 is probably the best choice, because it has almost the same communication cost rating with 6/0 but half the running cost rating, and in the long run the overhead can be ignored.

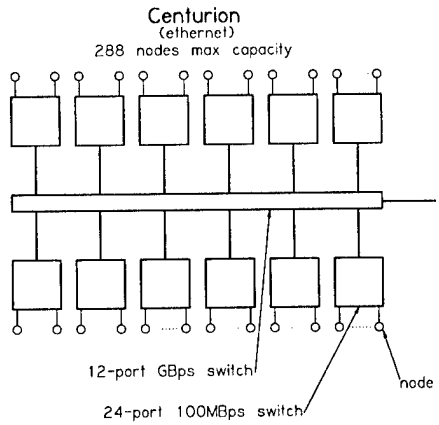


Figure 5. Centurion, ethernet connections

There are other factors that can affect the cost of each option. For example, if clusters A and B contain processors of different architectures, cross-cluster communication may be much more expensive than before, due to the necessary data conversions. This cost can be counterbalanced if the processors of cluster B are faster than the ones of cluster A, thus providing for higher execution rates. Finally, one has to take into account the effect of the already existing load of the processors and the network as it changes with time, as there can be other applications or tasks sharing the system resources.

5.3 The Experimental Computing Environment

The hardware we are using for experimentation is a metacluster comprising the Centurion machine [2] and the Syracuse Orange Grove cluster. Centurion has 256 nodes, with 128 533MHz DEC Alpha nodes and 128 400MHz dual Pentium II nodes. Each node is connected to a 100Mbps fast ethernet switch and multiple such switches are joined via gigabit ethernet switches (see figure 5). The initial 64 Alpha nodes are also joined in a complex mesh by a 1.2Gbps Myrinet fabric (see figure 6). The Orange Grove cluster has 16 533MHz DEC Alpha nodes and 48 450MHz dual Pentium III nodes, and has a system area network similar to Centurion's. Figure 7 presents a simplified overall view of

the total system. The only real restriction is that each CPU needs to run the modified Linux system.

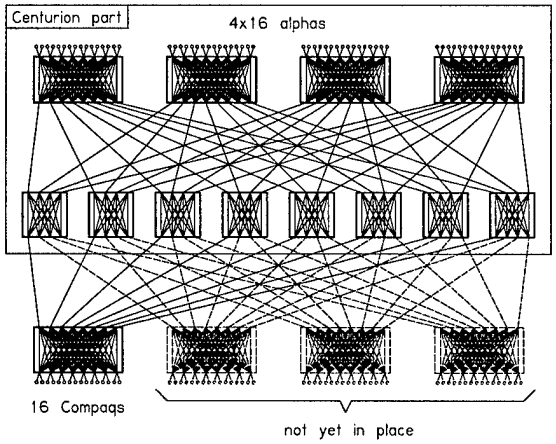


Figure 6. Myrinet connected part of Centurion

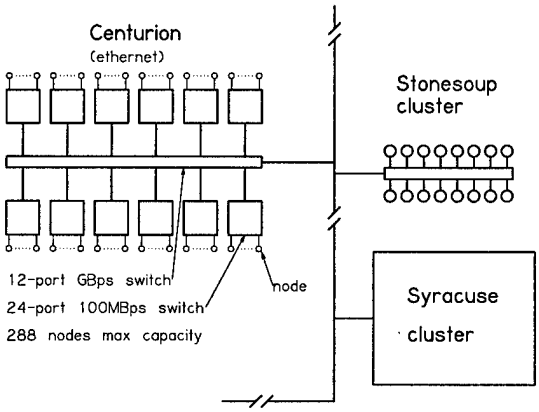


Figure 7. The experimental system

6 Related Work

Much of research work has been and is being dedicated to issues related to clusters of commodity workstations. Our approach is novel, to the best of our knowledge, in the way it supports clustering, in providing the mechanisms for dynamic resource sharing in this framework, and most of all for using cost models to estimate the cost/benefit of certain o.s. operations and aid in decision making. The current section presents work related to our research with respect to operating systems, distributed shared memory, and other relevant areas.

6.1 Related Work in Operating Systems

The first work of interest is a research project of the Computer Systems Research Institute of the University of Toronto, Canada, targeted to hierarchically structured NUMA clusters [14] Part of this project was the design and construction of the Hector hierarchically structured shared memory multiprocessor. Hurricane is the operating system that was specially developed for Hector. The structure of this operating system reflects the structure of the multiprocessor. Hurricane uses tight coupling within a local cluster and loose coupling within clusters. Operating system services are designed to take advantage of high speed connections and locality of data. Cluster size is determined statically. Processes of an application are scheduled on the same cluster, unless there is a benefit for a job to span multiple clusters. Within a cluster load is balanced at a fine granularity and cross-cluster scheduling performs coarse-grain load balancing by assigning and migrating processes to specific clusters. Tests of the system have shown that applications need to be adapted to Hector/Hurricane to perform well.

The computing environment targeted in our research is substantially different from the Hector/Hurricane environment. The Hector/Hurricane approach was an attempt to create a cost-effective, scalable NUMA machine, utilizing specially designed hardware. Experimenting with hardware is not in our intentions. Although the targeted environment maintains the notion of tightly-coupled sub-clusters within a larger hierarchical system, there is no hardware support for shared memory, except for the case of a multiprocessor PC box. Also, architecture homogeneity is not necessary, the cluster hierarchy is not of fixed depth, and the (logical) cluster size is not determined statically. However, there are several lessons from the Hector/Hurricane approach that can be useful to us, e.g. that the factors found to be crucial for application performance depend on application behavior and system behavior. This is exactly what our work investigates. The fact that applications needed to be specifically adapted to run efficiently on Hector under Hurricane underlines the importance of having operating systems that can provide the necessary information and allow applications to self-adapt to the dynamically changing resource availability of a system.

A more recent work is the Berkeley Network of Workstations (NOW) project [18]. This project is targeted at workstation clusters and includes operating-system level work software such as the GLUnix layer [19]. GLUnix is a multi-user, user-level system for a cluster of workstations. It is designed to provide transparent remote execution by maintaining a single-system image across an entire cluster and

supports interactive parallel and sequential jobs and load balancing. The proposed research differs in that it is not dedicated to single-cluster but to hierarchically clustered systems with several cluster levels. Although it is possible to provide a single-system image across clusters and levels of hierarchy, the approach taken here is to provide the users with the necessary data to make informed choices about state sharing issues. In any case, increasing the degree of sharing is usually accompanied by a decrease in performance, and users may be willing to sacrifice transparency for performance. GLUnix supports load balancing through intelligent job placement but doesn't support task migration nor any form of dynamic resource sharing based on cost/benefit prediction.

6.2 Related Work in Distributed Shared Memory

There is a vast variety of research works on distributed shared memory (DSM). A possible classification of DSM systems can be based on:

- the level of implementation (user, kernel, hybrid),
- the coherence protocols (SRSW, MRSW, MRMW, etc.),
- the consistency model (sequential, processor, release, etc.).

Three general techniques have been used [20]. The first technique simulates a multiprocessor by using a modified pagefault trap to do paging over the network, e.g. IVY [23]. The second technique focuses on shared variables rather than pages. The programmer is required to annotate the shared variables with their anticipated access patterns. These annotations are then used by the DSM runtime system for selecting the most suitable coherence protocol. This technique is used in Munin [5]. The third technique is object-oriented and requires a high-level programming model, e.g. Linda [21], Orca [22].

Relatively few approaches have dealt with the issue of heterogeneity. Mermaid [8] is an implementation of heterogeneous DSM as an extension to the IVY system. According to Mermaid's creators, heterogeneous DSM is feasible and presents comparable performance to homogeneous DSM. However, many issues need to be addressed due to heterogeneity.

Providing yet another DSM system is not in the scope of our research. Our main interest is in developing a DSM system that can provide cluster-oriented functionality and cost data on typical DSM operations. The preferred way to follow here is to adopt an existing system and subsequently modify and augment it to serve the intended purposes. Several decisions need to be made, but the main direction is to adopt the simplest approach possible, decide the level of implementation, and add the minimum functionality necessary. In this sense, the IVY and Mermaid systems seem to be attractive. The issue of fully supporting heterogeneity remains open, because forming clusters with machines of heterogeneous architectures or sharing memory among clusters containing homogeneous hardware but of different architecture from cluster to cluster are options of high complexity and questionable performance. The Munin approach is also attractive because of the focus on the different access patterns of shared variables. Because shared variable access patterns constitute an important parameter of the cost model, it would be interesting to see the impact the utilization of such a DSM approach has on performance. However, the use of a Munin-like system requires additional programmer effort, the annotations of shared variables, which is not particularly desirable here if it means many modifications to already existing programs.

Several other works on DSM need to be mentioned here, because they contain useful elements for our work. Iftode et al. [6] study the sharing patterns of applications running on DSM systems and identify several factors affecting performance. Lu et al. [7] add limited compiler support and modifications to TreadMarks [24] to eliminate unnecessary computation and communication during the execution of irregular applications. Yoon and Malek [9] propose the creation of a single address space per running program instead of a global address space shared by all processing nodes. This is similar to the definition of logical clusters with the difference that not only one program can run on a logical cluster, no matter what effect this fact has on performance. Kim and Vaidya [10] propose an adaptive DSM system where statistics about memory accesses, collected over a sampling period, are used for determining the protocol that has the minimum cost for each memory page. Erlichson et al. [11] present a kernel-level implementation of DSM on an 8-node, 8-processors/node cluster and study the cost of DSM primitives and the effects of clustering on performance. Finally, Quarks [12] is a relatively new simple DSM approach. Quarks is a descendant of Munin and is aimed at reducing the communication overhead. Its basic abstraction is that of shared regions, which are page-aligned byte ranges of variable length. For this DSM system there exists a freely available Linux port.

6.3 Other Relevant Works

In terms of functionality, the most closely related work to our approach is the Scalable Concurrent Programming Library (SCPlib) [4] of the Scalable Concurrent Programming Laboratory at Syracuse University. SCPlib is the descendant of the concurrent graph library [25] and is aimed at supporting irregular applications on parallel hardware. The

library provides heterogeneous communication and file I/O, load balancing, and dynamic task granularity control. The user needs to supply a set of special support routines for the library to successfully perform migration of tasks and granularity adjustments. The load balancing mechanism is based on the concept of heat diffusion [3]. Communication cost is taken into account when deciding task movements.

Our work is targeted at a more general and complex environment than SCPlib. The goal is to provide functionality for a hierarchy of clusters and help applications and middleware make decisions by estimating as accurately as possible, or to the desired degree of accuracy, the cost of certain operating system services especially when dynamically sharing resources. In view of this, the proposed research can address all issues covered by SCPlib. Although load balancing is not in the scope of our research, the use of the cost model can greatly facilitate a sophisticated load balancing mechanism by providing the cost of various task placement options when deciding how to redistribute the load and not only the cost of communications. Also, SCPlib is based on message-passing whereas we support DSM and mixed mode computations.

Another research work, which has common ground with the proposed research, is that by Kravets et al. [15]. This work proposes a cooperative solution to the dynamic management of communication resources. In this solution, application requirements, expressed in the form of "payoff" functions, and network resource availability, expressed in the form of service availability curves, are taken into account by a configurable communication layer. The goal is to better exploit communication resources by allowing applications and networks to adapt to each other. Our approach has as ultimate goal to allow applications to self-adapt to the changing availability of all system resources and doesn't focus specifically on the communication layer.

7 Summary

Modern clusters connected by high-speed networks are capable of outperforming supercomputers, but the performance delivered to scientific applications is only a fraction of this maximum. Identifying software overhead as a key reason for this discrepancy, we have described our approach to solving this problem. We focus on an hardware environment with hierarchically organized clusters of computing nodes. In this environment we form logical groups of nodes by using software DSM and multithreading, and augment the capabilities of the operating system with dynamic resource sharing primitives and a decision-making framework based on a parametrical cost/benefit model. The model works for a variety of situations with data availability ranging from minimal to high. Our goal is to predict the performance ramifications of policy choices and thus to allow applications and middleware to adapt to their computing environment and achieve better overall performance.

References

[1] G. Lindahl, S. Chapin, N. Beekwilder, A. Grimshaw. Experiences with Legion on the Centurion Cluster. Technical Report CS-98-27, Department of CS, University of Virginia, 1998

[2] Centurion: The Legion project testbed. http://legion.Virginia.EDU/centurion

[3] J. Watts, M. Rieffel, S. Taylor. Dynamic Management of Heterogeneous Resources. In *High Performance Computing: Grand Challenges in Computer Simulation*, pp.151-6, April 1998.

[4] SCP Laboratory. Scalable Concurrent Programming Library. http://www.scp.syr.edu/html/scp_library.html

[5] J. Bennett, J. Carter, W. Zwaenepoel. Munin: Distributed Shared Memory based on type-specific memory coherence. In *Proceedings of the Second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pp.168-176, March 1990.

[6] L. Iftode, J. PalSingh, K. Li. Understanding application performance on shared virtual memory systems. In *Proceedings of the 23rd annual international symposium on Computer architecture*, pp.122-133, 1996.

[7] H. Lu, A. Cox, S. Dwarkadas, R. Rajamony, W. Zwaenepoel. Compiler and software distributed shared memory support for irregular applications. In *Proceedings of the Sixth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pp.48-56, 1997.

[8] S. Zhou, M. Stumm, K. Li, D. Wortman. Heterogeneous Distributed Shared Memory. In *IEEE Transactions on Parallel and Distributed Systems*, Vol.3, No.5, pp.540-554, September 1992.

[9] M. Yoon, M. Malek. Configurable Shared Virtual Memory for Parallel Computing. Technical Report CS-TR-94-21, University of Texas, Austin, July 1994.

[10] J. Kim, N. Vaidya. A cost-comparison approach for adaptive distributed shared memory. In *Proceedings of the 1996 international conference on Supercomputing*, page 44, 1996.

[11] A. Erlichson, N. Nuckolls, G. Chesson, J. Hennessy. SoftFLASH: analyzing the performance of clustered

distributed virtual shared memory. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pp.210-220, 1996.

[12] M. Swanson, L. Stoller, J. Carter. Making Distributed Shared Memory Simple, Yet Efficient. To appear in the *Proceedings of the Third International Workshop on High-Level Parallel Programming Models and Supportive Environments*, March 1998.

[13] R. Unrau, M. Stumm, O. Krieger. Hierarchical Clustering: A Structure for Scalable Multiprocessor Operating System Design. Technical Report CSRI-268, Computer Systems Research Institute,. University of Toronto, March 1992.

[14] R. Unrau, O. Krieger, B. Gamsa, M. Stumm. Hierarchical Clustering: A Structure for Scalable Multiprocessor Operating System Design. *Journal of Supercomputing*, 1995.

[15] R. Kravets, K. Calvert, K. Schwan. Payoff-Based Communication Adaptation based on Network Service Availability. In *Proceedings of IEEE Multimedia Systems '98*, 1998.

[16] D. Culler, R. Karp, D. Patterson, A. Sahay, E. Santos, K. Schauser, R. Subramonian, T. von Eicken. LogP: a Practical Model of Parallel Computation. In *Communications of the ACM*, Vol.30, No.11, pp.79-85, November 1996.

[17] OpenMP
OpenMP C and C++ Application Program Interface, Version 1.0, October 1998.
http://www.openmp.org

[18] The Berkeley NOW project.
http://now.cs.berkeley.edu

[19] GLUnix: A Global Layer Unix for NOW.
http://now.cs.berkeley.edu/Glunix/glunix.html

[20] A. Tanenbaum. *Distributed Operating Systems.* Prentice-Hall, Inc., 1995.

[21] D. Gelernter. Generative Communication in Linda. In *ACM Transactions on Programming Languages and Systems*, vol.7,pp.80-112, January 1985.

[22] H. Bal, M. Kaashoek, A. Tanenbaum. Orca: A Language for Parallel Programming of Distributed Systems. In *IEEE Transactions on Software Engineering*, vol.18, pp.190-205, March 1992

[23] K. Li. IVY: A Shared Virtual Memory System for Parallel Computing. In *Proceedings of the International Conference on Parallel Processing*, pp.94-101, August 1988.

[24] C. Amza, A. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, W. Zwanenepoel. TreadMarks: Shared Memory Computing on Networks of Workstations. In *IEEE Computer*, Vol.29, No.2, pp.18-28, February 1996.

[25] S. Taylor, J. Watts, M. Rieffel, M. Palmer. The Concurrent Graph: Basic Technology for Irregular Problems. In *IEEE Parallel and Distributed Technology*, 4(2):15-25, 1996.

[26] R. Wolski. Dynamically Forecasting Network Performance to Support Dynamic Scheduling Using the Network Weather Service. In *Proceedings of the Sixth International Symposium on High-Performance Distributed Computing (HPDC-6)*, August 1997.

Steve Chapin is an Associate Professor of Electrical Engineering and Computer Science at Syracuse University. Prior to joining Syracuse, he served on the faculties of the University of Virginia and Kent State University. He received his Ph.D. in Computer Science from Purdue University in 1993.

Deepak Saxena and **Nehal Mehta** are M.S. students in the department of Electrical Engineering and Computer Science at Syracuse University.

Dimitrios Katramatos is a Ph.D. candidate at the University of Virginia. He received his M.S. in Computer Science from Kent State University.

SESSION 5-A
DESIGN TOOLS

Chair: S. Singh, *Oregon State University, USA*

The HARNESS PVM-Proxy: gluing PVM applications to distributed object environments and applications

Mauro Migliardi and Vaidy Sunderam
Emory University, Dept. of Math & Computer Science
1784 N. Decatur Rd. #100
Atlanta, GA, 30322, USA
{om, vss}@mathcs.emory.edu

Abstract

Metacomputing frameworks have received renewed attention of late, fueled both by advances in hardware and networking, and by novel concepts such as computational grids. HARNESS is an experimental metacomputing system based upon the principle of dynamic reconfigurability not only in terms of the computers and networks that comprise the virtual machine, but also in the capabilities of the VM itself. These characteristics may be modified under user control via a "plug-in" mechanism that is the central feature of the system. The system's capabilities have been used to develop a PVM compatibility suite, i.e. a set of plug-ins that allow users to run PVM applications of top of HARNESS. In this paper we describe the PVM-Proxy plug-in: a plug-in capable of gluing PVM applications to distributed object environments.

1 Introduction

HARNESS [1] is a metacomputing framework that is based upon several experimental concepts, including dynamic reconfigurability and fluid, extensible, virtual machines. The underlying motivation behind HARNESS is to develop a metacomputing platform for the next generation, incorporating the inherent capability to integrate new technologies as they evolve. The first motivation is an outcome of the perceived need in metacomputing systems to provide more functionality, flexibility, and performance, while the second is based upon a desire to allow the framework to respond rapidly to advances in hardware, networks, system software, and applications. Both motivations are, in some part, derived from our experiences with the PVM [2] system, whose monolithic design implies that substantial re-engineering is required to extend its capabilities or to adapt it to new network or machine architectures.

HARNESS attempts to overcome the limited flexibility of traditional software systems by defining a simple but powerful architectural model based on the concept of a software backplane. The HARNESS model is one that consists primarily of a kernel that is configured, according to user or application requirements, by attaching "plug-in" modules that provide various services. Some plug-ins are provided as part of the HARNESS system, while others might be developed by individual users for special situations, while yet other plug-ins might be obtained from third-party repositories. By configuring a HARNESS virtual machine using a suite of plug-ins appropriate to the particular hardware platform being used, the application being executed, and resource and time constraints, users are able to obtain functionality and performance that is well suited to their specific circumstances. Furthermore, since the HARNESS architecture is modular, plug-ins may be developed incrementally for emerging technologies such as faster networks or switches, new data compression algorithms or visualization methods, or resource allocation schemes – and these may be incorporated into the HARNESS system without requiring a major re-engineering effort.

HARNESS' reconfiguration capabilities allowed us to design and implement a PVM compatibility suite, i.e. a set of plug-ins that emulate the services provided by PVM demons and allows users to run unchanged PVM applications on top of HARNESS. The native distributed object programming model of HARNESS on which the compatibility suite is based has been leveraged to introduce a high level of modularity in the design of the compatibility suite. This modularity allows introducing new technologies and new services into PVM without requiring a complete redesign of the demon itself.

However, the compatibility suite only allows execution of traditional message passing PVM applications that cannot directly take advantage of the underlying distributed object infrastructure, while our perceived goal was to provide a seamless connection for PVM users to distributed objects technology.

0-7695-0556-2/00 $10.00 © 2000 IEEE

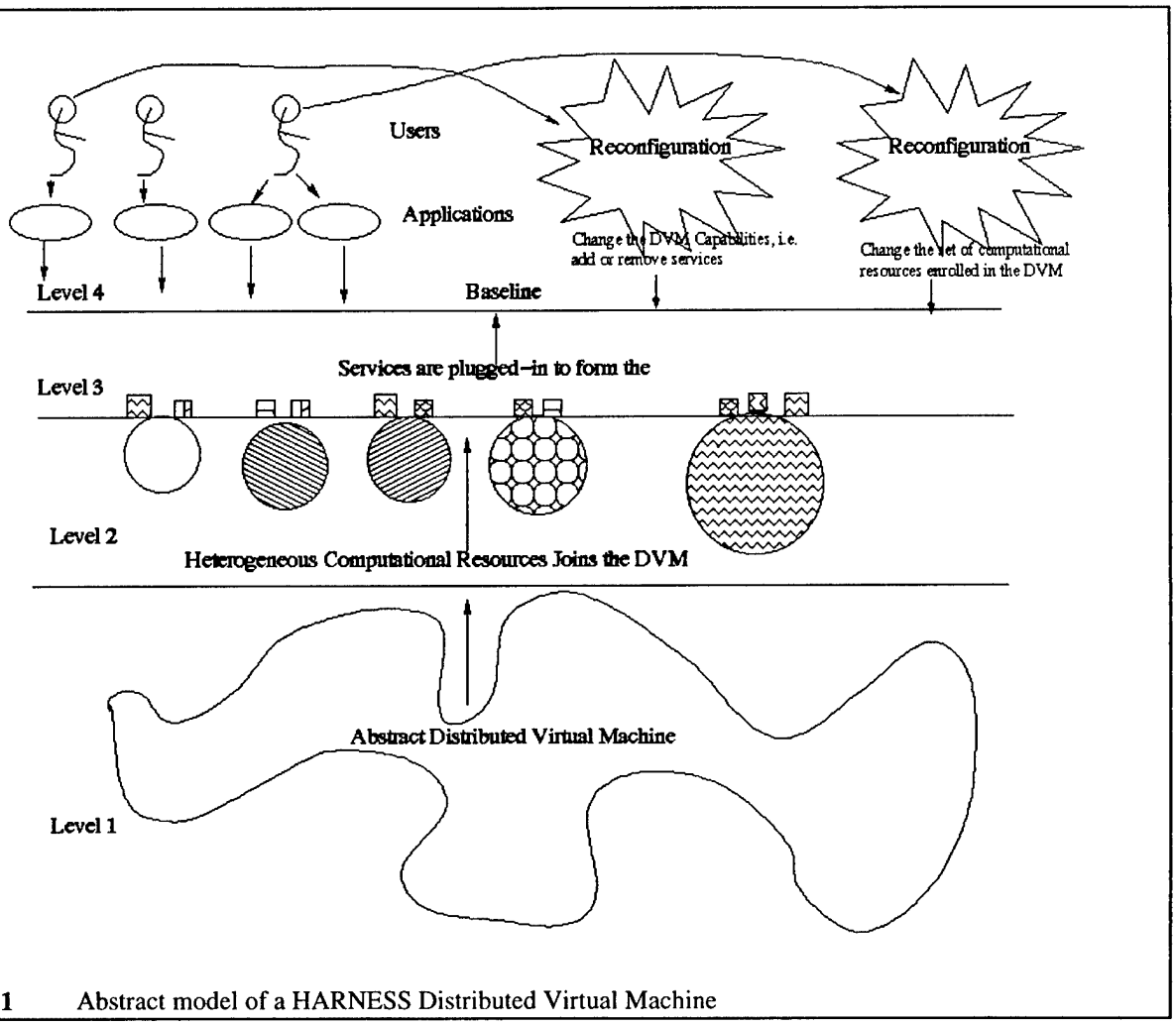


Figure 1 Abstract model of a HARNESS Distributed Virtual Machine

In this paper we describe the PVM-Proxy plug-in, a HARNESS plug-in that is able to bridge the gap between traditional PVM applications and HARNESS native distributed object infrastructure. This plug-in appears as a standard PVM task to any running PVM application, but is able to interface directly to distributed objects applications translating messages coming from the PVM side into RMI or CORBA procedure calls and procedure calls coming from the distributed object applications back into PVM messages. This plug-in can be used to make complete distributed object applications such as our reusable simulation framework [3] appear as PVM tasks to PVM applications, actually allowing traditional PVM applications to take complete advantage of the capabilities of the HARNESS system.

The full paper will be structured as follows: in section 2 we will give a detailed description of the HARNESS metacomputing framework and of its main architectural features; in section 3 we will describe the HARNESS PVM compatibility suite; in section 4 we will describe the architectural and implementation details of the HARNESS PVM-Proxy plug-in; in section 5 we will show how our system can be used to glue together heterogeneous applications using a distributed MPEG coder as an example application; finally, in section 6 we will provide some concluding remarks.

2 HARNESS System Architecture

The fundamental abstraction in the HARNESS metacomputing framework is the **Distributed Virtual Machine** (DVM) (see figure 1, level 1). Any DVM is associated with a symbolic name that is unique in the HARNESS name space, but has no physical entities connected to it. **Heterogeneous Computational Resources** may enroll into a DVM (see figure 1, level 2)

at any time, however at this level the DVM is not ready yet to accept requests from users. To get ready to interact with users and applications the heterogeneous computational resources enrolled in a DVM need to load **plug-ins** (see figure 1, level 3). A plug-in is a software component implementing a specific **service**. By loading plug-ins a DVM can build a consistent **service baseline** (see figure 1, level 4). Users may **reconfigure** the DVM at any time (see figure 1, level 4) both in terms of computational resources enrolled by having them **join** or **leave** the DVM and in terms of services available by **loading** and **unloading** plug-ins.

The main goal of the HARNESS metacomputing framework is to achieve the capability to enroll heterogeneous computational resources into a DVM and make them capable of delivering a consistent service baseline to users. This goal require the programs building up the framework to be as portable as possible over an as large as possible selection of systems. The availability of services to heterogeneous computational resources derives from two different properties of the framework: the portability of plug-ins and the presence of multiple searchable plug-in repositories. HARNESS implements these properties mainly leveraging two different features of Java technology. These features are the capability to layer a homogeneous architecture such as the Java Virtual Machine (JVM) [4] over a large set of heterogeneous computational resources, and the capability to customize the mechanism adopted to load and link new objects and libraries. However, the adoption of the Java language as the development platform for the HARNESS metacomputing framework has given us several other advantages:

- it allowed us to develop the framework as a collection of cooperating objects with consistent boundaries (Java Classes) and to guarantee to users an OO development environment;
- it allowed us to define a clear and consistent boundary for plug-ins, in fact each plug-in is required to appear to the system as a Java class;
- it allowed us to implement all the entities in the framework adopting a robust multithreaded architecture;
- it allows users to develop additional services both in a passive, library-like flavor and in an active thread-enabled flavor;
- it provided us an Object Oriented mechanism to require services from remote computational resources (Java Remote Method Invocation [5]);
- it provided us a generic methodology to transfer data over the network in a consistent format (Java Object Serialization [6]);
- it allowed us to provide to users the definition of interfaces to be implemented by plug-ins implementing the basic services;
- it allowed us to tune the trade-off between portability and efficiency for the different components of the framework.

This last capability is extremely important, in fact, although portability at large is needed in all the components of the framework, it is possible to distinguish three different categories among the components that requires different level of portability. The first category is represented by the components implementing the capability to manage the DVM status and load and unload services. We call these components **kernel level services**. These services require the highest achievable degree of portability, as a matter of fact they are necessary to enroll a computational resource into a DVM. The second category is represented by very commonly used services (e.g. a general, network independent, message passing service or a generic event notification mechanism). We call these services **basic services**. Basic services should be generally available, but it is conceivable for some computational resources based on specialized architecture to lack them. The last category is represented by highly architecture specific services. These services include all those services that are inherently dependent on the specific characteristics of a computational resource (e.g. a low-level image processing service exploiting a SIMD co-processor, a message passing service exploiting a specific network interface or any service that need architecture dependent optimization). We call these services **specialized services**. For this last category portability is a goal to strive for, but it is acceptable that they will be available only on small subsets of the available computational resources. These different degrees of required portability and efficiency over heterogeneous computational resources can optimally leverage the capability to link together Java byte code and system dependent native code enabled by the Java Native Interface (JNI) [7]. The JNI allows to develop the parts of the framework that are most critical to efficient application execution in ANSI C language and to introduce into them the desired level of architecture dependent optimization at the cost of increased development effort [8][9].

The use of native code requires a different implementation of a service for each type of heterogeneous computational resource that need to deliver that service. This fact implies a development effort multiplied for each plug-in including native code. However, if a version of the plug-in for a specific architecture is available, the HARNESS metacomputing framework is able to fetch and load it in a user transparent fashion, thus users are screened from the necessity to control the set of architectures their application is currently running on. To achieve this result HARNESS leverages the capability of the JVM to let users redefine the mechanism used to retrieve and load both Java classes bytecode and native shared libraries. In fact, each DVM in

the framework is able to search a set of plug-ins repositories for the desired library. This set of repositories is dynamically reconfigurable at run-time, users can add new repositories at any time.

The kernel level services of a Harness DVM are delivered by a distributed system composed of two categories of entities:

- a DVM status server, unique for each DVM;
- a set of Harness kernels, one and only one running on each computational resource currently enrolled or willing to be enrolled into a DVM.

To achieve the highest possible degree of portability for the kernel level services both the kernel and the DVM status server are implemented as pure Java programs. We have used the multithreading capability of the Java Virtual Machine to exploit the intrinsic parallelism of the different tasks the two entities have to perform, and we have built the framework as a set of Java packages.

Control messages and DVM status changes not related to the discovery-and-join protocol or the recover-from-failure protocol, are exchanged through a star shaped set of reliable unicast channels whose center is the DVM status server. These connections are implemented through the communication commodities delivered by the java.net package. It is important to notice that neither the star topology, nor the use of the java.net package are constraints imposed to all the communication services in the framework. On the contrary, user level communication services may adopt the connection topology that best suit their needs and are not required to use the java.net package to implement these commodities. For this reason, neither the star topology interconnecting the kernels and the DVM Server, nor the fact that the java.net package is used represent a major bottleneck in the Harness metacomputing framework. The kernels and the DVM server interacts to guarantee a consistent evolution of the status of the DVM both in front of users requesting new services to be added and in front of computational resources or network failures. This consistency is enforced by means of a set of protocols executed during the different phases of the DVM life.

A DVM may be started in three different ways:

- starting a DVM server;
- starting a kernel;
- starting an application.

In the first case, a user invokes the execution of the main method of the Java H_Server class from the edu.emory.mathcs.harness package providing as a parameter the name of the DVM this server is starting. The DVM server reads the configuration file harness.defaults to see if the user wants to use server based implementation or a multicast implementation of the HARNESS name space. In the former case the server gets from the same configuration file the port and address of the HARNESS name server the user wants to adopt and connects to it to register its presence. In the latter case it executes a hashing function to map the DVM name into a multicast IP address and port. Then it starts to multicast on the channel I'm alive packets and to listen for incoming packets.

In name-server mode the DVM server can get two types of packets:

- probe requests from the name-server;
- join requests from kernels.

Probe request are sent by the naming service every time it is requested to provide information about a DVM server. Before sending it's current data the name server validates them with a probe message. If the server receives a join packet then it generates a TCP connection to the sender kernel and it starts the Join protocol.

In multicast mode the DVM server can get three types of packets:

- I'm alive packets from a DVM server;
- join packets from kernels;
- query packets from applications.

The server checks the source address of any I'm alive packet it receives. If the packet comes from another server the server multicasts a train of I'm alive packets to notify its presence to the other server and then it exits. This will enforce the kernels running on computational resources enrolled in the DVM to start the server regeneration protocol and to regenerate a new, single server. This mechanism prevents the existence of multiple DVM servers with partial or outdated information and guarantees that a single DVM server is active in a DVM.

If the server receives a join packet then it generates a TCP connection to the sender kernel and it starts the Join protocol.

If the server receives a query packet then it checks if a kernel exists on the computational resource from which the application is querying. If a kernel is already active, then the server provides to the querying application the port number on which the kernel accepts connections from applications, otherwise it provides a null reply.

The second way to start a Harness DVM is to invoke the main method of the Main class in the edu.emory.mathcs.harness package providing as a startup parameter the name of the DVM the kernel wants to enroll into. The kernel reads the configuration file harness.defaults to check if the user wants to use server based implementation or a multicast implementation of the HARNESS name space. In the former case the kernel gets from the same configuration file the port and address of the HARNESS name server the user wants to adopt, it connects to it and asks if there is a DVM server for the DVM it wants to join. If there is one it sends a join request to it, if there is none it starts the DVM regeneration protocol.

In the latter case, the kernel executes the hash function to map the DVM name into an IP multicast address and port and sends send a Join packet on that channel. The

kernel performs three tries before giving up. After three tries have timed out without a DVM server activating a TCP connections the kernel assume no DVM server exists and spawns a new JVM to start a new DVM server. Then he starts again sending the Join packet.

The third way to start a Harness DVM is to instantiate the class H_core or H_RMIcore from the package edu.emory.mathcs.harness in an application providing the DVM name as a parameter. The H_RMIcore class constructor hashes the DVM name to obtain a port for the HARNESS RMI registry. The HARNESS RMI registry provides to the H_RMIcore class an RMI reference for the local kernel. If it cannot connect to that port, the H_RMIcore class assumes that no kernel for the given DVM is active on the local host and starts a new one.

The H_core class constructor executes the hashing function and drops a query packet on the multicast channel. If no answer comes back or if the answer says that no kernel is active on the computational resource the constructor spawns a new JVM starting a kernel and sets a flag to avoid starting a new one even in the case of another failed set of tries. The possibility of two or more applications racing to spawn to or more kernels on the same computational resource is prevented by the Join protocol.

The DVM server initiates the join protocol each time it receives a multicast join packet. The Join packet contains the IP address and a port number onto which the willing-to-join kernel is accepting a TCP connection. The first step of the join protocol is the instantiation of a TCP connection between the DVM server and the Joining kernel. Then the DVM server waits for the kernel to provide its baseline. At this point the server performs two checks: the baseline check and the uniqueness check. The baseline check consists of checking the compatibility of the kernel with the current implementation of the DVM server. The uniqueness check consists of checking that no other kernel has already joined from the same computational resource. In case of failure of one of these two checks an error message is sent back, the protocol terminates with a failure and the connection is closed. If the kernel passes both controls then the DVM servers checks if the kernel is Joining back after a failure (computational resource or network crash) or if the computational resource has never been enrolled in the DVM before. If the computational resource is coming back from a crash the DVM server sends to the kernel a crash token message and a copy of its pre-crash status, otherwise it sends a new token message. The following step is to get from the kernel its current status and to send back to it the current status of the DVM.

At this point the Join protocol is successfully completed, the DVM server generates a Join event that is distributed as described in next section while the kernel is now enrolled in the DVM.

The leave protocol is much simpler that the Join protocol. The leave protocol is always started by a kernel. A TCP connection between the kernel is guaranteed to be active, as a matter of fact it is not possible to start the Leave protocol before a successful completion of the Join protocol. The kernel sends an explicit Leave message to the DVM server and then closes the TCP connection. The DVM server generates a Leave event that is distributed to all the remaining kernels.

The status of the DVM consists of the set of computational resources currently enrolled in the DVM, the set of services available on each enrolled computational resource as well as the DVM's baseline. We call baseline of a DVM the minimum set of services a computational resource must be able to deliver in order to join the DVM. The dynamic nature of the framework make this state an evolving entity, thus the framework keeps it up to date and available for queries from any application or service in the DVM. It is important to notice that information about the applications currently using services or internal status of an application is not part of the DVM status and loosing track of it does not in any way compromise the existence of the DVM in itself. Any form of application tracking and check-pointing, while highly desirable for many applications, is a service in itself and the framework does not need to incorporate it in its status.

The Harness metacomputing framework guarantees that all the events that changes the status of the DVM are received by all the kernels enrolled in the DVM in the same order. In the current implementation the Total Order (TO) protocol is implemented adopting the DVM server as a central ordering entity and exploiting the stream nature of TCP connections to avoid subsequent losses of order. Although very simple, a centralized implementation of the TO protocol has in general two negative features:

- the central entity is a single point of failure;
- the central entity is a bottleneck.

However, these two problems do not represent a major flaw in the design and efficiency of our framework. In fact, the single point of failure is limited to the incapability of the framework to retrieve after a DVM server crash the status of a previously crashed kernel and the central bottleneck does not influences application level communication services. The status of a DVM as it is defined in the Harness metacomputing framework consists of the sum of the stati of each enrolled kernel. Each event that changes the status of the DVM changes the status of a kernel in a way that is recorded by the kernel itself with the only exception being the case of a kernel crash. Thus it is not possible for an event, except for kernel crash events, to get lost in a DVM server crash. On the contrary, in the case of a DVM server crash it is possible to reconstruct completely the current status of the DVM simply obtaining from every surviving kernel a copy of its current status.

```
User Setup
public int login(java.lang.String, java.lang.String)
public int logout()
public boolean setCResourceMapping( H_pname)
public boolean setServiceMapping( H_pname)

DVM Manipulation
public java.lang.String getName()
public H_StringKeyedTable getArchs()
public H_crname getAHosts()[]
public java.util.Enumeration getHosts()
public H_RetVal grabHost( H_crname[], H_QoS)
public H_RetVal deleteHost( H_crname[], H_QoS);
public void kill()

Plug-ins Manipulation
public H_RetVal getInterfaceDescriptor(H_phandle)
public H_RetVal load( H_pname, H_crname[], H_QoS)
public H_RetVal unload( H_phandle[], H_QoS)

Information Gathering
public H_Info getInfo()
public java.util.LinkedList getAll( H_pname)
public java.util.LinkedList getAll( H_pname, H_crname)
public java.util.LinkedList getAll(java.lang.Class)
public java.util.LinkedList getAll(java.lang.Class, H_crname)
public H_phandle getAny( H_pname)
public H_phandle getAny( H_pname, H_crname)
public H_phandle getAny(java.lang.Class)
public H_phandle getAny(java.lang.Class, H_crname)
public java.util.LinkedList getPlugins(H_crname)
```

Figure 2 Functional interface provided by the Java class H_RMIcore.

It is important to notice that the fact that this reconstruction process is not able to keep track of crashed kernels does not mean that applications relying on services delivered by the crashed kernels will have as their only choice to stop and fail. Reliable distributed check-pointing of application's status and restart of failing services are services themselves, thus their behavior in the event of kernel crashes is not constrained by the DVM status and the reconstruction of the DVM status is not concerned with them.

To evaluate the bottleneck represented by the star topology, it is important to notice that it involves only events requiring DVM status changes, as a matter of fact any traffic generated by user application exchanging data is not required to flow through the DVM server. The only events that the DVM status server needs to process are:

- a kernel joining the DVM;
- a kernel leaving the DVM;
- a kernel crash;
- the addition of a service to the DVM.

Thus the DVM server represents only a marginal bottleneck in the Harness metacomputing framework.

The current release of HARNESS provides 4 mechanisms for applications to interact with a HARNESS kernel:

- the H_RMIcore Java class that provides a set of fully object oriented methods and communicates with the kernel by means of RMI;
- the H_core Java class that provides the same functional interface as the H_RMIcore class on top of a string oriented protocol;
- a C library that exploits the JNI to invoke the methods of the above mentioned Java class;
- a language independent, string oriented protocol on top of a TCP reliable connection.

In figure 2 you can see the functional interface provided by the Java class H_RMIcore. The functions can be divided in four groups: user setup, DVM manipulation,

H_USER
username
password
H_ISROOTLIKE <present if this user is equivalent to root absent otherwise>
H_LOADABLECLASSES
<classname or packagename as in import statement in java files>
...
H_ENDLOADABLECLASSES
H_ACCESSIBLEPLUGINS
<classname or packagename as in import statement in java files>
...
H_ENDACCESSIBLEPLUGINS
H_REPOSITORIES
<repository URL>
...
H_ENDREPOSITORIES
H_ENDUSER

Figure 3 Syntax of the harness.policy file.

plug-ins manipulation and information gathering. A user need to log into the DVM to be allowed performing any other operation. The DVM stores the couple user name-password so that the same user will be recognized at log-in independently from the kernel he is loggin-in from. A user can set up a resource mapper service and a service mapper service. The resource mapper service performs a translation from a user-defined naming scheme into the HARNESS naming scheme for the names of the computational resources. The service mapper service performs a translation from a user-defined naming scheme into the HARNESS naming scheme for the names of the plug-ins. Thus it is possible to build user-defined naming schemes on top of the basic HARNESS naming scheme both for computational resources and for plug-ins. The only constraint is the need for a complete mapping from the user-defined scheme to the HARNESS scheme. As an example, we developed a simple resource mapper that is able to translate architecture-names into instances of that architecture available at Emory. Once logged-in a user can access all the other functionality provided to grab and remove hosts from the DVM, load and unload plug-ins and query the status of the DVM. The capability to request a service (e.g. deleting a host) does not imply that the system will fulfill the request, in fact every HARNESS kernel is configured at bootstrap with security options. These options define:

- which user is the root user for the local kernel;
- if root user access is required to force the kernel to leave a DVM;
- which plug-ins each user can load;
- which plug-ins each user can access;
- which repositories the kernel can retrieve plug-ins from.

It is important to notice that each kernel can set a different set of users who have root access to it. Thus, it is possible both to have a global system administrator for clusters owned by a single entity and to have a different administrator for each node of a DVM composed of personal workstations.

The sets of plug-ins loadable and accessible for each user are defined through a security configuration file, namely the harness.policy file. In figure 3 you can see the syntax of the harness.policy file, while in figure 4you can see an example instance of it. If a user is not explicitely cited in the policy file he is associated by default to user NOBODY. Thus it is possible to establish a minimum access level for anonymous users by means of the user ID NOBODY. A plug-in can be unloaded only by the user who loaded it or by the local root user. Thus it is not

```
H_USER
MAURO
Harness
H_ISROOTLIKE
H_LOADABLECLASSES
edu.emory.mathcs.harness.*
helloHarness.*
cgrowth4.*
H_ENDLOADABLECLASSES
H_ACCESSIBLEPLUGINS
edu.emory.mathcs.harness.*
helloHarness.*
cgrowth4.*
H_ENDACCESSIBLEPLUGINS
H_REPOSITORIES
http://www.mathcs.emory.edu/harness/REPOSITORY/
http://www.mathcs.emory.edu/~om/REPOSITORY/
H_ENDREPOSITORIES
H_ENDUSER

H_USER
NOBODY
NOBODY
H_LOADABLECLASSES
edu.emory.mathcs.harness.*
helloHarness.*
H_ENDLOADABLECLASSES
H_ACCESSIBLEPLUGINS
edu.emory.mathcs.harness.*
helloHarness.*
cgrowth4.*
H_ENDACCESSIBLEPLUGINS
H_REPOSITORIES
http://www.mathcs.emory.edu/harness/REPOSITORY/
H_ENDREPOSITORIES
H_ENDUSER
```

Figure 4 An example harness.policy file.

possible for a non-root user to remove a plug-in that is part of another user's application.

3 The Design of a PVM plug-in for a HARNESS DVM

The design of our PVM compatibility suite had three main objectives:

- requiring no changes into PVM applications to run in the new environment;
- minimizing the amount of changes to be inserted in the application side PVM library;
- achieving a modular design for the services provided by the PVM daemon.

We achieved these goals by designing a set of plug-ins able to understand the original PVM library to PVM

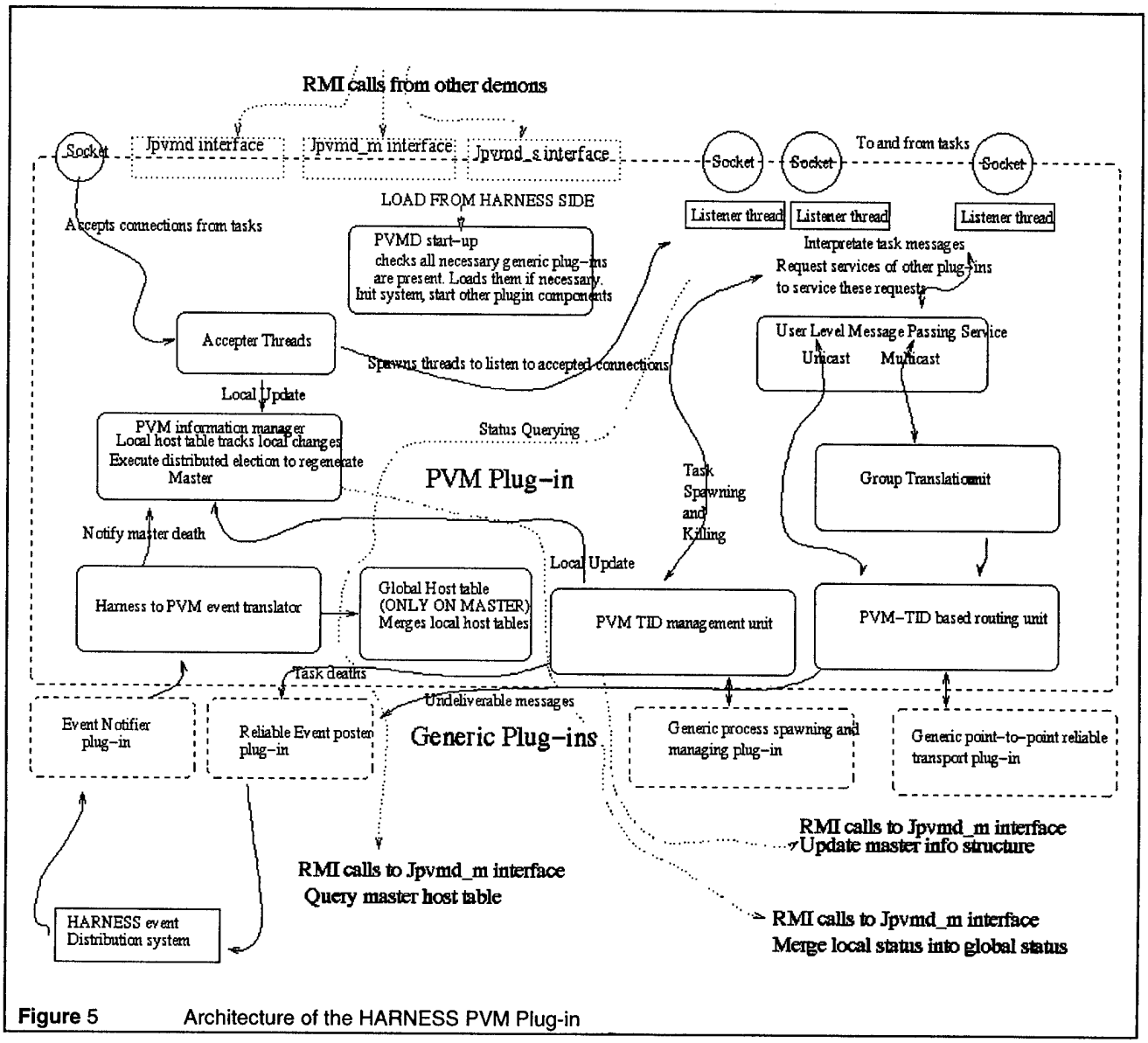


Figure 5 Architecture of the HARNESS PVM Plug-in

daemon protocol and to duplicate the services provided by the original PVM daemon. This approach provides complete compatibility with PVM legacy code, both in C and in FORTRAN, and requires only one change in the PVM library on the application side, namely the adoption of internet domain sockets for the communication channel between the library and the daemon.

Thus to run a legacy PVM application in the Harness PVM environment it is only necessary to link the original object code with the modified version of the library.

In our implementation, the services provided by the PVM daemons to applications are delivered by dedicated modules and general purpose Harness plug-ins such as a process-spawning plug-in and a message-passing plug-in (see figure 5). In figure 6 we show the actual sequence of the events in the Harness PVM startup, while figure 7 shows the chain of events serving an add_host request. At PVM startup a special PVM application (i.e. the HARNESS-PVM console) starts up the PVM demon by issuing to the Harness kernel the command to load the main PVMD plug-in. This plug-in takes care to request the Harness kernel to load the services that are required to provide full PVM compatibility. When a task requests an add-host operation the local PVMD plug-in translates it into a request for the remote Harness kernel to load the main PVMD plug-in which then takes care of requesting

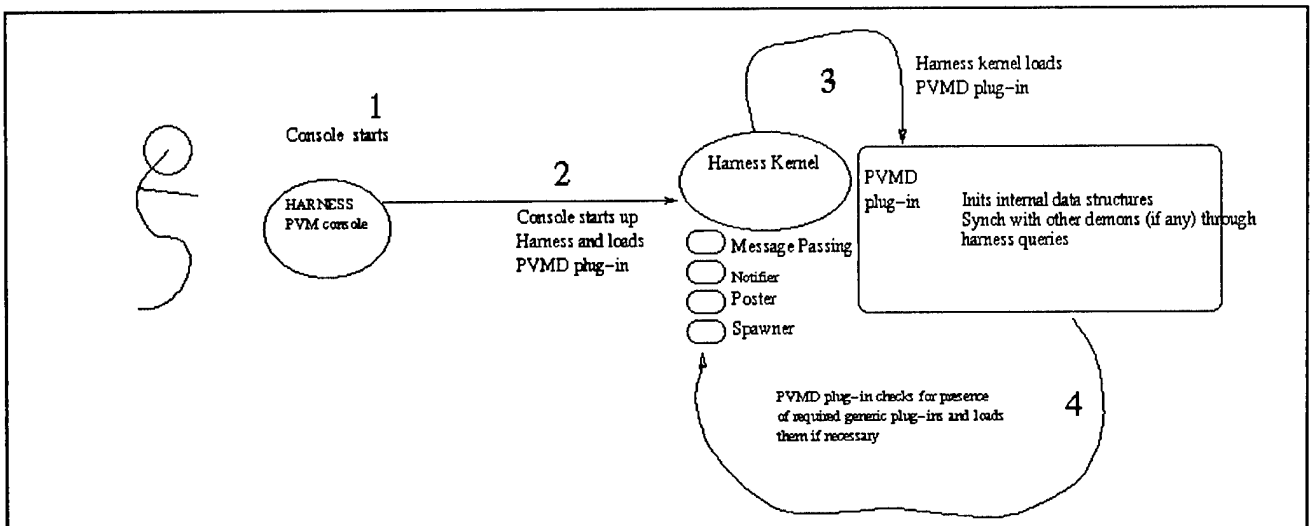


Figure 6 Sequence of events at Harness PVM startup

the loading of the other needed plug-ins.

The current version of the plug-in provides only the services required to emulate a completely functional subset of PVM daemon's capabilities. This subset includes:

- all the process control PVM commands;
- the pvm_parent, pvm_tidtohost and pvm_error information commands;
- all the message buffers commands;
- all the point-to-point sending commands;
- all the receive commands.

Multicast and group operations are currently not supported, however the development of these services as additional pluggable module is in progress.

Direct routing is supported as it completely bypasses the PVM demons and is completely implemented in the original PVM library.

The modularity of the design will easily let us substitute any plug-in with new versions in order to provide an enhanced version of the service. As an example, it will be easy to load a new version of the database plug-in to provide an extended system querying capability. Besides, the Harness capability to hot swap services allows run-time tuning of services to the set of hosts enrolled in the virtual machines, e.g. a specific version of message-passing plug-in can be loaded at run-time if a new communication fabric becomes available.

This design also has the following additional advantages. The first one derives from the fact that the message-passing service provided by the message-passing plug-in needs only to peek at the destination field of a message in order to route it and does not need to know anything about the actual content of the message. This is beneficial for two reasons:

- the marshalling and un-marshalling of the data types is performed inside the Harness PVM library in C or Fortran thus we don't incur in the typical marshalling inefficiency due to the strong typedness of Java;
- it is extremely easy to substitute the message passing plug-in with another plug-in optimized to a specific communication fabric (be it a proprietary local network or an unreliable Internet connection) because they only need to move arrays of bytes.

Another benefit of our design is the fact that PVM applications can rely on the Harness capability to soft-install applications to move executable and libraries to the hosts in the VM. Thus it is not necessary to install the application and PVM itself on all the hosts of the VM, the Harness loader will do it as long as they are available on any host in the Harness DVM or on any one of the enlisted repositories.

A third, very important benefit of our design is the removal of the single point of failure represented by the master PVM daemon. In fact, providing PVM compatibility on top of the Harness system by means of a set of cooperating plug-ins, allowed us to exploit the Harness event subscription/notification service to implement a distributed control algorithm in the information management plug-ins. This algorithm is capable of reconstructing a consistent, up-to-date version of the PVM status after the crash of any daemon.

The performance delivered by the HARNESS PVM plug-in is still not on par with the original PVM. In particular, due to the fact that the PVM plug-in shares the

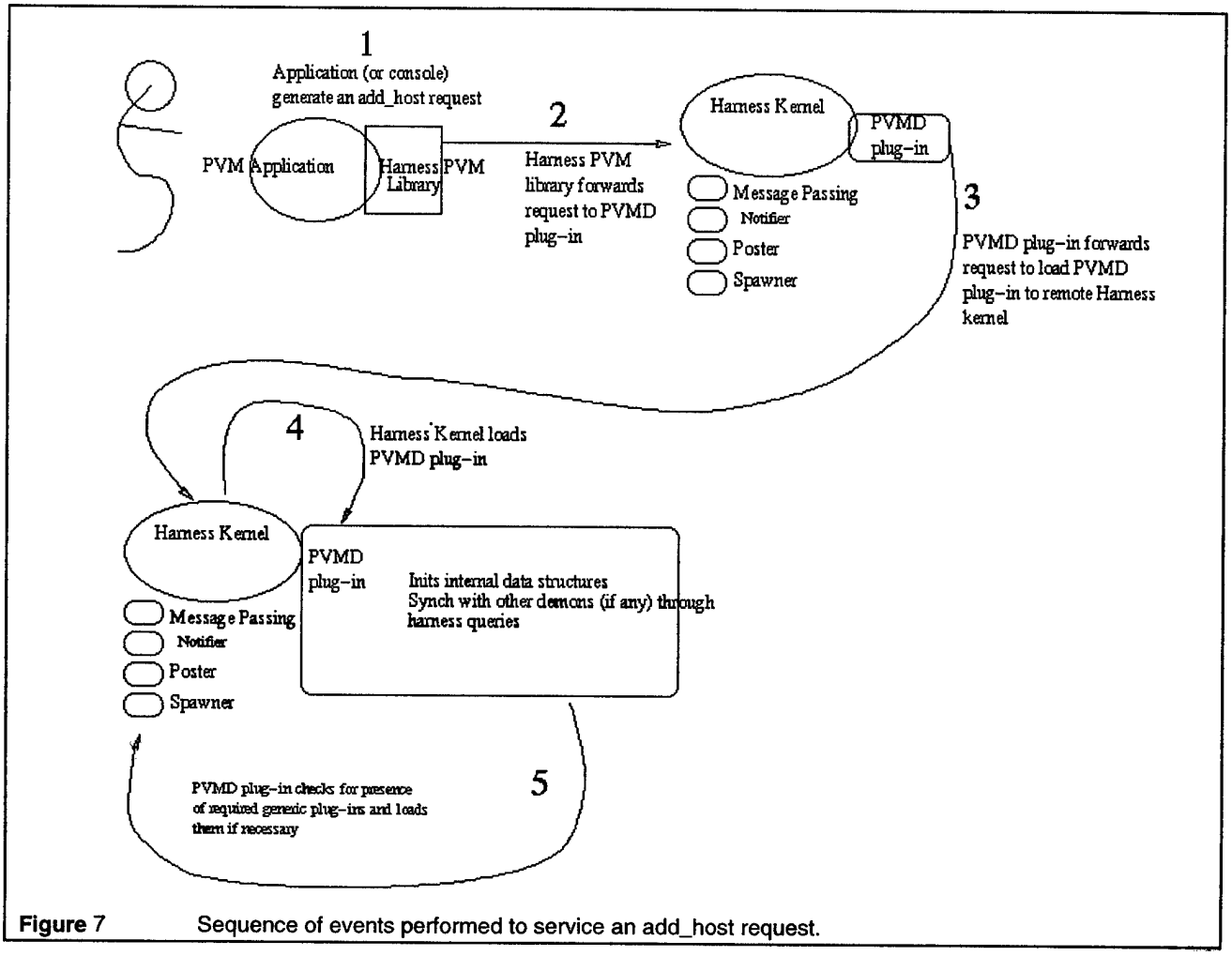


Figure 7 Sequence of events performed to service an add_host request.

JVM with any other plug-in on the same host, the performance degradation is extremely sensitive to the other activities currently on-going in the HARNESS DVM. To cope with this problem we are currently studying a mechanism to force HARNESS to dedicate a complete JVM to the components composing the PVM demon and applications. It is our opinion that such a mechanism will allow removing the dependency of the HARNESS PVM plug-in performance to external applications without compromising the modularity and expandability achieved so far.

4 The PVM-Proxy plug-in

The PVM-Proxy plug-in implement a generic translation of PVM user messages into function calls according to a user-defined protocol that can be plugged in the Proxy itself as a behavioral object. This arbitrary protocol takes care of interpreting the messages coming from the PVM side in order to generate the correct actions. The simplest possible protocol contains only data packet that need to be processed by the distributed object application connected to the PVM-Proxy. The restrictions that we need to impose onto a PVM task in order to be able to hot swap it are:

- the original task needs to be able to checkpoint itself;
- the original task must not be currently using the direct-routing capability of the PVM system.

The first restriction is a direct consequence of the fact that the original task will be substituted and there is no way to remove it. However, this restriction applies only if the PVM task is being swapped out during the execution. In the next section we will show how it is possible to remove it in the case in which a component of a PVM application is substituted at start-up time.

The second restriction, on the contrary, depends on our implementation of the PVM compatibility suite, in fact it derives from the fact that we left the original PVM library

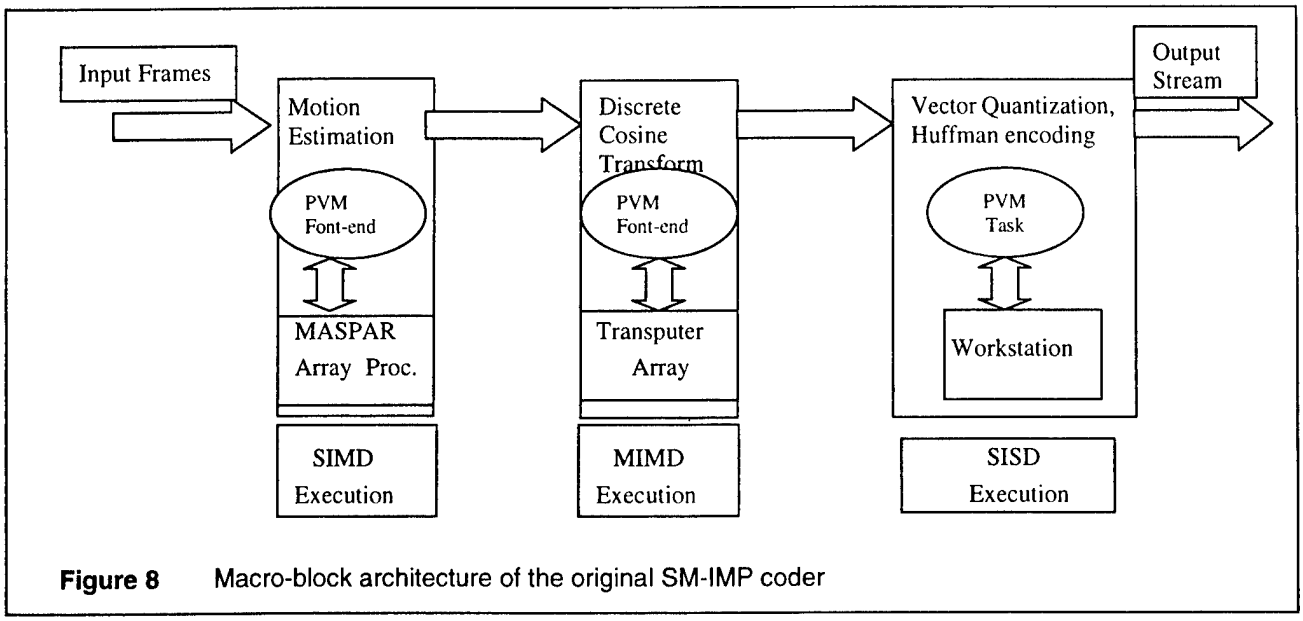


Figure 8 Macro-block architecture of the original SM-IMP coder

untouched and unaware of the changes that we introduced in the demons. However, we plan to remove this restriction in future versions of the PVM compatibility suite.

To execute the actual run-time substitution it is necessary to perform the following steps:

1. notify the PVM plug-in that all the traffic toward the given task has to be held;
2. checkpoint the PVM task;
3. load in HARNESS the PVM proxy plug-in;
4. kill the original PVM task;
5. give the saved status to the proxy plug-in;
6. tag the PVM task TID in the PVM plug-in with the proxy attribute and store the handle of the actual plug-in;
7. invalidate the possibly cached references to the swapped out task in all the components of the PVM plug-in;
8. remove the hold on the traffic toward the task.

The execution of the PVM application will continue undisturbed with the exception that every other PVM task will experience a temporary lag in the responsiveness of the swapped task while HARNESS performs the above described steps.

As soon as the proxy plug-in is in place it can start acting as a bridge between the legacy application and any HARNESS service such as the HARNESS reusable simulation. This capability allows extending legacy PVM simulation with distributed components technology and can be used to evolve a long-running application (e.g. a climate simulation) according to the results obtained.

The PVM proxy can be also used to substitute obsolete components of a distributed application at start-up time. This process requires the application to be capable of pausing between the set-up phase and the actual execution. However, it removes the requirement for the task to be substituted to be able to checkpoint its status, in fact, the actual substitution takes place before the tasks are initialized with the zero state. The actual sequence of steps for a start-up time task substitution is as follows:

1. notify the PVM plug-in that all the traffic toward the given task has to be held;
2. load in HARNESS the PVM proxy plug-in;
3. kill the original PVM task;
4. tag the PVM task TID in the PVM plug-in with the proxy attribute and store the handle of the actual plug-in;
5. remove the hold on the traffic toward the task.

The PVM-proxy plug-in automatically redirects all the messages sent to the original task to the proxied task. Thus the new implementation begins execution directly from the zero state and there is no need for the original task to be able to checkpoint itself.

5 An Example Use: Removing the Obsolete Components in a Distributed MPEG Coder

In this section we will describe how we have used the PVM-proxy plug-in to substitute the obsolete components of a legacy application, namely a distributed MPEG-1 coder [10] targeted at the heterogeneous parallel SM-IMP testbed architecture [11].

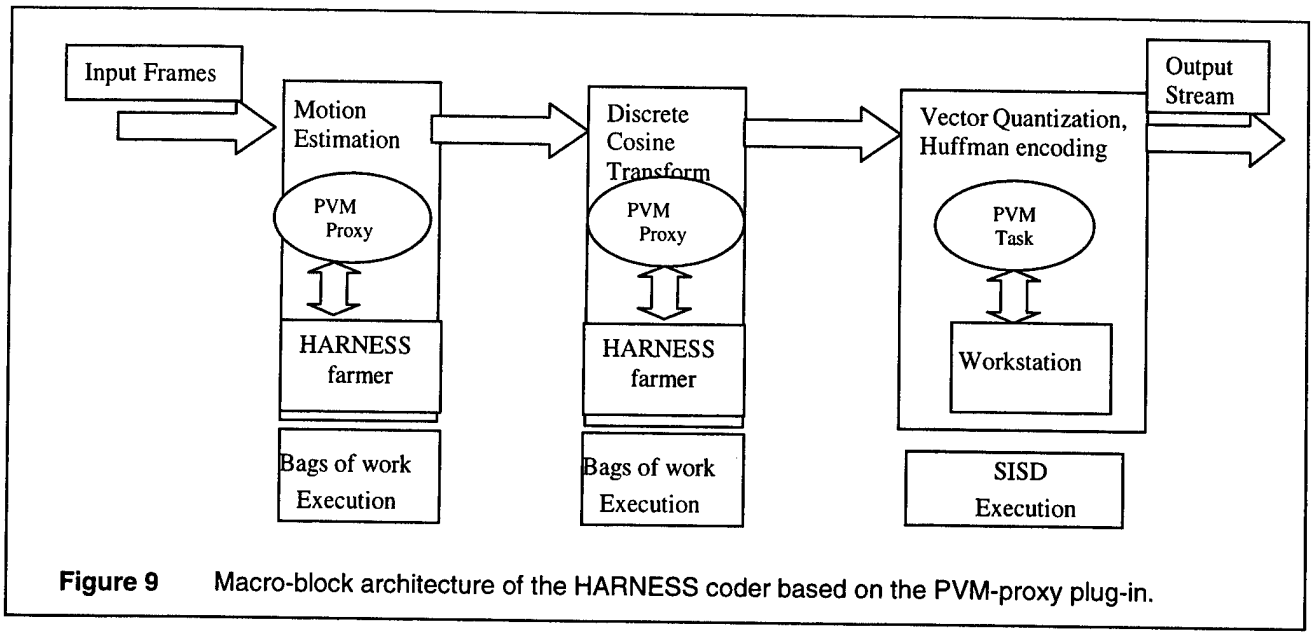


Figure 9 Macro-block architecture of the HARNESS coder based on the PVM-proxy plug-in.

MPEG-1 is an ISO standard for motion picture compression [12]. The algorithm defined in the standard requires the execution of several steps most of which are very computationally intensive. Thus it is very well suited to a distributed pipelined implementation. The SM-IMP project developed a prototype distributed MPEG coder for its heterogeneous SIMD-MIMD parallel architecture. This coder divided the MPEG algorithm in a sequence of pipelined steps. Each of these steps was performed by the component of the parallel architecture whose computational paradigm was best suited to the kind of parallelism of the computational step itself. The different activities were glued together using the PVM message passing service. The main parallelizable steps identified by the SM-IMP coder were:

- motion estimation;
- discrete cosine transform.

The former step was performed on a MASPAR MP1 SIMD array processor [13], while the latter step was performed by a MIMD transputer based multiprocessor. The remaining interconnecting steps were implemented as sequential PVM tasks.

To prove the feasibility of PVM tasks substitution by means of the PVM-proxy plug-in:

1. we substituted the obsolete parallelized components of the coder with a new distributed farmer/workers implementation based on the HARNESS reusable simulation framework [14]
2. we connected the components by means of the PVM-proxy plug-in.

In figure 8 you can see the macro-block architecture of the original SM-IMP coder and in figure 9 the macro-block architecture of the new one.

The presence of the PVM-proxy plug-in introduces a non negligible overhead. However, in our example the original PVM task has been substituted by a the farmer plug-in. This plug-in shares a JVM with both the PVM-plug-in and the PVM-proxy plug-in. Thus, all the data flowing through the PVM demon with destination the proxied task do not need to perform another hop through the TCP stack to reach it. This fact largely compensates the overhead introduced by the PVM-proxy plug-in.

6 Concluding remarks

In the field of metacomputing, features and capabilities are, by definition, subject to constant change. One possible approach to achieve the insulaton of applications from this aspect of platform evolution is to employ a model that is extremely abstract. However, this approach usually leads to very inefficient systems. Our alternative strategy is to build flexibility in the metacomputing framework itself, by permitting software-based reconfiguration in response to both new technological developments and application program requirements. As an example of this flexibility, in this paper we have described the PVM-Proxy plug-in. This plug-in leverages the HARNESS capability to reconfigure the services and programming environments provided by a Distributed Virtual Machine to transparently connecting legacy PVM applications to other Harness applications. This capability allows substituting obsolete components of a legacy application without requiring any change in the

remaining components of the application.

To prove our claim we have adopted as an example a PVM legacy application, namely an MPEG-1 coder, that was targeted to an obsolete heterogeneous parallel architecture. We successfully substituted the two main components of the PVM legacy application with newly developed modules based on the farmer/workers paradigm and Java distributed object technology.

We believe that this methodology endows applications with a great deal of flexibility and the capability to adapt to changing needs both in terms of evolving hardware and software. Besides, Besides, our experiments with the PVM-proxy plug-in show that the overhead introduced by this degree of flexibility does not significantly compromise performance. Nevertheless, future work will aim at further reducing this overhead.

7 References

1 M. Migliardi, V. Sunderam, A. Geist, J. Dongarra, Dynamic Reconfiguration and Virtual Machine Management in the Harness Metacomputing System, Proc. of ISCOPE98, pp. 127-134, Santa Fe', New Mexico (USA), December 8-11, 1998.

2 A. Geist, A. Beguelin, J. Dongarra, W. Jiang, B. Mancheck and V. Sunderam, PVM: Parallel Virtual Machine a User's Guide and Tutorial for Networked Parallel Computing, MIT Press, Cambridge, MA, 1994.

3 M. Migliardi and V. Sunderam, Distributed, Reconfigurable Simulation in Harness, Proc. of PDPTA'99, pgg. 730-736, Las Vegas, June 28-July 1, 1999.

4 T. Lindholm and F. Yellin, The Java Virtual Machine Specification, Addison Wesley, 1997.

5 Sun Mycrosystems, Remote Method Invocation Specification, available on line at http://java.sun.com/products/jdk/1.2/docs/guide/rmi/index.html, July 1998.

6 Sun Mycrosystems, Object Serialization Specification, available on line at http://java.sun.com/products/jdk/1.2/docs/guide/serialization/index.html

7 S. Liang, The Java Native Interface: Programming Guide and Reference, Addison Wesley, 1998.

8 V. Getov, S. Flynn-Hummel and S. Mintchev, High Performance Parallel Programming in Java: Exploiting Native Libraries, to appear on Concurrency: Practice and Experience, 1998.

9 S. Mintchev and V. Getov, Automatic Binding of Native Scientific Libraries to Java, Proceeding of ISCOPE97, December, 1997.

10 M. Migliardi, M. Maresca, Performance Evaluation of the SM-IMP Architecture: a Parallel, Heterogeneous, Image Processing Oriented Architecture, Proc. of the International Symposium on Problems of Modular Systems and Networks, St. Petersburg (Russia), June 26-30, 1995.

11 M. Migliardi, Parallel Heterogeneous Architectures for Image and Signal Processing and Coding, Ph.D. thesis (in Italian), National Library of Italy, Rome, February 1995.

12 MPEG-1 Standard, ISO/IEC International Standard 11172-2.

13 J. Nickolls, The design of the MasPar MP1: a cost effective massively parallel computer, Proceedings of Compcon Spring 1990, San Francisco 26/2 2/3 1990.

14 M. Migliardi, V. Sunderam, Distributed, Reconfigurable Simulation in Harness, Proc. of PDPTA '99 pgg. 730-736, Las Vegas, June 28- July 1, 1999.

Biographies

Mauro Migliardi is born in Genoa (Italy) in 1966. He got a Laurea degree in Electronic Engineering from the University of Genoa in 1991 and a PhD in Computer Engineering from the University of Genoa in 1995.

From 1995 to 1997 he has been a research associate at the University of Genoa where he studied hybrid SIMD-MIMD computers for image processing in research projects funded by the EU and the Italian government.

In 1998 and 1999 he has been a research associate at Emory University and one of the main investigators in the HARNESS heterogeneous metacomputing project.

Currently he is an assistant professor at the University of Genoa.

His main research interests are parallel, distributed, heterogeneous computing systems and architectures, metacomputing and high performance networking.

Vaidy Sunderam is Professor of Computer Science at Emory University, Atlanta, USA.

His current and recent research focuses on aspects of distributed and concurrent computing in heterogeneous networked environments. He is one of the principal architects of the PVM system, in addition to several other software tools and systems for parallel and distributed computing.

He has received several awards for teaching and research, including the IEEE Gordon Bell prize for parallel processing.

Recently his research activities have included novel techniques for multithreaded concurrent computing, input-output models and methodologies for distributed systems, and integrated computing frameworks for collaboration.

MoBiDiCK: A Tool for Distributed Computing on the Internet

Moyez Dharsee
Samuel Lunenfeld Research Institute,
Toronto, ON, Canada
dharsee@mshri.on.ca

Christopher W. V. Hogue
Samuel Lunenfeld Research Institute,
Toronto, ON, Canada
hogue@mshri.on.ca

Abstract

We have developed a software tool called MoBiDiCK ultimately intended for distributed computing. In this report we detail the design and show results using the core components of MoBiDiCK running two different clients on a local cluster. MoBiDiCK is a database driven system that can be used to marshal a large number of processors across the Internet in order to have them collaborate on a single computation. These utilize a message-passing API and control synchronization formalism we have developed that uses the HTTP standard and Web servers. CGI programs on the volunteer processors perform the computations. The problem domains best served by MoBiDiCK are parallel computing problems that are CPU-bound (not I/O-bound), and require minimal inter-process communication. The parallel tasks that we present include analysis of databases of three dimensional protein structures and Monte-Carlo simulations for ab-initio protein folding.

1. Introduction

We are principally interested in the protein folding problem and our motivation to build a distributed computing system arises from our fundamental desire to engineer software systems that have the computational capacity to tackle the high-dimensional problem of protein folding. We are not alone in the pursuit of computational resources, as IBM research has recently announced a project under their "Deep Computing" division to build a massive new computer specifically for the protein folding problem, one that will achieve petaflop performance in five years[1].

We have been designing optimized software for protein folding for some time and we have recently published a report of the first fast all-atom method for generating plausible protein structures in real space[2], and demonstrated that the program, FOLDTRAJ, has O(NlogN) time complexity in protein length.

FOLDTRAJ embodies over 10 years of software design and development work. FOLDTRAJ is an application developed using the National Center for Biotechnology Information (NCBI) software development toolkit[3]. The NCBI SDK comprises source code used in many bioinformatics applications such as Entrez, an integrated bioinformatics database; BLAST, a tool for searching DNA and protein sequence databases; Cn3D, a tool for three-dimensional molecular structure visualization; Sequin, an annotation tool for sequence databases, and a growing number of Web based applications including the PubMed system, one of the top scientific sites on the Internet. Within the NCBI toolkit lies rich source code resources including a suite of tools written in C for ASN.1 data specification, encoding, decoding and code generation, a comprehensive HTTP protocol interface, as well as our own work on a comprehensive programming interface for 3-D structure applications known as MMDB-API[4].

To tackle the protein folding problem, we wish to use FOLDTRAJ on thousands, possibly tens of thousands of computers. Distributed computing has already been explored for other large problems. Cryptographers studying brute-force methods to crack encryption schemes have already laid much of the groundwork for distributed computing[5]. The SETI@home project (Search for Extra-Terrestrial Intelligence) has demonstrated that over a million processors across the Internet can be brought to bear on a very large problem and that people are eager to volunteer spare CPU cycles for such causes[6]. FOLDTRAJ has been carefully implemented so that it runs on a large number of computing platforms including NT and several UNIX variants, making it an ideal client for distributed computing.

0-7695-0556-2/00 $10.00 © 2000 IEEE

The potential of the Internet as an infrastructure for distributed computing has been predicted to reach exaop performance in the year 2007, perhaps exceeding the performance of massively parallel supercomputers at that time by up to three orders of magnitude[7]. Current efforts have already demonstrated that reasonable coarse-grained parallelism can be achieved using processors in the Web.

MoBiDiCK is the Modular Big Distributed Computing Kernel. Our goal in designing MoBiDiCK was to allow us to marshal our own in-house computing resources (Sun and SGI servers, workstations, a Beowulf cluster, secretarial and laboratory computers). The system's design, however, allows any computer with an Internet connection and the ability to run HTTP server software to participate in a distributed computation.

The development of applications for heterogeneous computing environments can be undertaken in a variety of ways. Some methods require the adoption of a particular programming model, as in Java and MPI, that is coupled with a specific computing environment, like the Internet or a local network of workstations. Other approaches offer underlying communication and management infrastructures that can integrate various existing software models, as exemplified by the Globus project[8]. The MoBiDiCK effort fits best in the middle of this spectrum. MoBiDiCK is neither an infrastructure nor a programming model, rather it is the middleware to connect the two. The communication technology used by MoBiDiCK is far from novel: TCP/IP, HTTP and CGI are long standing Internet standards, and the idea of using Internet-connected hosts for distributed computing dates back several years.

Many scientific computations are essentially iterative, that is, the same set of instructions are repeatedly applied to elements in the problem domain space. These types of programs often lend themselves well to a distributed approach, in which the problem domain is divided among a set of nodes that simultaneously, and more or less independently, execute the same program. One of our goals is to minimize the effort required to introduce this kind of parallelism into existing code. We address this by providing methods to enable an application to operate as a CGI program in a distributed environment, without hampering the application's ability to be used in a stand-alone environment.

Finally, we sought to build a database driven system. The benefits of a database driven system include scalability of the system from cluster computing to wide-area and globally distributed computing, as well as providing records of past performance of applications that can be used to set up initial load balancing and improve the overall scheduling of distributed computing tasks.

2. Enabling Technologies

2.1. Hypertext Transfer Protocol

The Hypertext Transfer Protocol (HTTP) was introduced around 1990 to address the need for consistency in the manner in which computers connected to the Internet should exchange information, and has since evolved to become a *de facto* standard and the most widely used protocol on the Internet. HTTP relies on TCP/IP, in which IP (Internet Protocol) serves as an addressing scheme for naming and identifying Internet hosts, and TCP (Transmission Control Protocol) provides routing, error detection, error recovery, sequence control and sequence flow mechanisms for data transmitted from one host to another[9].

To access a document on some host, the user sends a request through a client program, such as a Web browser, to a HTTP daemon running on the host. The daemon, or Web server, processes the request and returns output back to the client. Authentication and access control functions are provided by the Web server to secure private data. Both client and server software must conform to the HTTP specification for the transmission and execution of requests.

The motivation for HTTP was to bring together and share disparate information located on geographically distributed machines. A shared document is attributed a Uniform Resource Identifier (URI) that denotes the document's location in the network, by identifying the computer's IP address and the file path. Files can be accessed directly with the URI and can be interconnected by reference links embedded in the document using the Hypertext Markup Language (HTML). Documents whose content is not plain-text can also be accessed. The Multimedia Internet Mail Extensions, (MIME) is applied to HTTP to identify a file format with a specific MIME-type and to inform the client about the type of data being requested[9].

2.2. Common Gateway Interface

Along with the ability to provide access to static documents, i.e. those represented by individual files on local storage, Web servers must also be able to produce dynamic content which depends on specific user input. For example, the Web server must be able to accept a search key from the user, perform a database query, and return the search results. Such capabilities transcend the scope of a general-purpose Web server. The Common Gateway Interface (CGI) was established to address the need for HTTP software to produce executable content[10]. When the client's HTTP request refers to an executable file, rather than a static one, the Web server launches the application as a new, separate, server-side

process. Input parameters required by the CGI program are appended to the request; the Web server simply passes these on to the application, often by way of environment variables. When the application completes, the server returns its output back to the client. CGI guidelines provide an API for the manner in which data should be exchanged between an application and a HTTP daemon. Since this API is purely syntactic, a CGI executable can be programmed using nearly any language or platform.

An important drawback of CGI is that a separate process must be started by the Web server for each client request. Process creation and initialization overhead can cause a significant performance bottleneck if multiple consecutive requests must be served. A CGI program also competes for system resources with other processes, including the HTTP server itself. Furthermore, HTTP is a stateless protocol that does not directly support the saving of information between requests.

The FastCGI open protocol addresses these drawbacks by enabling a CGI program to persist across multiple HTTP requests, thereby reducing process creation and initialization overhead and allowing state information to be maintained between requests[11]. The FastCGI application library facilitates new application development and easy migration of existing CGI programs. It also supports a distributed configuration whereby a program can be invoked remotely by the Web server over a TCP/IP connection. As a future direction, we intend to use the FastCGI application library for the migration and further development of MoBiDiCK modules.

3. System Design

3.1. System architecture

MoBiDiCK is a CGI-based approach to distributed and parallel computing. It operates on a set of nodes interconnected by a TCP/IP network. A node is simply a networked computer that can act as a Web server, i.e. it should be able to support the correct execution of HTTP server software. This requires that it be assigned a static or dynamic IP address. Network and node characteristics may affect performance, but there are no specific hardware or operating system requirements other than those imposed by HTTP server software. A node can be a standard workstation, a cluster node, or a multiprocessor. Local resource requirements such as disk space, memory, network and I/O bandwidth are commensurate to a particular computation. As a general guideline, applications are designed to minimize local resource consumption.

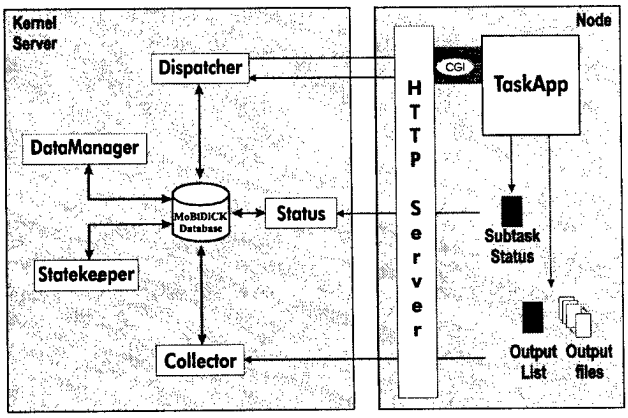


Figure 1. System architecture

In the simplest configuration, a *kernel server* is designated such that it can reach all processing *nodes* through HTTP; that is, Web servers running on the nodes should be able to process HTTP requests received from the kernel server. Five *kernel modules* are installed on the kernel server: Dispatcher, Status, Statekeeper, Collector, and DataManager. These are distinct CGI applications that carry out system and data management functions such as node registration, task definition, task mapping, job monitoring, fault tolerance, load balancing, task migration, output collection and cleanup. *TaskApp* modules, which are also CGI-driven applications, run on the nodes; one or more TaskApp modules may be installed on a given node, each embodying a distinct computational problem. For a computation to be distributed over a set of nodes, a corresponding TaskApp must be installed as a working CGI program in the Web server's published directory tree. Figure 1 illustrates the general system architecture.

Interactions between CGI modules involve only one or two consecutive HTTP requests. For example, when a computation is required of a node, the Dispatcher sends a request to the node's Web server, then simply terminates. Jjust before starting the subtask, the newly launched CGI process on the node responds by sending an acknowledgement request back to the kernel server. This invokes a new Dispatcher CGI process which records the acknowledgement and immediately exits. Meanwhile, the TaskApp process continues with the computation.

Data management is driven by a relational database implemented with the CodeBase API (Sequiter Software, Alberta, Canada, www.sequiter.com), an efficient library of high- and low-level data management functions that, when integrated with the kernel modules, provide a fully-functional, platform-independent embedded relational data management system. Access overhead is minimal since all database functions are performed through API procedure calls from within the application source code, and updates and queries do not require explicit connections to an external database engine[12]. The

database is used to hold and manage all system data, including descriptive and statistical information pertaining to nodes and computations, input parameter values, and output files.

The kernel's modularity allows CPU, I/O and network resources to be optimized. System management functions can be distributed by spreading the kernel modules across multiple servers. For example, a distributed kernel configuration may comprise four servers: the Dispatcher is installed on one server, the Collector on a second, Status and Statekeeper on a third, and the DataManager on a fourth server. The database disk is remotely mounted by each server and shared. The use of a multiprocessing kernel server is an alternative configuration that may benefit CPU bandwidth as kernel modules can run simultaneously on several processors. I/O contention during output collection can be minimized by high-performance storage solutions such as SCSI, RAID, and FibreChannel.

3.2. Communication model

MoBiDiCK modules communicate by embedding messages within HTTP "GET" and "POST" requests. Figure 2 illustrates the general communication model between a module m_1 running on node N_1 behind HTTP server S_1, and a module m_2 running on node N_2 behind HTTP server S_2. The steps involved in transmitting a message from m_1 to m_2 are outlined below.

1. m_1 opens a TCP connection, C, between N_1 and N_2
2. m_1 sends a HTTP request containing message, M, for m_2 over C
3. S_2 receives the request, starts m_2, and passes the request to m_2
4. m_2 extracts M from request, processes M, and writes a reply R to *stdout* stream
5. S_2 captures R, attaches an HTTP header to it, and sends it over C
6. m_1 receives the response from S_2 and reads R
7. m_1 closes C

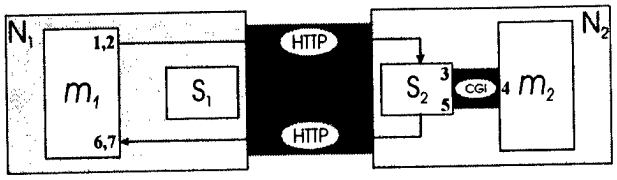


Figure 2. Module communication m_1 and m_2 are CGI modules running on nodes N_1 and N_2, behind HTTP servers S_1 and S_2.

Modules m_1 and m_2 designate any two CGI programs on any two nodes. For example, m_1 can represent the Dispatcher and m_2 a TaskApp module on a compute node, the message transmitted between them being a computation request. Only HTTP server S_2 on the receiving node participates in the exchange, while S_1 remains idle.

This framework does not preclude other communication models within a TaskApp module, such as MPI message-passing primitives and other programming models like PVM and Linda, as well as new parallel application development models.

3.3. Task definition

A *task* conceptually represents an entire computation. A task's attributes hold various characteristics of an associated TaskApp, such as application filename, node filesystem path, and input and output requirements. A particular execution of a task is an *instance*. An instance inherits the task's attributes and parameters, and has its own attributes such as start and end times, total execution time, and the set of nodes performing the instance. Runtime options such as output collection and logging are also associated with an instance.

Once instantiated, the task is partitioned to produce a set of *subtasks* that are mapped to the set of nodes selected to perform the desired computation. A subtask inherits the attributes and task parameters of its associated instance. Each subtask represents a unique TaskApp process. *Dispatching* a task consists of sending subtask requests to the nodes and launching the TaskApp processes. While a TaskApp is running, its corresponding subtask is *active*; a *complete* subtask represents a process that executed successfully; a subtask is *incomplete* if the process did not finish executing due to user intervention, node failure, or communication failure. A task instance is active as long as there remain nodes executing subtasks, and complete only when all subtasks have been completed.

Input arguments required by a task are specified by *task parameters*. A task parameter can be an integer, real, Boolean, or string, and can be constant or variable. A constant parameter is assigned a single specified value for all subtasks. A variable parameter has an associated range which represents the parameter's value space, delimited by start and end points. The range's *distribution* defines the manner in which the range should be divided among subtasks. An *incremental* distribution signifies that a specified step function should be applied to the range to calculate subtask parameter values. A *partitioned* distribution divides the range into a series of subranges, with each subrange being assigned to a subtask. The *upper boundary* and *lower boundary* parameter types are system-defined *dependent* parameters associated with the subrange boundaries of a partitioned parameter. A dependent parameter can also be defined as a mathematical or logical function of other parameters. Task attributes and parameter information is defined

through a browser user interface and stored in the database by the DataManager.

3.4. Node registration

A processing node is registered and scheduled using a browser interface, invoking the DataManager module to record the information in a database. Registration involves providing key attributes such as host name or IP address, CPU type and speed, number of CPUs, operating system, disk and main memory capacity, as well as contact information (in the case of a volunteer processor). Once registered, a processor can be scheduled by selecting the hourly time slots during which it will be available for each day of the week. The processor's participation can also be restricted by specifying which tasks it is available to perform.

This is an all-or-none scheduling model: a processor participates in a computation only when its schedule allows. The all-or-none model is well suited for people who wish to volunteer a known and fixed amount of CPU time, such as the times outside regular business hours for an office computer. By contrast, nodes that participate in the SETI@home project run a computation selectively through a specified "nice" value that assigns a local scheduling priority to the process[6], and restricting access to the node at some point in time requires explicit user intervention.

3.5. Task execution

A distributed computation is requested from the Dispatcher through a browser interface. Once invoked, the Dispatcher initiates *interactive node selection* by compiling a list of candidate nodes for the task. A node's candidacy for a given task is determined by the following conditions:

- *Registration*: the node is registered in the database
- *Participation*: the node is registered to participate in the task
- *Accessibility*: access to the node is currently permitted by the node's schedule
- *Connectivity*: the node is reachable by the Dispatcher
- *Configuration*: the TaskApp is operational on the node

Registration and participation are verified simply by querying the database: a node must be registered in the database if it is to be used at all, and the selected task must appear in the node's participation list to indicate its "willingness" to perform the task. Accessibility is determined by looking at the node's access schedule to see if the node is currently accepting requests and if it will remain usable for a sufficient amount of time. Connectivity and configuration are determined in a single handshaking step whereby the Dispatcher sends a "test" request to the TaskApp on the node. If no response is received from the node, or if an error is encountered in establishing a connection, then neither condition is met. If the node's Web server responds with an error then connectivity is achieved but the node fails to meet the configuration condition. If a correct response is received from the TaskApp then it is concluded that the node's Web server was able to launch the TaskApp successfully and therefore the node fulfills both conditions.

Other node selection conditions can be specified by a user to further constrain node candidacy, such as cut-off values for node rating, storage space, total main and temporary memory, and number of CPUs. Selection can be restricted to specific categories of nodes, such as local, remote, dedicated, or shared. Manual selection of specific nodes can also be done to bypass preset conditions.

After node selection is settled, the Dispatcher is prompted to carry out the *task mapping* phase by partitioning the task into subtasks and mapping the subtasks to selected nodes. More than one subtask may be assigned to a node, as may be desired for a multiprocessor; a node's subtask-to-CPU relationship is definable in its registration information and can be modified by the user prior to dispatching.

Task mapping incorporates an *initial load balancing* step that computes the load of each subtask based on the associated node performance rating. *Subtask load* represents a fraction of the task's total workload that should be allotted to the node to which the subtask is mapped. A node's rating can be obtained by running a benchmarking TaskApp module that measures the node's ability in floating-point and integer arithmetic, memory access, disk access, and communication. If the rating is known, subtask load can be calculated as follows.

For a set of processing nodes $p_1, p_2, \ldots, p_n$ with corresponding ratings $r_1, r_2, \ldots, r_n$, the *weighted rating* for p_i is,

$$R_i = \frac{r_i}{\Sigma (r_1, \ldots, r_n)} \quad (1)$$

Subtask load is obtained by dividing a node's weighted rating by the number of subtasks assigned to that node. Hence subtask load for node p_i that is assigned k subtasks is,

$$L_i = \frac{R_i}{k} \quad (2)$$

In the *parameter allocation* step, task parameters are inherited by each subtask and values are assigned to these parameters. The value assigned to a subtask parameter depends primarily on the parameter's type. If it is a constant then it simply takes on the value defined for the task. If it is a variable parameter with a partitioned

numerical range, then the value is found by applying the subtask load to this range. For example, given a numerical parameter, X, with a range delimited by lower boundary x_L and upper boundary x_U, the value of X for node p_i is,

$$X_i = L_i(x_U - x_L). \quad (3)$$

The mapping process can be repeated if the user wishes to modify node candidacy conditions or task attributes and parameters. At the user's request, the Dispatcher begins the computation by sending *subtask requests* to the selected nodes. A subtask request is successful if the Dispatcher receives a confirmation message from the TaskApp module. If all requests are successful, the task instance is in the *active* state.

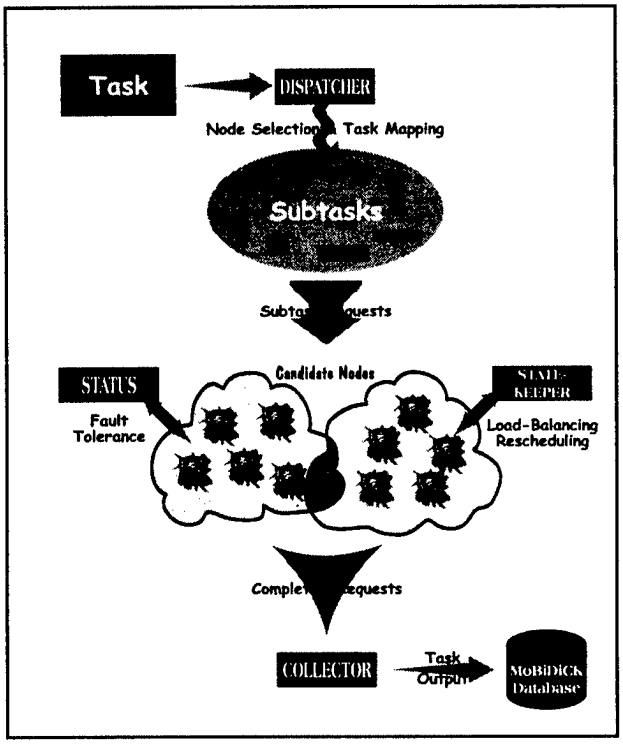


Figure 3. Task execution The Dispatcher partitions and distributes tasks to currently scheduled nodes, while the Collector gathers subtask results. During the execution of a task, the Status and Statekeeper modules ensure fault-tolerance, load-balancing and correct scheduling.

If the task is successfully launched, The Dispatcher invokes a new Status process and a new Statekeeper process to monitor the TaskApp processes. During its execution, a TaskApp regularly updates a local SubtaskStatus file (shown in Figure 1) with subtask progress information. Fault-tolerance during a computation is assured by the Status module by periodically downloading each node's SubtaskStatus file. If a subtask fails to complete on a node, the Status module re-assigns the subtask. The Statekeeper's role is to monitor node schedule overflows and maintain dynamic load-balancing. It can reschedule subtasks by terminating existing subtasks and starting new ones, thus maintaining the computation in the "all-or-none" state according to each node's schedule in the database. It can remap the entire task to a new set of nodes if too many schedule conflicts are encountered during a computation or if node availability changes dramatically.

Local output files produced by a TaskApp are recorded in an OutputList file. When a TaskApp completes a subtask, it sends a *completion request* to the Collector. The Collector updates the subtask's status in the database, obtains the OuputList from the node, gathers the output files from the node and stores them in the database if required. The Collector is capable of storing arbitrary data objects as binary files in the database, and iterators are provided in the API for summarizing or combining results once the dispatched subtasks are all completed. After all output has been received, the Collector sends a *cleanup request* to the TaskApp, asking it to delete the output files it produced on the node's filesystem, as reported in the TaskApp's OutputList.

3.6. Task API

Task modules are programmed using the Task API, itself integrated with NCBI Toolkit libraries. The Task API facilitates the development of platform-independent, CGI-enabled applications which can be operated both as stand-alone executables through a regular command-line interface, and as CGI programs that can be invoked by a Web server daemon when it receives a client's HTTP request to run the application. In other words, the use of the Task API does not restrict the application to the MoBiDiCK system. Using the same executable, the user can choose to perform the computation manually, or to install the program behind HTTP servers on a set of nodes so that the task can be distributed using the MoBiDiCK kernel. The core Task API functions are shown in Table 1.

This flexibility is achieved by formally defining input parameters in the TaskApp program code. Each task parameter is defined in a TaskArg structure. The members of the TaskArg structure contain, among other information, a parameter's command line tag, CGI field name, type (integer, string, boolean, etc.), range of allowed values, and default value if any. A set of task parameters is defined by declaring an array of TaskArg structures. Parameter values are obtained by the application with the GetTaskArgs function, which detects the input method (terminal or HTTP), and accordingly reads, validates and copies parameter values in the TaskArg array for subsequent access.

Many computations consist of a main loop that iterates over a fixed and predetermined range of values, such as a

sequence of numbers, a list of strings, or a collection of records. Tracking the progress of these types of programs is achieved by calling the TaskSetSize function prior to entering the main loop, and by placing a call to the TaskIterate function in the loop body. TaskSetSize sets the "problem size", e.g. the total number of iterations to be performed in the main loop. Each time TaskIterate is called, the loop's progress is computed by dividing the current iteration number by the total number of iterations. The progress is thus the percentage of the loop that has been completed. If the time required for each iteration is more or less uniform (the loop body is deterministic), then progress is proportional to the computation's current running time by a more less constant value. When the loop body is non-deterministic, as is the case for random-walk algorithms, the progress still provides a measure of where in the loop the process is currently positioned. Subtask progress is written to a SubtaskStatus file in the Web server's published directory tree. This file is periodically obtained by the Status kernel module while it monitors the computation.

To communicate with a running TaskApp process, messages or signals can be placed in its SubtaskStatus file. For example, when the Statekeeper module must cancel an active subtask, it invokes a "kill" TaskApp process on the node, requesting it to interrupt the execution of the subtask. The kill process writes a "cancel" signal to the appropriate SubtaskStatus file. The signal is detected and carried out in the TaskIterate function at the next iteration of the subtask process, as it reads the SubtaskStatus file before updating it. Thus TaskIterate not only reports a subtask's progress to the kernel, but also serves to communicate with a TaskApp process during execution.

Table 1. Core Task API functions

GetTaskArgs	GetTaskID
InitTask	GetInstanceNum
TaskSetSize	GetSubtaskID
TaskIterate	TaskLogWrite
TaskInterrupt	ErrLogPostEx
RecordOutput	TaskComplete
GetInputMethod	

Output produced by a TaskApp is written in the form of one or more output files on the compute node's filesystem. If output is to be collected by the kernel server, the RecordOutput function is used to record the names of files to be collected in an OutputList file written in a Web-published directory on the node. At the end of its computation, a TaskApp process informs the Collector module that output is ready to be collected. After obtaining the OuputList file from the compute node, the Collector proceeds to get all the files contained in OutputList and stores them in the central database or other specified storage area.

Core library functions provided in Task API are listed in Table 1. Several other routines are also available as part of the NCBI SDK, on top of which the Task API is built, such as ASN.1 encoding, which we use in the FOLDTRAJ TaskApp module. The general structure of a TaskApp program is shown in Figure 4. A simple set of rules, outlined below, must be followed in order to produce TaskApp programs that can correctly interact with the kernel.

a) Define all input parameters in the TaskArg array, allowing the TaskApp to receive arguments from the Dispatcher.

b) Obtain input arguments with GetTaskArgs. This enables the program to get arguments from either the command-line console or from the Web server. If invoked by the Dispatcher, the TaskApp also obtains required MoBiDiCK arguments (e.g. Task ID, Dispatcher and Collector IP addresses).

c) Use InitTask to initialize the computation. Node-specific settings are read from a TaskApp configuration file; log files are initialized.

d) Set the size of the computation with TaskSetSize prior to main loop.

e) Call TaskIterate in main loop to record subtask progress information and read signals from other modules.

```
…
function and variable definitions
…
TaskArg array definition
…
main(…) {
 …
 GetTaskArgs(…);
 …
 InitTask();
 …
 TaskSetSize(…);
 …
 mainLoop {
       …
       TaskIterate();
       …
       RecordOutput(…);
       …
 }
 …
 RecordOutput();
 …
 TaskComplete();
 …
}
```

Figure 4. TaskApp structure and core Task API functions

f) Use RecordOutput to report all output files produced during the computation. This enables output collection (by the Collector module) as well as output cleanup.
g) Call TaskComplete before exiting. This invokes the Collector to collect output, request output cleanup, and record timing and status information.
h) If the TaskApp must exit prematurely during the computation, use TaskInterrupt.
i) Record informational and error messages in subtask-specific log files using TaskLogWrite and ErrLogPostEx.

4. Results

4.1. RAMAPLOT

Using the Task API we developed a TaskApp module called RAMAPLOT. This program uses three-dimensional protein structure information from NCBI's Molecular Modeling Database (MMDB) to generate a Ramachandran plot for the structure, which is a graph of the distribution of the protein's α-carbon bond angles in angular 2-D space (written as a GIF file). We performed this task using 15 nodes on our Beowulf cluster, each node configured with two Intel Pentium II 400MHz processors, 512Mb of RAM, the RedHat Linux operating system, and the Apache HTTP server. The nodes were interconnected by a 100Base-T network. The kernel server was a Sun Sparc Ultra-1 running Solaris 2.6. The MMDB database was copied to each node's hard disk.

The goal of the task was to generate one Ramachandran plot for each of 851 protein structures in the MMDB. Two input parameters, *dbsize* and *dbstart*, were defined for the RAMAPLOT task. The partitioned parameter, *dbsize*, represents the number of records to be processed, ranging from 1 to 851 (corresponding to the first and last record indexes of the database, respectively). *dbstart* is defined as a lower boundary for the subranges of *dbsize* serving to inform the program of the starting database record number. We performed 15 instances of the RAMAPLOT task, starting with a single node and adding an additional node for every new instance. Only one CPU per node was used. Execution time, speedup, and efficiency were determined for each instance and are summarized in Table 2 and plotted in Figures 5, 6 and 7.

Table 2. RAMAPLOT timing results using MoBiDiCK

Instance	Subtasks	Time (s)	Speedup	Efficiency
1	1	714	0.972	97.2
2	2	378	1.84	92.0
3	3	265	2.62	87.2
4	4	195	3.57	89.1
5	5	165	4.20	84.1
6	6	144	4.82	80.3
7	7	116	6.01	85.8
8	8	106	6.55	81.9
9	9	107	6.47	71.9
10	10	96	7.19	71.9
11	11	79	8.79	79.9
12	12	87	8.00	66.7
13	13	76	9.09	69.9
14	14	66	10.49	74.9
15	15	69	10.07	67.1

Speedup is defined as the ratio T_P/T_S, where T_P is parallel execution time and T_S is the best serial execution time. T_S, obtained by running RAMAPLOT on one cluster node through a command-line interface, was found to be 695 seconds. Efficiency is defined as the ratio S_O/S_{max} of observed speedup over ideal speedup, with S_{max} always equal to the number of nodes.

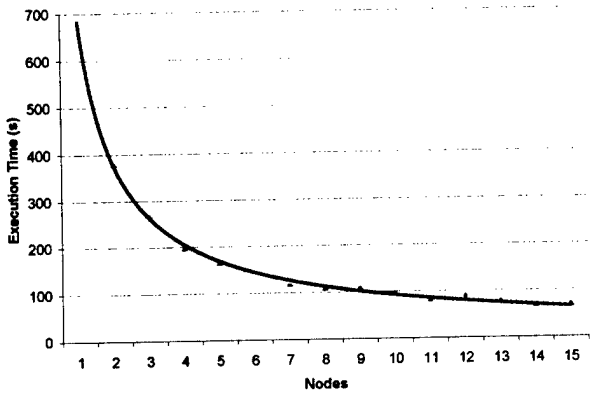


Figure 5. Execution time for RAMAPLOT. The task was repeated by varying number of nodes.

A steady speedup was obtained as the number of nodes was incremented from 1 to 15. The use of all 15 nodes yielded a speedup of 10.1, corresponding to 67.5% efficiency.

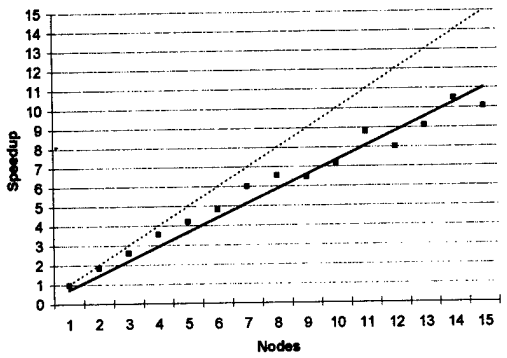


Figure 6. Speedup for RAMAPLOT. Ideal speedup is represented by the broken line.

An increase in the number of subtasks raises the probability that any two subtasks complete at the same. This leads to increased network and I/O contention on the kernel server as subtask completion requests invoke Collector processes. These effects are reflected in Figure 7 by a regular decrease in efficiency as the number of nodes rises. An average efficiency of 80.0% was achieved over all 15 instances.

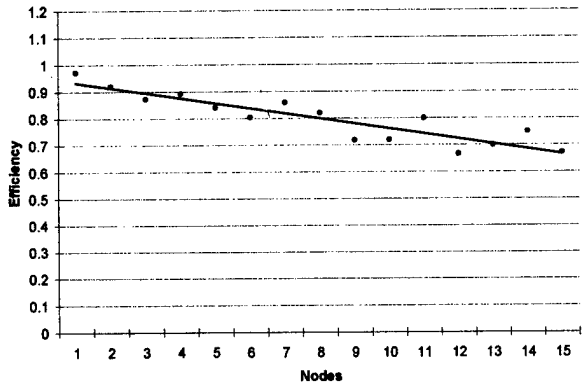


Figure 7. Efficiency for RAMAPLOT

4.2. FOLDTRAJ

Much of our work is dedicated to the protein folding problem. This problem in the field of structural biology represents our inability to computationally predict the three-dimensional conformation of arbitrary proteins given only primary amino-acid (AA) sequence information. Given an input file known as a "trajectory distribution" containing angular 2-D space amino-acid frequency information about a particular protein, FOLDTRAJ can generate a number of random yet chemically valid protein conformers, placing each in a separate output file in binary ASN.1 or ASCII PDB format. The correctness of a predicted structure, important in developing methods to score the fitness of generated proteins, is measured by calculating its Root Mean Squared Deviation (RMSD) relative to the protein's native fold.

FOLDTRAJ was developed independently and intended to be used both as a stand-alone application and in a distributed computing framework. We used the Task API to migrate FOLDTRAJ to operate as a TaskApp under MoBiDiCK. Integration with the Task API also enabled the same FOLDTRAJ executable to be operated through an HTML-based interface in both stand-alone and client-server modes.

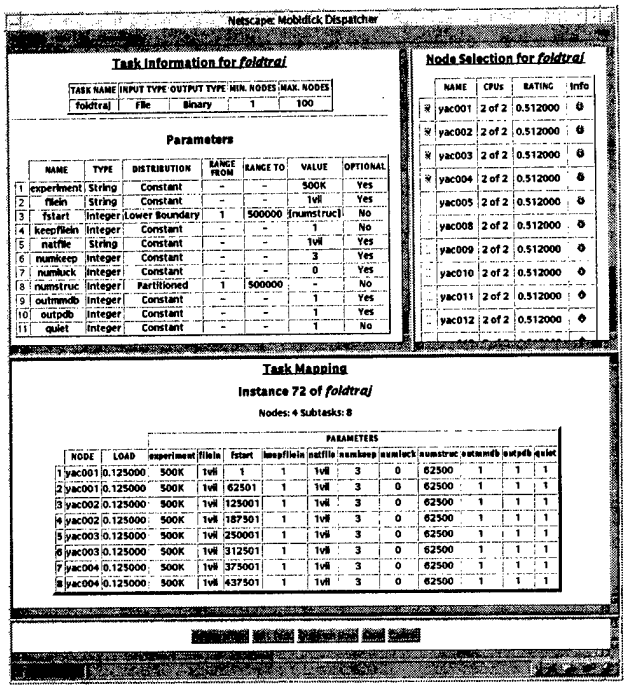


Figure 8. Task definition and mapping of FOLDTRAJ. The definition includes general task information and parameter information (top left frame). Four dual-processor nodes are selected (top-right frame). The task has been partitioned into 8 subtasks, each mapped to a node's processor. Subtask parameters are inherited from the task and assigned values (middle frame). The task can be re-partitioned after editing the task or modifying node selection (through menu in bottom frame).

An example task definition and mapping for FOLDTRAJ is shown in Figure 8. The task has three significant parameters: *filein, numstruc* and *fstart*. The first, *filein*, is a constant string parameter that holds the name of the input file; in the figure, the input filename "1vii" corresponds to the 36 amino-acid Villin headpiece protein, a small protein we use for testing. The partitioned integer parameter, *numstruc*, is the total number of structures to be generated; its range is set from 1 to 500,000, signifying that half a million structures are to be generated. The integer parameter *fstart* denotes the

starting structure number and is set as a lower boundary of *numstruc*, with the same range. The example instance was mapped to four of our dual-processor cluster nodes, using both CPUs per node, resulting in 8 subtasks as shown in Figure 8. Subtask loads are equal (0.125) since the nodes were rated equally. Since *filein* is constant, each subtask is assigned the same value. The range of *numstruc* is divided equally among the 8 subtasks. A subtask's *fstart* value informs the TaskApp where in the range it should start numbering its structures and is used to produce unique output filenames.

Using MoBiDiCK we regularly perform distributed FOLDTRAJ computations to carry out prediction experiments on various proteins. Figure 9 plots the results of four such experiments. In each, 50,000 protein structures were generated using 15 of our dual-processor cluster nodes. The frequency distribution of the resulting Root Mean Squared Deviation values indicate the accuracy of protein backbone atom prediction of random protein conformers made with FOLDTRAJ. From this ongoing testing we are obtaining a better understanding of the relationship between sample size, protein size and how well entities in the sampled protein ensembles fit the true structure of a protein.

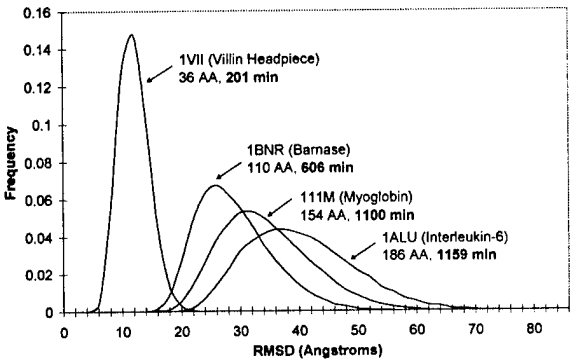


Figure 9. RMSD frequency distribution of random protein structures generated with FOLDTRAJ. Each curve represents a separate instance of FOLDTRAJ generating 50,000 structures using 30 subtasks on 15 nodes; protein name, size in number of amino-acids (AA), and execution time are indicated for each experiment.

5. Related work

Existing cluster computing tools that are publicly available include PVM (Parallel Virtual Machine) and MPI (Message Passing Interface). Commonly used on LANs and clusters of workstations, these systems provide resource encapsulation and monitoring functions, and transparent heterogeneity utilities through a messaging API. PVM provides a high-level system for a user to coordinate tasks spread across a network of heterogeneous workstations. The set of nodes is perceived as a single virtual machine through a message-passing abstraction and a library of functions for task creation and management[13][14].

Other systems aside from the more widely used PVM and MPI are numerous and many of their aspects can be directly compared with MoBiDiCK. Their applications to cluster and distributed computing have provided us with a useful study, including Globus, SuperWeb, Condor, Linda, Piranha, NOW, Legion, WebOS, Atlas, ParaWeb, Bayanihan, Popcorn, Charlotte, JPVM, RMI, CORBA, Javelin, Nimrod, Clustor, JICE, LSF. At the time we began building MoBiDiCK (Oct. 1997) it was not clear to us that other systems were capable of doing the multiple duty that FOLDTRAJ required for distributed computing over the Internet with clients at this level of sophistication. We therefore set out to develop MoBiDiCK with goals that it provide an integrated environment for process control as shown in Figure 8, and be capable of large scale, high performance, database-driven, heterogeneous distributed computing.

6. Future Directions

6.1. Estimating subtask execution time

In general, the execution time of a subtask on a node is influenced by (1) subtask load and (2) node rating. Subtask load is a computed fraction of the total work to be done in the task. Node rating is a measure of a node's performance as determined by a benchmarking procedure that incorporates limiting factors associated with network bandwidth and congestion, CPU performance, memory and storage availability, and I/O efficiency. A node's rating is directly proportional to this measured performance.

The execution time T of subtask, S, with load L assigned to a node with rating R can be estimated by $T = hF(L)/R$. F is the task's time complexity as a function of subtask load. If known, F can be supplied as an attribute of the task and saved in the database. The constant h is given by $F(L_i)/R_i$, where L_i is the average subtask load over previous instances (with the same parameter definitions) and R_i is the average rating of all nodes that computed these subtasks. h thus captures a task's performance history over all previous instances and nodes. Since timing and node rating information are stored in the database for every task instance, k can be quickly updated after each task execution, keeping its value readily available to the Dispatcher to calculate T during the node selection phase of future instances.

6.2. Selective fault tolerance

The problem of managing distributed computations on a collection of heterogeneous nodes is challenging: both

the availability and performance of nodes are unpredictable, and effective mechanisms must exist to detect and handle failures. Fault-tolerance duties in MoBiDiCK are centralized at the kernel level, instead of being distributed as in PVM. This is because in a distributed computing model the authority granted to compute on a node is much more limited. Malfunctions that occur during a computation are abstracted from the TaskApp, which in many cases is favorable to the application developer since it removes the burden of appropriately responding to failures. Two conditions should remain satisfied throughout the execution of a task: the task will complete and performance is maximized.

Task completion is ensured by detecting and remedying process-level and node-level failures. Process failure occurs when a running TaskApp program is prematurely terminated either by the process itself due to execution error, by the operating system, by the HTTP server, or by direct user intervention at the node console. Causes of node failure include operating system instability, faulty hardware, HTTP server malfunction or misconfiguration, and communication link breakdown.

To detect and respond to such occurrences, the Dispatcher invokes the Status module as soon as a task is dispatched. Status carries out monitoring, fault detection and fault recovery functions for an active task instance. From each node performing a task, Status periodically downloads the SubtaskStatus file produced by the TaskApp process. This file contains a current percentage-done progress of the assigned subtask; the subtask's database record is updated with each new progress value. Process failure is suspected if no change in subtask progress is observed over a sufficient length of time. If the SubtaskStatus file cannot be obtained from the node's HTTP server after several attempts, node failure is suspected. Node failure implies the failure of all active subtasks assigned to the node.

The Status module can respond to subtask failure in a variety of ways. The exact measures to be taken can be user-specified before and during the computation. Possible fault recovery behaviors are:

(a) *Carry on with the computation*
The remaining subtasks are left running and the failure is disregarded.

(b) *Cancel execution*
Status sends a "cancel" signal to the participating nodes in order to terminate the remaining subtasks for the instance. The computation is terminated and no further action is taken.

(c) *Restart the execution*
The instance is cancelled as in (b) and the Dispatcher is invoked to launch a new instance, replacing the faulty nodes with new ones if possible.

(d) *Re-allocate the failed subtask*
This can be done in at least two ways:
- reassign or migrate the subtask to a new node if one is available, or
- redistribute the subtask's load to other active nodes.

6.3. Task migration

The all-or-none model that MoBiDiCK uses offers some unique cases to consider for task migration. The *access period* represents how long a node is available at a given point in time; it is computed from the node schedule, a block of 24x7 cells representing each hour in the week. For example, if a node's access schedule permits use from 6 p.m. to 11 p.m. on a given day then, at 8:30 p.m. the same day, the access period is 2.5 hours. A *schedule overflow* occurs when the time to completion of a TaskApp process running on a node exceeds the node's access period. During a computation, a Statekeeper kernel process periodically scans the database to detect possible schedule overflows, by checking a subtask's progress (updated by the Status module) against the access period of the node performing the subtask. If an overflow is anticipated, the subtask is migrated to another node by the Statekeeper, by invoking a new Dispatcher process to re-send the failed subtask.

Prior to task dispatching, an initial load balancing step determines the load of each subtask based on the rating of the associated node. Yet a node's performance may vary during the computation due to, for example, increased CPU or I/O contention, causing a TaskApp process to slow down. If left unchecked, this can lead to significant idle time and hence reduced speedup. To avoid this, we intend to equip the Statekeeper module with dynamic load balancing and migration functions through which the load of a slow subtask can be wholly or partially transferred to faster nodes.

6.4. Other directions

In addition to the above work showing the use of MoBiDiCK in a controlled cluster environment, further components are being implemented to allow wide distributed computing.

6.4.1. Accessing clusters with hierarchical kernels. Many Beowulf clusters are set up using IP addresses in the 192.168.x.x or other non-public ranges. This precludes them from being seen over the Internet and used

in the MoBiDiCK system described so far. However most are configured with a "gateway" node, which is usually the cluster "head" and has a public IP address and is set up to use IP masquerading so that nodes can access the Internet. We have devised a method to allow MoBiDiCK to operate on these clusters, forming "clusters of clusters". This is a configuration that consists of a *root kernel* and several *child kernels* that manage computations on local site nodes only. Since the Dispatcher and other kernel modules are already CGI modules, they can themselves be made into TaskApp processes, and arranged in a hierarchy. A *child kernel* receives a single subtask from a *parent kernel.* The parent kernel sees the *child kernel* as a single multiprocessor system with a subtask load based on the cumulative rating of the child kernel's local nodes. The child Dispatcher interprets the subtask as a local task, and thus applies the same partitioning and mapping mechanisms as the parent Dispatcher. Output is first collected locally; the child Collector then passes local output to its parent Collector. Child Status and Statekeeper modules maintain fault-tolerance and task migration locally, while sending periodic summary progress reports to their parent counterparts. This configuration may enable nodes hidden behind firewalls as well as internal cluster nodes to participate in distributed computations through the child kernel. This approach may also be taken on large multiprocessor SMP machines; it is not limited to cluster use. It may also be used to enhance scalability by distributing the task management load to multiple servers and making it easier to manage a large number of nodes.

6.4.2. Kernel redundancy. To avoid single points of failure, kernel modules can be mirrored across several failover servers. Nodes are made aware of alternate kernel locations so that if one server fails, another can assume task management and output collection functions.

6.4.3. FastCGI. Performance of kernel modules may be significantly improved by migrating them to the FastCGI extension. This will be of particular benefit to the Collector module. Under standard CGI, a new Collector process is created for each subtask completion request, hence repeatedly incurring process creation and initialization overhead. Under FastCGI, one or just a few persistent Collector processes would handle all requests.

6.4.4. Volunteer computing. We hope to involve the general public in our distributed protein folding experiments, by asking them to register their Web server nodes and volunteer idle CPU cycles. The node access schedule gives full control to node administrators and owners as to when and how long their nodes can be used. Additional security features, such as Web server level authentication, will be required in the kernel modules to ensure safe access of volunteer nodes.

7. Summary

We presented MoBiDiCK as a tool for distributed computing based on well established protocols, HTTP and CGI. The relatively high communication latencies of these protocols over the Internet render the system to be most suitable for data-parallel tasks that are CPU-intensive and that require minimal inter-node communication. Node accessibility is controlled by a real-time access schedule. A central relational database is used to hold static information about nodes and tasks, as well as dynamic data relating to node availability and task progress. The database also stores useful task timing information that can be used to build performance reports and histories of past computations, which in turn can serve to predict and optimize the performance of future instances. We reported the development of two TaskApps, RAMAPLOT and FOLDTRAJ. The former yielded encouraging speedup and efficiency results using a local cluster of Web server nodes, despite I/O contention on the kernel node during collection of output. This problem is, however, not unique to our system. FOLDTRAJ is an application we regularly employ in the MoBiDiCK environment in order to perform our computational protein folding experiments. We are continuing with our development of MoBiDiCK and look forward to carry on with our future directions.

8. References

[1] IBM, Corp. "Blue Gene" to Tackle Protein Folding Grand Challenge. http://www.research.ibm.com/news/detail/-bluegene.html

[2] Feldman, H.J., Hogue, C.W.V. A Fast Method to Sample Real Protein Conformational Space. *Proteins: Structure, Function, and Genetics*, 2000. In Press.

[3] Ostell, J and et al. NCBI Software Development ToolKit. (6.0). ftp://ftp.ncbi.nlm.nih.gov/toolbox/.

[4] Hogue, C.W.V. Cn3D: A New Generation of Three Dimensional Molecular Structure Viewer. *Trends.Biochem.Sci.*, 22:314-316, 1997.

[5] Distributed.net website, http://www.distributed.net/.

[6] Search for Extra-Terrestrial Intelligence; SETI@home website, http://www.seti.org/science/setiathome.html.

[7] Fox, G.and Furmanski. W. Computing on the Web: New Approaches to Parallel Processing; Petaop and Exaop Performance in the Year 2007. *IEEE Internet Computing*, 1(2), March-April 1997.

[8] Foster, I., Kesselman, C. Globus Project, 2000. http://www.globus.org.

[9] Fielding, R., Gettys, J., Mogul, J., Frysyk, H., Masinter, L., Leach, P. and T. Berners-Lee. Hypertext Transfer Protocol - HTTP/1.1. RFC 2616, Network Working Group, World Wide Web Consortium, June 1999.

[10] Coar, K., Robinson, D. The WWW Common Gateway Interface Version 1.1. NCSA Internet-draft in progress. Linked from http://Web.Golux.Com/coar/cgi/. June 1999.

[11] Open Market, Inc. FastCGI: A High-Performance Web Server Interface. White paper. Linked from http://www.fastcgi.com.

[12] Sequiter Software Inc. CodeBase (Version 6): Database Management for Programmers, Getting Started, p 1-7.

[13] Geist, G., Kohl, J., Papadopoulos, P., Scott, S. Beyond PVM 3.4: What We've Learned, What's Next, and Why. *J Proceedings of EuroPVM-MPI 97*, November 1997.

[14] Pacheco, P. Parallel Programming with MPI. Morgan Kaufmann Publishers Inc., San Francisco. pp 245-269, 1997.

Moyez Dharsee is a Senior Bioinformatics Developer with the Samuel Lunenfeld Research Institute, where he currently manages the MoBiDiCK project. He obtained his B.Sc. in Computer Science from the University of Toronto in 1998.

Christopher W. V. Hogue is a Principal Investigator at the Samuel Lunenfeld Research Institute (SLRI), Toronto, Canada, where he is cross-appointed with the Department of Biochemistry at the University of Toronto. He obtained his B.Sc. in Biochemistry in 1990 from the University of Windsor and his Ph.D. in Biochemistry at the University of Ottawa while working in the laboratories of the National Research Council in the area of time-resolved protein fluorescence spectroscopy. Prior to joining SLRI, Chris worked as a postdoctoral researcher at the U.S. National Institutes of Health in the National Center for Biotechnology Information (NCBI) where the GenBank sequence database is maintained and distributed. At NCBI, Chris helped develop a new three-dimensional structure database called MMDB and wrote Cn3D, a widely used program for visualizing biological molecules.

RsdEditor: A Graphical User Interface for Specifying Metacomputer Components

R. Baraglia, D. Laforenza
CNUCE - Instituto del Consiglio Nazionale delle Ricerche
Via S. Maria, 36 - I-56100 Pisa, Italy
e-mail: (Ranieri.Baraglia, Domenico.Laforenza)@cnuce.cnr.it

A. Keller
Paderborn Center for Parallel Computing
Fürstenallee 11, 33102 Paderborn, Germany
e-mail: kel@upb.de

A. Reinefeld
Konrad-Zuse-Zentrum für Informationstechnik Berlin
Takustr. 7, D-14195 Berlin-Dahlem, Germany
e-mail: ar@zib.de

Abstract

*RsdEditor is a graphical user interface which produces specifications of computational resources. It is used in the RSD (*Resource and Service Description*) environment for specifying, registering, requesting and accessing resources and services in a metacomputer.*

RsdEditor was designed to be used by the administrators and users of metacomputing environments. At the administrator level, the GUI is used to describe the available computing and networking components of a metacomputer. At the user level RsdEditor can be used to specify which characteristics of the computational resources are needed to execute a meta-application.

This paper is organized as follows: Section 1 introduces the RsdEditor. Section 2 briefly describes the RSD environment, and Section 3 highlights various features and implementation issues of the RsdEditor.

Keywords: Metacomputing, Resource Management, Resource and Service Description.

1. Introduction

RsdEditor is a graphical user interface for specifying metacomputer resources. It was developed by CNUCE-CNR in cooperation with PC^2 Paderborn as part of the Metacomputer On-Line (MOL) project [1]. MOL exploits the Computing Center Software (CCS) [2, 3] in order to manage the resources of a computing center. Within CCS, the resource and service description language *RSD* [4] is used to describe the metacomputer resources managed by CCS.

RsdEditor provides a user-friendly visual support to automatically generate ASCII files. These are structured according to the RSD language which describes the specified resources by using the graphical features of the interface. *RsdEditor* was designed to be used by administrators and users of a metacomputer.

As shown in Figure 1, system administrators use the *RsdEditor* to specify particular characteristics of the metacomputer's resources (computational nodes, networks, software services, etc.). Specifically the administrator can assign attributes such as type and number of processors, memory size, software environments, architectural classes, latency, or bandwidth to the available computational resources. Likewise, users can specify which characteristics of the computational resources are needed to execute their meta-application. This phase is not intended to select specific resources but to indicate the general attributes belonging to a class of resources. The specifications made by the administrator and the user can be used to generate two graphs representing the metacomputer's configuration and the user's requirements, respectively. The allocation of the resources needed to execute the meta-application on the

0-7695-0556-2/00 $10.00 © 2000 IEEE

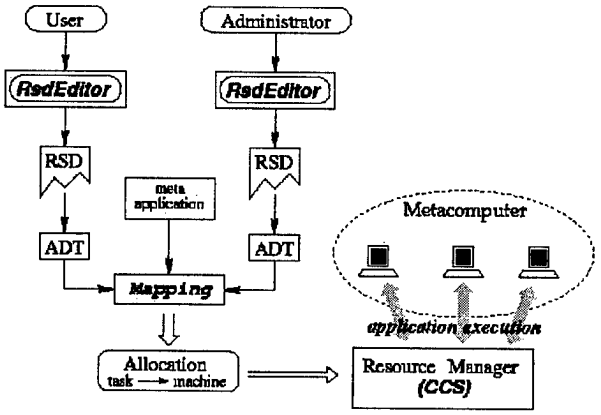


Figure 1. The use of the specifications generated by RsdEditor

metacomputer is a question of mapping the user graph onto the metacomputer graph.

Several mapping algorithms [5, 6, 7, 8] have been put forward to solve this problem.

The RSD resource specification file, automatically generated by *RsdEditor*, is analyzed by a parser to obtain *Abstract Data Type (ADT)* objects [4]. ADT objects can only be accessed through the *RsdAPI*which provides an abstract interface to the RSD data structures. As shown in Figure 1, the mapping of the task graph onto the available resources as generated by the mapping algorithm is used by CCS to start and control the execution of a meta-application.

2. RSD Environment

RsdEditor is part of the RSD environment [4] that provides services and tools for specifying, registering, requesting and accessing computer resources in heterogeneous computing environments. RSD is comprised of three major components:

- a compiler system that transforms resource descriptions into ADTs (described in [4]),
- an ADT object library with API (outlined in [4]),
- a graphical user interface and editor *RsdEditor* (described in this paper).

In RSD, resources and services are represented by hierarchical graphs with attributed nodes and edges describing static and dynamic properties such as communication bandwidth, message latency, or CPU load. Tools exist for end-users as well as for system administrators (Figure 2).

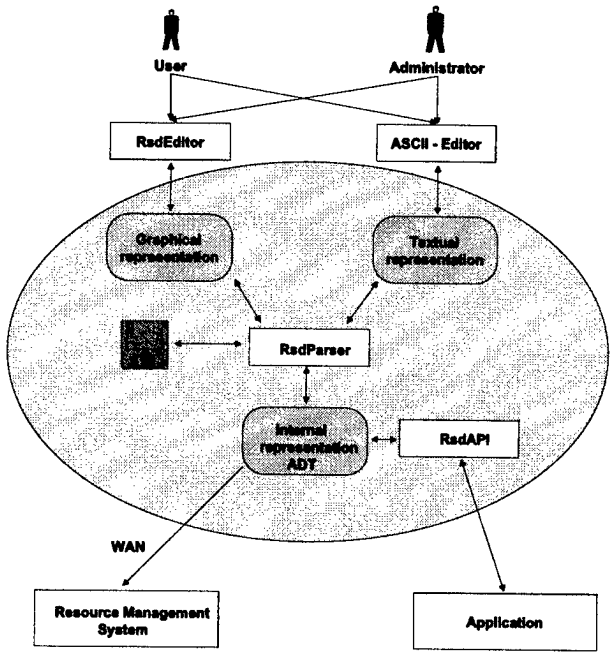


Figure 2. RSD environment

The output of the graphical *RsdEditor* is sent through the RsdParser which generates abstract data objects that can be stored or submitted for further processing by (remote) resource management systems. In addition, ASCII RSD files can also be translated by the parser into abstract objects. By bundling objects with the corresponding methods the data can be interpreted and manipulated on other machines. Internal RSD objects can only be accessed through the *RsdAPI*which provides the data structures with an abstract interface. For later modification, the data structures are re-translated into their original form with the graphical and textual components. This is possible because the internal data representation also contains a description of the component's graphical layout.

2.1 Textual versus Graphical Interface

RsdEditor has been designed to provide a user-friendly alternative to the textual RSD representation [4]. From a theoretical point of view, both representations are equivalent. In fact, the *RsdEditor* output is parsed and compiled by the same RSD compiler system used for the language representation. Hence, *RsdEditor* is no more expressive than the language. On the other hand, it is easy to prove that the language is no more expressive than *RsdEditor* by looking at the more advanced features of the language, such as dynamic attributes and macros.

Dynamic attributes [4] provide a means of handling dy-

namic data that are obtained at runtime. For example, when running a WAN distributed application, the optimal (re-)mapping of the processes may depend on the current network performance. For this purpose, dynamic attributes provide up-to-date information on the current network status. When a dynamic attribute (keyword DYNAMIC) is parsed, the compiler system generates a corresponding object with appropriate access methods. These are then used by the dynamical data manager at runtime to provide up-to-date data in a synchronous or asynchronous way. Dynamic attributes can be specified in the same way in the *RsdEditor* and in the textual representation.

One feature not included in the *RsdEditor* are macros. In the textual representation, they provide a shortcut for tedious repetitive declarations.

In *RsdEditor*, this is done by copying the corresponding edges or (hyper-)nodes.

2.2 RSD Tools in CCS

Maximizing the system utilization, and maintaining a high degree of system independence for improved portability and easier adaptation to new systems have been the two main goals of the CCS [3] project. CCS tackles these two conflicting goals by splitting the scheduling process into two parts. Figure 3 depicts the RSD flow in CCS.

The hardware independent part is located in the *Queue Manager (QM)*. It has no information on the mapping constraints such as the minimum cluster size, or the location of I/O-nodes. The hardware dependent task is performed by the *Machine Manager (MM)*. It verifies whether a schedule computed by the QM can be mapped onto the hardware at the specified time.

The RSD tools are used in the CCS management system for describing system resources and user requests. At boot time, all CCS components read the RSD specification created by the administrator and extract the relevant information (by use of the *RsdAPI*). For example, the MM reads the machine topology and attributes, whereas the QM only extracts information such as the number of PEs or the available operating system(s).

The UI (User Interface) generates an RSD description from the user's parameters (or from a given RSD description) and sends the description to the *Access Manager (AM)*.

The AM, which is responsible for authentication, authorization, and accounting, checks whether the request matches the administrator's given limits and forwards it to the QM.

The QM extracts the information, computes a schedule and sends it to the MM. MM verifies this schedule by mapping the user given RSD description against the static (e.g. topology) and dynamic (e.g. PE availability) information on the system resources. If there is no mapping possible,

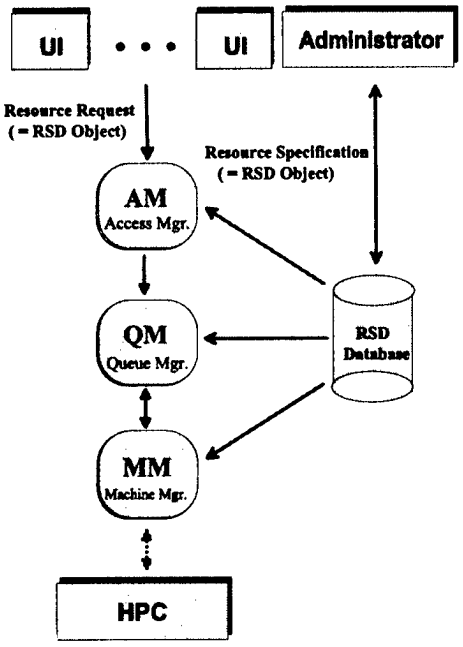


Figure 3. RSD flow in the CCS system.

the MM returns an alternative schedule. QM either accepts this schedule or uses it to compute a new one.

Although all CCS components are based on RSD, in the past we disguised the complexity of the RSD language by an easy-to-use command line interface. There was no need for a versatile resource description facility because most of the systems were homogeneous, their topologies simple and regular, and nearly all applications ran on only one system.

With the trend of metacomputing (now often called grid based computing), resource description has become more and more important, because now the system (instead of the user) decides which of the available resources are used. Hence, the users need a convenient tool to specify their requests, and the applications need an API to negotiate their requirements with the resource management system.

The RSD systems fulfill both requirements by providing the *RsdEditor* and the *RsdAPI*, respectively.

2.3 Related Work

Like RSD, the Globus [9] resource specification language RSL [10], its corresponding metacomputer directory service MDS [10] and the underlying LDAP services have also been devised for specifying distributed resources.

However, the Globus approach is somewhat asymmetric: it uses RSL for specifying resource requests and LDAP (based on X.500) for specifying resource offers.

RSD, in contrast, uses the same representation for both

purposes, thereby allowing us to use common graph matching mechanisms for brokerage.

The brokerage aspect is emphasized in the ClassAds approach used in the Condor [11] framework for matching resource offers with requests. Compared to RSD, the ClassAds project focuses on protocols for advertising resources and on the matchmaking process, rather than on the specification aspect. As a result, the expressions used in the ClassAds seem to be less powerful than our hierarchical, graph-based RSD expressions.

The *Resource Cataloging and Distribution System RCDS* [12] developed at the University of Tennessee is another interesting approach. RCDS supports flexible, scalable, and secure access to various types of data (e.g. files) on WAN connected computers.

Resources are named by URNs (Uniform Resource Names) which provides stable names for resources which may change in content or location over time. This is achieved by putting *resolution servers* between the location dependent URLs and the end user.

A middle software layer guarantees integrity and persistence of resources in an environment of dynamically changing information.

3. RsdEditor: Features and Implementation

Figure 4 shows the *RsdEditor* starting window. Currently, it is possible to choose between two different languages, English and Italian. Moreover, it can operate in Administrator or User mode.

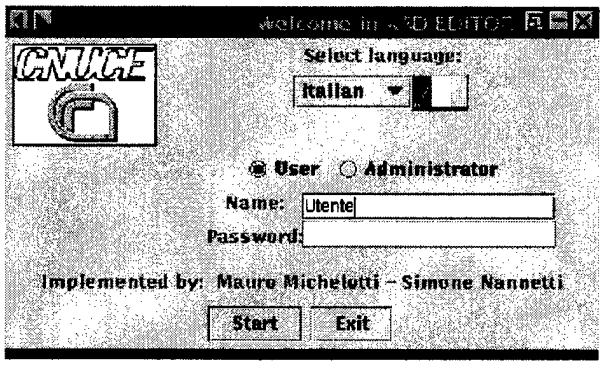


Figure 4. RsdEditor Start Window

Figure 5 depicts an example of a working session. A status bar is shown at the bottom of the window in which error and information messages are displayed. The central part of the window, called the workspace, is the working area for the graphical resource specification.

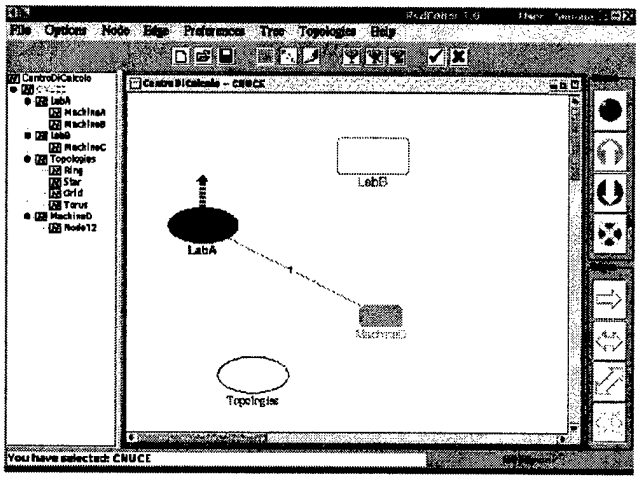


Figure 5. Example of a work session

The menu bar contains the following items: File, Options, Node, Edge, Preferences, Tree, Topologies, and Help.

File enables the creation/editing of a resource specification file.

Options displays the current RSD file, refreshes the workspace, etc..

Node allows the creation, editing, or deletion of a node or hypernode (a node containing other nodes in a recursive way). In the RSD syntax a node represents a computational resource characterized by graphical and RSD attributes.

Figures 6 and 7 show the definition of the graphical characteristics and the assignment of RSD attributes to a node, respectively.

The RSD syntax requires each node to have at least one port (a node's interface toward other nodes) in order to link it to another node by using an edge. RSD attributes can be assigned to a port (see Figure 8).

RSD allows nodes to be defined recursively and to create hypernodes. A hypernode contains the specifications of other resources such as nodes, physical and virtual edges. On the left hand side of Figure 5 the hypernodes and nodes are indicated by the letters H and N, respectively.

Edge enables the creation, editing, or deletion of a physical or virtual edge. A physical edge represents a link between two nodes. The RSD syntax permits uni- or bi-directional physical edges. By using the windows shown in Figures 9 and 10 it is possible to select a port connected by an edge (*binding*).

The edge binding is an RSD syntax constraint. Therefore, the ports must be defined before the binding. The notion of a virtual or vertical edge is used to provide a link between different levels of a hierarchy in the specification

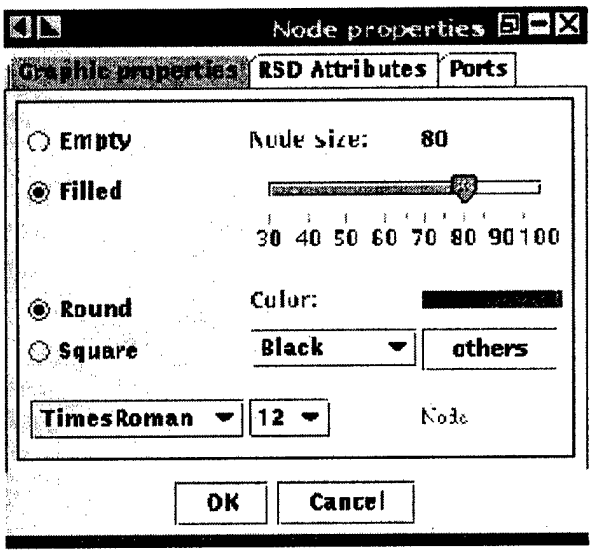


Figure 6. Specification of the graphical properties of a node

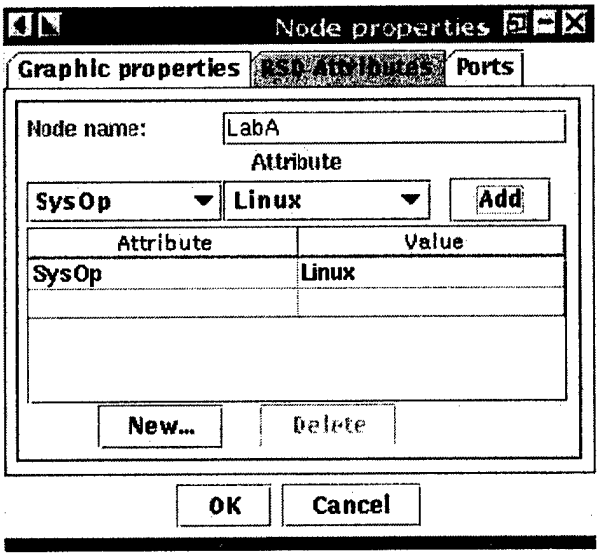


Figure 7. Specification of the RSD attributes of a node

graph. A virtual edge is defined using the windows shown in Figures 11 and 12, and it is represented by an arrow (see Figure 5).

Preferences permits the definition of various graphical features, such as size, shape, color, etc.

Tree enables the managing of the resource tree. This tree is shown in a synoptical way; it is thus useful for the user who can see and navigate each level of the resource specification tree. On the left hand side of Figure 5 an example of a resource specification tree is shown.

Topologies allows the creation, editing, or deletion of nodes representing some of the most common homogeneous interconnection topologies (Ring, Grid, Star, Torus). Figure 13 shows an example specifying a Grid composed of 4 × 8 nodes. This prevents the user from having to manually specify 32 nodes and 52 edges.

Help accesses the *on-line* manual.

RsdEditor saves the current resource specifications by creating two files: filename.`rsd` and filename.`gui` containing the resource specifications in RSD syntax and the formal descriptions of graphical objects, respectively. filename is the name specified by the user when the resource specification is created.

The graphical interface provides the option of importing and exporting resource specifications, some of which may have been previously recorded, in order to reuse them. As shown in Figure 14 *RsdEditor* allows the RSD code, produced during a specification phase, to be displayed.

For portability reasons, *RsdEditor* was implemented in Java [13, 14] and it has been tested successfully on Microsoft Windows (98, NT), RedHat Linux and Sun Solaris. The modular structure adopted to implement *RsdEditor* facilitates its maintenance and extension.

A more detailed description of the *RsdEditor* functions can be found in [15, 16].

4. Example of *RsdEditor* Utilization

As an example of how *RsdEditor* can be used, we show how to describe the computing resources of a computing center. Figure 15 shows the structure of the Paderborn Center for Parallel Computing. There are four parallel computers (CC_48, GCel_System, SCI_64 and GCPP_64) connected by Ethernet, and a computer (Uranus) acting as gateway towards the outside world.

As sketched in Figure 16, each parallel computer (represented by a torus icon) and the gateway are connected to a central node representing the Ethernet hub. Those nodes are included in a hypernode denoted as `Paderborn_Park`. In order to specify the attributes of each computer the windows shown in Figures 13 need to be used.

The complete resource description automatically produced by *RsdEditor* is shown in the following. It is worth highlighting the usefulness of *RsdEditor* by looking at the pages containing the RSD code. In fact, the resource specifications made by the RSD language is a relatively long and, potentially error prone, task.

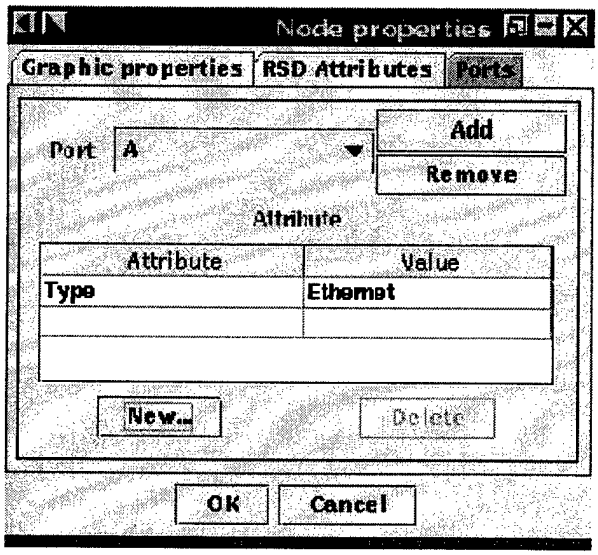


Figure 8. Specification of port attributes

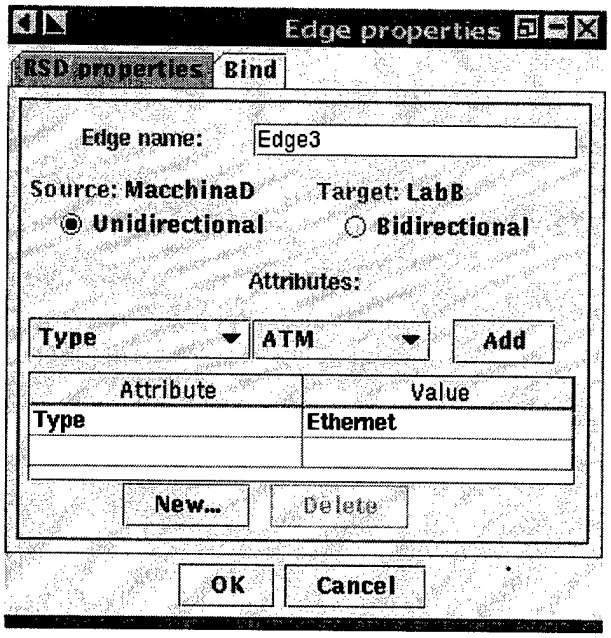


Figure 9. Specification of the RSD attributes of a physical edge

```
=======================================================
=    Resource Description Produced by RsdEditor       =
=======================================================
ROOTNODE Paderborn_Park
{
NODE Ethernet
{
   PORT Eth1 { Type = Ethernet; };
   PORT Eth2 { Type = Ethernet; };
   PORT Eth3 { Type = Ethernet; };
   PORT Eth4 { Type = Ethernet; };
   PORT Eth5 { Type = Ethernet; };
   PORT Eth6 { Type = Ethernet; };

   Bandwidth = 100;
};
NODE Uranus
{
   PORT Ethernet;
   PORT ATM;
};
NODE CC_48
{
   CONST n = 2;
   CONST m = 24;

   FOR i=1 TO n DO
      FOR i=1 TO n DO
         NODE Torus_$i_$j
         {
            PORT cc48;
            IF ((i=1) && (j=1)) THEN
               PORT CC_48-Esterna;
            FI

            CPU = PowerPC;
            Memory = 64MByte;
            PeakPerformance = 12,76GFlops;
            SysOp = AIX4.1;
         };
      OD
   OD

   FOR i=1 TO n-1 DO
      FOR j=1 TO m DO
            EDGE Edge_$i_$j_to_$i_$((j+1) MOD m)
            {
               NODE Torus_$i_$j PORT cc48 <=>
                    NODE Torus_$i_$((j+1) MOD m) PORT cc48;

            };
      OD
   OD

   FOR j=1 TO m-1 DO
      FOR i=1 TO n DO
            EDGE Edge_$i_$j_to_$((i+1) MOD n)_$j
            {
               NODE Torus_$i_$j PORT cc48 <=>
                    NODE Torus_$((i+1) MOD n)_$j PORT cc48;

            };
      OD
   OD

   ASSIGN Torus_1_1 PORT CC_48-Esterna;
};
NODE GCel_System
{
   CONST n = 32;
   CONST m = 32;

   FOR i=1 TO n DO
      FOR i=1 TO n DO
         NODE Torus_$i_$j
         {
            PORT TransputerLink;
            IF ((i=1) && (j=1)) THEN
               PORT GCel-Esterna;
            FI

            CPU = T805;
            Memory = 4MByte;
```

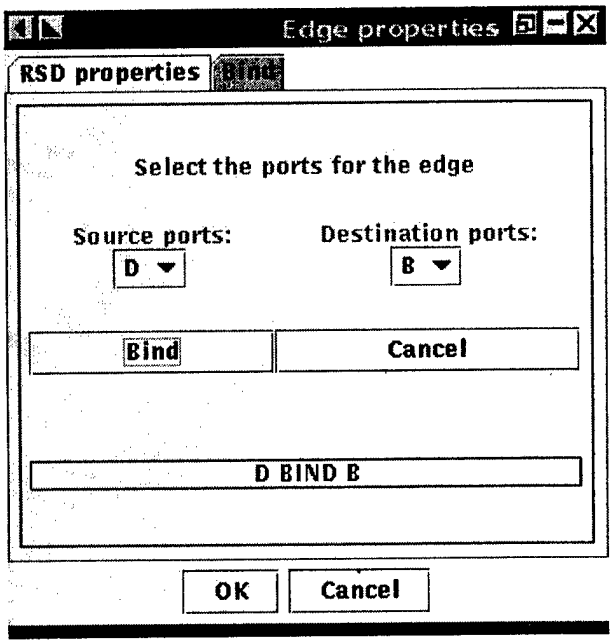


Figure 10. Specification of the RSD attributes of a physical edge

```
            PeakPerformance = 4.4GFlops;
         };
      OD
   OD

   FOR i=1 TO n-1 DO
      FOR j=1 TO m DO
            EDGE Edge_$i_$j_to_$i_$((j+1) MOD m)
            {
               NODE Torus_$i_$j PORT TransputerLink <=>
                    NODE Torus_$i_$((j+1) MOD m)
                         PORT TransputerLink;

            };
      OD
   OD

   FOR j=1 TO m-1 DO
      FOR i=1 TO n DO
            EDGE Edge_$i_$j_to_$((i+1) MOD n)_$j
            {
               NODE Torus_$i_$j PORT TransputerLink <=>
                    NODE Torus_$((i+1) MOD n)_$j
                         PORT TransputerLink;

            };
      OD
   OD

   ASSIGN Torus_1_1 PORT GCel-Esterna;
};
NODE GCPP_64
{
   CONST n = 4;
   CONST m = 8;

   FOR i=1 TO n DO
      FOR i=1 TO n DO
         NODE Torus_$i_$j
```

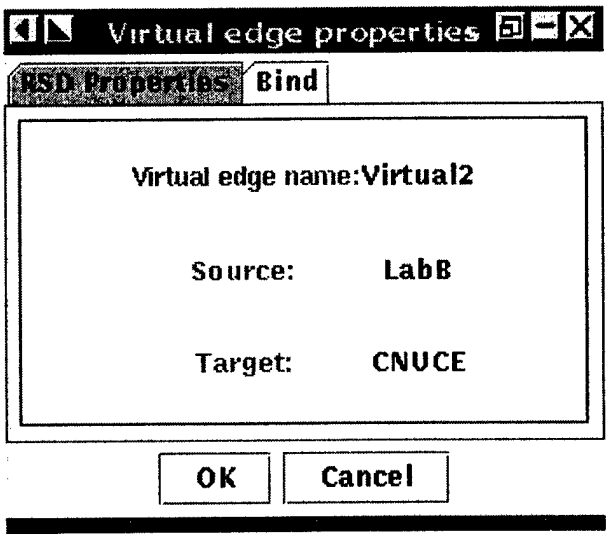


Figure 11. Specification of the RSD attributes of a virtual edge

```
         {
            PORT gcpp;
            IF ((i=1) && (j=1)) THEN
               PORT GCPP-Esterna;
            FI

            CPU = PowerPC601;
            CPUNumber = 2;
            Memory = 32MByte;
            PeakPerformance = 5.12GFlops;
         };
      OD
   OD

   FOR i=1 TO n-1 DO
      FOR j=1 TO m DO
            EDGE Edge_$i_$j_to_$i_$((j+1) MOD m)
            {
               NODE Torus_$i_$j PORT gcpp <=>
                    NODE Torus_$i_$((j+1) MOD m) PORT gcpp;

            };
      OD
   OD

   FOR j=1 TO m-1 DO
      FOR i=1 TO n DO
            EDGE Edge_$i_$j_to_$((i+1) MOD n)_$j
            {
               NODE Torus_$i_$j PORT gcpp <=>
                    NODE Torus_$((i+1) MOD n)_$j PORT gcpp;

            };
      OD
   OD

   ASSIGN Torus_1_1 PORT GCPP-Esterna;
};
NODE SCI_64
{
   CONST n = 4;
   CONST m = 8;

   FOR i=1 TO n DO
```

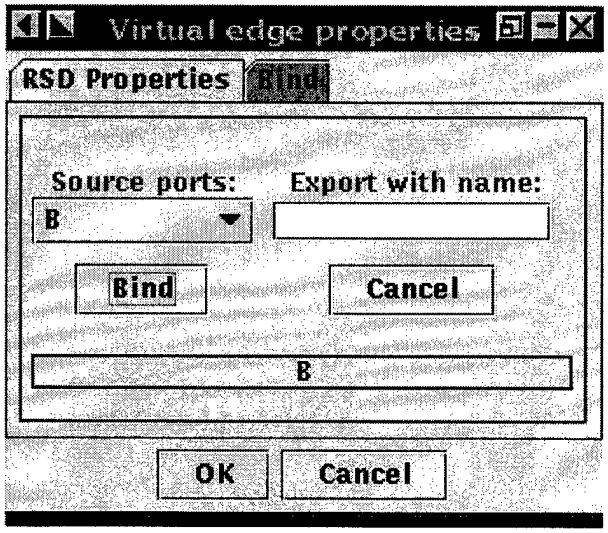


Figure 12. Specification of the RSD attributes of a virtual edge

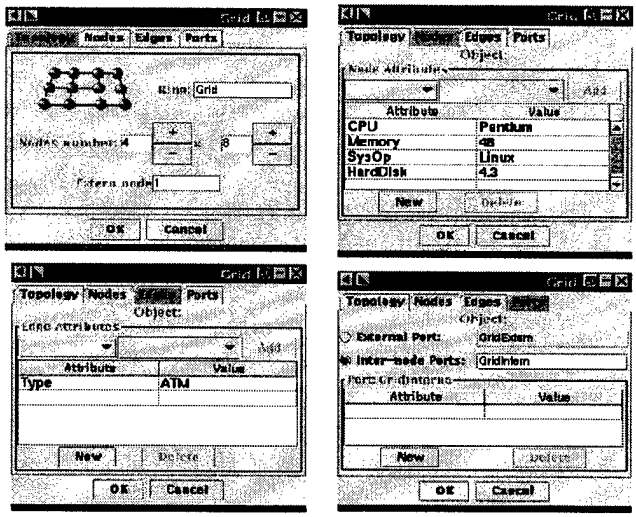


Figure 13. Supported topologies

```
    FOR i=1 TO n DO
       NODE Torus_$i_$j
       {
          PORT sci64;
          IF ((i=1) && (j=1)) THEN
             PORT SCI_64-Esterna;
          FI

          CPU = PentiumII;
          CPUNumber = 2;
          Memory = 256MByte;
          PeakPerformance = 19.2GFlops;
       };
    OD
 OD

 FOR i=1 TO n-1 DO
    FOR j=1 TO m DO
          EDGE Edge_$i_$j_to_$i_$((j+1) MOD m)
          {
             NODE Torus_$i_$j PORT sci64 <=>
                  NODE Torus_$i_$((j+1) MOD m) PORT sci64;

             Bandwidth = 500MByte/s;
          };
    OD
 OD

 FOR j=1 TO m-1 DO
    FOR i=1 TO n DO
          EDGE Edge_$i_$j_to_$((i+1) MOD n)_$j
          {
             NODE Torus_$i_$j PORT sci64 <=>
                  NODE Torus_$((i+1) MOD n)_$j PORT sci64;

             Bandwidth = 500MByte/s;
          };
    OD
 OD

 ASSIGN Torus_1_1 PORT SCI_64-Esterna;
};

EDGE Edge0
{
   NODE Uranus PORT Ethernet <=> NODE Ether-
net PORT Eth1;
   Type = Ethernet;
};

EDGE Edge1
{
   NODE Ethernet PORT Eth2 <=>
        NODE GCel_System PORT GCel-Esterna;
   Type = Ethernet;
};

EDGE Edge2
{
   NODE Ethernet PORT Eth3 <=> NODE SCI_64 PORT SCI_64-
Esterna;
   Type = Ethernet;
};

EDGE Edge3
{
   NODE Ethernet PORT Eth4 <=> NODE GCPP_64 PORT GCPP-
Esterna;
   Type = Ethernet;
};

EDGE Edge4
{
   NODE Ethernet PORT Eth5 <=> NODE CC_48 PORT CC_48-
Esterna;
   Type = Ethernet;
};

ASSIGN NODE Uranus PORT ATM;

}
```

5. Summary

In this paper we have presented *RsdEditor*, a graphical editor for specifying computational resources and services in distributed environments. Computing components (com-

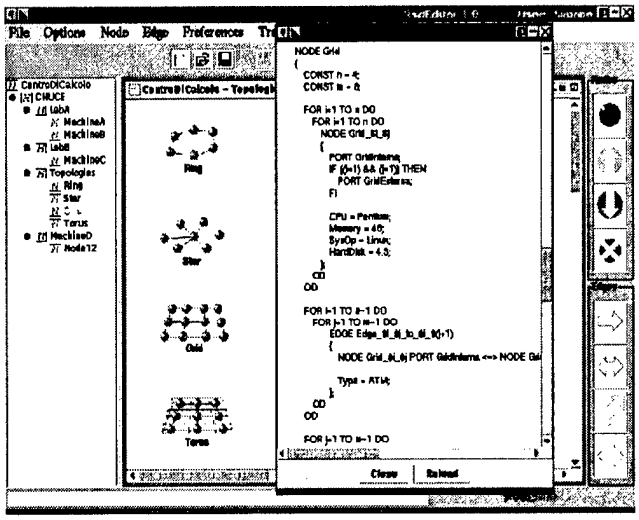


Figure 14. RSD code generation

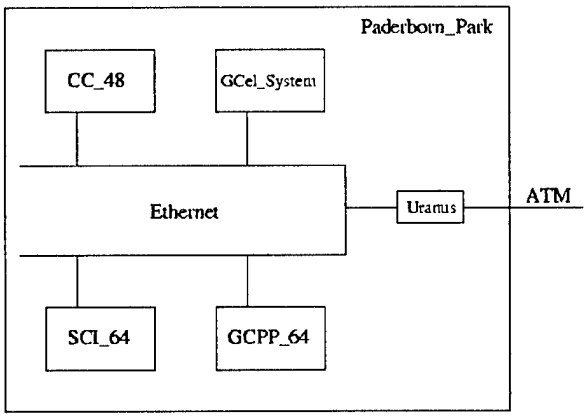


Figure 15. Example: structure of a computing center (Paderborn Park).

puters, processors) are represented by nodes and their interconnects (WAN, LAN, or internal computer links) by edges. Both may be attributed.

Compared to other approaches RSD is used by both, users and administrators, thereby allowing the use of simple graph matching algorithms for mapping resource requests onto resource offers.

RsdEditor currently generates (via the RSD language) C++ objects. For improved portability, we plan to adapt *RsdEditor* to generate XML code. In addition, work is under way to implement resource brokers with different strategies on top of the RSD framework.

Acknowledgements

The authors would like to thank the Master Thesis students who worked with us during the design and the development phases of RsdEditor. In particular, thanks are due to Simone Nannetti and Mauro Micheletti. Moreover, we would like to thank Giancarlo Bartoli, technical manager at the CNUCE Parallel Computing Laboratory.

References

[1] A. Reinefeld, R. Baraglia, T. Decker, J. Gehring, D. Laforenza, F. Ramme, T. Römke, J. Simon. *The MOL Project: An Open Extensible Metacomputer.* Proc. Heterogenous Computing Workshop HCW'97, IEEE Computer Society Press, pp. 17-31.

[2] CCS: Computing Center Software. http://www.uni-paderborn.de/pc2/projects/ccs.

[3] A. Keller, A. Reinefeld. *CCS Resource Management in Networked HPC Systems.* Proc. Heterogeneous Computing Workshop HCW'98 at IPPS, Orlando, pp. 44-56.

[4] M. Brune, A. Reinefeld, J. Varnholt. *A Resource Description Environment for Distributed Computing Systems.* Proc. 8th Intern. Sympos. High-Performance Distributed Computing HPDC'99, Redondo Beach, 1999, 279–286.

[5] R.R. Freund. *Optimal selection theory for superconcurrency.* In Proceedings of Supercomputing 89, p.699-703, 1989.

[6] T.D. Braun, H.J. Siegel, N. Beck, L.L. Boloni, M. Maheswaran, A.I. Reuther, J.P. Robertso, M.D. Theys, B. Yao, D. Hensgen, R.F. Freund. *A Compasrison Study of Static Mapping Heuristics for a Class of Meta-task on Heterogeneous Computing Systems* 8th IEEE Heterogeneous Computing Workshop, IEEE Computer Society Press, April 1999.

[7] M.M. Eshaghian, Y.C. Wu. *Heterogeneous Task Graphs onto Heterogeneous System Graphs* Proceedings of Sixth Heterogeneous Computing Workshop, IEEE Computer Society Press, 1:147-160, April 1997.

[8] R. Baraglia, D. Laforenza, A. Panciatici, F. Ravaglia *A Suboptimal Mapping of Parallel Applications on Metacomputers.* Proceedings of IASTED International Conference, Parallel and Distributed Comput-

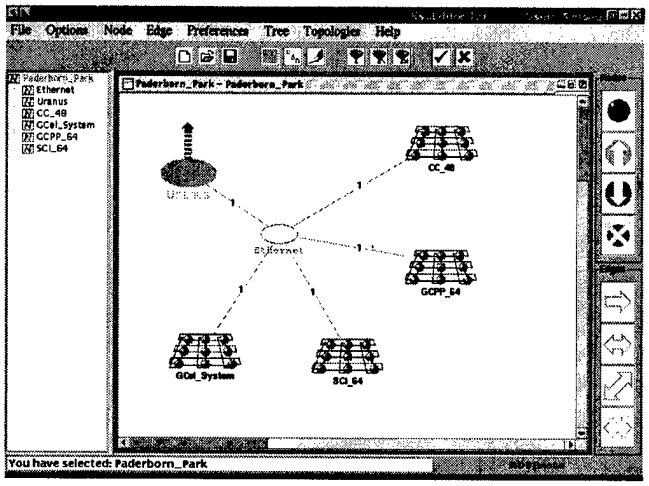


Figure 16. Resources description of the Paderborn Park Hypernode.

ing and Systems, Cambridge Massachusetts, USA, November 3-6, 1999.

[9] I. Foster, C. Kesselman. *The Globus Project: A Status Report.* Proc. IPPS/SPDP '98, Heterogeneous Computing Workshop, Orlando, pp. 4-18, 1998.

[10] K. Czajkowski, I. Foster, N. Karonis, C. Kesselman, S. Martin, W. Smith, S. Tuecke. *A Resource Management Architecture for Metacomputing Systems.* Proc. IPPS/SPDP '98, Workshop on Job Scheduling Strategies for Parallel Processing, Orlando, 1998.

[11] M. Livny, R. Raman. *High-Throughput Resource Management.* In The Grid: Blueprint for a new computing infrastructure, ed. I. Foster and C. Kesselman, San Francisco, Morgan Kaufmann, 1998.

[12] K. Moore, S. Browne, J. Cox, and J. Gettler. *Resource Cataloging and Distributed Systems.* Univ. of Tennessee, Techn. Report, January 1997.

[13] The $Java^{TM}$ Development Kit (JDK^{TM}). http://java.sun.com/products/jdk/.

[14] Jason J. Manger. *Essential Java*.* McGraw-Hill, 1998.

[15] R. Baraglia, D. Laforenza, M. Michelotti, S. Nannetti. *RsdEditor: un'interfaccia grafica per la specifica delle risorse in un metacomputer. Descrizione e guida per l'utilizzo.* CNUCE-CNR Technical Report CNUCE-B4-1999-013 (in Italian. Translation into English is in progress), 1999.

[16] M. Michelotti, S. Nannetti. *Un'interfaccia grafica per la definizione e descrizione delle risorse e servizi di un metacomputer.* Tesi di Laurea (in Italian), Università degli Studi di Pisa, Facoltà di Scienze MM. FF. NN., Febbraio 1999.

Biographies

Ranieri Baraglia graduated in Computer Sciences in 1982 at the University of Pisa, Italy. He is currently a researcher at CNUCE, an Institute of the Italian National Research Council (CNR) in Pisa, where he is a member of the Parallel Processing Group. He was a contract professor at the Department of Mathematics at Perugia University from 1991 to 1996 where he taught operating systems and parallel architectures. Ranieri Baraglia has been involved in both Italian and European projects. His main research interests are in the fields of heterogeneous computing and parallel algorithms and applications. He is a member of IEEE and ACM.

Axel Keller received his diploma in computer science from the University of Paderborn in 1993. He is currently working as a technical associate for the Paderborn Center for Parallel Computing.

Domenico Laforenza is currently a researcher at CNUCE, Institute of the Italian National Research Council (CNR) in Pisa where he is responsible of the Advanced Computing Department. He also has a joint appointment at the Department of Computer Science of University of Pisa as professor of Parallel Applications. Dr. Laforenza has written numerous technical and scientific papers and has served as a program committee member and organiser of many national and international workshops and conferences related to High Performance Computing. He is a member of AICA and IEEE.

Alexander Reinefeld is the head of the Computer Science department at Konrad-Zuse Supercomputing Center (ZIB) in Berlin, Germany, and a professor for Parallel and Distributed Systems at the Humboldt University Berlin. From 1992 to 1998, he was managing director of the Paderborn Center for Parallel Computing, Germany. His research interests focus on parallel and distibuted high-performance computing, parallel programming models, performance benchmarking, and applications in discrete optimization. He is a member of IEEE, ACM and GI.

SESSION 5-B
SCHEDULING II

Chair: I. Ahmad, *Hong Kong University of Science and Technology, China*

Heuristics for Scheduling Parameter Sweep Applications in Grid Environments

Henri Casanova* Arnaud Legrand† Dmitrii Zagorodnov* Francine Berman*

* Computer Science and Engineering Department
University of California, San Diego, USA
[casanova,dzagorod,berman]@cs.ucsd.edu

† Laboratoire de l'Informatique et du Parallélisme
École Normale Supérieure de Lyon, France
alegrand@ens-lyon.fr

Abstract

The Computational Grid provides a promising platform for the efficient execution of parameter sweep applications *over very large parameter spaces. Scheduling such applications is challenging because target resources are heterogeneous, because their load and availability varies dynamically, and because independent tasks may share common data files. In this paper, we propose an adaptive scheduling algorithm for parameter sweep applications on the Grid. We modify standard heuristics for task/host assignment in perfectly predictable environments (*Max-min, Min-min, Sufferage*), and we propose an extension of Sufferage called* XSufferage. *Using simulation, we demonstrate that XSufferage can take advantage of file sharing to achieve better performance than the other heuristics. We also study the impact of inaccurate performance prediction on scheduling. Our study shows that: (i) different heuristics behave differently when predictions are inaccurate; (ii) increased adaptivity leads to better performance.*

1. Introduction

Fast networks make it possible to aggregate CPU, network and storage resources into *Computational Grids* [8]. Such environments can be used effectively to support very large-scale runs of distributed applications. An ideal class of applications for the Grid is the class of *parameter sweep applications*, applications structured as a set of multiple "experiments", each of which is executed with a distinct set of parameters.

Executing a parameter sweep on the Grid involves the assignment of tasks to resources. Although the experiments (or *tasks*) of a parameter sweep application are independent, a number of issues make scheduling such applications challenging. First, resources in the Grid are typically *shared* so that the contention created by multiple applications creates dynamically fluctuating delays and qualities of service. In addition, Grid resources are *heterogeneous* and may not perform similarly for the same application. Moreover, although parameter sweep tasks are independent, they may share common input files which reside at remote locations, hence the performance-efficient assignment and scheduling of the application must include consideration of the impact of data transfer times. Previous work [3] has demonstrated that run-time, adaptive, application-scheduling based on dynamic information about the status of computing resources is a good general approach for achieving performance on the Grid.

In [20], three heuristics (*Max-min, Min-min* and *Sufferage*) were proposed for the scheduling of independent tasks in single-user, homogeneous environments. **In this paper, we modify existing heuristics to schedule parameter sweep applications with file I/O requirements, we propose an extended version of Sufferage, XSufferage, and we study the impact of inaccurate performance prediction on scheduling.** We integrate these heuristics into a gen-

This research was supported in part by NSF Grant ASC-9701333, NASA/NPACI Contract AD-435-5790, DARPA/ITO under contract #N66001-97-C-8531, and CNRS/INRIA project ReMaP

0-7695-0556-2/00 $10.00 © 2000 IEEE

eral adaptive scheduling algorithm and compare them in various simulated computing environments and for various application scenarios. We will use a standard performance metric to evaluate our heuristics: the application *makespan* [22], i.e. the time between the first input files is sent to a computational server and the last output file is returned to the user. Our ultimate goal is to include our adaptive scheduling algorithm in a software framework, a Parameter Sweep Template (PST), developed as part of the AppLeS project [13]. PST will be the subject a a future paper.

In a Grid environment it is usually difficult to obtain accurate predictions for computing and networking resource performance; moreover most scheduling heuristics make use of such predictions. We designed a simulator that allows us to experiment with different levels of performance prediction accuracy. In this paper we present a preliminary study of the effect of increasing inaccuracy on the heuristics under consideration and discuss how adaptivity can be used to promote performance in Grid environments.

This paper is organized as follows. In Section 2, we present our models for both the application and the underlying Grid environment. In Section 3, we present our scheduling algorithm. Section 4 focuses on the different task/host assignment heuristics whereas Sections 5 discusses adaptivity and performance prediction accuracy. Section 7 references related research work, and Section 8 concludes the paper.

2. A Scheduling Model for Parameter Sweeps on the Grid

2.1. Application Model

We define a **parameter sweep** application as a set of n independent sequential tasks $\{T_i\}_{i=1,..,n}$. By *independent* we mean that there are no inter-task communications or data dependencies (i.e. task precedences). We assume that the input to each task is a set of files and that a single file might be input to more than one task. In our model, without loss of generality, each task produces exactly one output file. Figure 1 shows an example with input file sharing among tasks. We assume that the size of each input and output file is known a-priori.

This model is motivated by our primary target application for PST: MCell [29], a micro-physiology application that uses 3-D Monte-Carlo simulation techniques to study molecular bio-chemical interactions within living cells. An MCell run is composed of multiple Monte-Carlo simulations for cell regions whose geometries are described in (potentially very large) files. For instance,

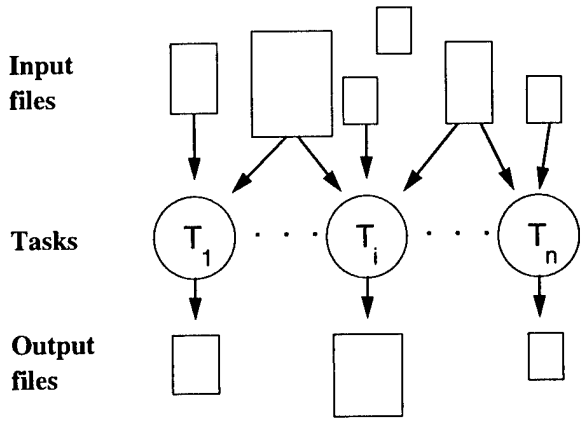


Figure 1. Application Model

MCell can be used to study the trajectories of neurotransmitters in the 3-D space between two cell membranes for different deformations of the membranes, where each deformation is described in a geometry file. Additional files of variable sizes are also needed for describing the initial locations of different molecules. The model described above is adequate for our purpose and should be general enough to accommodate other applications (e.g. general Monte-Carlo simulations).

MCell users and developers anticipate large-scale runs that contain tens of thousands of tasks with each task processing hundreds of MBytes of input and output data, with various task-file usage patterns. Furthermore, research work outside the scope of this paper addresses the question of *steerable* MCell runs when users can add new tasks on-the-fly, and modify the computational targets of existing tasks. Such runs will lead to fairly intricate task-file usage patterns and a model as general as the one we describe will be needed to study scheduling issues in the presence of computational steering.

2.2. Grid Model

We assume that the Computational Grid available to the user has the following *topology*: it is a set of k clusters of computing resources $\{C_j\}_{j=1,..,k}$ that are accessible to the user via k distinct network links. This is a logical topology, and this work does not attempt to take into account the actual physical network topology of the Grid. Our intent is to model a wide-area system, such as a *Worldwide Flock of Condors* [24] for instance. Each cluster contains a certain number of *hosts* where a host can be any computing platform, from a single-processor workstation to an MPP system, and is available for computation. From now on, we call

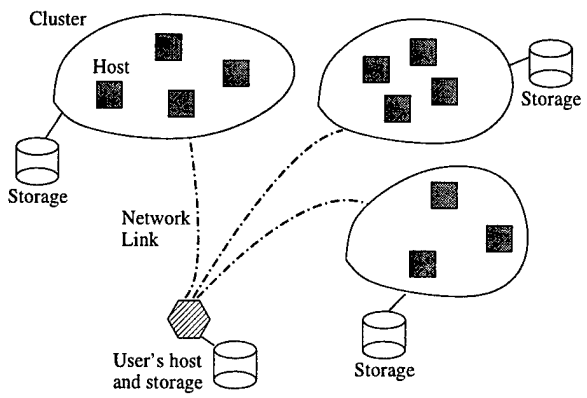


Figure 2. Environment Models

both hosts and network links *resources*. We do not impose any constraint on the performance characteristics of the resources and our simulator allows for arbitrary performance variation. The only requirement is that for each computation and file transfer, an estimate of running time is available. On interactive hosts, the estimate is the task *execution time* whereas on batch resources, it is the *turnaround time* (defined as the waiting time plus the execution time). Such estimates can be provided by the user, computed from analytical models or historical information, or provided by facilities such as the Network Weather Service (NWS) [31], ENV [28], Remos [19], Grid services such as those found in Globus [10], or computed from a combination of the above. Recent work shows that the accuracy of the performance estimates have an impact on the effectiveness of scheduling heuristics and that information about the probability distribution of the estimates should be exploited for scheduling [18, 26]. We explore scenarios for different levels of estimate accuracy in Section 5.

We assume that a storage facility (e.g. NFS, GASS [9], IBP [23]) is available at each cluster so that files can be shared among the processes running on different hosts in the cluster. For the first implementation of PST, we are planning to use IBP for storage management. Figure 2 shows an example of a Grid of a Grid with three clusters. In this work we assume that all input files are initially stored on the user's host, that all output files must be returned to this location, and that there are no inter-cluster file exchanges. We assume for now that once assigned, tasks do not migrate between resources. This scenario fits the current usage of several real-life, parameter sweep applications (e.g. MCell, INS2D [25]), and we leave alternate usage scenarios for future work. In this work we ignore possible storage constraints and assume unlimited storage space. Our model assumptions are discussed in the following section, but we believe that they make it possible to obtain initial meaningful results about a realistic environment while keeping the simulation tractable.

2.3. Model Discussion

Our Grid model makes several simplifying assumptions. Even though we allow network links to have arbitrary dynamic performance characteristics, we do not model network contention caused by the application itself. Instead we view the network as a set of distinct links emanating from the user's host and that can all be used in parallel. We believe that this assumption will need to be relaxed in future work. Since our purpose is not simulation per se, we will aim at using simulators developed by other research groups. For instance, the Micro-Grids simulator [15], when it becomes available, will allow us to precisely simulate network contention and study its impact on our current results.

Similarly, we do not take into account contention within a cluster for shared file access. Our justification is that wide-area file transfer cost dominate the cost of file access within the cluster, even in the presence of contention. While this is true in certain environments, it is certainly not general and we will need to enhance our own simulator so that it can simulate model contention for shared storage. For instance, this will be necessary to simulate high-bandwidth wide-area research networks such as the vBNS. At the moment we are planning to deploy the PST software on non-dedicated commercial wide-area networks with many clusters and we believe our simulation results will hold in those environments. The assumption of unlimited storage is realistic for current runs of MCell on our current testbed, but that assumption will be relaxed in future versions of our scheduling algorithm.

Our Grid model also assumes that there are no direct network links between clusters in the sense that file transfers cannot be performed by our scheduling algorithm between clusters. In other words, the only authoritative source of input files is the user's host. This prevents schedulers from making some optimizations when disseminating input files among the clusters. However, no heuristic we study in this paper is able to support such optimizations as this would require a considerably more precise understanding of the network. Our next step in this research will be to use a more complete network model and to consider any storage device for any file retrieval. This will allow not only for more flexible application scenarios, but also for the investigation of more sophisticated scheduling algorithms.

```
schedule() {
    (1) compute the next scheduling event
    (2) create a Gantt Chart, G
    (3) foreach computation and file transfer currently underway
            compute an estimate of its completion time
            fill in the corresponding slots in G
    (4) select a subset of the tasks that have not started execution: T
    (5) until each host has been assigned enough work
            heuristically assign tasks to hosts (filling slots in G)
    (6) convert G into a plan
}
```

Figure 3. Scheduling Algorithm Skeleton

3. Adaptive Scheduling for Parameter Sweeps

3.1. The Scheduling Algorithm

We call our scheduling algorithm `schedule()`. The general strategy is that it takes into account resource performance estimates to generate a *plan* for the assigning file transfers to network links and tasks to hosts. To account for the Grid's dynamic nature, `schedule()` can be called repeatedly so that the schedule can be modified and refined. We denote the points in time at which `schedule()` is called *scheduling events*, according to the terminology in [20]. We assume that at each scheduling event our scheduler has knowledge of: (i) the current topology of the Grid (number of clusters, number of hosts in those clusters, network and CPU loads), (ii) the number and location of copies of all input files, and (iii) the list of computations and file transfers currently underway or already completed.

Figure 3 shows the general skeleton for `schedule()` whose steps can be described as follows:

(1) determines the time of the next scheduling event. This can take into account environment behavior to increase or decrease the scheduling event frequency. A higher frequency means a higher adaptivity but also a higher scheduling cost.

(2) creates a Gantt chart [7], G, that will be used to keep track of task/host assignments. G contains as many columns as resources. Figure 4 shows an example of a Gantt chart for an environment containing two clusters with respectively two and three hosts.

(3) inserts slots corresponding to tasks that are currently running into the chart. Two examples are shown on Figure 4 as black-filled rectangular slots at the beginning of the chart (one file transfer and one computation).

(4) performs a task-space reduction that can be used to reduce `schedule()`'s execution time. This will be necessary for runs of real parameter sweep applications since we expect them to contain thousands of tasks.

(5) is the core of the algorithm, determining which task should be performed on which host. This step is detailed in Section 4. Examples of slot assignments are depicted on Figure 4 in gray. In this example, input file transfers are scheduled on the network link to cluster 2, the computation is then scheduled on a host within that cluster, and the output file is scheduled to be returned to the user's host.

(6) converts the Gantt chart into a *plan*, or a sequence of instructions. These instructions can then provide a schedule for deployment with Grid software services (for job submission and monitoring, data motion, etc.).

3.2. Discussion

Several steps of our scheduling algorithm can be implemented independently and this makes it possible to experiment with different techniques and strategies. Our ultimate goal is to instantiate the algorithm so that it is optimized for specific environments and applications. Furthermore, this instantiation should be

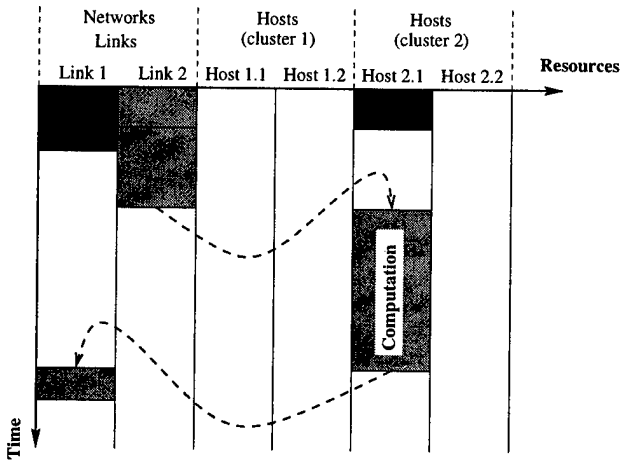


Figure 4. Sample Gantt chart

as dynamic and automatic as possible as the algorithm should be able to reconfigure itself on-the-fly to accommodate changing Grid conditions.

Step (1) allows for dynamic adjustment of the scheduling event frequency. A higher frequency leads to more adaptivity and should be better for unstable Grid environments. However, a higher frequency also means that `schedule()` is called more frequently. Depending on the computational cost of the scheduling, a high frequency might not be desirable. In the case of the PST software, a processor is usually dedicated to scheduling. Furthermore, given the granularity of the applications we are considering, there should be no need for very high frequencies. However, steps (3) and (5) might include remote access to Grid information services (e.g. NWS [31]) to perform performance prediction. It may then become necessary to reduce the scheduling event frequency because of latencies associated with Grid services. One can envision algorithms to dynamically tune the frequency in step (1). For instance, one could compute the deviation of the computation from what was planned during the previous call to `schedule()`. Large deviations suggest higher frequencies whereas low deviations suggest that the frequency can be decreased.

Step (3) obtains estimates for completion of ongoing file transfers and computations in order to start filling in slots in the Gantt chart. This is a little different from estimating just running times because more information is available. It is indeed conceivable that more precise forecasting techniques can be used because the required prediction is in the near future and because there are ways to compute percentage to completion. It may be that applications provides means to check on computation progress (this is however not the case for MCell). More generally, techniques using historical information from Grid services can lead to estimations of the percentage to completion. We have started experimenting with such techniques and will present results in a subsequent paper.

Step (5) in Figure 3 states that tasks are assigned to hosts until "enough" work has been assigned. Like step (4), this is intended to limit the time spent computing the schedule. Indeed, it makes little sense to assign tasks to hosts for times that are well beyond the next scheduling event since the schedule will be re-evaluated then. Since real runs will not be perfectly predictable, it is good practice to leave some margin of error and assign work until after the next scheduling event so that resources are utilized even if the performance predictions were pessimistic.

Step (6) processes the Gantt chart and transforms it into a set of task lists associated to each resource. The Grid infrastructure software in use is then responsible for sequencing file transfers and computations on the appropriate resources. Here there is some latitude for some choices concerning the actual implementation of the task sequencing. It may be that, due to unexpected performance misprediction, some resource cannot execute the next task on its list but could execute one of the subsequent ones. For instance, a file transfer for the output of a task that is unexpectedly lagging might cause a network link to stay unused. A solution is to relax the ordering of the list and allow subsequent file transfers to be performed immediately.

Our experience indicates that allowing output file transfers to be delayed until they can effectively occur is usually a good idea as it allows for better network bandwidth utilization while not disrupting the overall schedule to a great extent. This is the scheme used by `schedule()` in this paper. Further experiments would be required in order to investigate the trade-offs between resource utilization and schedule disruption.

Steps (1) and (5) use dynamic information about the status of the Grid resources and are key to the algorithm efficacy. **Our main focus in this paper is step (5) of the algorithm** and our results are presented in the following section. We also present preliminary experiments concerning step (1) in Section 5.

4. Performing Task/Host Assignment Decisions

4.1. Heuristics

We must identify heuristics that are applicable in Grid environments to perform assignment of file transfers to network links and of computations to hosts.

Moreover these heuristics must be reasonably computationally inexpensive with respect to the duration of a typical application task. Three simple heuristics for scheduling independent tasks for a uniform single-user environment are proposed in [17, 20]: *Min-min*, *Max-min*, and *Sufferage*. These three heuristics iteratively assign tasks to processors by considering all tasks not scheduled and computing Minimum Completion Times (MCTs). For each task, this is done by tentatively scheduling it to each resource, estimating the task's completion time, and computing the minimum completion time over all resources. For each task, a *metric* is computed using these MCTs, and the task with the "best" metric is assigned to the resource that lets it achieve its MCT. The process is then repeated until all tasks have been scheduled.

Min-min uses the Minimum MCT as a metric, meaning that the task that can be complete the earliest is given priority. The motivation behind Min-min is that assigning tasks to hosts that will execute them the fastest will lead to an overall reduced makespan. *Max-min*'s metric is the Maximum MCT. The expectation is to overlap long-running tasks with short-running ones. The rationale behind *Sufferage* is that a host should be assigned to the task that would "suffer" the most if not assigned to that host. For each task, its sufferage value is defined as the difference between its best MCT and its second-best MCT. Tasks with high sufferage value take precedence. Note that this definition of sufferage is a little different from the one presented in [20]. We found our definition easier to implement and experiments showed no differences between our version of sufferage and the one in [20]. We modified all three heuristics so that they (i) include input and output data transfer times when computing MCTs and (ii) take into account the fact that some files may already be present on remote storage devices. In addition, we implemented an extended version of the Sufferage heuristic: *XSufferage*.

In XSufferage the sufferage value is computed not with MCTs, but with *cluster-level MCTs*, i.e. by computing the minimum MCTs over all hosts in each cluster. Our first intuition was that Sufferage should be a nice way to exploit file locality issues without any a-priori analysis of the task-file dependence pattern. The idea is that if a file required by some task is already present at a remote cluster, that task would "suffer" if not assigned to a host in that cluster, provided the file is large compared to the available bandwidth on the cluster's network link. The sufferage value would then be a simple way of capturing such situations and ensuring maximum file re-use. This is somewhat reminiscent of the idea of *task/host affinities* introduced in [20], where some hosts are better for some tasks but not for others.

However that early experiments showed that the Sufferage heuristic as described above does not lead to makespans as good as the ones we expected. This can be explained easily. Assume that a task, say T_0, requires a large input file that is already stored on a remote cluster. If that cluster contains two (or more) hosts with nearly identical performance, which is often the case in practice, then both those hosts can achieve nearly the same MCT for that task. If the file is of significant size compared to network bandwidths available, then it is likely that those two hosts lead to the best and second-best MCTs for T_0. This means that the sufferage value will be close to zero, giving the task low priority. Other tasks may be scheduled in its place, generate load on the hosts in the cluster, and eventually force T_0 to be scheduled on some other cluster, thereby requiring an additional file transfer. This can have a dramatic impact on the overall application makespan as it leads to poor file re-use among tasks, especially in wide-area bandwidth-constrained environments.

We solved this problem in XSufferage by using a modified sufferage value definition. For each task and for each cluster we compute the task's MCT only for hosts in the given cluster and call that value the *cluster-level MCT*. The *cluster-level sufferage* value is computed as the difference between the best and second-best cluster-level MCT. The task with the highest cluster-level sufferage is given priority and is scheduled to the host that achieves the earliest MCT within the cluster that achieves the earliest cluster-level MCT. Appendix 8 gives formal descriptions of Max-min, Min-min, Sufferage, and XSufferage.

4.2. Simulating Parameter Sweeps in Grid Environments

In order to evaluate the efficacy of the heuristics described earlier we developed a Grid parameter sweep simulator. At present, little software is available for Grid simulation. Among the most promising work, the Bricks project [30] addresses the question of simulating heterogeneous distributed environment for the purpose of evaluating scheduling strategies, but no public implementation is available at the moment. Furthermore, Bricks targets "global computing systems" [5, 27, 11, 10] rather than application schedulers. It assumes constant task and data arrival rates to servers and uses queuing theory in an attempt to model many users who asynchronously interact with a global computing system. By contrast, our simulator is purely event-driven which is more appropriate in our

framework where the scheduler knows all tasks a-priori and is in charge of only one application. The Micro-Grids [15] project will also be of interest for gaining insight on how our simulation results hold under more realistic assumptions. At present, it cannot be used to perform large numbers of runs of large-scale applications as it emulates the Grid rather than simulates runs of the application. However, Micro-Grids uses a network simulator that could help us model network traffic more accurately by taking into account physical network topology and link contention due to that topology.

Our simulator allows us to compare heuristics under the same load conditions, in a reproducible manner, for a wide variety of system states and application scenarios. In addition, we verified the accuracy of our simulated results by comparing experimental runs in shared, production environments with similarly loaded simulation application execution times. Our simulator takes as input `schedule()`, a task/host assignment heuristic, a description of the application tasks and input/output files, and a description of the Grid topology with performance characteristics of Grid resources. These characteristics can be constant values, samples from random distributions, or traces from the NWS [31]. In this work we use only NWS traces as they lead to more realistic models. The simulator also allows for adding and removing resources dynamically, but we do not perform any experiments with transient resources in this paper. The output of the simulator is a makespan value based on the set of input parameters. More details on the simulator can be found in [6].

4.3. Simulation Results

4.3.1. Random Grids and Applications

In order to perform a fair comparison of task/host selection heuristics we generated 1000 simulated Grids and 2000 simulated applications. We then randomly picked Grid/application pairs among the 2,000,000 possible, and ran our simulator for each pair with all heuristics. The simulations in this section assume 100% accurate performance estimation and scheduling events occur every 500 seconds. The expectation is that computing statistical characteristics of makespans achieved by each heuristic is representative if the sample size is large enough, that is if enough Grid/application pairs are simulated. Before presenting the results, let us describe how Grids and applications were generated.

In what follows we denote by $U(x, y)$ the discrete integer uniform probability distribution on the interval $[x, y]$ where x and y are integers. Each Grid contains a $U(2, 12)$ number of clusters and each of those clusters contains a $U(2, 32)$ number of hosts. The performance of each host is modeled by a CPU load trace randomly picked among 50 different actual traces obtained from the NWS for various hosts. Each trace is then shifted by a random offset, so that two hosts using the same CPU trace do not exhibit the same behavior at the same time. Similarly network link performance is modeled by randomly picking latency/bandwidth traces among 20 different NWS traces. All the NWS traces that we use for simulations in this paper typically span 4 days of real time and were obtained for hosts in various US research institutions and for network links between these institutions (commercial Internet or vBNS). Traces were collected during the first week of November 1999.

In accordance with typical MCell scenarios we generate applications as sets of independent Monte-Carlo simulations, with the tasks of a simulation sharing a (potentially large) input data file for describing 3-D geometries. All tasks take as input one additional file of 1 KByte and generate an output file of 10 KBytes. An application is composed of a $U(2, 10)$ number of Monte-Carlo simulations, with each simulation composed of a $U(20, 1000)$ number of tasks. Each task requires a $U(100, 300)$ number of seconds of computation on an unloaded base CPU. Finally, the size in KBytes of the geometry file associated with each Monte-Carlo simulation is $U(400, 100000)$, meaning that those files can reach the size of approximatively 95 MBytes. These distributions are representative of what can be expected from real MCell applications. We generated one-thousand applications following exactly this method, and we also generated another thousand by adding random file/task dependencies. The idea is to create some perturbation of the regular structure of the file/task dependency graph and investigate if such a perturbation has an impact on the relative performances of the different task/host selection heuristics. The perturbation consisted in adding a number of random additional dependencies on the order of one fifth of the total number of tasks. Even though such perturbations take us away from typical MCell applications, it can be interesting to see how they affect the heuristics.

The results are summarized in Table 4.3 for both standard applications and applications with file/task dependencies perturbation. For each scheduling heuristic (including a self-scheduled workqueue [12]) and for each type of application, the table contains three performance values. The results are computed over 1000 random Grid/application pairs. We use the geometric mean of the makespans rather than the arithmetic mean to account for the fact that, depending on the Grid and the application, makespans in the sample

Table 1. Results for Random Grid/Application pairs

Scheduling	No Perturbation			Perturbation		
	Geometric Mean (sec)	Av. Degradation from Best (%)	Average Rank	Geometric Mean (sec)	Av. Degradation from Best (%)	Average Rank
Max-min	2390	17.3	3.1	2549	18.9	3.1
Min-min	2452	21.2	3.0	2619	23.2	3.0
Sufferage	2329	14.1	2.8	2505	16.7	2.9
XSufferage	2174	6.2	1.8	2316	7.9	1.8
Workqueue	2850	42.2	4.3	3091	48.1	4.2

space can be of different orders of magnitude (one large makespan could easily dominate the arithmetic mean).

The second performance value is the "average degradation from the best" which is a measure of how far a heuristic is from the best heuristic on average. For each heuristic, it is computed as the arithmetic mean over all Grid/application pairs of the relative difference of the makespans for that heuristic and for the best heuristic. The smaller that value, the closer the heuristic to being the best one on average.

The third performance value is the "average rank" of a heuristic over all Grid/application pairs. The average rank is computed as the arithmetic mean of the rank (1 to 5), where the heuristic leading to the best makespan is of rank 1 and the one leading to the worst makespan is of rank 5.

These three different performance values all have slightly different interpretations and make it possible to gain a clear understanding of how the heuristics compare with one another. For instance, it could be that heuristic 1 is the best one in most cases, and that heuristic 2 is the second-best one in most cases as well. In that case, their average ranks will be close to 1 and 2 respectively. This might lead us to think that heuristic one is preferable. However, it is possible that the average degradation from best are respectively 20% and 5%. For instance, it can be that when heuristic 1 is not the best one it is far worse than the best one, whereas heuristic 2 might not be best often, but is never far behind the best one in practice. In that case, we would probably conclude that heuristic 2 is preferable.

The main message from Table 4.3 is that XSufferage is the best heuristic as it leads to the best geometric mean, average degradation from the best, and average rank for perturbed and non-perturbed applications. Its average degradation from the best is at least twice smaller than that of any other heuristics for standard applications and applications with perturbation. Its average rank is better than any other by 1 unit. Note that the workqueue, in these experiments, is the least efficient scheduling algorithm as its average rank is larger than 4 units. Max-min and Sufferage are comparable with a slight advantage to Max-min, and Min-min seems less efficient.

Note that all the results in Table 4.3 are averages over a large number of experiments. In the following sections we will see cases where Min-min leads to good makespans when compared to Max-min and Sufferage. We claim that the experimental results presented in this section are sufficient to show that XSufferage is the one of the four heuristics that leads to best schedules for parameter sweep applications, given the models described in Section 2.

4.3.2. Varying Shared File Sizes

Figure 5(a) show simulation results for the following application and Grid. The application consists of 1600 tasks, where each task takes as input a 10K un-shared file and one of eight identical shared files, each shared by 200 tasks. All tasks are identical in terms of computational cost (200 seconds on an unloaded base CPU) and produce a 10K output file. This application setting is comparable to what some of our target parameter sweep applications require. We simulate a Grid such as one that could realistically be used by a user based at UCSD. That Grid contains 5 clusters containing respectively 6, 6, 8, 20, and 20 hosts. The performance characteristics of the hosts are based on actual CPU traces obtained via the Network Weather Service. The network links are also modeled from NWS traces obtained over the course of a day between a workstation at UCSD and several remote sites accessible by commercial Internet links or the vBNS. Bandwidths on these links varies from as little as 6 KBytes/sec to 600 KBytes/sec depending on the link and on the time of the day. In terms of bandwidth averages, one can classify two of the links as *fast* (500 KBytes/sec), three of the links as *moderate* (between 100 and 200 KBytes/sec), and one as *slow* (50 KBytes/sec). One

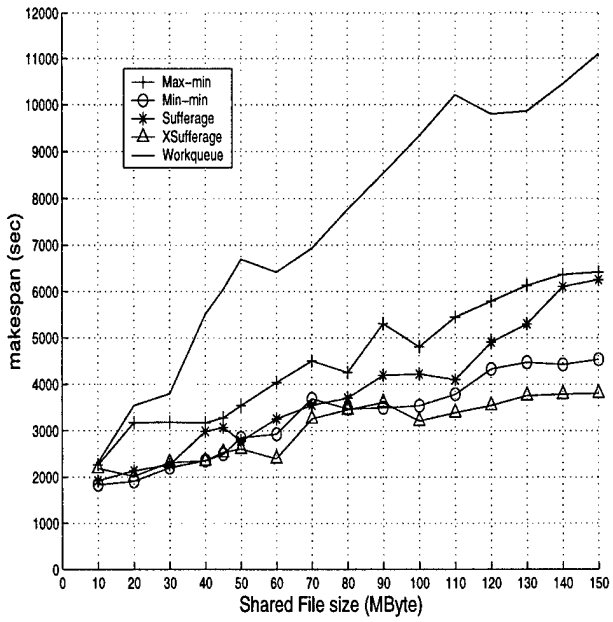


Figure 5. Makespan vs. shared file size for different heuristics

of the two large 20-host clusters is accessible via a fast link. For large shared file sizes on the graphs, the average ratio between file transfer time and computation time for one task is about 3 on a fast link and 30 on a slow link. For the smallest shared file sizes in our study, that ratio is about 0.2 on a fast link and 2 on a slow link. For these experiments, the interval between scheduling events was always 500 seconds, and we assumed 100% accurate performance estimations.

The graph in Figure 5 plots makespans vs. shared file shared file sizes between 10 and 150 MBytes for the four heuristics and the self-scheduled workqueue. For very small shared file sizes (up to 100 KBytes, not plotted on the graph), the workqueue leads to better makespans than other heuristics when files are so small that the effect of file sharing becomes negligible. However, the workqueue quickly becomes inefficient when shared file size increases. The other heuristics perform similarly for small shared file sizes but one can see on the graph that XSufferage performs at least about 20% better than Min-min and 40% better than Max-min and Sufferage for a file size of 150 MBytes. We obtained similar results with different Grid configurations. We also performed experiments with much larger file sizes. Even though those experiments are not very realistic given the current networking capabilities they provide information about what happens when file re-use is the only constraint for achieving good scheduling. The results showed that XSufferage constantly outperforms all other heuristics by at least 50%. These results show that XSufferage does a better job at capturing and taking advantage of file sharing patterns to maximize file re-use.

5. Adaptive Scheduling

5.1. Quality of Information

A new avenue of research that we are beginning to explore is the study of *Quality of Information* (QoI) on scheduling, that is the impact of the performance estimation accuracy and other qualitative attributes on different scheduling strategies . We expect different heuristics to react differently to degrading levels of accuracy and that strategies that do not depend on performance estimation and forecast (e.g. self-scheduled ones [16]) will be more performance-efficient when QoI is low. Low QoI can also be accounted for in adaptive scheduling algorithms such as `schedule()`. The following section presents our first simulation results for different levels of QoI and for increasingly adaptive versions of `schedule()`.

Our initial model for simulating different levels of QoI is simple. Our simulator allows us to obtain 100% accurate estimates for all file transfer or computational times and we add random noise to those estimates to simulate inaccurate performance estimates. For each estimate used by the scheduling algorithm we introduce a percentage error that is uniformly distributed on the interval $[-p, +p]$ where p is a value between 0% and 100%. Perfectly accurate QoI corresponds to $p = 0$, whereas $p = 10$ means that every 100% accurate estimate will be randomly increased or decreased by up to 10%.

This model is sufficient for obtaining initial results concerning the impact of QoI on the scheduling of parameter sweep applications, however it makes two assumptions that are not realistic for real forecasting services that will be deployed in Computational Grids. First, it assumes that QoI behavior is the same for all estimates (for file transfer times and computational times) and for all resources. This is clearly not the case as network behavior is significantly different from CPU behavior for performance prediction purposes [32], and some resources will generally be more predictable than others on a regular basis. Second, it assumes that QoI behavior does not depend on whether a forecast

The term "quality of Information is used to describe qualitative aspects of performance predictions in the Performance Prediction Engineering Project [14].

is needed for an event in the short-term or in the long-term. For instance, our model uses the same error model for predicting a file transfer time if the transfer is initiated in the next minute or in an hour. A more realistic model should probably try to capture some *decay* of the QoI as predictions become more and more long-term. Note that this issue becomes less critical for high scheduling event frequencies. Future work will aim at providing a more realistic model of QoI based on experiments with deployed Grid services, such as the Network Weather Service [31], and with a variety of Grid resources.

5.2. Simulation Results

Figure 6 shows simulation results for four different scheduling event frequencies and decreasing QoI levels.

We use the same simulated Grid as the one used in Section 4.3.2 and the application is modeled after an MCell computation that performs eight moderate-size Monte-Carlo simulations (100 tasks each) for eight different geometry configurations of a neuro-muscular junction. Geometry files are on the order of 40 MBytes, meaning that network transfer times for those files take on average 80 seconds on a fast link and about 800 seconds on a slow link. The average task computational time over all hosts is about 110 seconds, can be as fast at 90 seconds, and as slow as 350 seconds depending on the host and on its load when the task is running (as simulated by an offset in an NWS CPU load trace).

All data points in the graphs of Figure 6 are computed as the average makespan over 50 simulated runs. This is necessary since we introduce random noise to performance estimates in order to simulate different levels of QoI. All graphs plot average makespans vs. values of p (defined in Section 5.1, for the heuristics presented in Section 4.1 as well as for a self-scheduled workqueue algorithm. Since the workqueue does not make use of performance estimates it is not sensitive to QoI. It is shown as a horizontal solid line on the four graphs (with a makespan of 1730 seconds). The variances associated with the 50 samples for each data point were small for all heuristics: coefficients of variations were on the order of 5%.

Graph 6(a) plots results when there is only one initial scheduling event, meaning that the scheduling algorithm is not adaptive. All heuristics but Max-min lead to better makespans than the workqueue for perfect QoI ($p = 0$), but their performance degrades very rapidly when p increases. Max-min leads to the worst makespans, but over all, all heuristics lead to makespans at least 40% larger than the workqueue when p is greater than 50. This result is not surprising as the cumulative errors of performance estimates impact the computations of the various MCTs required by the heuristics.

Graph 6(b) shows the results when there is a scheduling event every 500 seconds, or 3 times during each run of the application in this case. One can notice that some heuristics outperform the workqueue for values of p up to 20. Max-min and Sufferage exhibit less performance as soon as the QoI is not perfect.

Graph 6(c) shows the results when there is a scheduling event every 250 seconds, or between 5 and 8 times for each run depending on the heuristic being used. The effect observed on graph 6(b) is more pronounced in that heuristics become more tolerant to low QoI thanks to increased adaptivity, even though Max-min still leads to large makespans. Sufferage outperforms the workqueue for values of p lower than 30, whereas Min-min and XSufferage lead to better makespan than the workqueue for all values of p. For perfect accuracy, XSufferage outperforms workqueue by as much as 25%.

Finally, Graph 6(d) shows results for scheduling events every 125 seconds, or between 11 and 14 times per run. Sufferage now outperforms the workqueue for p up to 80, whereas Min-min and XSufferage keep benefiting from increased adaptivity. The results show little improvements for higher scheduling frequencies. This is due to the granularity of the application: since tasks take at least 90 seconds, little can be gained by calling `schedule()` more than once every 125 seconds. For "good" QoI ($p < 5\%$), XSufferage always outperforms Min-min.

Note that these results are preliminary and that it is difficult to use them to rank the different heuristics according to their respective robustness to inaccurate performance predictions. It will be necessary to perform experiments for large numbers of different Grid configurations and application structures as was done in Section 4.3.1. A future paper will contain results from such experiments as well as a more in-depth study of QoI issues.

Note also that in these experiments we assume that the QoI does not depend on the scheduling event frequency (see the discussion in Section 5.1). However, a high scheduling frequency implies that the heuristics do not use long-term predictions (see the discussion on step (5) in Section 3.2). Assuming that short-term predictions are typically more accurate than long-term predictions, higher scheduling frequencies lead to improved QoI. On Graph 6(d) we show simulation results for values of p up to 100, but we expect that in reasonably stable Grid environments with appropriate forecasting services, the performance estimation error will not be as large as +/- 100% for short-term predictions.

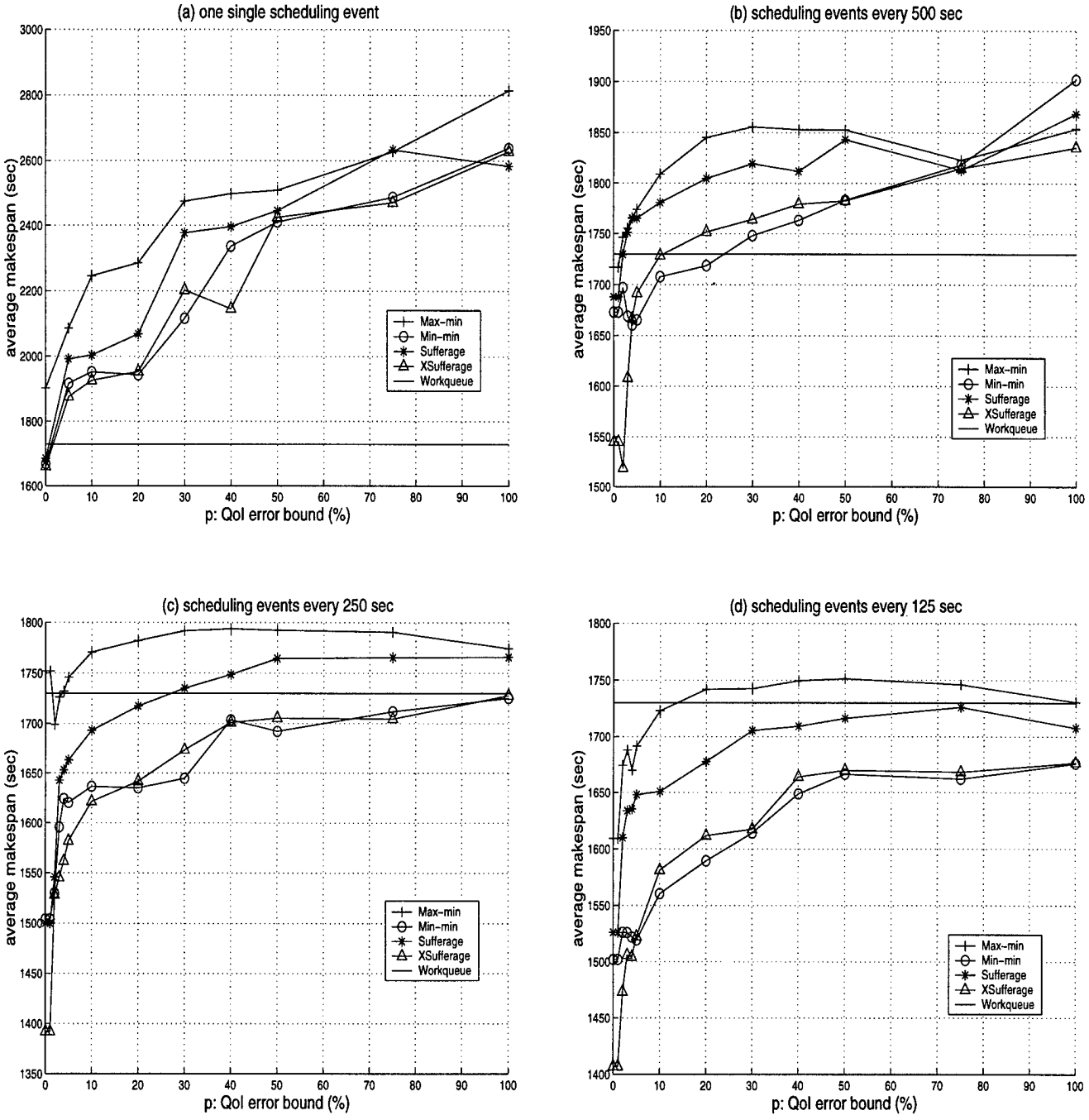


Figure 6. Simulation results for various levels of QoI and adaptivity

The left part of the graph should be more representative of what we can expect for real systems.

Finally, the scheduling event frequency impacts the performance of the adaptive algorithm even for perfect QoI. For instance, XSufferage has an average makespan around 1550 seconds for $p = 0$ and scheduling events every 500 seconds, whereas that average makespan becomes lower than 1400 seconds for larger frequencies. All heuristics but Max-min achieve better makespan for higher frequencies in the case of perfect QoI. This is rather counter-intuitive as one would expect that perfect QoI would not mandate any adaptivity at all. The fact is that applying those heuristics on small sets of tasks leads to better scheduling decisions and shorter makespans. As tasks complete, successive calls to `schedule()` apply the heuristics to the decreasing sets of tasks, leading to better overall makespans. This suggests that calling the scheduling algorithm repeatedly is a good idea for Sufferage, Min-min, and XSufferage, even if the environment is very predictable.

6. Summary of Simulation Results

The simulation results in Section 4.3.1 showed that XSufferage is on average a better heuristic than the traditional Max-min, Min-min, Sufferage, and self-scheduled workqueue for scheduling parameter sweep applications such as MCell on Computational Grids, provided the models of Section 2 hold. We believe that Xsufferage leads to better results because it better captures the file/task dependencies of the application and leads to improved file re-use. This claim is supported by the results in Section 4.3.2. Indeed, all experiments we have conducted with very large shared files seem to indicate that XSufferage leads to better makespans than its contenders. Finally, the preliminary study of Quality of Information and adaptivity in Section 5 showed that all heuristics can benefit from increased adaptivity and from increased QoI. The results seem to indicate that XSufferage leads to very good results for good QoI and compares well with other heuristics for poor QoI.

It is always difficult to make general statements about the relative efficiency of scheduling algorithms since the space of possible Grid configurations and application structures is very large. The solution is to sample both the Grid and the application space as much as possible as was done in Section 4.3.1. Future work will contain such sampling for experiments similar to the ones presented in Section 4.3.2 and 5 in order to make the results concerning shared file sizes and QoI more general. Ironically, performing such large-scale simulations in the Grid/application space is itself a parameter sweep application and we will probably use the PST software to distribute it on a real computational Grid.

7. Related Work

A large number of research papers address the question of mapping sets of tasks onto sets of processors in a view to minimizing overall execution time. Many of these papers address the case where tasks are independent [17, 12, 16, 20]. Scheduling heuristics found in [20] were adapted to our framework as discussed in Section 4. All these papers make simplifying assumptions for task execution times (constant, following a truncated Gaussian distribution, etc.) and none of them take into account data storage issues.

The work described in [2] focuses on scheduling applications structured as DAGs on heterogeneous sets of processors and uses heuristics that are related to Max-min and Min-min for *Level-by-Level* scheduling of the graphs. However, special attention is paid to data storage issues which makes that work related to the research presented in this paper. A major difference between our work and the development in [2] is that the latter assumes constant perfectly predictable performance characteristics for resources as should be available in advanced reservation QoS environments. Also, the different application structures (Parameter Sweep vs. DAGs) lead to many differences between the models in this paper and in [2]. For instance, data repositories are located anywhere on the network (as opposed within a cluster) and datasets are pre-staged to these repositories. Finally, [4] contains a survey that encompasses several heuristics in addition to the ones described in [20]. We will consider these heuristics in our future work. This work contrasts to others in that we: (i) take into account application data storage; (ii) model shared, heterogeneous computational and network resources with realistic dynamic performance characteristics; (iii) study the impact of the accuracy of performance prediction; (iv) introduce a new heuristic for scheduling parameter sweep applications (XSufferage).

This work is also related to our work on an AppLeS Parameter Sweep Template (PST) in that the results in this paper provide a good justification that XSufferage should be implemented as part of the PST scheduler. PST will provide with a practical way to deploy and schedule parameter sweep on the computational Grid using available software infrastructures and will be described in a future paper. PST itself is related to the Nimrod project [1]. Nimrod targets parameter sweep applications but its scheduling approach is dif-

ferent from ours as it is based on deadlines and on a Grid economy model. Also, to the best of our knowledge, Nimrod does not take into account dynamic Grid conditions or file locality constraints for scheduling. In fact, the work in this paper and our work on the PST software should be applicable to Nimrod and one can envision an implementation of PST as a Nimrod module.

8. Conclusion and Future Work

In this paper we have proposed an adaptive scheduling algorithm for parameter sweep applications in Grid environments. In particular, we address the case of applications where tasks can share input files (e.g. MCell [29]) and the case of non-dedicated Computational Grids that span non-dedicated wide-area networks. After precisely defining our application and Grid model, we adapted three standard heuristics for performing task/host assignment (*Max-min*, *Min-min*, *Sufferage*) and proposed an extension of the *Sufferage* heuristic, *XSufferage*. We also introduced the notion of *Quality of Information* (QoI) to account for inaccuracies in performance predictions. We use simulation to compare the four heuristics and a self-scheduled workqueue algorithm in multiple settings with various shared files sizes, levels of QoI, application structures, Grid topologies and resources. The simulation results demonstrated that: (i) XSufferage leads to better schedules on average by quite a large margin; (ii) Increased adaptivity benefits all four heuristics even for perfect QoI; (iii) XSufferage leads to better schedules for larger shared file sizes; (iv) XSufferage is as tolerant as the other heuristics to poor QoI and more efficient for good QoI.

Future work will provide improvements to our models such as more realistic network and storage models (encompassing shared-storage and link contention, and limited storage space), and alternate application usage scenarios. We will also study the concept of QoI further by investigating realistic QoI models and performing more QoI-related experiments with our simulator. New heuristics such as the ones found in [4, 21] will considered for implementing step (5) of our algorithm. As discussed in Section 3.2, all steps of the algorithm can lead to new research in different directions (performance prediction and forecasting, task-space reduction, trade-offs between schedule disruption vs. maximum resource utilization). Also, the algorithm can be adapted to provide ways to perform adaptive scheduling for other classes of applications by using different heuristics in step (5). Finally, we will incorporate the results here, as well as many of these future improvements, into a practical programming environment and adaptive scheduler for parameter sweep applications on the Grid, an AppLeS Parameter Sweep Template. We believe that such software will provide a useful first step in achieving performance and programmability for applications in Grid environments.

Acknowledgements

The authors would like to thank the reviewers for their insightful comments as well as members of the AppLeS group for their help.

Appendix A: Task/host Selection Heuristics

Let $H_{j,k}$ denote the k^{th} host within the j^{th} cluster and $C(T_i, H_{j,k})$ the estimated completion time of task T_i on host $H_{j,k}$. Let us define the *argmin* operator:

Definition: Given a function f from $\mathbb{R}^n$ into $\mathbb{R}$,

$$f(\text{argmin}_{x \in \mathbb{R}^n} f(x)) = \min_{x \in \mathbb{R}^n} f(x).$$

The operator denotes one of the possible vectors that achieves the minimum of the function f. The way ties are broken is left the implementation and in this work they are broken randomly. An *argmax* operator can be defined in a similar fashion. Assuming a task set T, we can now describe each heuristic as follows:

Min-min

while $(T \neq \varnothing)$
  foreach $(T_i \in T)$
    $(c_i^{(1)}, h_i^{(1)}) = \text{argmin}_{j,k}(C(T_i, H_{j,k}))$
  end foreach
  $s = \text{argmin}_i(C(T_i, H_{c_i^{(1)}, h_i^{(1)}}))$
  assign T_s to $H_{c_s^{(1)}, h_s^{(1)}}$
  $T = T - \{T_s\}$
end while

Max-min

while $(T \neq \varnothing)$
  foreach $(T_i \in T)$
    $(c_i^{(1)}, h_i^{(1)}) = \text{argmin}_{j,k}(C(T_i, H_{j,k}))$
  end foreach
  $s = \text{argmax}_i(C(T_i, H_{c_i^{(1)}, h_i^{(1)}}))$
  assign T_s to $H_{c_s^{(1)}, h_s^{(1)}}$
  $T = T - \{T_s\}$
end while

Sufferage

while $(T \neq \varnothing)$
  foreach $(T_i \in T)$
    $(c_i^{(1)}, h_i^{(1)}) = \text{argmin}_{j,k}(C(T_i, H_{j,k}))$
    $(c_i^{(2)}, h_i^{(2)}) = \text{argmin}_{j \neq c_i^{(1)}, k \neq h_i^{(1)}}(C(T_i, H_{j,k}))$
    $suf_i = C(T_i, H_{c_i^{(2)}, h_i^{(2)}}) - C(T_i, H_{c_i^{(1)}, h_i^{(1)}})$
  end foreach
  $s = \text{argmax}_i(suf_i)$
  assign T_s to $H_{c_s^{(1)}, h_s^{(1)}}$
  $T = T - \{T_s\}$
end while

XSufferage

while $(T \neq \varnothing)$
  foreach $(T_i \in T)$
    foreach cluster j
      $h_{i,j} = \text{argmin}_k(C(T_i, H_{j,k}))$
    end foreach
    $c_i^{(1)} = \text{argmin}_j(C(T_i, H_{j,h_{i,j}}))$
    $c_i^{(2)} = \text{argmin}_{j \neq c_i^{(1)}}(C(T_i, H_{j,h_{i,j}}))$
    $suf_i = C(T_i, H_{c_i^{(2)}, h_{i,c_i^{(2)}}}) - C(T_i, H_{c_i^{(1)}, h_{i,c_i^{(1)}}})$
  end foreach
  $s = \text{argmax}_i(suf_i)$
  assign T_s to $H_{c_i^{(1)}, h_{i,c_i^{(1)}}}$
  $T = T - \{T_s\}$
end while

References

[1] D. Abramson and J. Giddy. Scheduling Large Parametric Modelling Experiments on a Distributed Metacomputer. In *PCW'97*, Sep. 1997.

[2] A. Alhusaini, V. Prasanna, and C. Raghavendra. A Unified Resource Scheduling Framework for Heterogeneous Computing Environments. In *Proceedings of the 8th IEEE Heterogeneous Computing Workshop (HCW'99)*, pages 156–165, Apr. 1999.

[3] F. Berman. *The Grid, Blueprint for a New computing Infrastructure*, chapter 12. Morgan Kaufmann Publishers, Inc., 1998. Edited by Ian Foster and Carl Kesselman.

[4] R. Braun, H. Siegel, N. Beck, L. Boloni, M. Maheswaran, A. Reuther, J. Robertson, M. Theys, B. Yao, D. Hensgen, and R. Freund. A Comparison Study of Static Mapping Heuristics for a Class of Meta-tasks on Heterogeneous Computing Systems. In *Proceedings of the 8th Heterogeneous Computing Workshop (HCW'99)*, pages 15–29, Apr. 1999.

[5] H. Casanova and J. Dongarra. NetSolve: A Network Server for Solving Computational Science Problems. *The International Journal of Supercomputer Applications and High Performance Computing*, 1997.

[6] H. Casanova, A. Legrand, D. Zagorodnov, and F. Berman. Using Simulation to Evaluate Scheduling Heuristics for a Class of Applications in Grid Environments. Technical Report RR1999-46, École Normale Supérieure de Lyon, France, Sep. 1999.

[7] W. Clark. *The Gantt chart.* Pitman and Sons, London, 3rd edition, 1952.

[8] I. Foster and C. Kesselman, editors. *The Grid, Blueprint for a New computing Infrastructure.* Morgan Kaufmann Publishers, Inc., San Francisco, USA, 1998.

[9] I. Foster, C. Kesselman, J. Tedesco, and S. Tuecke. GASS: A Data Movement and Access Service for Wide Area Computing Systems. In *Proceedings of the Sicth workshop on I/O in Parallel and Distributed Systems*, May 1999.

[10] I. Foster and K. Kesselman. Globus: A Metacomputing Infrastructure Toolkit. *International Journal of Supercomputer Applications*, 11(2):115–128, 1997.

[11] A. Grimshaw, A. Ferrari, F. Knabe, and M. Humphrey. Wide-area computing: Resource sharing on a large scale. In *IEEE Computer 32(5)*, volume 32(5), May 1999. page 29-37.

[12] T. Hagerup. Allocating Independent Tasks to Parallel Processors: An Experimental Study. *Journal of Parallel and Distributed Computing*, 47:185–197, 1997.

[13] http://apples.ucsd.edu.

[14] http://apples.ucsd.edu/perf.html.

[15] http://www csag.ucsd.edu/projects/grid/microgrid.html.

[16] S. F. Hummel, J. Schmidt, R. N. Uma, and J. Wein. Load-sharing in heterogeneous systems via weighted factoring. In *Proceedings of the 8th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 318–328, June 1996.

[17] O. H. Ibarra and C. E. Kim. Heuristic algorithms for scheduling independent tasks on nonindentical processors. *Journal of the ACM*, 24(2):280–289, Apr. 1977.

[18] T. Kidd and D. Hensgen. Why the Mean is Inadequate for Accurate Scheduling Decisions. In *Proceedings of the 3rd International Symposium on Parallel Architectures, Algorithms, and Networks (ISPAN'99)*, Jun. 1999.

[19] B. Lowekamp, N. Miller, D. Sutherland, T. Gross, P. Steenkiste, and J. Subhlok. A Resource Query Interface for Network-Aware Applications. In *Proceedings of the 7th IEEE Smposium on High-Performance Distributed Computing*, July 1998.

[20] M. Maheswaran, S. Ali, H. J. Siegel, D. Hensgen, and R. Freund. Dynamic Matching and Scheduling of a Class of Independent Tasks onto Heterogeneous Computing Systems. In *8th Heterogeneous Computing Workshop (HCW'99)*, Apr. 1999.

[21] M. Mitzenmacher. How useful is old information. In *Proceedings of the 16th ACM Symposium on Principles of Distributed Computing*, pages 83–91, 1997.

[22] M. Pinedo. *Scheduling: Theory, Algorithms, and Systems.* Prentice Hall, Englewood Cliffs, NJ, 1995.

[23] J. Plank, M. Beck, W. Elwasif, T. Moore, , M. Swany, and R. Wolski. The Internet Backplane Protocol: Storage in the Network. In *Proceedings of NetSore'99: Network Storage Symposium, Internet2*, 199.

[24] J. Pruyne and M. Livny. A Worldwide Flock of Condors : Load Sharing among Workstation Clusters . *Journal on Future Generations of Computer Systems*, 12, 1996.

[25] S. Rogers and D. Ywak. Steady and Unsteady Solutions of the Incompressible Navier-Stokes Equations. *AIAA Journal*, 29(4):603–610, Apr. 1991.

[26] J. Schopf and F. Berman. Stochastic Scheduling . In *Proceedings of SuperComputing'99, Portland*, 1999.

[27] S. Sekiguchi, M. Sato, H. Nakada, S. Matsuoka, and U. Nagashima. Ninf : Network based Information Library for Globally High Performance Computing. In *Proc. of Parallel Object-Oriented Methods and Applications (POOMA), Santa Fe*, pages 39–48, February 1996.

[28] G. Shao, F. Breman, and R. Wolski. Using Effective Network Views to Promote Distributed Application Performance. In *Proceedings of the 1999 International Conference on Parallel and Distributed Processing Techniques and Applications*, 1999.

[29] J. Stiles, T. Bartol, E. Salpeter, , and M. Salpeter. Monte Carlo simulation of neuromuscular transmitter release using MCell, a general simulator of cellular physiological processes. *Computational Neuroscience*, pages 279–284, 1998.

[30] A. Takefusa, S. Matsuoka, H. Nakada, K. Aida, and U. Nagashima. Overview of a performance evaluation system for global computing scheduling algorithms. In *Proceedings of the 8th IEEE International Symposium on High Performance Distributed Computing (HPDC8)*, pages 97–104, Aug 1999.

[31] R. Wolski. Dynamically Forecasting Network Performance Using the Network Weather Service. In *6th High-Performance Distributed Computing Conference*, pages 316–325, August 1997.

[32] R. Wolski, N. Spring, and J. Hayes. Predicting the CPU Availability of Time-shared Unix Systems on the computational Grid. In *Proceedings of the 8th IEEE International Symposium on High Performance Distributed Computing (HPDC8)*, Aug 1999.

Henri Casanova is a Computer Science and Engineering project scientist at the University of California, San Diego. His research interests include all areas of metacomputing, and in particular theoretical models for the efficient scheduling of distributed application in computational Grid environments. He received his BS from the École Nationale Supérieure d'Électrotechnique, d'Électronique, d'Informatique et d'Hydraulique de Toulouse (ENSEEIHT), his MS from the Université Paul Sabatier, Toulouse, and his PhD from the University of Tennessee, Knoxville.

Arnaud Legrand is a Computer Science graduate student at the École Normale Supérieure, the leading French scientific research and teaching institution, Lyon, France. He is interested in parallelism, metacomputing and numerical simulation and is currently working at the Laboratoire de l'Informatique et du Parallélisme (Parallel Computing Laboratory, ENS Lyon) on linear algebra algorithms that are taylored to heterogeneous environments.

Dmitrii Zagorodnov is currently pursuing a PhD at the Department of Computer Science and Engineering at the University of California, San Diego. He has received B.S. and M.S. degrees in computer science from the University of Alaska Fairbanks in 1995 and 1997, respectively. His current research interest is in fault tolerance for distributed systems.

Francine Berman is a Professor of Computer Science and Engineering at U. C. San Diego, Senior Fellow at the San Diego Supercomputer Center, Fellow of the ACM, and founder of the Parallel Computation Laboratory at UCSD. Her research interests over the last two decades have focused on parallel and distributed computation, and in particular the areas of programming environments, tools, and models that support high-performance computing. She received her B.A. from the University of California, Los Angeles, and her M.S. and Ph.D. from the University of Washington.

Parallel Program Execution on a Heterogeneous PC Cluster Using Task Duplication

YU-KWONG KWOK

Department of Electrical and Electronic Engineering
The University of Hong Kong, Pokfulam Road, Hong Kong

Email: *ykwok@eee.hku.hk*

Abstract†—In this paper, we propose to use a duplication based approach in scheduling tasks to a heterogeneous cluster of PCs. In duplication based scheduling, critical tasks are redundantly scheduled to more than one machine in order to reduce the number of inter-task communication operations. The start times of the succeeding tasks are also reduced. The task duplication process is guided given the system heterogeneity in that the critical tasks are scheduled or replicated in faster machines. The algorithm has been implemented in our prototype program parallelization tool for generating MPI code executable on a cluster of Pentium PCs. Our experiments using three numerical applications have indicated that heterogeneity of PC cluster, being an inevitable feature, is indeed useful for optimizing the execution of parallel programs.

Keywords: Scheduling, task graphs, algorithms, parallel processing, heterogeneous systems, PC cluster computing, task duplication, resource management.

1 Introduction

Recently we have witnessed an increasing interest in employing a network of PCs connected by a high-speed network to tackle many computationally intensive parallel applications [9], [18]. Parallel processing using a cluster of machines, also commonly called *cluster computing*, enables a much larger community of users than ever before to efficiently tackle many difficult optimization problems on a readily available platform [9], [18]. However, realizing the goal of efficient cluster computing entails handling a number of resource management chores [18]. One of the most important problems is the scheduling of tasks. Indeed, to effectively harness the aggregate computing power of such a heterogeneous cluster, it is crucial to judiciously allocate and sequence the tasks on the machines. In a broad sense, the scheduling problem exists in two forms: *dynamic* and *static*. In dynamic scheduling, few assumptions about the parallel program can be made before execution, and thus, scheduling decisions have to be made on-the-fly. The goal of a dynamic scheduling algorithm as such includes not only the minimization of the program completion time but also the minimization of scheduling overhead, which represents a significant portion of the cost paid for running the scheduler. In a cluster of PCs environment, such dynamic scheduling algorithms usually employ the so-called "idle-cycle-stealing" approach [5] which attempts to dynamically balance the work load evenly across all the machines. However, when the objective of scheduling is to minimize the execution time of a parallel application, such dynamic scheduling strategies are not suitable.

On the other hand, the approach of using static scheduling algorithms [11], [12], [22], which can afford to use longer time to generate an optimized schedule off-line, is particularly effective for many scientific applications such as the adaptive simulation of N-body problem, object recognition using iterative image processing algorithms, and

† This research was jointly supported by a research initiation grant from HKU CRCG under contract number 10202518, a research grant from the Hong Kong Research Grants Council under contract number HKU 7124/99E, and a seed funding grant from HKU URC under contract number 10203010.

0-7695-0556-2/00 $10.00 © 2000 IEEE

some other numerical applications [1], [3], [4], [13], [14], [19], [25] because the characteristics of such applications can be determined at compile-time. A parallel program, therefore, can be represented by a directed acyclic task graph [3], in which the node weights represent task processing times and the edge weights represent data dependencies as well as the communication times between tasks [3], [6]. The static scheduling problem is, in general, NP-complete [5], [8] and there have been many heuristics suggested in the literature for scheduling a parallel machine. However, the problem of scheduling tasks to a cluster is a relatively less explored topic. Specifically, there are two difficult research issues to be tackled in the scheduling problem for cluster computing:

1) *Communication overhead*: The communication overhead in a network of PCs is still very significant relative to the processing power of the machines [9]. Thus, to avoid offsetting the gain from parallelization by excessive communication overhead, the tasks should be scheduled in such a manner that the number of communications is kept small.
2) *Heterogeneity*: In a PC cluster, which typically undergoes continual upgrading, heterogeneity in the hardware configuration is unavoidable. Heterogeneity can be a potential problem for some highly regular applications (e.g., some data parallel problems). However, it has been demonstrated that heterogeneity is useful for further enhancing the performance of irregularly structured parallel application [7], [21], by exploiting the affinity of different tasks to different machines.

In this study, we propose to use a *duplication* approach to scheduling the tasks to the cluster. In duplication based scheduling, critical tasks are redundantly scheduled to more than one machines in order to reduce the number of inter-task communication operations. The start times of the succeeding tasks are also reduced. There have been many duplication approaches suggested in the literature [1], [10], [15], [16], [17], [20]. However, all these methods are designed for homogeneous parallel architectures. Furthermore, the previous approaches are all evaluated based on simulations rather than using real applications with a parallelizing compiler. In our proposed approach, the task duplication process is guided by tracking the critical path of the task graph given the system heterogeneity in that the critical tasks are scheduled or replicated in faster machines. Task duplication is indeed particularly effective for heterogeneous systems because the overall completion time of an application is usually determined by a subset of tasks (i.e., the *critical-path*, discussed in detailed in Section 2) which can be scheduled to execute efficiently on the faster machines. We have implemented this duplication based scheduling algorithm in the parallel code generator of a prototype program parallelization tool [2], which generates MPI code executable on a network of Pentium PCs. The system on which we tested our approach is shown schematically in Figure 1. Our experiments using several real applications have demonstrated that the duplication technique is very effective in reducing the completion time of the applications on a heterogeneous cluster of Pentium II PCs connected via a Fore Fast Ethernet switch.

The remainder of this paper is organized as follows. In the next section, we describe in detail the model used and the design considerations of the duplication algorithm. Section 3 includes the results of our performance study. The last section concludes the paper.

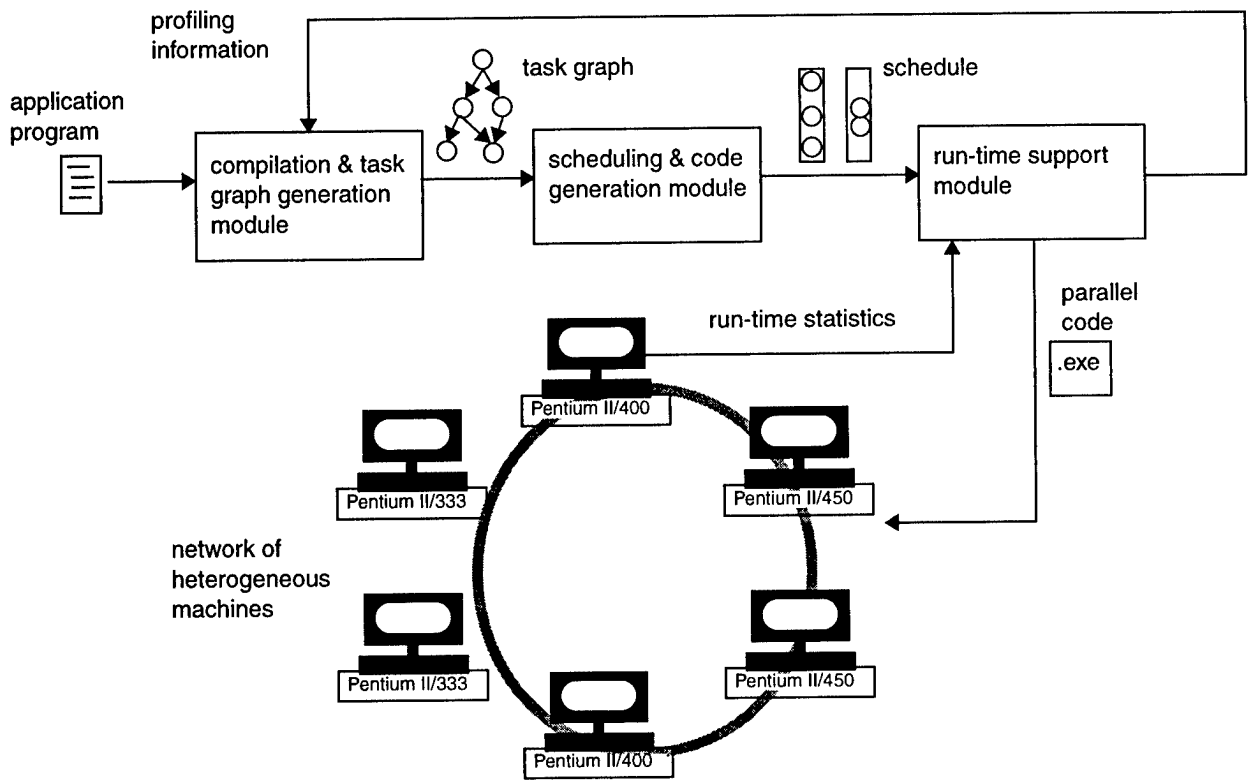


Figure 1: System support for high-performance computing on a heterogeneous cluster.

2 Scheduling for a Heterogeneous PC Cluster

In this section, we first describe our scheduling model, followed by a discussion of the duplication techniques employed in our scheduling module of the parallel code generator.

2.1 The Model

A parallel program is composed of n tasks $\{T_1, T_2, ..., T_n\}$ in which there is a partial order: $T_i < T_j$ implies that T_j cannot start execution until T_i finishes due to the data dependency between them. Thus, a parallel program can be represented by a directed acyclic *task graph* [3]. Parallelism exists among independent tasks—T_i and T_j are said to be independent if neither $T_i < T_j$ nor $T_j < T_i$. Each task T_i is associated with a nominal execution cost τ_i which is the execution time required by T_i on a reference machine in the heterogeneous system. Similarly, a nominal communication cost c_{ij} is associated with the message M_{ij} from T_i to T_j. Assume there are e messages where $(n-1) \le e < n^2$ so that the task graph is a connected graph.

To model heterogeneity of the target system which consists of m processors $\{P_1, P_2, ..., P_m\}$, *heterogeneity factors* are used. For example, if a task T_i is scheduled to a processor P_x, then its actual execution cost is given by $h_{ix}\tau_i$ where h_{ix} is the heterogeneity factor which is determined by measuring the difference in processing capabilities (e.g., speed) of processor P_x and the reference machine with respect to task T_i. Similarly, if a message M_{ij} is scheduled to the communication link L_{xy} between processors P_x and P_y, its actual communication cost is given by $h'_{ijxy}c_{ij}$. An example parallel program graph is shown in Figure 2.

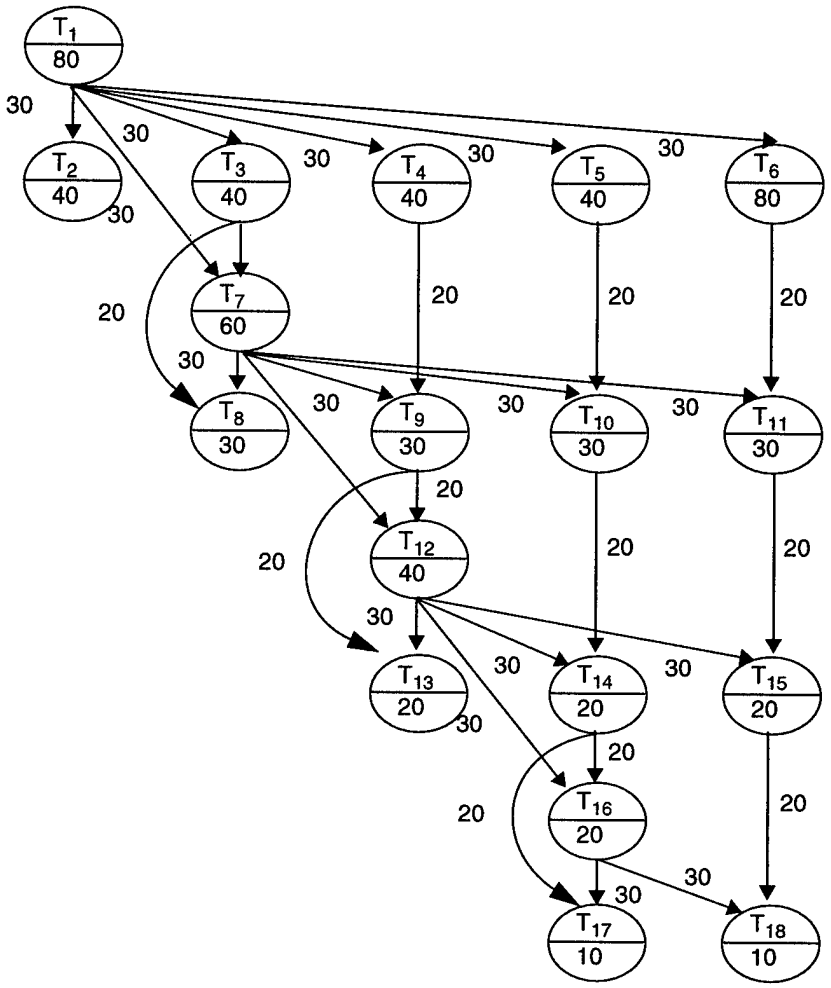


Figure 2: A Gaussian elimination task graph.

The start time and finish time of a message M_{ij} from T_i to T_j on a communication link L_{xy} are denoted by $MST(M_{ij}, L_{xy})$ and $MFT(M_{ij}, L_{xy})$, respectively. Obviously, we have:

$$MFT(M_{ij}, L_{xy}) = MST(M_{ij}, L_{xy}) + h'_{ijxy}c_{ij}$$

The start time of a task T_i on processor P_x is denoted by $ST(T_i, P_x)$ which critically depends on the task's *data ready time* (DRT). The DRT of a task is defined as the latest arrival time of messages from its predecessors. The finish time of a task T_i is given by $FT(T_i, P_x) = ST(T_i, P_x) + h_{ix}\tau_i$. The objective of scheduling is to minimize the maximum FT, which is called the *schedule length* (SL).

2.2 Parallel Code Generation with Duplication Based Scheduling

The proposed duplication scheduler is designed as a core module in the CASCH (Computer-Aided SCHeduling) tool [2]. The system organization of the CASCH tool is shown in Figure 3. It generates a task graph from a sequential program, uses a scheduling algorithm to perform scheduling, and then generates the parallel code in a scheduled form for a cluster of workstations. The timings for the tasks and messages are assigned through a timing database which was obtained through profiling of the basic operations [2], [6]. As soon as the task graph is generated, the duplication based scheduler is invoked.

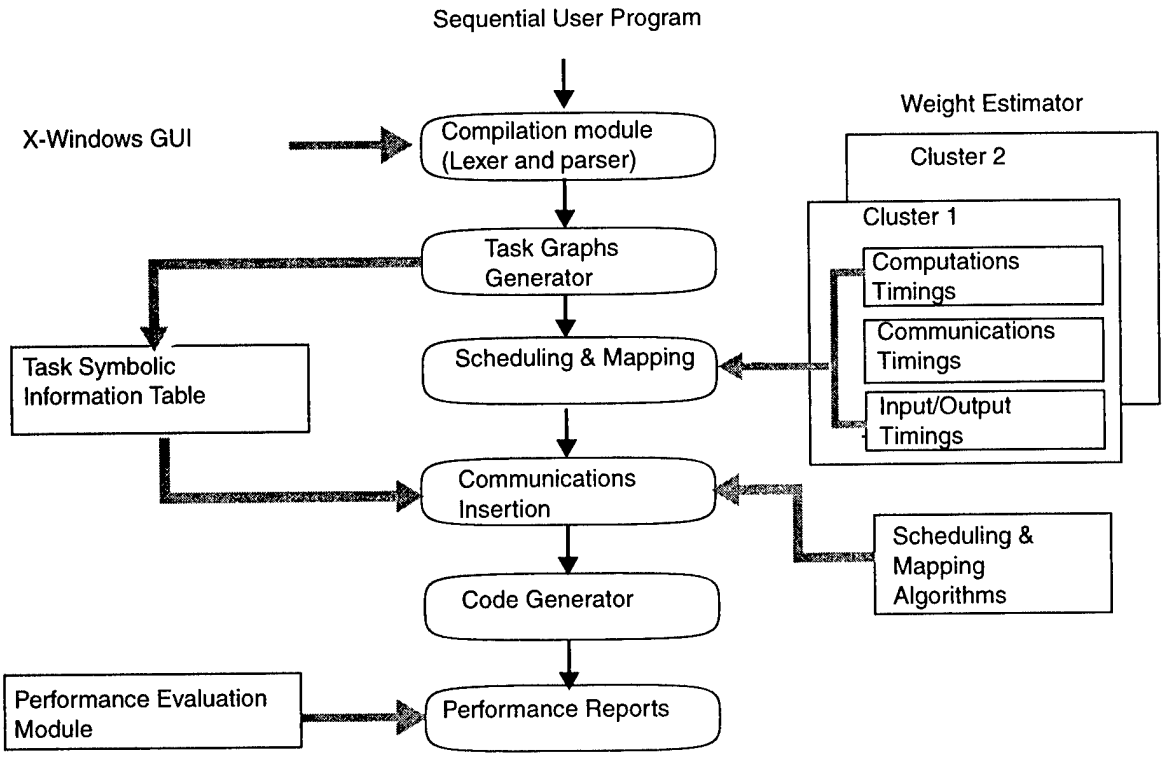


Figure 3: The organization of the CASCH tool.

To minimize the overall execution time of the application on the cluster, the scheduler first determines which tasks are more critical so that they need to be scheduled to start at earlier time slots, possibly by duplicating their ancestors. In a task graph, the *critical-path* (CP), which consists of tasks forming the longest path, is such an important structure because the tasks on the CP potentially determine the overall execution time. To determine whether a task is a CP task, we can use two attributes: *t-level* (top level) and *b-level* (bottom level) [13], [24]. The *b-level* of a task is the length of the longest path beginning with the task. The *t-level* of a task is the length of the longest path reaching the task. Thus, all tasks on the CP have the same value of (*t-level* + *b-level*), which is equal to the length of the CP. Based on this observation, we can easily partition the parallel program into three categories: CP (critical path), IB (in-branch), and OB (out-branch) tasks. The IB tasks are ancestors of CP tasks but are not CP tasks themselves. The OB tasks are neither CP nor IB tasks and as such, are relatively less important. This partitioning can be performed in $O(e)$ time because the *t-level* and *b-level* of all tasks can be computed by using depth-first search. A task with a larger *b-level* implies that it is followed by a longer chain of tasks, and thus, is given a higher priority. A procedure is outlined below for constructing a scheduling list based on the partitioning.

ALOGRITHM 1: CONSTRUCTION OF SCHEDULING LIST

Input: a program task graph with n tasks $\{T_1, T_2, \ldots, T_n\}$

Output: a serial order of the tasks

1. compute the *t-level* and *b-level* of each task by using depth-first search;
2. identify the CP; if there are multiple CPs,

select the one with the largest sum of execution cost and ties are broken randomly;

3. put the CP task which does not have any predecessor to the first position of the serial order;
4. $i \leftarrow 2$; $T_x \leftarrow$ the next CP task
5. while not all the CP tasks are included do
6. if T_x has all its predecessors in the serial order then
7. put T_x at position i and increment i;
8. else let T_y be the predecessor of T_x which is not in the serial order and has the largest *b-level* (ties are broken by choosing the predecessor with a smaller *t-level*);
9. if T_y has all its predecessors in the serial order then put T_y at position i and increment i; otherwise, recursively include all the ancestors of T_y in the serial order such that the tasks with a larger *b-level* are included first;
10. repeat the above step until all the predecessors of T_x are in the serial order;
11. put T_x at position i and increment i;
12. $T_x \leftarrow$ the next CP task ;
13. append all the OB tasks to the serial order in descending order of *b-level*;

Using the above scheduling list, we can determine which tasks have to be considered first in the duplication process. During scheduling, the CP tasks are always considered first. However, we cannot attempt to schedule the CP tasks unless all of their ancestor tasks, which need not be CP tasks themselves, are scheduled. Thus, we use a recursive approach. For each CP task, we first recursively check whether its ancestors are scheduled. If not, then the candidate for scheduling will be changed to the unscheduled ancestor which is at the earliest position on the scheduling list. To actually schedule a task, we try to minimize its finish time by attempting to schedule it to the fastest machine. Duplication is employed for the minimization of finish times in that as many ancestors as possible are inserted before the task. The duplication process will stop when the finish time of the task starts to increase or the time slot has been used up. The order of selecting ancestors for duplication is governed by the scheduling list. The heterogeneity factors h_{ix} are also used for determining the finish times. After all the CP tasks are scheduled (and hence all the IB tasks), the OB tasks are considered for scheduling. To avoid using an excessive number of machines, we attempt to schedule the OB tasks without using duplication. This is useful because the OB tasks usually do not affect the overall completion time and, thus, need not be scheduled to finish as soon as possible. However, if such a conservative approach fails—that is, the overall completion time is increased by scheduling a certain OB task without using duplication, then the same recursive duplication process will be applied to the OB task. The whole duplication based task scheduling process is summarized in Alogrithm 2 below.

ALOGRITHM 2: HETEROGENEOUS DUPLICATION BASED SCHEDULING

Input: a program task graph with n tasks $\{T_1, T_2, \ldots, T_n\}$, a heterogeneous system with m machines $\{P_1, P_2, \ldots, P_m\}$, and the relative speeds of the machines;

Output: a duplication based schedule

1. Construct the scheduling list (use Alogrithm 1);
2. For each CP task, first recursively schedule each of its unscheduled ancestor IB tasks to a machine so that they can finish as soon as possible by trying to duplicate on the machine as many ancestors as the time slot allows (use the heterogeneity factors h_{ix} for determining the finish times); the order of selecting tasks for duplication is governed by the scheduling list; finally apply the same recursive duplication process to the CP task itself;
3. Without using any duplication, schedule each of the remaining tasks (i.e., OB tasks) to the fastest machine provided that the schedule length does not increase; if this fails, employ recursive duplication

technique to schedule the OB task;

To illustrate how the heterogeneity of the machines is exploited, consider in Figure 4 the two schedules of the Gaussian elimination task graph (shown earlier in Figure 1). The schedule on the left is the best schedule without duplication using homogeneous machines. On the right is a schedule using six heterogeneous machines in which P_2 is of the same speed as the machines in the left schedule, while P_0 and P_1 are two times and 1.3 times faster than P_2, respectively. The remaining machines are slower than P_2. We can see that the CP of the task graph is scheduled to the fastest machine P_0. The critical IB tasks, T_4 and T_5, are also scheduled to finish as early as possible on fast machines P_1 and P_2, respectively, by duplicating T_1. The resulting schedule has an overall completion time of 182 units which is significantly smaller than that of the homogeneous schedule without duplication (330 units)†. Due to space limitations, detailed steps of producing the two schedules are not shown.

After a symbolic schedule is generated, the code generator is invoked to actually implement the schedule using the SPMD (Single Program Multiple Data) model [2], [23]. The program statements or procedures constituting a task T_i are allocated to the specified machine P_j for execution using conditional statements checking the ID of the machine, as shown in Figure 5. Data structures associated with a task are also replicated. The output of the code generator is a C program in which MPI communication primitives are inserted. The resulting parallel program is then compiled and executed on the cluster of workstations.

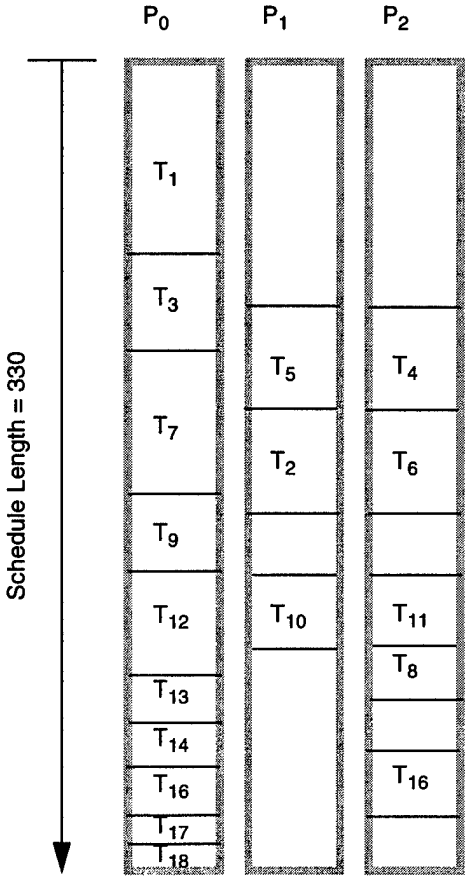
3 Performance Results

We have implemented the duplication based scheduling algorithm in the code generator module of the CASCH tool (see Figure 3), which is executable on a Linux-based Pentium II PC in our cluster. We have parallelized several numerical applications on CASCH. Here, we present and discuss some preliminary results obtained by measuring the execution time of three applications: Gaussian elimination, Gauss-Seidel algorithm, and N-Body problem. By varying the problem sizes (i.e., the dimensions of the matrices in these applications, from 32 to 256) and the granularities (from 1-column block to 8-column block, using 1-D decomposition), we generated four task graphs for each application with roughly 100, 200, 400, and 800 tasks.

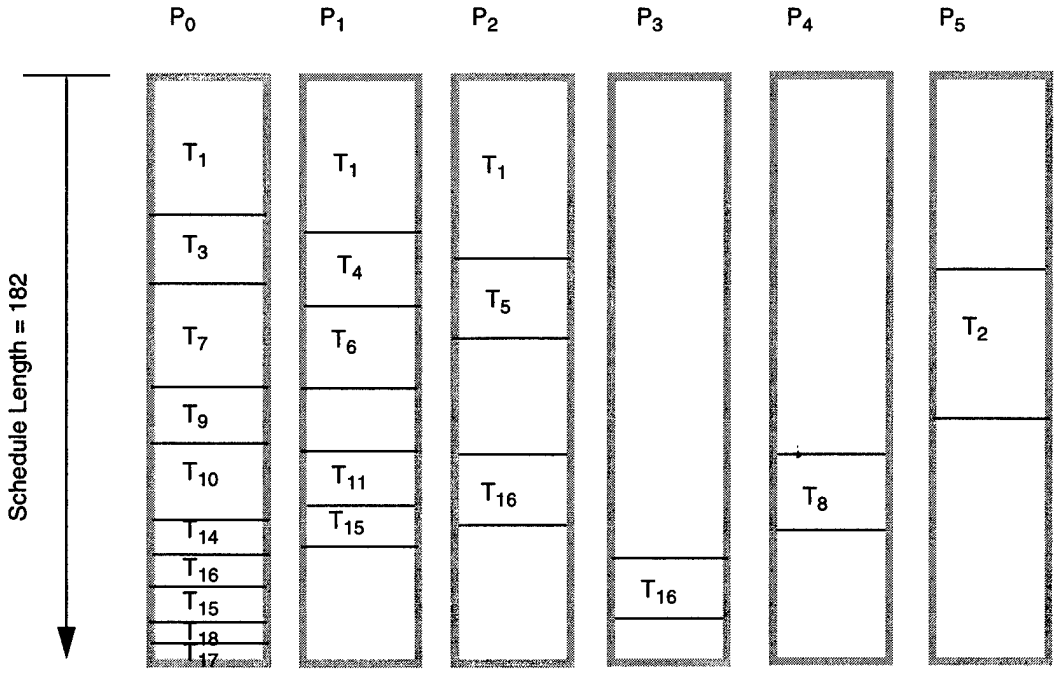
Our heterogeneous cluster consists of twelve PCs: eight Pentium II 333 MHz with 32 MB memory and four Pentium II 450 MHz with 64 MB memory. The PCs are connected by a Fore Fast Ethernet switch. All the experiments were performed using eight PCs but with different configurations: (1) eight homogeneous machines (i.e., all are Pentium II 333 MHz); (2) five Pentium II 333 MHz plus two Pentium II 450 MHz; (3) two Pentium II 333 MHz plus four Pentiume II 450 MHz. The aggregate computing power of the three configurations are approximately the same because we found that a Pentium II 450 MHz is about 1.5 times faster than a Pentium II 333 MHz. The rationale behind selecting these configurations is that we wanted to investigate the benefit of heterogeneity. These configurations are denoted as 8S (for eight slow machines), 2F+5S (two fast plus five slow machines), and 4F+2S (four fast plus two slow machines), respectively. Ten different runs for each size of the three applications were done and the average application execution times were noted.

These average execution times of the three applications are shown in Figure 6. As can be seen,

† In the homogeneous case, the scheduler is also given six machines. However, to arrive at the best schedule shown, it needs only three.



(a) Schedule with homogeneous machines.



(b) Schedule with heterogeneous machines.

Figure 4: The effect of heterogeneity.

```
if (mynode() == j) {
  /* execution of task i */

}
```

Figure 5: SPMD implementation of a schedule.

heterogeneity has a significant impact on the overall execution time of an application in that using more fast machines (albeit the total number of machines is smaller), in general, can speedup the application considerably. The improvement in the Gaussian elimination application is the most remarkable. This can be explained by the fact that the Gaussian elimination graph has a distinctive critical-path (see Figure 2), the tasks on which can be scheduled to the fastest machine. On the other hand, as the Gauss-Seidel task graph has many intersecting critical-paths [23], the duplication approach is less effective in exploiting the advantage of heterogeneity. The improvement of the heterogeneous approaches for the N-Body problem, which has a slightly less regular task graph structure [23], is also considerable.

4 Conclusions and Future Work

In this paper, we have presented a duplication based approach in scheduling tasks to a heterogeneous cluster of PCs. The scheduling algorithm works by recursively duplicating critical tasks to the faster machines in order to minimize the finish times. The algorithm has been implemented in our prototype program parallelization tool for generating MPI code executable on a cluster of Pentium PCs. Our experiments using three numerical applications have indicated that heterogeneity of PCs cluster is indeed useful for optimizing the execution of parallel programs. One important issue related to using a PC cluster is fault-tolerance. Unlike a tightly couple parallel architecture (e.g., the IBM SP2), a PC in a cluster may experience intermittent failure, possibly due to user reboots. Thus, the task schedule has to be fault-tolerant so that the application can finish its execution even in the presence of such faults. We believe that task duplication, augmented with check-pointing and roll-back recovery techniques, is a viable approach to achieve this goal. A performance model is being developed to quantitatively analyze the merits of this approach.

References

[1] I. Ahmad and Y.-K. Kwok, "On Exploiting Task Duplication in Parallel Program Scheduling," *IEEE Trans. Parallel and Distributed Systems*, vol. 9, no. 9, pp. 872-892, Sept. 1998.

[2] I. Ahmad, Y.-K. Kwok, M.-Y. Wu, and W. Shu, "CASCH: A Software Tool for Automatic Parallelization and Scheduling of Programs on Multiprocessors," *IEEE Concurrency*, accepted for publication and scheduled to appear in 2000.

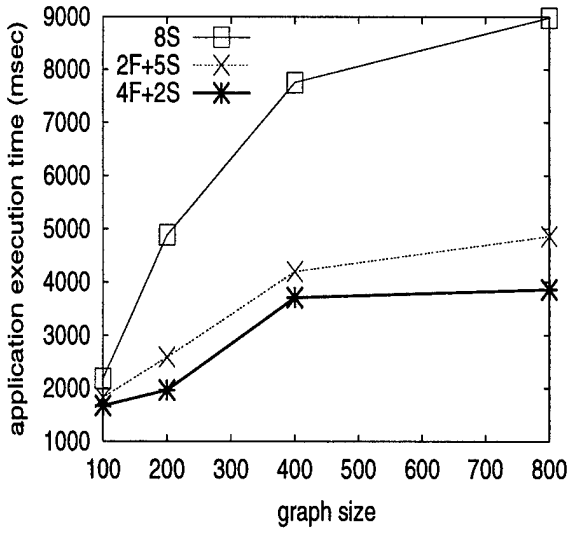
[3] M. Cosnard and M. Loi, "Automatic Task Graphs Generation Techniques," *Parallel Processing Letters*, vol. 5, no. 4, pp. 527-538, Dec. 1995.

[4] M. Cosnard, M. Marrakchi, Y. Robert, and D. Trystam, "Parallel Gaussian Elimination on An MIMD Computer," *Parallel Computing*, vol. 6, pp. 275-296, 1988.

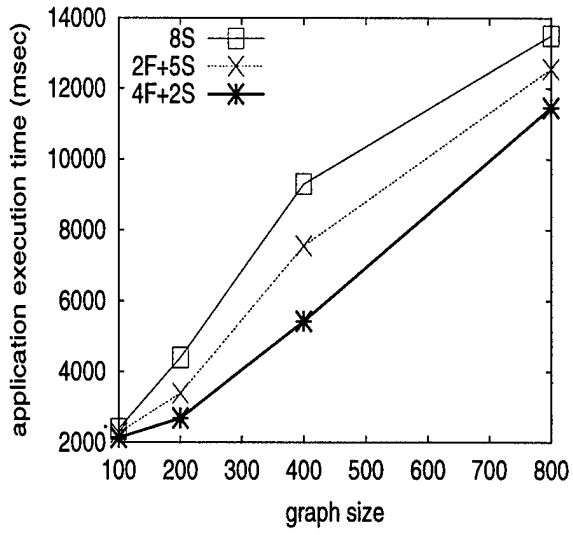
[5] H. El-Rewini, T.G. Lewis, and H.H. Ali, *Task Scheduling in Parallel and Distributed Systems*, Englewood Cliffs, New Jersey: Prentice Hall,1994.

[6] T. Fahringer, "Compile-Time Estimation of Communication Costs for Data Parallel Programs," *J. Parallel and Distributed Computing*, vol. 39, pp. 46-65, 1996.

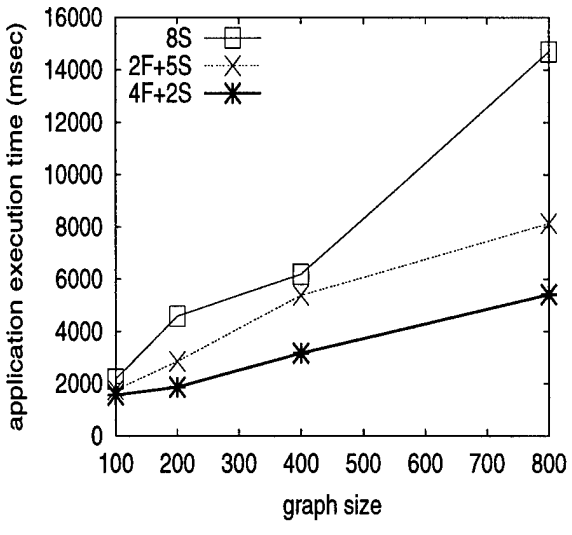
[7] R.F. Freund and H.J. Siegel, "Heterogeneous Processing," *IEEE Computer*, vol. 26, no. 6, pp.



(a) Gaussian elimination



(b) Gauss-Seidel



(c) N-body

Figure 6: The average execution time of the three applications with three different cluster configurations.

13-17, June 1993.

[8] M.R. Garey and D.S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W.H. Freeman and Company, 1979.

[9] K. Hwang, X. Zu, and C.L. Wang, "Viable Approaches to Realizing Single System Image in Multicomputer Clusters," *IEEE Concurrency,* accepted for publication and to appear.

[10]B. Kruatrachue and T.G. Lewis, "Grain Size Determination for Parallel Processing," *IEEE Software*, vol. 5, no. 1, pp. 23-32, Jan. 1988.

[11]Y.-K. Kwok and I. Ahmad, "Dynamic Critical

Path Scheduling: An Effective Technique for Allocating Tasks Graphs to Multiprocessors," *IEEE Trans. Parallel and Distributed Systems*, vol. 7, no. 5, pp. 506-521, May 1996.

[12]—, "Benchmarking and Comparison of the Task Graph Scheduling Algorithms," *Journal of Parallel and Distributed Computing*, accepted for publication and scheduled to appear in 1999.

[13]—, "Static Scheduling Algorithms for Allocating Directed Task Graphs to Multiprocessors," *ACM Computing Surveys*, accepted for publication and to appear in 2000.

[14]M.G. Norman and P. Thanisch, "Models of Machines and Computation for Mapping in Multicomputers," *ACM Computing Surveys*, vol. 25, no. 3, pp. 263-302, Sept. 1993.

[15]M.A. Palis, J.-C. Liou, and D.S.L. Wei, "Task Clustering and Scheduling for Distributed Memory Parallel Architectures," *IEEE Trans. Parallel and Distributed Systems*, vol. 7, no. 1, pp. 46-55, Jan. 1996.

[16]C. Papadimitriou and M. Yannakakis, "Toward an Architecture Independent Analysis of Parallel Algorithms," *SIAM J. Computing*, vol. 19, no. 2, pp. 322-328, Apr. 1990.

[17]G.-L. Park, B. Shirazi, and J. Marquis, "DFRN: A New Approach for Duplication Based Scheduling for Distributed Memory Multiprocessor Systems," *Proc. 11th Int'l Parallel Processing Symposium*, pp. 157-166, Apr. 1997.

[18]G.F. Pfister, *In Search of Clusters*, second edition, Englewood Cliffs, New Jeresy: Prentice Hall, 1998.

[19]V. Sarkar, *Partitioning and Scheduling Parallel Programs for Multiprocessors*, MIT Press, Cambridge, MA, 1989.

[20]B. Shirazi, H. Chen, and J. Marquis, "Comparative Study of Task Duplication Static Scheduling versus Clustering and Non-Clustering Techniques," *Concurrency: Practice and Experience*, vol. 7, no. 5, pp. 371-390, Aug. 1995.

[21]H.J. Siegel, H.G. Dietz, and J.K. Antonio, "Software Support for Heterogeneous Computing," *ACM Computing Surveys*, vol. 28, no. 1, pp. 237-239, Mar. 1996.

[22]G.C. Sih and E.A. Lee, "A Compile-Time Scheduling Heuristic for Interconnection-Constrained Heterogeneous Processor Architectures," *IEEE Trans. Parallel and Distributed Systems*, vol. 4, no. 2, pp. 75-87, Feb. 1993.

[23]M.-Y. Wu and D.D. Gajski, "Hypertool: A Programming Aid for Message-Passing Systems," *IEEE Trans. Parallel and Distributed Systems*, vol. 1, no. 3, pp. 330-343, July 1990.

[24]T. Yang and A. Gerasoulis, "List Scheduling with and without Communication Delays," *Parallel Computing*, vol. 19, no. 12, pp. 1321-1344, Dec. 1993.

[25]A.Y. Zomaya, M. Clements, and S. Olariu, "A Framework for Reinforcement-Based Scheduling in Parallel Processor Systems," *IEEE Trans. Parallel and Distributed Systems*, vol. 9, no. 3, pp. 249-260, Mar. 1998.

Author Biography:

YU-KWONG KWOK received his BSc degree in computer engineering from the University of Hong Kong in 1991, the MPhil and PhD degrees in computer science from the Hong Kong University of Science and Technology in 1994 and 1997, respectively. Currently, he is an assistant professor in the Department of Electrical and Electronic Engineering at the University of Hong Kong. Before joining the University of Hong Kong, he was a visiting scholar for one year in the parallel processing laboratory at the School of Electrical and Computer Engineering at Purdue University. His research interests include mobile computing, wireless networking, software support for parallel and distributed computing, heterogeneous cluster computing, and distributed multimedia systems. He is a member of the IEEE Computer Society and the ACM. For more information about Kwok, visit `http://www.eee.hku.hk/~ykwok`.

Segmented Min-Min: A Static Mapping Algorithm for Meta-tasks on Heterogeneous Computing Systems

Min-You Wu and Wei Shu
Department of Electrical and Computer Engineering
University of New Mexico

Hong Zhang
Department of Electrical and Computer Engineering
University of Central Florida

Abstract

The Min-min algorithm is a simple algorithm. It runs fast and delivers good performance. However, the Min-min algorithm schedules small tasks first, resulting in some load imbalance. In this paper, we present an algorithm which improves the Min-min algorithm by scheduling large tasks first. The new algorithm, Segmented min-min, balances the load well and demonstrates even better performance in both makespan and running time.

1. Introduction

A heterogeneous computing environment utilizes a suite of different machines interconnected by high-speed networks to execute different computationally intensive applications that have diverse computational requirements [8, 12, 13]. The general problem of mapping tasks to machines has been shown to be NP-complete [10]. Many useful heuristics to perform this mapping function have been developed. Among many sophisticated algorithms, the Min-min algorithm [10] is a simple algorithm which runs fast and delivers satisfactory performance. It selects from all tasks the task that minimizes the completion time on a machine. In most situations, it maps as many tasks as possible to their first choice of machine. However, the Min-min algorithm is unable to balance the load well since it usually schedules small tasks first. In this paper, we propose a simple alternative of the Min-min algorithm by scheduling large tasks first. The proposed algorithm retains the advantage of the Min-min algorithm and achieves good load balance at the same time.

This paper presents the new algorithm, named the *Segmented min-min* algorithm. In section 2, previous heuristic algorithms are reviewed. Section 3 presents the new algorithm. Section 4 exhibits the simulation model and experimental results. Section 5 concludes the paper.

2. Previous Heuristics

In this section, we review a set of heuristic algorithms which schedule meta-tasks to heterogeneous computing systems. A meta-task is defined as a collection of independent tasks with no data dependences. Meta-tasks are mapped onto machines statically; each machine executes a single task at a time. For static mapping, it is assumed that the number of tasks, t, and the number of machines, m, are known a priori.

A large number of heuristic algorithms have been designed to schedule tasks to machines on heterogeneous computing systems. In [2], eleven commonly used algorithms have been evaluated, listed as follows.

OLB : Opportunistic Load Balancing (OLB) assigns each task, in arbitrary order, to the next available machine [1, 7, 8].

UDA : User-Directed Assignment (UDA) assigns each task, in arbitrary order, to the machine with the *best expected execution time* for the task [1, 7].

Fast Greedy : Fast Greedy assigns each task, in arbitrary order, to the machine with the *minimum completion time* for that task [1].

Min-min : In Min-min, the minimum completion time for each task is computed respect to all machines. The task with the *overall minimum completion time* is selected and assigned to the corresponding machine. The newly mapped task is removed, and the process repeats until all tasks are mapped [1, 7, 10].

0-7695-0556-2/00 $10.00 © 2000 IEEE

Max-min : The Max-min heuristic is very similar to the Min-min algorithm. The set of minimum completion times is calculated for every task. The task with *overall maximum completion time* from the set is selected and assigned to the corresponding machine [1, 7, 10].

Greedy : The Greedy heuristic is literally a combination of the Min-min and Max-min heuristics by using the better solution [1, 7].

GA : The Genetic algorithm (GA) is used for searching large solution space. It operates on a population of chromosomes for a given problem. The initial population is generated randomly. A chromosome could be generated by any other heuristic algorithm. When it is generated by Min-min, it is called "seeding" the population with Min-min [15, 14].

SA : Simulated Annealing (SA) is an iterative technique that considers only one possible solution for each meta-task at a time. SA uses a procedure that probabilistically allows solution to be accepted to attempt to obtain a better search of the solution space based on a system temperature [5, 11].

GSA : The Genetic Simulated Annealing (GSA) heuristic is a combination of the GA and SA techniques [3].

Tabu : Tabu search is a solution space search that keeps track of the regions of the solution space which have already been searched so as not to repeat a search near these areas [6, 9].

A* : A* is a tree search beginning at a root node that is usually a null solution. As the tree grows, intermediate nodes represent partial solutions and leaf nodes represent final solutions. Each node has a cost function, and the node with the minimum cost function is replaced by its children. Any time a node is added, the tree is pruned by deleting the node with the largest cost function. This process continues until a complete mapping (a leaf node) is reached [4].

The experimental results from [2] show that OLB, UDA, Max-min, SA, GSA, and Tabu do not produce good schedules in general. Min-min, GA, and A* are able to deliver good performance. The difference between the completion times of the schedules (makespans) generated by these three algorithms is within 10%. GA is consistently better than Min-min by a few percents, since it is seeding the population with a Min-min chromosome. A*, on the other hand, produces better or worse schedules than Min-min and GA in different situations. Among the three algorithms, Min-min is the fastest algorithm, GA is much slower, and A* is very slow. For 512 tasks and 16 machines, the running time of Min-min is about 1 second, GA 30 seconds, and A* 1200 seconds [2].

Min-min is a simple algorithm, fast, and able to deliver good performance. Even GA has to be "seeding" the population with a Min-min chromosome to obtain its good performance. Min-min schedules the "best case" tasks first and generates relatively good schedules. The drawback of Min-min is that it assigns the small task first. Thus, the smaller tasks would execute first and then a few larger tasks execute while several machines sit idle, resulting in poor machine utilization. We propose a simple method to enforce large tasks to be scheduled first. Tasks are partitioned into segments according to their execution times. The segment with larger tasks is scheduled first with the Min-min algorithm being applied within the segment. This is called *Segmented min-min (Smm).*

3. The Segmented Min-Min Algorithm

Every task has a *ETC (expected time to compute)* on a specific machine. If there are t tasks and m machines, we can obtain a $t \times m$ *ETC matrix*. $ETC(i, j)$ is the estimated execution time for task i on machine j.

The *Segmented min-min* algorithm sorts the tasks according to ETCs. The tasks can be sorted into an ordered list by the average ETC, the minimum ETC, or the maximum ETC. Then, the task list is partitioned into segments with the equal size. The segment of larger tasks is scheduled first and the segment of smaller tasks last. For each segment, Min-min is applied to assign tasks to machines. The algorithm is described as follows.

Segmented min-min (Smm)

1. Compute the sorting key for each task:

SUB-POLICY 1 — *Smm-avg*: Compute the average value of each row in ETC matrix

$$key_i = \sum_j ETC(i,j)/m.$$

SUB-POLICY 2 — *Smm-min*: Compute the minimum value of each row in ETC matrix

$$key_i = \min_j ETC(i,j).$$

SUB-POLICY 3 — *Smm-max*: Compute the maximum value of each row in ETC matrix

$$key_i = \max_j ETC(i,j).$$

2. Sort the tasks into a task list in decreasing order of their keys.
3. Partition the tasks evenly into N segments.
4. Schedule each segment in order by applying Min-min.

Different from the Min-min algorithm, Segmented min-min performs task sorting before scheduling. Sorting implies that larger tasks are promoted to be scheduled earlier. Then, Min-min is applied locally within each segment. The problem here is how to define the sorting key. Tasks with long execution time deserve promotion to early scheduling. However, in a heterogeneous system, the execution time of a task varies in different machines. Therefore, we test three sub-policies by defining the execution time of a task as the average, the minimum, or the maximum of its ETCs.

The third step of the Segmented min-min algorithm partitions tasks into N segments. Determining the optimal value of N is a trade-off. More segments result in better load balance. On the other hand, too many segments will lose advantages of the Min-min algorithm. Intuitively, as long as we partition the tasks into a few segments, such as large, medium, and small tasks, the load can be balanced fairly well. Experimental results confirm this as shown in Figure 1 where the curves show the improvement of *Smm-avg* over Min-min for different values of N. Each point in these curves is the average of five runs. In general, the optimal value of N is relevant to the ratio $c = \frac{t}{m}$. When c is large, Min-min performs well. For small c, which means the number of tasks per machine is not large, the optimal value of N is about 4 or 5. Therefore, we fix the value of N to 4, which means that we always partition the tasks into four segments.

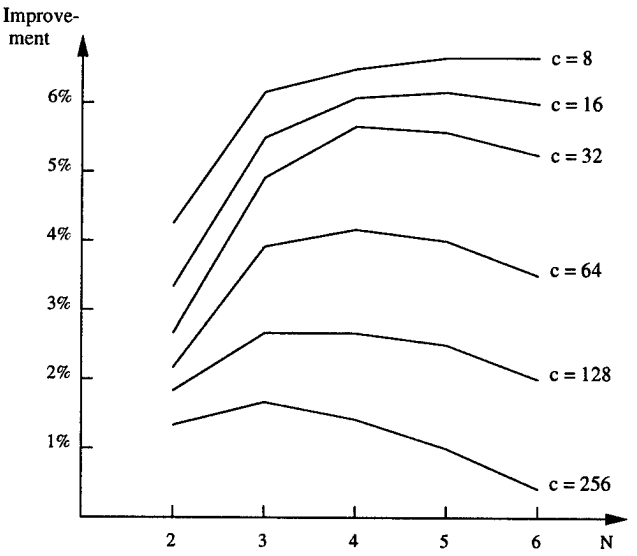


Figure 1. The N Value.

4. Experiments

4.1. Performance Comparison

For the experimental studies, we use the same method in [2] to generate the test set. The parameters include *Consistant, Inconsistant,* or *Semi-Consistant*; *High* or *Low Task Heterogeneity*; and *High* or *Low Machine Heterogeneity*. For details, see [2]. All experiment results are based on 512 tasks, 16 or 32 machines, 100 trails and N = 4. The results for 16 machines are shown in Tables I to XII and that for 32 machines are shown in Tables XIII to XXIV. In these tables, the second column shows the utilization of machines which is defined as $1 - \frac{\sum idle\ time}{m \times makespan}$. The third column is the makespan (the completion time) of schedules. The fourth column is the improvement of each Segmented min-min algorithm over the Max-min algorithm and the fifth column is that over the Min-min algorithm. The last column shows the running time of each algorithm.

4.2. Discussion

From these results, we found that the Segmented min-min algorithm is able to balance the load very well compared to the Max-min and the Min-min algorithms. The system utilization of Min-min is relatively low while that of Segmented min-min is very high. This is because Segmented min-min schedules larger tasks first and smaller tasks can run in parallel with large tasks. Although the Max-min algorithm produces very good load balancing, it does not schedule tasks to their "best case." Thus, its performance is far worse than that of the Segmented min-min algorithm. Higher system utilization makes three Segmented min-min algorithms better than Min-min in almost all cases. *Smm-avg* enhances the performance of Min-min from 2% to 12%. *Smm-min* shows better performance than *Smm-avg* in some cases but is worse than *Smm-avg* in most cases. *Smm-max* is worse than Smm-avg in almost all cases. Thus, we use *Smm-avg* for the Segmented min-min algorithm, which improves the Min-min algorithm by 6.1% in average.

In addition, the running time of the Segmented min-min algorithm is much less than Min-min. This is not difficult to explain because Min-min spends the large amount of time to search entire matrix to map one task each time, while Segmented min-min, taking advantage of the divide-and-conquer strategy, only searches the minimum value within a single partition. In summary, this partitioning method improves the makespan and running time simultaneously.

Table I. 16 Machines, Inconsistent, Low Task, Low Machine Heterogeneity

Algorithm	System Utilization	Makespan ($\times 10^3$ Sec.)	Improvement over Max-min	Improvement over Min-min	Running Time (Sec.)
Max-min	99.9%	5.425	–	–	1.19
Min-min	91.0%	2.915	–	–	1.06
Smm-avg	98.1%	2.767	96.0%	5.3%	0.33
Smm-min	98.4%	2.746	96.3%	6.1%	0.33
Smm-max	97.8%	2.784	94.9%	4.7%	0.33

Table II. 16 Machines, Inconsistent, Low Task, High Machine Heterogeneity

Algorithm	System Utilization	Makespan ($\times 10^5$ Sec.)	Improvement over Max-min	Improvement over Min-min	Running Time (Sec.)
Max-min	99.8%	2.513	–	–	1.19
Min-min	83.3%	1.214	–	–	1.06
Smm-avg	96.9%	1.113	125.8%	9.1%	0.33
Smm-min	98.2%	1.064	136.2%	14.2%	0.33
Smm-max	95.9%	1.135	121.4%	7.0%	0.33

Table III. 16 Machines, Inconsistent, High Task, Low Machine Heterogeneity

Algorithm	System Utilization	Makespan ($\times 10^4$ Sec.)	Improvement over Max-min	Improvement over Min-min	Running Time (Sec.)
Max-min	99.9%	15.943	–	–	1.20
Min-min	91.0%	8.588	–	–	1.07
Smm-avg	98.2%	8.139	95.9%	5.5%	0.33
Smm-min	98.5%	8.087	97.1%	6.2%	0.33
Smm-max	97.9%	8.190	94.7%	4.8%	0.33

Table IV. 16 Machines, Inconsistent, High Task, High Machine Heterogeneity

Algorithm	System Utilization	Makespan ($\times 10^6$ Sec.)	Improvement over Max-min	Improvement over Min-min	Running Time (Sec.)
Max-min	99.8%	7.375	–	–	1.20
Min-min	83.4%	3.573	–	–	1.07
Smm-avg	96.8%	3.279	124.9%	8.9%	0.33
Smm-min	98.3%	3.131	135.5%	14.1%	0.33
Smm-max	95.9%	3.344	125.5%	6.9%	0.33

Table V. 16 Machines, Consistent, Low Task, Low Machine Heterogeneity

Algorithm	System Utilization	Makespan ($\times 10^3$ Sec.)	Improvement over Max-min	Improvement over Min-min	Running Time (Sec.)
Max-min	99.9%	7.415	–	–	1.22
Min-min	94.0%	5.857	–	–	1.07
Smm-avg	98.6%	5.705	30.0%	2.7%	0.33
Smm-min	98.2%	5.813	27.6%	0.7%	0.33
Smm-max	98.4%	5.749	29.0%	1.9%	0.33

Table VI. 16 Machines, Consistent, Low Task, High Machine Heterogeneity

Algorithm	System Utilization	Makespan ($\times 10^5$ Sec.)	Improvement over Max-min	Improvement over Min-min	Running Time (Sec.)
Max-min	99.9%	4.125	–	–	1.23
Min-min	89.0%	2.866	–	–	1.07
Smm-avg	97.7%	2.805	47.1%	2.1%	0.33
Smm-min	96.7%	2.910	42.3%	-2.0%	0.33
Smm-max	97.2%	2.867	43.9%	0.0%	0.33

Table VII. 16 Machines, Consistent, High Task, Low Machine Heterogeneity

Algorithm	System Utilization	Makespan ($\times 10^5$ Sec.)	Improvement over Max-min	Improvement over Min-min	Running Time (Sec.)
Max-min	100.0%	2.181	–	–	1.24
Min-min	93.9%	1.725	–	–	1.08
Smm-avg	98.6%	1.679	29.9%	2.8%	0.33
Smm-min	98.2%	1.710	27.5%	0.9%	0.33
Smm-max	98.4%	1.693	28.8%	1.9%	0.33

Table VIII. 16 Machines, Consistent, High Task, High Machine Heterogeneity

Algorithm	System Utilization	Makespan ($\times 10^6$ Sec.)	Improvement over Max-min	Improvement over Min-min	Running Time (Sec.)
Max-min	99.9%	12.152	–	–	1.24
Min-min	88.9%	8.437	–	–	1.07
Smm-avg	97.7%	8.258	47.2%	2.2%	0.33
Smm-min	96.7%	8.564	41.9%	-1.5%	0.33
Smm-max	97.4%	8.430	44.2%	0.0%	0.33

Table IX. 16 Machines, Semi-Consistent, Low Task, Low Machine Heterogeneity

Algorithm	System Utilization	Makespan ($\times 10^3$ Sec.)	Improvement over Max-min	Improvement over Min-min	Running Time (Sec.)
Max-min	99.9%	6.339	–	–	1.21
Min-min	91.8%	3.745	–	–	1.07
Smm-avg	98.2%	3.595	76.3%	4.2%	0.33
Smm-min	98.1%	3.624	74.9%	3.3%	0.33
Smm-max	98.0%	3.624	74.9%	3.3%	0.33

Table X. 16 Machines, Semi-Consistent, Low Task, High Machine Heterogeneity

Algorithm	System Utilization	Makespan ($\times 10^5$ Sec.)	Improvement over Max-min	Improvement over Min-min	Running Time (Sec.)
Max-min	99.8%	3.199	–	–	1.21
Min-min	84.4%	1.664	–	–	1.07
Smm-avg	96.8%	1.569	103.9%	6.1%	0.33
Smm-min	96.5%	1.593	100.8%	4.5%	0.33
Smm-max	96.3%	1.590	101.2%	4.6%	0.33

Table XI. 16 Machines, Semi-Consistent, High Task, Low Machine Heterogeneity

Algorithm	System Utilization	Makespan ($\times 10^5$ Sec.)	Improvement over Max-min	Improvement over Min-min	Running Time (Sec.)
Max-min	99.9%	1.862	–	–	1.21
Min-min	91.7%	1.104	–	–	1.07
Smm-avg	98.2%	1.058	76.0%	4.4%	0.33
Smm-min	98.1%	1.066	74.7%	3.5%	0.33
Smm-max	98.0%	1.067	74.5%	3.4%	0.33

Table XII. 16 Machines, Semi-Consistent, High Task, High Machine Heterogeneity

Algorithm	System Utilization	Makespan ($\times 10^6$ Sec.)	Improvement over Max-min	Improvement over Min-min	Running Time (Sec.)
Max-min	99.9%	9.370	–	–	1.21
Min-min	84.6%	4.882	–	–	1.07
Smm-avg	96.9%	4.619	102.9%	5.7%	0.33
Smm-min	96.6%	4.693	99.7%	4.0%	0.33
Smm-max	96.5%	4.673	100.5%	4.7%	0.33

Table XIII. 32 Machines, Inconsistent, Low Task, Low Machine Heterogeneity

Algorithm	System Utilization	Makespan ($\times 10^3$ Sec.)	Improvement over Max-min	Improvement over Min-min	Running Time (Sec.)
Max-min	99.5%	1.954	–	–	2.23
Min-min	85.1%	1.294	–	–	2.16
Smm-avg	93.1%	1.199	63.0%	7.9%	1.16
Smm-min	93.9%	1.188	64.5%	8.9%	1.10
Smm-max	92.6%	1.206	62.0%	7.3%	1.10

Table XIV. 32 Machines, Inconsistent, Low Task, High Machine Heterogeneity

Algorithm	System Utilization	Makespan ($\times 10^4$ Sec.)	Improvement over Max-min	Improvement over Min-min	Running Time (Sec.)
Max-min	98.8%	6.395	–	–	2.24
Min-min	68.0%	3.959	–	–	2.16
Smm-avg	81.8%	3.523	81.5%	12.4%	1.16
Smm-min	79.3%	3.678	73.9%	7.6%	1.10
Smm-max	82.1%	3.502	82.6%	13.0%	1.10

Table XV. 32 Machines, Inconsistent, High Task, Low Machine Heterogeneity

Algorithm	System Utilization	Makespan ($\times 10^4$ Sec.)	Improvement over Max-min	Improvement over Min-min	Running Time (Sec.)
Max-min	99.5%	5.755	–	–	2.23
Min-min	85.2%	3.804	–	–	2.16
Smm-avg	93.2%	3.525	63.3%	7.9%	1.16
Smm-min	93.9%	3.498	64.5%	8.7%	1.10
Smm-max	92.4%	3.556	61.8%	7.0%	1.10

Table XVI. 32 Machines, Inconsistent, High Task, High Machine Heterogeneity

Algorithm	System Utilization	Makespan ($\times 10^6$ Sec.)	Improvement over Max-min	Improvement over Min-min	Running Time (Sec.)
Max-min	99.8%	1.882	–	–	2.21
Min-min	67.9%	1.167	–	–	2.16
Smm-avg	81.7%	1.038	74.3%	12.4%	1.16
Smm-min	79.5%	1.079	74.4%	8.2%	1.09
Smm-max	81.2%	1.044	80.3%	11.8%	1.09

Table XVII. 32 Machines, Consistent, Low Task, Low Machine Heterogeneity

Algorithm	System Utilization	Makespan ($\times 10^3$ Sec.)	Improvement over Max-min	Improvement over Min-min	Running Time (Sec.)
Max-min	99.8%	3.502	–	–	2.26
Min-min	88.4%	3.129	–	–	2.18
Smm-avg	94.8%	2.982	17.4%	4.9%	1.17
Smm-min	93.4%	3.025	15.8%	3.4%	1.09
Smm-max	94.1%	3.005	16.5%	4.1%	1.09

Table XVIII. 32 Machines, Consistent, Low Task, High Machine Heterogeneity

Algorithm	System Utilization	Makespan ($\times 10^5$ Sec.)	Improvement over Max-min	Improvement over Min-min	Running Time (Sec.)
Max-min	99.7%	1.707	–	–	2.27
Min-min	76.4%	1.296	–	–	2.18
Smm-avg	89.3%	1.245	37.1%	4.1%	1.17
Smm-min	87.0%	1.279	33.5%	1.3%	1.10
Smm-max	87.9%	1.260	35.5%	2.9%	1.09

Table XIX. 32 Machines, Consistent, High Task, Low Machine Heterogeneity

Algorithm	System Utilization	Makespan ($\times 10^4$ Sec.)	Improvement over Max-min	Improvement over Min-min	Running Time (Sec.)
Max-min	99.9%	10.305	–	–	2.27
Min-min	88.5%	9.196	–	–	2.19
Smm-avg	94.8%	8.775	17.4%	4.8%	1.18
Smm-min	93.6%	8.887	16.6%	3.5%	1.10
Smm-max	94.1%	8.849	16.5%	3.9%	1.09

Table XX. 32 Machines, Consistent, High Task, High Machine Heterogeneity

Algorithm	System Utilization	Makespan ($\times 10^6$ Sec.)	Improvement over Max-min	Improvement over Min-min	Running Time (Sec.)
Max-min	99.8%	5.016	–	–	2.26
Min-min	76.5%	3.814	–	–	2.18
Smm-avg	89.3%	3.668	36.8%	4.0%	1.18
Smm-min	87.0%	3.768	33.1%	1.2%	1.09
Smm-max	87.9%	3.717	34.9%	2.6%	1.09

Table XXI. 32 Machines, Semi-Consistent, Low Task, Low Machine Heterogeneity

Algorithm	System Utilization	Makespan ($\times 10^3$ Sec.)	Improvement over Max-min	Improvement over Min-min	Running Time (Sec.)
Max-min	99.6%	2.586	–	–	2.23
Min-min	85.1%	1.773	–	–	2.19
Smm-avg	92.8%	1.674	54.5%	5.9%	1.17
Smm-min	92.2%	1.679	54.0%	5.6%	1.09
Smm-max	92.3%	1.683	53.7%	5.3%	1.09

Table XXII. 32 Machines, Semi-Consistent, Low Task, High Machine Heterogeneity

Algorithm	System Utilization	Makespan ($\times 10^4$ Sec.)	Improvement over Max-min	Improvement over Min-min	Running Time (Sec.)
Max-min	99.3%	10.230	–	–	2.22
Min-min	66.4%	6.121	–	–	2.20
Smm-avg	84.3%	5.604	82.5%	9.2%	1.19
Smm-min	84.6%	5.714	79.0%	7.1%	1.09
Smm-max	82.6%	5.682	80.0%	7.7%	1.09

Table XXIII. 32 Machines, Semi-Consistent, High Task, Low Machine Heterogeneity

Algorithm	System Utilization	Makespan ($\times 10^4$ Sec.)	Improvement over Max-min	Improvement over Min-min	Running Time (Sec.)
Max-min	99.7%	7.603	–	–	2.23
Min-min	85.1%	5.226	–	–	2.20
Smm-avg	92.8%	4.925	54.3%	6.1%	1.19
Smm-min	92.4%	4.937	54.2%	5.9%	1.10
Smm-max	92.1%	4.967	53.1%	5.2%	1.10

Table XXIV. 32 Machines, Semi-Consistent, High Task, High Machine Heterogeneity

Algorithm	System Utilization	Makespan ($\times 10^6$ Sec.)	Improvement over Max-min	Improvement over Min-min	Running Time (Sec.)
Max-min	99.3%	3.012	–	–	2.22
Min-min	66.3%	1.797	–	–	2.19
Smm-avg	84.1%	1.645	83.1%	9.2%	1.18
Smm-min	84.5%	1.682	79.0%	6.8%	1.09
Smm-max	82.8%	1.674	80.0%	7.3%	1.10

5. Concluding Remarks

The Segmented min-min algorithm starts from a set of large tasks while Min-min starting from small tasks. *Smm* can balance the load very well and runs faster. We will compare it in the near future to the Genetic algorithm that delivered the best performance among eleven selected algorithms.

Acknowledgments

The authors would like to thank the anonymous reviewers for their thorough comments which caused us to improve the presentation and level of detail. This research was partially supported by NSF grants CCR-9505300 and CCR-9625784.

References

[1] R. Armstrong, D. Hensgen, and T. Kidd. The relative performance of various mapping algorithms is independent of sizable variances in run-time predictions. In *7th IEEE Heterogeneous Computing Workshop (HCW '98)*, pages 79–87, Mar. 1998.

[2] T. Braun, H. Siegel, N. Beck, L. Boloni, M. Maheswaran, A. Reuther, J. Robertson, M. Theys, B.Yao, D. Hensgen, and R. Freund. A comparison study of static mapping heuristics for a class of meta-tasks on heterogeneous computing systems. In *8th IEEE Heterogeneous Computing Workshop (HCW '99)*, pages 15–29, Apr. 1999.

[3] H. Chen, N. S. Flann, and D. W. Watson. Parallel genetic simulated annealing: a massively parallel SIMD approach. *IEEE Transactions on Parallel and Distributed Computing*, 9(2):126–136, Feb. 1998.

[4] K. Chow and B. Liu. On mapping signal processing algorithms to a heterogeneous multiprocessor system. In *ICASSP 91*, pages 1585–1588, May 1991.

[5] M. Coli and P. Palazzari. Real time pipelined system design through simulated annealing. *Journal of Systems Architecture*, 42(6-7):465–475, Dec. 1996.

[6] I. D. Falco, R. D. Balio, E.Tarantino, and R. Vaccaro. Improving search by incorporating evolution principles in parallel tabu search. In *IEEE Conference on Evolutionary Computation*, pages 823–828, 1994.

[7] R. Freund, M. Gherrity, S. Ambrosius, M. Campbell, M. Halderman, D. Hensgen, E. Keith, T. Kidd, M. Kussow, J. Lima, F. Mirabile, L. Moore, B. Rust, and H. Siegel. Scheduling resources in multi-user, heterogeneous, computing environments with SmartNet. In *7th IEEE Heterogeneous Computing Workshop (HCW '98)*, pages 184–199, Mar. 1998.

[8] R. F. Freund and H. J. Siegel. Heterogeneous processing. *IEEE Computer*, 26(6):13–17, June 1993.

[9] F. Glover and M. Laguna. *Tabu Search*. Kluwer Academic Publishers, 1997.

[10] O. Ibarra and C. Kim. Heuristic algorithms for scheduling independent tasks on nonidentical processors. *Journal of the ACM*, 77(2):280–289, Apr. 1977.

[11] S. Kirkpatrick, J. C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, May 1983.

[12] M. Maheswaran, T. D. Braun, and H. J. Siegel. *Encyclopedia of Electrical and Electronics Engineering*, chapter Heterogeneous Distributed Computing. John wiley & sons, 1999.

[13] H. J. Siegel, H. G. Dietz, and J. K. Antonio. *The Computer Science and Engineering Handbook*, chapter Software support for heterogeneous computing, pages 1886–1909. CRC Press, 1997.

[14] H. Singh and A. Youssef. Mapping and scheduling heterogeneous task graphs using genetic algorithms. In *5th IEEE Heterogeneous Computing Workshop (HCW '96)*, pages 86–97, Apr. 1996.

[15] L. Wang, H. J. Siegel, V. P. Roychowdhury, and A. A. Maciejewski. Task matching and scheduling in heterogeneous computing environments using a genetic-algorithm-based approach. *Journal of Parallel and Distributed Computing*, 47(1):1–15, Nov. 1997.

Biographies

Min-You Wu is an Associate Professor in the Department of Electrical and Computer Engineering at the University of New Mexico. He received the M.S. degree from the Graduate School of Academia Sinica, Beijing, China, and the Ph.D. degree from Santa Clara University, California. He has held various positions at University of Illinois at Urbana-Champaign, University of California at Irvine, Yale University, Syracuse University, State University of New York at Buffalo, and University of Central Florida. His research interests include parallel and distributed systems, compilers for parallel computers, programming tools, VLSI design, and multimedia systems. He has published over 80 journal and conference papers in the above areas and edited two special issues on parallel operating systems. He is a senior member of IEEE and a member of ACM. He is listed in International Who's Who of Information Technology and Who's Who in America.

Wei Shu received the Ph.D. degree from the University of Illinois at Urbana-Champaign in 1990. Since then, she worked at Yale University, the State University of New York at Buffalo, and University of Central Florida. She is currently an Associate Professor in the Department of Electrical and Computer Engineering, University of New Mexico. Her current interests include dynamic scheduling, resource management, runtime support systems for parallel and distributed processing, multimedia networking, and operating system support for large-scale distributed simulation. She

is a senior member of IEEE and a member of ACM.

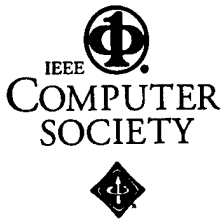
Hong Zhang is a graduate student and Teaching Assistant in the Department of Electrical and Computer Engineering at the University of Central Florida. She received her Bachelor of Science degree in Electrical Engineering from Zhejiang University, Hangzhou, P.R. China, in 1986. She worked as a software engineer in the Computer Center of the Institute of High Energy Physics from 1987-1997, mainly engaged in design and maintenance of the computer system. Her research interests include distributed algorithms, computer networks and database management. She will get her master degree in May, 2000.

AUTHOR INDEX

NOTES

NOTES

NOTES



Press Activities Board

Vice President and Chair:
Carl K. Chang
Dept. of EECS (M/C 154)
The University of Illinois at Chicago
851 South Morgan Street
Chicago, IL 60607
ckchang@eecs.uic.edu

Editor-in-Chief
Advances and Practices in Computer Science and Engineering Board
Pradip Srimani
Colorado State University, Dept. of Computer Science
601 South Hows Lane
Fort Collins, CO 80525
Phone: 970-491-7097 FAX: 970-491-2466
srimani@cs.colostate.edu

Board Members:
Mark J. Christensen
Deborah M. Cooper – Deborah M. Cooper Company
William W. Everett – SPRE Software Process and Reliability Engineering
Haruhisa Ichikawa – NTT Software Laboratories
Annie Kuntzmann-Combelles – Objectif Technologie
Chengwen Liu – DePaul University
Joseph E. Urban – Arizona State University

IEEE Computer Society Executive Staff
T. Michael Elliott, Executive Director and Chief Executive Officer
Angela Burgess, Publisher

IEEE Computer Society Publications

The world-renowned IEEE Computer Society publishes, promotes, and distributes a wide variety of authoritative computer science and engineering texts. These books are available from most retail outlets. Visit the Online Catalog, *http://computer.org*, for a list of products.

IEEE Computer Society Proceedings

The IEEE Computer Society also produces and actively promotes the proceedings of more than 141 acclaimed international conferences each year in multimedia formats that include hard and softcover books, CD-ROMs, videos, and on-line publications.

For information on the IEEE Computer Society proceedings, send e-mail to *cs.books@computer.org* or write to Proceedings, IEEE Computer Society, P.O. Box 3014, 10662 Los Vaqueros Circle, Los Alamitos, CA 90720-1314. Telephone +1 714-821-8380. FAX +1 714-761-1784.

Additional information regarding the Computer Society, conferences and proceedings, CD-ROMs, videos, and books can also be accessed from our web site at *http://computer.org/cspress*

Revised 9 November 1999