Proceedings

Seventh Heterogeneous Computing Workshop (HCW '98)

Proceedings

Seventh Heterogeneous Computing Workshop (HCW '98)

March 30 1998
Orlando, Florida, U.S.A.

Edited by
John K. Antonio, Texas Tech University

Co-Sponsored by
IEEE Technical Committee on Parallel Processing
Office of Naval Research

IEEE COMPUTER SOCIETY

Los Alamitos, California

Washington • Brussels • Tokyo

IEEE Computer Society Order Number PR08365
ISBN 0-8186-8365-1
ISBN 0-8186-8367-8 (microfiche)
ISSN 1097-5209
IEEE Order Plan Catalog Number 98EX126

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
13, Avenue de l'Aquilon
B-1200 Brussels
BELGIUM
Tel: + 32-2-770-2198
Fax: + 32-2-770-8505
euro.ofc@computer.org

IEEE Computer Society
Ooshima Building
2-19-1 Minami-Aoyama
Minato-ku, Tokyo 107
JAPAN
Tel: + 81-3-3408-3118
Fax: + 81-3-3408-3553
tokyo.ofc@computer.org

Editorial production by Kristine Kelly

Cover art production by Alex Torres

Printed in the United States of America by Technical Communication Services

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave Blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	Sept. 18, 1998	Final, Nov. 1, 1997 to Sept. 30, 1998

4. TITLE AND SUBTITLE
A 1998 Workshop on Heterogeneous Computing

5. FUNDING NUMBERS
N00014-98-1-0122

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 7th Heterogeneous Computing Workshop (HCW '98), which was held on March 30, 1998. HCW '98 was part of the first merged symposium of the 12th International Parallel Processing Symposium and the 9th Symposium on Parallel and Distributed Processing (IPPS/SPDP 1998), which was sponsored by the IEEE Computer Society Technical Committee on Parallel Processing and held in cooperation with ACM SIGARCH. 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/or administrative domains. Topics representative of those in the proceedings 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

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. Z39-18
298-102

J61171
10-08-96

Table of Contents

Message from the General Chair

Welcome to the 7th Heterogeneous Computing Workshop, also known as HCW '98. Heterogeneous computing is a growing research area within Computer Science and Engineering, and is at the confluence of a number of other sub-disciplines, including parallel processing, distributed systems, scheduling and resource management algorithms, and metacomputing. 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/or administrative domains. The HCW workshop series includes research presentations on these and related topics and is an established forum for the dissemination of recent developments and results in heterogeneous computing. These proceedings contain the set of papers from the 1998 workshop; I hope you find these informative and interesting.

HCW '98 is the result of the dedication and hard work of a number of people. I thank Richard F. Freund, NRaD, for founding this series of workshops and for working hard to ensure its ongoing continuity and success. John Antonio of Texas Tech University was this year's Program Chair. With the able assistance of a terrific program committee, he has put together an excellent program and collection of papers in these proceedings. The Vice-General Chair was Dan Watson, who helped with the organization of HCW '98 in many ways, including handling workshop publicity.

Special thanks are due to H. J. Siegel of Purdue University for an enormous amount of help and advice with both programmatic and organizational matters. Without his guidance and assistance HCW '98 would not have been possible, and we truly appreciate his efforts. Traditionally, HCW has been held in conjunction with the International Parallel Processing Symposium (IPPS), which has merged with the Symposium on Parallel and Distributed Processing (SPDP) this year. I thank the General Co-Chairs of IPPS/SPDP, Viktor Prasanna and Behrooz Shirazi for their cooperation and assistance, with special thanks to Viktor for his continued support for HCW since its inception. This year, the workshop is sponsored by the IEEE Computer Society and the Office of Naval Research. We thank Dr. Andre M. van Tilborg, Director of the Math, Computer, and Information Sciences Division of the Office of Naval Research, for supporting publication of these proceedings under ONR grant number N00014-98-1-0122. Kristine Kelly, IEEE Computer Society Press, deserves special thanks for her promptness and professional, efficient handling of all the papers included here, and publication of these proceedings.

Vaidy Sunderam
Emory University

Message from the Program Chair

The field of heterogeneous computing (HC) is motivated by the diverse requirements of computational tasks, and the realization that the features of a single architecture are not always ideal for a wide range of task requirements. Research in HC ranges from the use of diverse computing systems interconnected over a geographically distributed network to the design and implementation of a parallel computer architecture consisting of a number of different processor types or modes of operation. Thus, the field of HC is quite broad, and requires research in areas such as parallel and distributed processing, performance estimation, matching and scheduling, task profiling, compiling, and portable programming languages.

The papers published in these proceedings represent some of the latest and most innovative research in the field of HC. The first session in the program consists of four invited papers describing case studies and reports on existing HC systems. These papers are very important in that they illustrate the practicality of HC, and often involve implementations based on past research findings. The papers included in the second and third sessions were selected by the program committee from submitted manuscripts. These papers cover topics of great importance to HC, including resource management, matching, scheduling, modeling issues, and group communications. The program concludes with what is sure to be a lively panel discussion on the use of Java for programming HC systems. Position papers from the panel chair and each of the panelists are included in these proceedings.

It has truly been an honor to serve as Program Chair for HCW '98, and I am very proud of the quality of this year's program. But coordinating the program was not done in isolation; many people contributed. I would like to thank the Program Committee members for their careful and prompt review of the papers assigned to them. I would also like to thank all of the authors for their technical contributions and insights, and for the careful editing and revising they performed on their papers based on reviewer comments. I am grateful to Gul Agha for his willingness to organize the Java panel session on relatively short notice. His success in assembling and coordinating the outstanding (and diverse) collection of panelists is no doubt a reflection of the well-deserved respect he has earned from his peers.

I am thankful to have worked with Vaidy Sunderam on this workshop. His leadership was illustrated in many ways, including his capacity to effectively organize tasks, coordinate ideas and concerns, and generally keep things running smoothly.

It has been a pleasure working with both Deborah Plummer and Kristine Kelly of the IEEE Computer Society Press in getting these proceedings published. Special thanks are due to Kristine for her patience in implementing some last minute changes before going to press. I would also like to give special thanks to my secretary, Marcelia Sawyers, for her assistance with my duties related to this workshop.

Finally, I am indebted to H. J. Siegel for his tireless dedication to this workshop. H. J. provided the Program Committee with numerous ideas and suggestions for organizing the program. For example, the original idea for including the Java panel came from H. J. In addition to providing ideas for the program, H. J. also helped keep me on track by providing "gentle reminders" to complete the many tasks that are involved in implementing a successful program.

John K. Antonio
Texas Tech University

Committees

General Chair

Vaidy Sunderam, *Emory University*

Vice General Chair

Dan Watson, *Utah State University*

Program Chair

John K. Antonio, *Texas Tech University*

Steering Committee

Richard F. Freund, Chair, *NOEMIX, Inc.*
Francine Berman, *UCSD*
Jack Dongarra, *University of Tennessee*
Debra Hensgen, *Naval Postgraduate School*
Paul Messina, *Caltech*
Jerry Potter, *Kent State University*
Viktor K. Prasanna, *USC*
H. J. Siegel, *Purdue University*
Vaidy Sunderam, *Emory University*

Program Committee Members

John K. Antonio, Chair, Texas Tech University
Francine Berman, University of California, San Diego
Steve J. Chapin, University of Virginia
Partha Dasgupta, Arizona State University
Mary Eshaghian, New Jersey Institute of Technology
Allan Gottlieb, New York University and NEC Research
Babak Hamidzadeh, University of British Columbia
Salim Hariri, Syracuse University
Taylor Kidd, Naval Postgraduate School
Domenico Laforenza, CNUCE - Institute of the Italian NRC
Yan Alexander Li, Intel Corporation
David J. Lilja, University of Minnesota
Noe Lopez-Benitez, Texas Tech University
Piyush Maheshwari, The University of New South Wales
Richard C. Metzger, Rome Laboratory
Viorel Morariu, Concurrent Technologies Corporation
Viktor K. Prasanna, University of Southern California
Ranga S. Ramanujan, Architecture Technology Corporation
Behrooz A. Shirazi, University of Texas at Arlington
H. J. Siegel, Purdue University
Min Tan, Cisco Systems, Inc.
Dan Watson, Utah State University
Charles C. Weems, University of Massachusetts, Amherst
Elizabeth Williams, Center for Computing Sciences
Albert Y. Zomaya, University of Western Australia

Session I

Invited Case Studies and Status Reports on Existing Systems

Session Chair

John K. Antonio
Texas Tech University, Lubbock, TX, USA

Scheduling Resources in Mult-User, Heterogeneous, Computing Environments with SmartNet*

Richard F. Freund+
Michael Gherrity
Stephen Ambrosius
Mark Campbell
Mike Halderman
Debra Hensgen
Elaine Keith
Taylor Kidd
Matt Kussow
John D. Lima
Francesca Mirabile
Lantz Moore
Brad Rust
H.J. Siegel

+NOEMIX, Inc.
14781 Pomerado Road, #133
Poway, CA 92064
rffreund@noemix.com

*see addendum, page 184

The Globus Project: A Status Report

Ian Foster

Mathematics and Computer Science Division
Argonne National Laboratory
Argonne, IL 60439

Carl Kesselman

Information Sciences Institute
University of Southern California
Marina Del Rey, CA 90292-6695

Abstract

The Globus project is a multi-institutional research effort that seeks to enable the construction of computational grids *providing pervasive, dependable, and consistent access to high-performance computational resources, despite geographical distribution of both resources and users. Computational grid technology is being viewed as a critical element of future high-performance computing environments that will enable entirely new classes of computation-oriented applications, much as the World Wide Web fostered the development of new classes of information-oriented applications. In this paper, we report on the status of the Globus project as of early 1998. We describe the progress that has been achieved to date in the development of the Globus toolkit, a set of core services for constructing grid tools and applications. We also discuss the Globus Ubiquitous Supercomputing Testbed (GUSTO) that we have constructed to enable large-scale evaluation of Globus technologies, and we review early experiences with the development of large-scale grid applications on the GUSTO testbed.*

1 Introduction

Advances in networking technology and computational infrastructure make it possible to construct large-scale high-performance distributed computing environments, or *computational grids* that provide dependable, consistent, and pervasive access to high-end computational resources. These environments have the potential to change fundamentally the way we think about computing, as our ability to compute will no longer be limited to the resources we currently have on hand. For example, the ability to integrate TFLOP/s computing resources on demand will allow us to integrate sophisticated analysis, image processing, and real-time control into scientific instruments such as microscopes, telescopes, and MRI machines. Or, we can call upon the resources of a nationwide *strategic computing reserve* to perform time-critical computational tasks in times of crisis, for example to perform diverse simulations as we plan responses to an oil spill.

In the past, high-performance distributed computation has been achieved on a limited scale by heroic efforts such as the CASA Gigabit testbed [26] and the I-WAY [12]. The work of ourselves and others on computational grids differs from these ground-breaking efforts in that we seek to make commonplace the integration of remote resources into a computation. To a large extent, the development of usable computational grids is hindered not by available hardware capabilities but by limitations in the software abstractions and services that are currently in use. Existing network tools are focused on supporting communication, not computation, while current distributed computing systems are not performance driven and typically are limited to client/server models of computation. Clearly, the success of computational grids will depend on the existence of grid-specific middleware that addresses the needs of computations including dynamic resource allocation, resource co-allocation, heterogeneous and dynamic computational and communication substrates, and process-oriented security.

We have been studying the problems associated with constructing usable computational grids since 1995, first in the context of the I-WAY networking experiment [12] and subsequently as part of a project called Globus. The goal of Globus is to understand application requirements for a usable grid and to develop the essential technologies required to meet these requirements. In pursuit of this goal, we have developed a research program comprising three broad activities:

- developing the basic technology and high-level tools required for computational grids;
- constructing a large-scale, prototype computational grid (i.e., testbed) using the basic technologies and tools we have developed; and

0-8186-8365-1/98 $10.00

- executing realistic applications on the prototype grid, in order to evaluate the utility of our technologies and of the grid concept.

In this paper, we describe the status of the Globus project in each of these three areas, as of early 1998. This description updates the original Globus paper [13] and a subsequent project summary in [14] by providing a more complete and up-to-date description of the Globus toolkit and by reviewing early experiments with the Globus Ubiquitous Supercomputing Testbed (GUSTO) grid prototype, the largest computational grid constructed to date.

The organization of this paper is as follows. In the next section, we outline the basic architecture of the Globus system, identifying the basic principles that motivate its design. In Sections 3–7, we describe the set of basic services that constitute the Globus toolkit that underlies our approach, and in Section 8 we review some of the higher-level tools that have been constructed with this toolkit. In Section 9, we describe our experiences deploying these tools in the GUSTO grid testbed, and in Section 10 we review our experiences developing applications. We conclude the paper with a brief survey of some related work (Section 11) and a description of our future plans (Section 12).

2 Globus Overview

A central element of the Globus system is the Globus Metacomputing Toolkit, which defines the basic services and capabilities required to construct a computational grid. The design of this toolkit was guided by the following basic principles.

The toolkit comprises a set of components that implement basic services for security, resource location, resource management, communication, etc.. The services currently defined by Globus are listed in Table 1. Computational grids must support a wide variety of applications and programming models. Hence, rather than providing a uniform programming model, such as the object-oriented model defined by the Legion system [18], the Globus toolkit provides a "bag of services" from which developers of specific tools or applications can select to meet their needs.

Because services are distinct and have well-defined interfaces, they can be incorporated into applications or tools in an incremental fashion. We illustrate this mix-and-match approach to metacomputing in Sections 8 and 10, where we describe how different parallel tools and a large application can be made grid aware by incorporating different services.

The toolkit distinguishes between local services, which are kept simple to facilitate deployment, and global services, which are constructed on top of local services and may be more complex. Computational grids require that a wide range of services be supported on a highly heterogeneous mix of systems and that it be possible to define new services without changing the underlying infrastructure. An established architectural principle in such situations, as exemplified by the Internet Protocol suite [6], is to adopt a layered architecture with an "hourglass" shape (Figure 1). A simple, well-defined interface—the neck of the hourglass—provides uniform access to diverse implementations of local services; higher-level global services are then defined in terms of this interface. To participate in a grid, a local site need provide only the services defined at the neck, and new global services can be added without local changes. We discuss this organization in greater detail in Section 3.

Interfaces are defined so as to manage heterogeneity, rather than hiding it. These so-called translucent interfaces provide structured mechanisms by which tools and applications can discover and control aspects of the underlying system. Such translucency can have significant performance advantages because, if an implementation of a higher-level service can understand characteristics of the lower-level services on which the interface is layered, then the higher-level service can either control specific behaviors of the underlying service or adapt its own behavior to that of the underlying service. Translucent interfaces do not imply complex interfaces. Indeed, we will show that translucency can be provided via simple techniques, such as adding an attribute argument to the interface. We discuss these issues at greater length in Section 4, when we describe Globus communication services.

An information service is an integral component of the toolkit. Computational grids are in a constant state of flux as utilization and availability of resources change, computers and networks fail, old components are retired, new systems are added, and software and hardware on existing systems are updated and modified. It is rarely feasible for programmers to rely on standard or default configurations when building applications. Rather, applications must *discover* characteristics of their execution environment dynamically and then either *configure* aspects of system and application behavior for efficient, robust execution or *adapt* behavior during program execution. A fundamental requirement for discovery, configuration, and adaptation is an *information-rich environment* that provides pervasive and uniform access to information about the current state of the grid and its underlying components. In the Globus toolkit, a component

Table 1: Core Globus services. As of early 1998, these include only those services deemed essential for an evaluation of the Globus design philosophy on realistic applications and in medium-scale grid environments. Other services such as accounting, auditing, and instrumentation will be addressed in future work

Service	Name	Description
Resource management	GRAM	Resource allocation and process management
Communication	Nexus	Unicast and multicast communication services
Security	GSI	Authentication and related security services
Information	MDS	Distributed access to structure and state information
Health and status	HBM	Monitoring of health and status of system components
Remote data access	GASS	Remote access to data via sequential and parallel interfaces
Executable management	GEM	Construction, caching, and location of executables

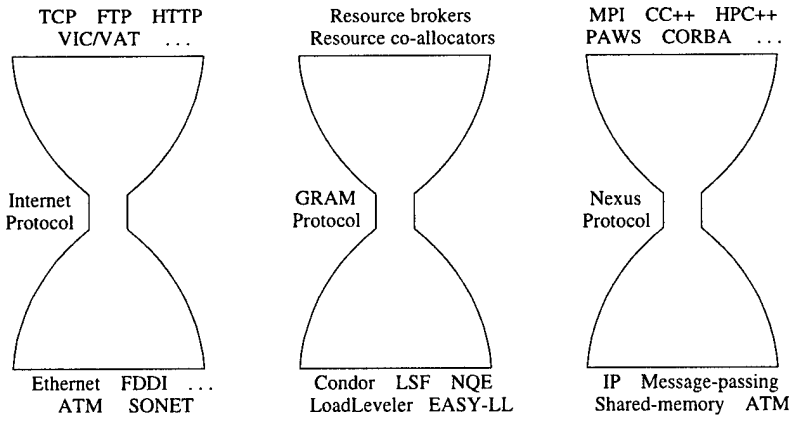


Figure 1: The hourglass principle, as applied in the Internet Protocol suite, Globus resource management services, and Globus communication services

called the Metacomputing Directory Service [9], discussed in Section 5, fulfills this role.

The toolkit uses standards whenever possible for both interfaces and implementations. We envision computational grids as supporting an important niche of applications that must co-exist with more general-purpose distributed and networked computing applications such as CORBA, DCE, DCOM, and Web-based technologies. The Internet community and other groups are moving rapidly to develop official and de facto standards for interfaces, protocols, and services in many areas relevant to computational grids. There is considerable value in adopting these standards whenever they do not interfere with other goals. Consequently, the Globus components we will describe are not, in general, meant to replace existing interfaces, but rather seek to augment them. The utility of standards is emphasized in Section 6, which describes the Globus security infrastructure.

3 Resource Management

We now dexcribe more fully the Globus components listed in Table 1. We start by considering resource management. Both this discussion and the current Globus implementation focus on the management of computational resources. Management of memory, storage, networks, and other resources is clearly also important and is being considered in current research.

Globus is a layered architecture in which high-level global services are built on top of an essential set of core local services. At the bottom of this layered architecture, the Globus Resource Allocation Manager (GRAM) provides the local component for resource management [8]. Each GRAM is responsible for a set of resources operating under the same site-specific allocation policy, often implemented by a local resource management system, such as Load Sharing Facility (LSF) or Condor. For example, a single manager could provide access to the nodes of a parallel computer, a cluster of workstations, or a set of machines operating within a Condor pool [25]. Thus, a computational grid built with Globus typically contains many GRAMs, each responsible for a particular "local" set of resources.

GRAM provides a standard network-enabled interface to local resource management systems. Hence, computational grid tools and applications can express resource allocation and process management requests in terms of a standard application programming interface (API), while individual sites are not constrained in their choice of resource management tools. GRAM can currently operate in conjunction with six different local resource management tools: Network Queuing Environment (NQE), EASY-LL, LSF, LoadLeveler, Condor, and a simple "fork" daemon. Within the GRAM API, resource requests are expressed in terms of an extensible *resource specification language* (RSL); as we describe below, this language plays a critical role in the definition of global services.

GRAM services provide building blocks from which we can construct a range of global resource management strategies. Building on GRAM, we have defined the general resource management architecture [8] illustrated in Figure 2. RSL is used throughout this architecture as a common notation for expressing resource requirements. Resource requirements are expressed by an application in terms of a high-level RSL expression. A variety of *resource brokers* implement domain-specific resource discovery and selection policies by transforming abstract RSL expressions into progressively more specific requirements until a specific set of resources is identified. For example, an application might specify a computational requirement in terms of floating-point performance (MFLOPs). A high-level broker might narrow this requirement to a specific type of computer (an IBM SP2, for example), while another broker might identify a specific set of SP2 computers that can fulfill that request. At this point, we have a so-called ground RSL expression in which a specific set of GRAMs are identified.

The final step in the resource allocation process is to decompose the RSL into a set of separate resource allocation requests and to dispatch each request to the appropriate GRAM. In high-performance computations, it is often important to *co-allocate* resources at this point, ensuring that a given set of resources is available for use simultaneously. Within Globus, a *resource co-allocator* is responsible for providing this service: breaking the RSL into pieces, distributing it to the GRAMs, and coordinating the return values. Different co-allocators can be constructed to implement different approaches to the problems of allocating and managing ensembles of resources. We currently have two allocation services implemented. The first defines a simple *atomic co-allocation* semantics. If any of the requested resources are unavailable for some reason, the entire co-allocation request fails. In practice, this strategy has proven to be too inflexible in many situations. Based on this experience, we have implemented a second co-allocator, which allows components of the submitted RSL expression to be modified until the application or broker issues a commit operation.

Notice that a consequence of the Globus resource management architecture is that resource and com-

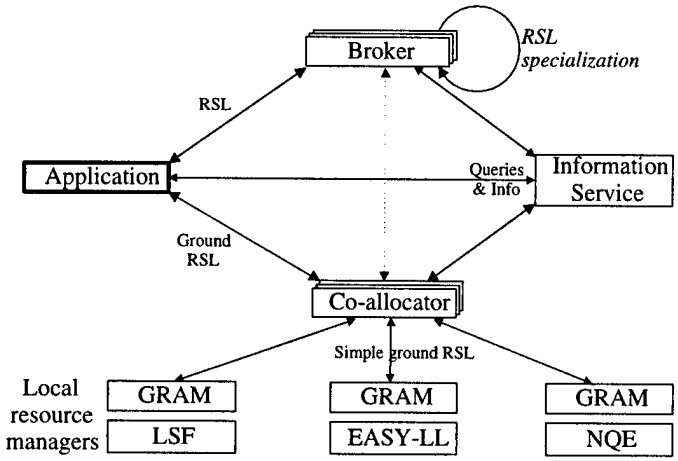


Figure 2: The Globus resource management architecture, showing how RSL specifications pass between application, resource brokers, resource co-allocators, and local managers (GRAMs). Notice the central role of the information service.

putation management services are implemented in a hierarchical fashion. An individual GRAM supports the creation and management of a set of processes, or Globus job, on a set of local resources. A computation created by a global service may then consist of one or more jobs, each created by a request to a GRAM and managed via management functions implemented by that GRAM.

This discussion of Globus resource management services illustrates how simple local services, if appropriately designed, can be used to support a rich set of global functionality.

4 Communication

Communication services within the Globus toolkit are provided by the Nexus communication library [15]. As illustrated in Figure 1, Nexus defines a relatively low-level communication API that is then used to support a wide range of higher-level communication libraries and languages, based on programming models as diverse as message passing, as in the Message Passing Interface (MPI) [10]; remote procedure call, as in CC++ [5]; striped transfer, as in the Parallel Application Workspace (PAWS); and distributed database updates for collaborative environments, as in CAVERNsoft. Nexus communication services are also used extensively in the implementation of other Globus modules.

The communication needs of computational grid applications are diverse, ranging from point-to-point message passing to unreliable multicast communication. Many applications, such as instrument control and teleimmersion, use several modes of communication simultaneously. In our view, the Internet Protocol does not meet these needs: its overheads are high, particularly on specialized platforms such as parallel computers; the TCP streaming model is not appropriate for many interactions; and its interface provides little control over low-level behavior. Yet traditional high-performance computing communication interfaces such as MPI do not provide the rich range of communication abstractions that grid applications will require. Hence, we define an alternative communication interface designed to support the wide variety of underlying communication protocols and methods encountered in grid environments and to provide higher-level tools with a high degree of control over the mapping between high-level communication requests and underlying protocol operations. We call this interface Nexus [15, 11].

Communication in Nexus is defined in terms of two basic abstractions. A *communication link* is formed by binding a communication startpoint to a communication endpoint (Figure 4); a communication operation is initiated by applying a *remote service request* (RSR) to a startpoint. This one-sided, asynchronous remote procedure call transfers data from the startpoint to the associated endpoint(s) and then integrates the data into the process(es) containing the endpoint(s) by invoking a function in the process(es). More than one startpoint can be bound to an endpoint and vice versa,

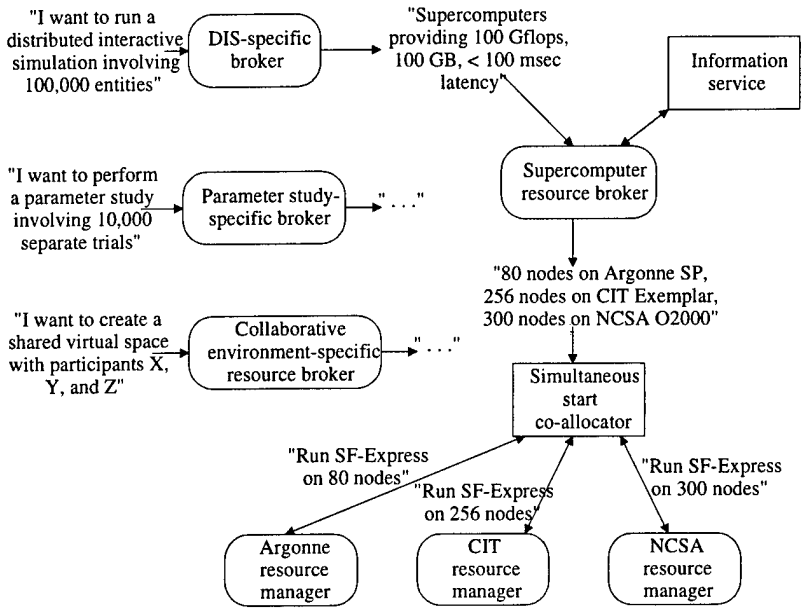


Figure 3: This view of the Globus resource management architecture shows how different types of broker can participate in a single resource request

allowing for the construction of complex communication structures.

The communication link/RSR communication model can be mapped into many different communication methods, each with potentially different performance characteristics [11]. Communication methods include not only communication protocols, but also other aspects of communication such as security, reliability, quality of service, and compression. By associating *attributes* with a specific startpoint or endpoint, an application can control the communication method used on a per-link basis. For example, an application in which some communications must be reliable while others require low latencies can establish two links between two processes, with one configured for reliable—and potentially high-latency—communication and the other for low-latency unreliable communication.

High-level selection and configuration of low-level methods is useful only if the information required to make intelligent decisions is readily available. Within Globus, MDS (discussed in Section 5) maintains a wealth of dynamic information about underlying communication networks and protocols, including network connectivity, protocols supported, and network bandwidth and latency. Applications, tools, and higher-level libraries can use this information to identify available methods and select those best suited for a particular purpose.

High-level management of low-level communication methods has many uses. For example, an MPI implementation layered on top of Nexus primitives can not only select alternative low-level protocols (e.g., message passing, IP, or shared memory) based on network topology and the location of sender and receiver [10], but can simultaneously apply selective use of encryption based on the source and destination of a message. The ability to attach network quality of service specifications to communication links is also useful.

Nexus illustrates how Globus services use translucent interfaces to allow applications to manage rather than hide heterogeneity. An application or higher-level library can express all operations in terms of a single uniform API; the resulting programs are portable across, and will execute efficiently on, a wide variety of computing platforms and networks. To this extent Nexus, like other Globus services, hides heterogeneity. However, in situations where performance is critical, properties of low-level services can be discovered. The higher-level library or application can then either adapt its behavior appropriately or use a control API to manage just how high-level behavior is implemented: for example, by specifying that it is ac-

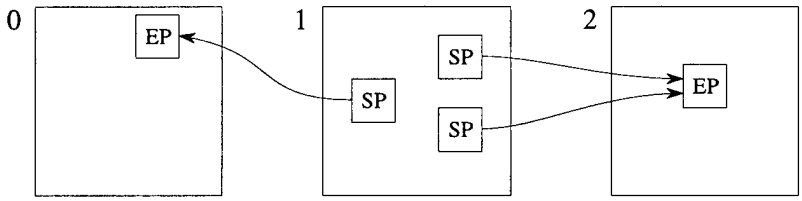


Figure 4: Nexus communication mechanisms. The figure shows three processes and three communication links. Three startpoints in process 1 reference endpoints in processes 0 and 2.

ceptable to use an unreliable communication protocol for a particular set of communications.

5 Information

The dynamic nature of grid environments means that toolkit components, programming tools, and applications must be able to adapt their behavior in response to changes in system structure and state. The Globus Metacomputing Directory Service (MDS) [9] is designed to support this type of adaptation by providing an information-rich environment in which information about system components is always available. MDS stores and makes accessible information such as the architecture type, operating system version and amount of memory on a computer, network bandwidth and latency, available communication protocols, and the mapping between IP addresses and network technology.

MDS provides a suite of tools and APIs for discovering, publishing, and accessing information about the structure and state of a computational grid. As in other Globus components, official or de facto standards are used in MDS whenever possible. In this case, the standards in question are the data representation and API defined by the Lightweight Directory Access Protocol (LDAP) [22], which together provide a uniform, extensible representation for information about grid components. LDAP defines a hierarchical, tree-structured name space called a *directory information tree* and is designed as a distributed service: arbitrary subtrees can be associated with distinct servers. Hence, the local service required to support MDS is exactly an LDAP server (or a gateway to another LDAP server, if multiple sites share a server), plus the utilities used to populate this server with up-to-date information about the structure and state of the resources within that site. The global MDS service is simply the ensemble of all these servers.

An information-rich environment is more than just mechanisms for naming and disseminating information: it also requires agents that produce useful information and components that access and use that information. Within Globus, both these roles are distributed over every system component—and potentially over every application. Every Globus service is responsible for producing information that users of that service may find useful, and for using information to enhance its flexibility and performance. For example, each local resource manager (Section 3) incorporates a component called the *GRAM reporter* responsible for collecting and publishing information about the type of resources being managed, their availability, and so forth. Resource brokers use this and other information for resource discovery.

6 Security

Security in computational grids is a multifaceted issue, encompassing authentication, authorization, privacy, and other concerns. While the basic cryptographic algorithms that form the basis of most security systems—such as public key cryptography—are relatively simple, it is a challenging task to use these algorithms to meet diverse security goals in complex, dynamic grid environments, with large and dynamic sets of users and resources and fluid relationships between users and resources.

The Globus security infrastructure developed for the initial Globus toolkit focuses on just one problem, *authentication*: the process by which one entity verifies the identity of another. We focus on authentication because it is the foundation on which other security services, such as authorization and encryption, are built; these issues will be addressed in future work.

Authentication solutions for computational grids must solve two problems not commonly addressed by standard authentication technologies. The first problem that must be addressed by a grid authentication solution is support for *local heterogeneity*. Grid resources are operated by a diverse range of entities,

each defining a different *administrative domain.* Each domain will have its own requirements for authentication and authorization, and consequently, domains will have different local security solutions, mechanisms, and policies, such as one-time passwords, Kerberos [29], and Secure Shell. We will have limited ability to change these administrative decisions, and any security solution must confront this heterogeneity.

The second problem facing security solutions for computational grids is the need to support *N-way security contexts.* In traditional client-server applications, authentication involves just a single client and a single server. In contrast, a grid computation may acquire, start processes on, and release many resources dynamically during its execution. These processes will communicate by using a variety of mechanisms, including unicast and multicast. These processes form a single, fully connected logical entity, although low-level communication connections (e.g., TCP/IP sockets) may be created and deleted dynamically during program execution. A security solution for a computational grid must enable the establishment of a security relationship between any two processes in a computation.

A first important step in the design of a security architecture, often overlooked, is to define a security policy: that is, to provide a precise definition of what it means for the system in question to be secure. This policy identifies what components are to be protected and what these components are to be protected against, and defines security operations in terms of abstract algorithms. The policy defined for Globus is shaped by the need to support N-way security contexts and local heterogeneity. The policy specifies that a user authenticate just once per computation, at which time a credential is generated that allows processes created on behalf of the user to acquire resources, and so forth, without additional user intervention. Local heterogeneity is handled by mapping a user's *Globus identity* into local user identities at each resource.

One important aspect of the security policy defined by Globus is that encrypted channels are not used. Globus is intended to be used internationally, and several countries (including the United States and France) have restrictive laws with respect to encryption technology. The Globus policy relies only on digital signature mechanisms, which are more easily exportable from the United States.

The Globus security policy is implemented by the Globus security infrastructure (GSI). GSI, like other Globus components, has a modular design in which diverse global services are constructed on top of a simple local service that addresses issues of local heterogeneity. As illustrated in Figure 5, the local security service implements a security gateway that maps authenticated Globus credentials into locally recognized credentials at a particular site: for example, Kerberos tickets, or local user names and passwords. A benefit of this approach is that we do not require "group" accounts and so can preserve the integrity of local accounting and auditing mechanisms.

The internal design of GSI emphasizes the important role that standards have to play in the definition of grid services and toolkits. Several of the problems that GSI is designed to solve, namely, support for different local mechanisms and N-way security contexts, are not supported by any existing system. Nevertheless, GSI's ability to interoperate with other systems, to achieve independence from low-level mechanisms, and to leverage existing code is enhanced by coding all security algorithms in terms of the Generic Security Service (GSS) standard [24]. GSS defines a standard procedure and API for obtaining credentials (passwords or certificates), for mutual authentication (client and server), and for message-oriented signature, encryption and decryption. GSS is independent of any particular security mechanism and can be layered on top of different security methods. To promote interoperability, the GSS standard defines how GSS functionality should be implemented on top of Kerberos and public key cryptography. GSS also defines a negotiation mechanism that allows two parties to select a mutually agreeable suite of security mechanisms, should alternatives exist.

GSI currently supports two security mechanisms, both accessible through the GSS interface. The first is a plaintext password system, which basically implements Unix `rlogin` type authentication. The plaintext implementation has the advantage of being easy to develop and debug and is not encumbered by export controls. The second mechanism uses public key cryptography and is based on the authentication protocol defined by the Secure Socket Layer (SSL) [21]. This implementation has the advantages of much stronger security and interoperability with a variety of commodity services, including LDAP and HTTP. We note that GSS supports a negotiation mechanism, which allows us to support both security mechanisms simultaneously in the Globus environment.

7 Other Globus Services

We briefly describe the other three Globus services listed in Table 1: health and status monitoring, remote access to files, and executable management.

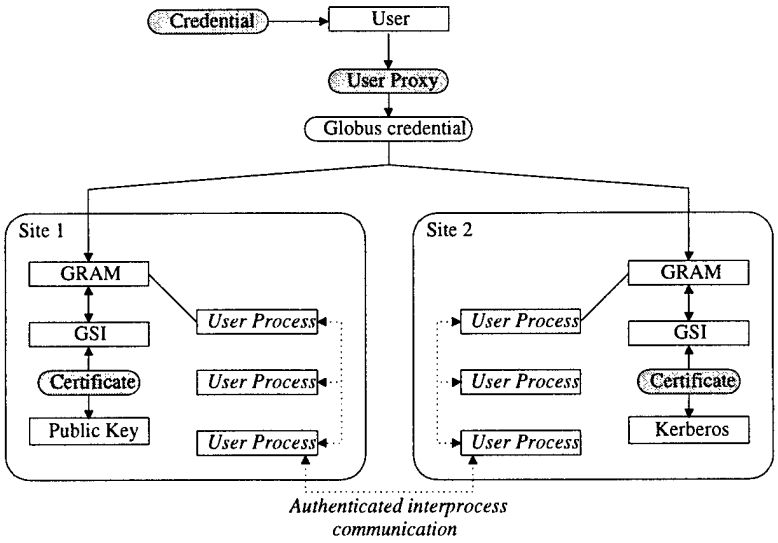


Figure 5: The Globus security infrastructure, showing its support for single sign-on and local heterogeneity

The Heartbeat Monitor (HBM) service provides simple mechanisms for monitoring the health and status of a distributed set of processes. The HBM architecture comprises a client interface and a data-collector API. The client interface allows a process to register with the HBM service, which then expects to receive regular heartbeats from the process. If a heartbeat is not received, the HBM service attempts to determine whether the process itself is faulty or whether the underlying network or computer has failed. The data-collector API allows another process to obtain information regarding the status of registered process; this information can then be used to implement a variety of fault detection and, potentially, fault recovery mechanisms. HBM mechanisms are used to monitor the status of core Globus services, such as GRAM and MDS. They can also be used to monitor distributed applications and to implement application-specific fault recovery strategies.

Access to remote files is provided by the Global Access to Secondary Storage (GASS) subsystem. This system allows programs that use the C standard I/O library to open and subsequently read and write files located on remote computers, without requiring changes to the code used to perform the reading and writing. As illustrated in Figure 6, files opened for reading are copied to a local file cache when they are opened, hence permitting subsequent read operations to proceed without communication and also avoiding repeated fetches of the same file. Reference counting is used to determine when files can be deleted from the cache. Similarly, files opened for writing are created locally and copied to their destination only when they are closed. A similar copying strategy is used in UFO [2], but our implementation does not rely on the Unix-specific `proc` file system. GASS also allows files to be opened for remote appending, in which case data is communicated to the remote file as soon as it is written; this mode is useful for log files, for example. In addition, GASS supports remote operations on caches and hence, for example, program-directed prestaging and migration of data. HTTP, FTP, and specialized GASS servers are supported.

Finally, the Globus Executable Management (GEM) service, still being designed as of January 1998, is intended to support the identification, location, and creation of executables in heterogeneous environments. GEM provides mechanisms for matching the characteristics of a computer in a computational grid with the runtime requirements of an executable or library. These mechanisms can be used in conjunction with other Globus services to implement a variety of distributed code management strategies, based for example on online executable archives and compile servers.

8 High-Level Tools

While Globus services can be used directly by application programmers, they are more commonly ac-

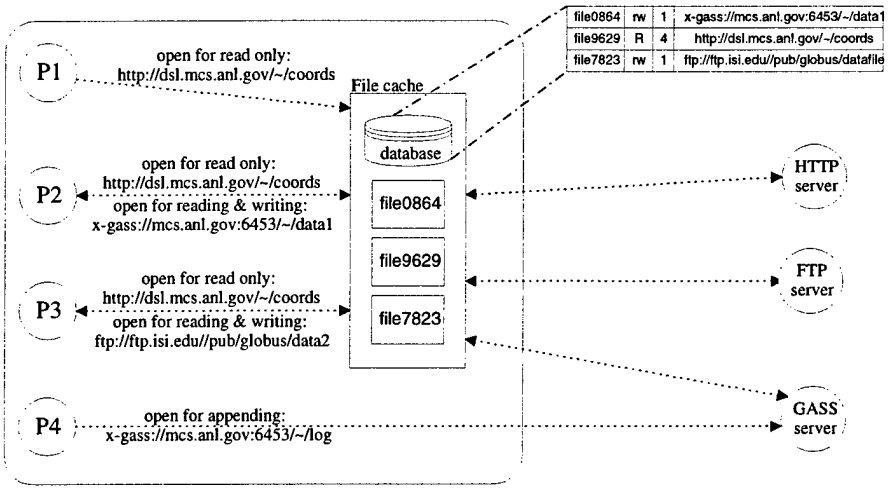


Figure 6: The Global Access to Secondary Storage (GASS) subsystem allows processes on local computers to read and write remote files. Copies of remote files opened for reading and/or writing are maintained in a local file cache. A simple database keeps track of the local file name, access mode, reference count, and remote file URL.

cessed via higher-level tools developed by tool developers. We illustrate this type of use with four examples: a message-passing library, a parallel language, a remote I/O library, and a parameter study system. Each tool uses different Globus services in a different way to support a particular programming model; in each case, availability of the Globus toolkit has allowed existing tools to be adapted for wide-area execution with relatively little effort.

The Message Passing Interface (MPI) defines a standard API for writing message-passing programs and is widely used in parallel computing. For grid applications, message passing has the advantage of providing a higher-level view of communication than TCP/IP sockets, while preserving for the programmer a high degree of control over how and when communication occurs. Globus services have been used to develop a grid-enabled MPI [10] based on the MPICH library [20], with Nexus used for communication, GRAM services for resource allocation, and GSI services for authentication. The result is a system that allows programmers to use simple, standard commands to run MPI programs in a variety of metacomputing environments (freely combining heterogeneous workstation and MPP metacomputing resources), while making efficient use of underlying networks. In future work, the developers of this system plan to use MDS information to construct communication structures—in particular, collective operations—that are optimized for wide-area execution.

Compositional C++ [5], or CC++, is a high-level parallel programming language based on C++. CC++ defines a global name space through the use of *global pointers*, dynamic resource allocation, and support for threading and remote procedure call style communication. The Globus implementation of CC++ uses the same services as the grid-enabled MPI, except that while the MPI implementation relies on Globus co-allocation services for resource allocation, the task-parallel CC++ model interfaces to GRAM directly.

The **Remote I/O** (RIO) library [16] is a tool for achieving high-speed access from parallel programs to files located on remote filesystems. RIO adopts the parallel I/O interface defined by MPI-IO [7, 27] and hence allows any program that uses MPI-IO to operate unchanged in a wide-area environment. The RIO implementation, like that of MPI, is constructed by using Globus services to adapt an existing system—the ROMIO implementation of MPI-IO—to support wide-area execution. Specifically, Nexus services are used for communication, GSI services for authentication, and MDS services for configuration.

Nimrod-G is a wide-area version of Nimrod [1], a tool that automates the creation and management of large parametric experiments. Nimrod allows a user to run a single application under a wide range of input conditions and then to aggregate the results of

these different runs for interpretation. In effect, it transforms file-based programs into interactive "meta-applications" that invoke user programs much as we might call subroutines. Nimrod-G uses MDS services to locate suitable resources when a user first requests a computational experiment, and GSI and GRAM services to schedule jobs to resources identified by MDS queries. In effect, Nimrod-G implements resource brokering services specialized for a particular class of application.

9 The GUSTO Testbed

Globus technologies have been deployed in the Globus Ubiquitous Supercomputing Testbed (GUSTO), by several measures the largest computational grid testbed ever constructed as of early 1998. This testbed uses both dedicated OC3 and commodity Internet services to link (as of early 1998) 17 sites, 330 computers, and 3600 processors, providing an aggregate peak performance of 2 Tflop/s. GUSTO sites span the continental United States, Hawaii, Sweden, and Germany; additional sites are being added rapidly. We discuss briefly our experiences deploying, administering, and using this testbed.

GUSTO was created during the three months prior to the November 1997 Supercomputing conference, held in San Jose. During this time, the first version of the Globus toolkit was completed, deployed at 15 sites, and applied in 10 different application projects.

One lesson learned early during this effort was that the approach of defining simple local services (and the considerable effort put into automatic configuration and information discovery tools) was a big win: we were able to deploy Globus software at 15 sites with relative ease, admittedly with considerable help from local staff in some cases. At several sites, computer security officers reviewed and approved our code. The hardest part of the deployment process was typically the development of the GRAM interface to the local scheduler.

Once Globus was deployed, MDS and HBM proved valuable as tools for administering a complex collection of computer systems. The standard interface provided by MDS ensured that GUSTO administrators always had up-to-date information about the structure and state of the system at their fingertips. This information was accessed via both an MDS browser and various specialized Web-based tools developed to publish specific views of the testbed.

Ten different groups developed applications for GUSTO. One of these applications is discussed in the next section; others included remote visualization of scientific simulations, real-time analysis of data from scientific instruments (meteorological satellite and X-ray source), and distributed parameter studies. The tools and services used by these different applications varied tremendously, with some programming in sockets and using just the bare minimum of Globus services, and others exploiting the full range of services.

The security model used for initial GUSTO deployment was based on the plain-text GSS implementation that we have developed. While the plain-text authentication model is quite weak, it had the advantage of avoiding export control issues. However, the need for the stronger, public key implementation was universally expressed. An export license for this technology is pending, and the currently deployed system will be upgraded to this authentication mechanism once such a license is issued.

10 Application Overview

We provide a brief description of one application demonstrated on the initial GUSTO prototype. SF-Express is a distributed interactive simulation (DIS) application that harnesses multiple supercomputers to meet the computational demands of large-scale network-based simulation environments. A large simulation may involve many tens of thousands of entities and requires thousands of processors. Globus services can be used to locate, assemble, and manage those resources. For example, in one experiment in November 1997, SF-Express was run on 852 processors distributed over 6 GUSTO sites. A more detailed discussion of SF-Express and how Globus is being used to support its execution across multiple supercomputers can be found in [4]

An advantage of the Globus bag of services architecture is that an application need not be entirely rewritten before it can operate in a grid environment: services can be introduced into an application incrementally, with functionality increasing at each step. As illustrated in Table 2 and described briefly in the following, this approach is being followed as the original SF-Express is converted into a grid-enabled application.

SF-Express Startup and Configuration Prior to the use of Globus services, simply starting SF-Express on multiple supercomputers was a painful task. The user had to log in to each site in turn and recall the arcane commands needed to allocate resources and start a program. This obstacle to the use of distributed resources was overcome by encoding resource allocation requests in terms of the GRAM API. GRAM and associated GSI services are used to handle authentication, resource allocation, and process creation at each site.

Table 2: A grid-aware version of SF-Express is being constructed incrementally: Globus services are incorporated one by one to improve functionality and reduce application complexity. The Status field indicates code status as of early 1998: techniques are in use (Y), are experimental or in partial use (y), or remain to be applied in the future (blank).

Services	How used	Benefits	Status
GRAM, GSI	Start SF-Express on supercomputers	Avoid need to log in to and schedule each system	Y
+ Co-allocator	Distributed startup and management	Avoid application-level check-in and shutdown	Y
+ MDS	Use MDS information to configure computation	Performance, portability	y
+ Resource broker	Use broker to locate appropriate computers	Code reuse, portability	y
+ Nexus	Encode communication as Nexus RSRs	Uniformity of interface, access to unreliable comms	y
+ HBM	Components check in with application-level monitor	Provides degree of fault tolerance	Y
+ GASS	Use to access terrain database files etc.	Avoid need to prestage data files	
+ GEM	Use to generate and stage executables	Avoid configuration problems	

Currently, the resources used for a simulation are manually specified, using MDS tools to help locate, select, and construct RSL specifications for appropriate supercomputers. As illustrated in Figure 2, these tasks can be avoided if we have access to resource brokers that can automatically construct the required RSL, using information such as the available network bandwidth and CPU power to determine the number of nodes required from the number of entities being simulated, and the number of nodes each router can handle. Once the resource set is identified and the RSL specification generated, Globus co-allocation services are employed to coordinate startup across multiple supercomputers, ensuring that the application has started on the desired resources before allowing the simulation to proceed.

After startup, the simulation must configure itself. In order to execute efficiently on parallel computers that have nonuniform access to network interfaces or secondary storage, SF-Express is organized such that intercomputer communication and I/O activities are performed only within specialized servers. Using information contained in the MDS, SF-Express can configure itself to place these services on appropriate nodes within a parallel computer, that is, the node with the attached disk or network interface card.

Finally, SF-Express must read various files describing the simulation scenario and the terrain on which the simulation is to be performed. In the initial SF-Express prototype, these files had to be staged manually to each site at which SF-Express executed. To simplify this task, we are migrating these file operations to use the GASS service provided in the Globus toolkit.

Communication. The SF-Express demonstrated at SC'97 uses MPI for communication within a simulation group, but handwritten socket code for communication between routers. This approach leads to considerable application code complexity and hinders portability. One approach we are considering is to rewrite the inter-supercomputer communication code to use MPI. The grid-enabled MPI discussed in Section 8 can then be used, eliminating the need for application socket code.

A second approach is to rewrite SF-Express so that communication operations are expressed directly by using Nexus operations. SF-Express communication operations are concerned primarily with the remote enqueing of simulation events and, hence, are expressed more naturally as Nexus RSRs than as MPI calls. A second benefit to using Nexus is that we can then, as discussed in Section 4, select an unreliable communication protocol for the distribution of

information to routers. This usage is desirable because SF-Express, unlike many other distributed simulations, does not maintain a global simulation clock. Instead, nodes simply discard incoming events with timestamps earlier than the local simulation clock. Hence, an unreliable protocol that tends to deliver most events sooner than an equivalent reliable protocol may be preferable.

11 Related Work

The primary purpose of this paper is to report on the current status of the Globus project rather than to provide detailed comparisons with related work. Hence, we provide pointers here to just a few representative efforts; the reader is referred to our other papers listed in the references for more detailed discussion.

The Legion project [19], like Globus, is investigating issues relating to software architectures and base technologies for grid environments. In contrast to the Globus bag of services architecture, Legion is organized around an object-oriented model in which every component of the system is represented by an object [23]. In principle, Globus services can be used to implement the Legion object model, so the two projects are in many respects pursuing complementary goals.

Condor [25] is a high-throughput computing environment whose goal is to deliver large amounts of computational capability over long periods of time (weeks or months), rather than peak capacity for limited time durations (hours or days). Condor addresses the needs of a limited, although important class of applications whose components are loosely coupled, often organized into a task-pool style computation. Currently, the GRAM interface to Condor enables Globus users to submit jobs to Condor pools. We are working with the Condor team to integrate other aspects of the systems, such as authentication.

A number of projects are attempting to build distributed computing environments on top of technologies and infrastructure developed for the World Wide Web. These include specialized systems such as SuperWeb [3] and WebOS [30] as well as systems leveraging basic Web technologies, such as Java Remote Method Invocation.

SNIPE [28] is a metacomputing project that builds on the resource management and communication facilities provided by the PVM message-passing library [17]. Like Globus, SNIPE recognizes the importance of information services and uses the Resource Cataloging and Distribution System to provide access to system resources and metadata.

12 Summary and Future Work

We have described the current status of the Globus project, which seeks to develop the basic technologies required to support the construction and use of computational grids. A particular focus of the Globus effort is the development of a small metacomputing toolkit providing essential services that can then be used to implement a variety of higher-level programming models, tools, and applications. As we have explained in this brief review, Globus components have been deployed in large testbeds and used to implement a variety of applications.

We have referred above to the advantages that we perceive in the Globus toolkit approach: in particular, the wide range of global services that can be supported, because of the decoupling of global and local services, and the ability to construct "grid-enabled" applications incrementally, by incorporating services one by one and/or by taking increasing advantage of translucent interfaces. Identification of the weaknesses of the approach will require the construction of larger testbeds and further experimentation with applications. One concern is that the basic techniques might not scale, perhaps because the local services defined by the Globus toolkit are too complex for broad deployment, or because the accuracy of the information provided by MDS declines below a useful level. We are investigating these issues.

We believe that the creation of large-scale testbeds must be a central part of any computational grid project. Hence, we are working with a variety of institutions around the world to create a permanent infrastructure to support experimentation with grid applications and grid software. The initial version of this GUSTO testbed already includes resources at some 17 institutions, and we expect this number to increase.

In current work, we are investigating both grid applications and more sophisticated grid services. We have started to investigate the construction of sophisticated resource brokers and robust co-allocation strategies. We are also studying how MDS can be used to support dynamic configuration and adaptation, so that applications can maintain high levels of performance in the face of dynamic changes in underlying system infrastructure. Finally, we are integrating quality of service mechanisms into the Globus framework. Our initial focus is on guaranteeing communication performance. However, we will also be studying how to integrate processor and memory scheduling into this framework.

For more information on the Globus project and toolkit, see the papers cited here and also the material

at `www.globus.org`.

Acknowledgments

We gratefully acknowledge the numerous contributions of the Globus team, without which the accomplishments detailed here would not have been possible: in particular, Steven Tuecke, Joe Bester, Joe Insley, Nick Karonis, Gregor von Laszewski, Stuart Martin, Warren Smith, and Brian Toonen at Argonne National Laboratory; Karl Czajkowski and Steve Fitzgerald at USC/ISI; and Craig Lee and Paul Stelling at The Aerospace Corporation. SF-Express was developed by Sharon Brunett, Paul Messina, and others at Caltech and JPL. The development of GUSTO was made possible by the considerable assistance offered by staff at each participating site.

This work was supported in part by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Computational and Technology Research, U.S. Department of Energy, under Contract W-31-109-Eng-38; by the Defense Advanced Research Projects Agency under contract N66001-96-C-8523; and by the National Science Foundation.

Author Biographies

Ian Foster received his Ph.D. in computer science from Imperial College, England. He holds a joint appointment as a scientist at Argonne National Laboratory and associate professor in the Department of Computer Science at the University of Chicago. He is the author of three books and over 100 articles and reports on various topics relating to parallel and distributed computing and computational science. In 1995, he led development of the software infrastructure for the I-WAY networking experiment. Dr. Foster co-leads the Globus project with Carl Kesselman.

Carl Kesselman is a project Leader at the Information Sciences Institute and a research associate professor in computer science at the University of Southern California. He received a Ph.D. in computer science at the University of California at Los Angeles. He co-leads the Globus project with Ian Foster. Dr. Kesselman's research interests include high-performance distributed computing, parallel computing, and parallel programming languages.

References

[1] D. Abramson, R. Sosic, J. Giddy, and B. Hall. Nimrod: A tool for performing parameterised simulations using distributed workstations. In *Proc. 4th IEEE Symp. on High Performance Distributed Computing*. IEEE Computer Society Press, 1995.

[2] A. D. Alexandrov, M. Ibel, K. E. Schauser, and C. J. Scheiman. Extending the operating system at the user level: The UFO global file system. In *1997 Annual Technical Conference on UNIX and Advanced Computing Systems (USENIX'97)*, January 1997.

[3] A. D. Alexandrov, M. Ibel, K. E. Schauser, and C. J. Scheiman. Superweb: Towards a global web-based parallel computing infrastructure. In *11th International Parallel Processing Symposium*, April 1997.

[4] S. Brunett, D. Davis, T. Gottschalk, P. Messina, and C. Kesselman. Implementing distributed synthetic forces simulations in metacomputing environments. In *Proceedings of the Heterogeneous Computing Workshop*, 1998. to appear.

[5] K. M. Chandy and C. Kesselman. CC++: A declarative concurrent object oriented programming notation. In *Research Directions in Object Oriented Programming*, pages 281–313. The MIT Press, 1993.

[6] D. E. Comer. *Internetworking with TCP/IP*. Prentice Hall, 3rd edition, 1995.

[7] P. Corbett, D. Feitelson, Y. Hsu, J.-P. Prost, M. Snir, S. Fineberg, B. Nitzberg, B. Traversat, and P. Wong. MPI-IO: A parallel file I/O interface for MPI. Technical Report NAS-95-002, NAS, NASA Ames Research Center, Moffett Field, CA, January 1995. Version 0.3.

[8] K. Czajkowski, I. Foster, C. Kesselman, S. Martin, W. Smith, and S. Tuecke. A resource management architecture for metacomputing systems. Preprint, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, Ill., 1997.

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

[10] I. Foster, J. Geisler, W. Gropp, N. Karonis, E. Lusk, G. Thiruvathukal, and S. Tuecke. A wide-area implementation of the Message Passing Interface. *Parallel Computing*, 1998. to appear.

[11] I. Foster, J. Geisler, C. Kesselman, and S. Tuecke. Managing multiple communication methods in high-performance networked computing systems. *Journal of Parallel and Distributed Computing*, 40:35–48, 1997.

[12] I. Foster, J. Geisler, W. Nickless, W. Smith, and S. Tuecke. Software infrastructure for the I-WAY metacomputing experiment. *Concurrency: Practice & Experience*, 1998. to appear.

[13] I. Foster and C. Kesselman. Globus: A metacomputing infrastructure toolkit. *International Journal of Supercomputer Applications*, 11(2):115–128, 1997.

[14] I. Foster and C. Kesselman, editors. *Computational Grids: The Future of High-Performance Distributed Computing*. Morgan Kaufmann Publishers, 1998.

[15] 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.

[16] I. Foster, D. Kohr, R. Krishnaiyer, and J. Mogill. Remote I/O: Fast access to distant storage. In *Proc. IOPADS'97*, pages 14–25. ACM Press, 1997.

[17] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, B. Manchek, and V. Sunderam. *PVM: Parallel Virtual Machine—A User's Guide and Tutorial for Network Parallel Computing*. MIT Press, 1994.

[18] A. Grimshaw, A. Nguyen-Tuong, and W. Wulf. Campus-wide computing: Results using Legion at the University of Virginia. Technical Report CS-95-19, University of Virginia, 1995.

[19] A. Grimshaw, W. Wulf, J. French, A. Weaver, and P. Reynolds, Jr. Legion: The next logical step toward a nationwide virtual computer. Technical Report CS-94-21, Department of Computer Science, University of Virginia, 1994.

[20] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing*, 22:789–828, 1996.

[21] K. Hickman. The SSL protocol. *Internet Draft RFC*, 1995.

[22] T. Howes and M. Smith. The ldap application program interface. RFC 1823, 08/09 1995.

[23] M. Lewis and A. Grimshaw. The core Legion object model. In *Proc. 5th IEEE Symp. on High Performance Distributed Computing*, pages 562–571. IEEE Computer Society Press, 1996.

[24] J. Linn. Generic security service application program interface. *Internet RFC 1508*, 1993.

[25] M. Litzkow, M. Livny, and M. Mutka. Condor - a hunter of idle workstations. In *Proc. 8th Intl Conf. on Distributed Computing Systems*, pages 104–111, 1988.

[26] P. Lyster, L. Bergman, P. Li, D. Stanfill, B. Crippe, R. Blom, C. Pardo, and D. Okaya. CASA gigabit supercomputing network: CALCRUST three-dimensional real-time multi-dataset rendering. In *Proc. Supercomputing '92*, 1992.

[27] Message Passing Interface Forum. *MPI-2: Extensions to the Message-Passing Interface*, 1997. http://www.mpi-forum.org.

[28] K. Moore, G. Fagg, A. Geist, and J. Dongarra. Scalable networked information processing environment (SNIPE). In *Proceedings of Supercomputing '97*, 1997.

[29] J. Steiner, B. C. Neuman, and J. Schiller. Kerberos: An authentication system for open network systems. In *Usenix Conference Proceedings*, pages 191–202. 1988.

[30] A. Vahdat, P. Eastham, C. Yoshikawa, E. Belani, T. Anderson, D. Culler, and M. Dahlin. WebOS: Operating system services for wide area applications. Technical Report UCB CSD-97-938, U.C. Berkeley, 1997.

NetSolve: a Network-Enabled Solver; Examples and Users.

Henri Casanova

Dept. of Computer Science
University of Tennessee
Knoxville, TN 37996-1301

Jack J. Dongarra

Dept. of Computer Science
University of Tennessee
Knoxville, TN 37996-1301
and
Mathematical Science Section
Oak Ridge National Laboratory
Oak Ridge, TN 37821-6367

Abstract

The NetSolve project, underway at the University of Tennessee and at the Oak Ridge National Laboratory, allows users to access computational resources distributed across the network. These resources are embodied in computational servers and allow the user to easily perform scientific computing tasks without having any computing facility installed on his/her computer. The user access to the servers is facilitated by a variety of interfaces: Application Programming Interfaces (APIs), Textual Interactive Interfaces and Graphical User Interfaces (GUIs). There are many research issues involved in the NetSolve system, including fault-tolerance, load balancing, user-interface design, computational servers, and network-based computing. As the project matures, several promising extensions and applications of NetSolve will emerge. In this article, we provide an overview of the project and examine some of the extensions being developed: An interface to the Condor system, an interface to the ScaLAPACK parallel library, a bridge with the Ninf system, and an integration of NetSolve and ImageVision.

1 The NetSolve project

1.1 Basics

Thanks to advances in hardware, networking infrastructure and algorithms, computing intensive problems in many areas can now be successfully attacked using networked, scientific computing. In the networked computing paradigm, vital pieces of software and information used by a computing process are spread across the network, and are identified and linked together only at run time. This is in contrast to the current software usage model where one acquires a copy (or copies) of task-specific software package for use on local hosts. One can distinguish three main paradigms for such systems. In *proxy computing*, the data and the program reside on the user's machine and are both sent to a server that runs the code on the data and returns the result. In *code shipping*, the program resides on the server and is downloaded to the user's machine, where it operates on the data and generates the result on that machine. This is the paradigm used by Java applets within Web browsers. NetSolve uses the *remote computing* paradigm: the program resides on the server; the user's data is sent to the server, where the appropriate programs or numerical libraries operate on it; the result is then sent back to the user's machine.

Figure 1 depicts the typical layout of the system. NetSolve provides the user with a pool of computational resources. These resources are computational servers that have access to ready-to-use numerical software. As shown in the figure, the computational servers can be running on single workstations, net-

0-8186-8365-1/98 $10.00 © 1998 IEEE

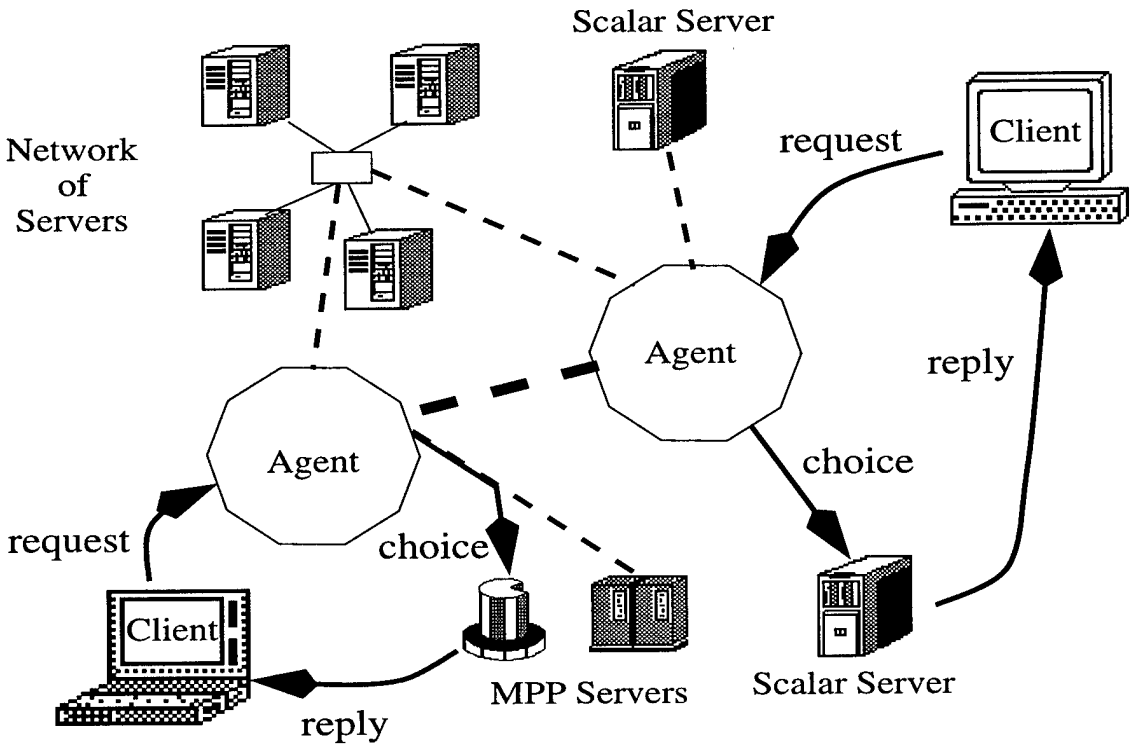


Figure 1: **NetSolve's organization**

works of workstations that can collaborate for solving a problem, or MPP (Massively Parallel Processor) systems. The user is using one of the NetSolve client interfaces. Through these interfaces, he can send requests to the NetSolve system asking for his numerical computation to be carried out by one of the servers. The main role of the NetSolve agent is to process this request and to choose the most suitable server for this particular computation in terms of execution time. Once a server has been chosen, it is assigned the computation, uses its available numerical software, and eventually returns the results to the user. One of the major advantages of this approach is that the agent performs load-balancing among the different resources.

As shown on Figure 1, there can be multiple instances of the NetSolve agent on the network, and different clients can contact different agents depending on their locations. The agents can exchange information about their different servers and allow access from any client to any server if desirable. NetSolve can be used either via the Internet or on an intranet, such as inside a research department or a university, without participating in any Internet based computation. Another important aspect of NetSolve is that the configuration of the system is entirely flexible: any server/agent can be stopped and (re-)started at any time without jeopardizing the integrity of the system.

1.2 The computational resources

When building the NetSolve system, one of the challenges was to design a suitable model for the computational servers. The NetSolve servers are configurable so that they can be easily upgraded to encompass ever-increasing sets of numerical functionalities. The NetSolve servers are also pre-installed, meaning that the end-user does not have to install any numerical software. Finally, the NetSolve servers provide uniform access the the numerical software, in the sense that the end-user has the illusion that she is accessing numerical subroutines from a single, coherent numerical library.

To make the implementation of such a computational server model possible, we have designed a general, machine-independent way of describing a numerical computation, as well as a set of tools to generate new computational modules as easily as possible. The main component of this framework is a *descriptive language* which is used to describe each separate nu-

merical functionality of a computational server. The description files written in this language can be compiled by NetSolve into actual computational modules executable on any UNIX or NT platform. These files can then be exchanged by any institution wanting to set up servers: each time a new description file is created, the capabilities of the entire NetSolve system are increased.

A number of description files have been generated for a variety of numerical libraries: ARPACK [1], FitPack [2], ItPack [3], MinPack [4], FFTPACK [5], LAPACK [6], BLAS [7, 8, 9], QMR [10], and ScaLAPACK [11]. These numerical libraries cover several fields of computational science; Linear Algebra, Optimization, Fast Fourier Transforms, etc.

NetSolve computational servers providing access to these libraries are currently running at the University of Tennessee and at other locations world-wide. Real-time information on the running servers can be found on the NetSolve web-page located at

`http://www.cs.utk.edu/netsolve`

1.3 The client interfaces

A major concern in designing NetSolve was to provide several interfaces for a wide range of users. NetSolve can be invoked through C, Fortran, Java, as well as on Matlab. In addition, there is a Web-enabled Java GUI. Another concern was keeping the interfaces as simple as possible. For example, there are only two calls in the MATLAB interface, and they are sufficient to allow users to submit problems to the NetSolve system. Each interface provides asynchronous calls to NetSolve in addition to traditional synchronous or blocking calls. When several asynchronous requests are sent to a NetSolve agent, they are dispatched among the available computational resources according to the load-balancing schemes implemented by the agent. Hence, the user—with virtually no effort—can achieve coarse-grained parallelism from either a C or Fortran program, or from interaction with a high-level interface. All the interfaces are described in detail in the "NetSolve's Client User's Guide" [12].

1.4 The NetSolve agent

1.4.1 The agent as a database

Keeping track of what software resources are available and on which servers they are located is perhaps the most fundamental task of the NetSolve agent. Since the computational servers use the same framework to contribute software to the system (see Section 1.2), it is possible for the agent to maintain a database of different numerical functionalities available to the users.

Each time a new server is started, it sends an application request to an instance of the NetSolve agent. This request contains general information about the server and the list of numerical functions it intends to contribute to the system. The agent examines this list and detects possible discrepancies with the other existing servers in the system. Based on the agent's verdict, the server can be integrated into the system and available for clients.

1.4.2 The agent as a resource broker

The goal of the NetSolve agent is to choose the best-suited computational server for each incoming request to the system. For each user request, the agent determines the set of servers that can handle the computation and makes a choice between all the possible resources. To do so, the agent uses computation-specific and resource-specific information. Computation-specific information is mostly included in the user request whereas resource-specific information is partly static (server's host processor speed, memory available, etc.) and partly dynamic (processor workload). The agent thus provides the user with location transparency for the processor performing her computation. Rationale and further detail on these issues can be found in [13], as well as a description of how NetSolve ensures fault-tolerance among the servers.

1.5 Conclusion

Agent-based computing seems to be a promising strategy. NetSolve will evolve into a more elaborate system in the future and a major part of this evolution is to take place within the agent. Such issues as user accounting, security, data encryption for instance are only partially addressed in the current implementation of NetSolve and will be the object of much work in the future. As the types of hardware resources and the types of numerical software available on the computational servers become more and more diverse, the resource broker embedded in the agent will need to become increasingly sophisticated. There are many difficulties in providing a uniform performance metric that encompasses any type of algorithmic and hardware considerations in a metacomputing setting, especially when different numerical resources, or even entire frameworks are integrated into NetSolve. Such integrations are described in the following sections.

2 An interface to the Condor system

2.1 Overview of Condor

Condor [14, 15, 16], developed at the University of Wisconsin, Madison, is an environment that can manage very large collections of distributively owned workstations. Its development has been motivated by

the ever increasing need for scientists and engineers to exploit the capacity of such collections, mainly by taking advantage of otherwise unused CPU cycles.

A brief description of Condor's software architecture follows. A Condor pool consists of any number of machines, that are connected by a network. Condor daemons constantly monitor the status of the individual computers in the cluster. Two daemons run on each machine, the *startd* and the *schedd*. The *startd* monitors information about the machine itself (load, mouse/keyboard activity, etc.) and decides if it is available to run a Condor job. The *schedd* keeps track of all the Condor jobs that have been submitted to the machine. One of the machine, the *Central Manager*, keeps track of all the resources and jobs in the pool. When a job is submitted to Condor, the scheduler on the central manager matches a machine in the Condor pool to that job. Once the job has been started, it is periodically checkpointed, can be interrupted and migrated to a machine whose architecture is the same as the one of the machine on which the execution was initiated. This organization is partly depicted in Figure 2. More details on the Condor system and the software layers can be found in [14].

2.2 A Condor pool as a NetSolve resource

2.2.1 Motivation

Interfacing NetSolve and Condor is a very natural idea. NetSolve provides remote easy access to computational resources through multiple, attractive user interfaces. Condor allows users to harness the power of a pool of machines while using otherwise unused CPU cycles. The users at the consoles of those machines are therefore not penalized by the scheduling of Condor jobs. If the pool of machines is reasonably large, it is usually the case that Condor jobs can be scheduled almost immediately. This could prove to be very interesting for a project like NetSolve. Indeed, NetSolve servers may be started so that they grant local resource access to outside users. Interfacing NetSolve and Condor could then give priority to local users and provide underutilized only CPU cycles to outside users.

2.2.2 Implementation

Figure 2 shows how an entire Condor pool can be seen as a single NetSolve computational resource. The Central Manager runs two daemons in addition to the usual *startd* and *schedd*: the *negotiator* and the *collector*. One machine also runs a customized version of the NetSolve server. When this server receives a request from a client, instead of creating a local child process running a computational module, it uses the Condor tools to submit that module to the Condor pool. The negotiator on the Central Manager then chooses a target machine for the computational module. Due to fluctuations in the state of the pool, the computational module can then be migrated among the machines in the pool. When the results of the numerical computation are obtained, the NetSolve server transmits that result back to the client.

The actual implementation of the NetSolve/Condor interface was made easy by the Condor tools provided to the Condor user. However, the restrictions that apply to a Condor jobs concerning system calls were difficult and required quite a few changes to obtain a Condor-enabled NetSolve server. A major issue however still needs to be addressed; how does the NetSolve agent perceive a Condor pool as a resource? Indeed, it is rather difficult to predict when the job will be scheduled or how often it will be suspended and migrated. Finding the appropriate performance prediction technique will be at the focus of the next step in the NetSolve/Condor collaboration.

3 Integrating parallel numerical libraries

3.1 Motivation

Integrating parallel packages into NetSolve will allow a user on a workstation to access MPP systems to perform large computation. This access can be extremely simple and the user may not even be aware that he is using a parallel library.

3.2 Integrating parallel software packages into NetSolve

ScaLAPACK (Scalable Linear Algebra Package) is a library of high-performance linear algebra routines for distributed-memory message-passing MIMD computers as well as networks of workstations supporting PVM [17] or MPI [18]. ScaLAPACK was developed at the University of Tennessee, Knoxville, the Oak Ridge National Laboratory and the University of California, Berkeley. It is a continuation of the LAPACK [6] project, and contains routines for solving systems of linear equations, least squares problems, and eigenvalue problems. ScaLAPACK views the underlying multi-processor system as a rectangular process *grid*. Global data is mapped to the local memories of the processes in that grid assuming specific data-distributions. For performance reasons, ScaLAPACK uses the two-dimensional block cyclic distribution scheme for dense matrix computations. Interprocess communication within ScaLAPACK is done via the BLACS (Basic Linear Algebra Communication

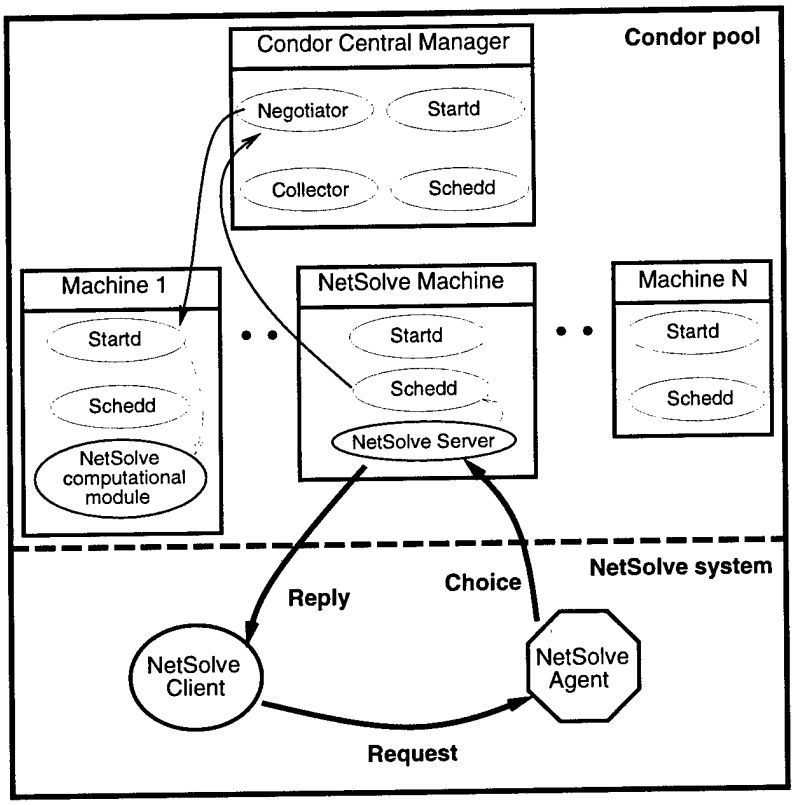


Figure 2: **NetSolve and Condor**

subprograms) [19, 20]. All the details on ScaLAPACK and its software hierarchy can be found in the latest edition of the User's Guide [11].

Figure 3 is a very simple description of how the NetSolve server has been customized to use the ScaLAPACK library. The customized server receives data input from the client in the traditional way. The NetSolve server uses BLACS calls to set up the ScaLAPACK processor grid. ScaLAPACK requires that the data already be distributed among the processors prior to any library call. This is the reason why each user input is first 2-D block cyclic distributed in that grid when necessary. The server can then initiate the call to ScaLAPACK and wait until completion of the computation. When the ScaLAPACK call returns, the result of the computation is usually available on the processors and is 2-D block cyclic distributed as well. The server then gathers that result and sends it back to the client in the expected format. This process is completely transparent to the user who does not even realize that a parallel execution is taking place.

This approach is very promising. A client can use MATLAB on a PC and issue a simple call like `[x] = netsolve('eig',a)` and have an MPP system use a high-performance library to perform a large eigenvalue computation. We have designed a prototype of the customized server running on top of PVM [17] or MPI [18]. There are many research issues arising with integrating parallel libraries in NetSolve, including performance prediction, choices of processor-grid/matrix-block size, choice of numerical algorithm, processor availability, accounting, etc.

4 NetSolve and Ninf

4.1 A brief overview of Ninf

Ninf, developed at the Electrotechnical Laboratory, Tsukuba, is a global network-wide computing infrastructure project which allows users to access computational resources including hardware, software, and scientific data distributed across a wide area network with an easy-to-use interface. Computational resources are shared as Ninf remote libraries and are executable at remote Ninf servers. Users can build an application by calling the libraries with the Ninf Remote Procedure Call, which is designed to provide a programming interface similar to conventional function calls in existing languages, and is tailored for scientific computation. In order to facilitate location

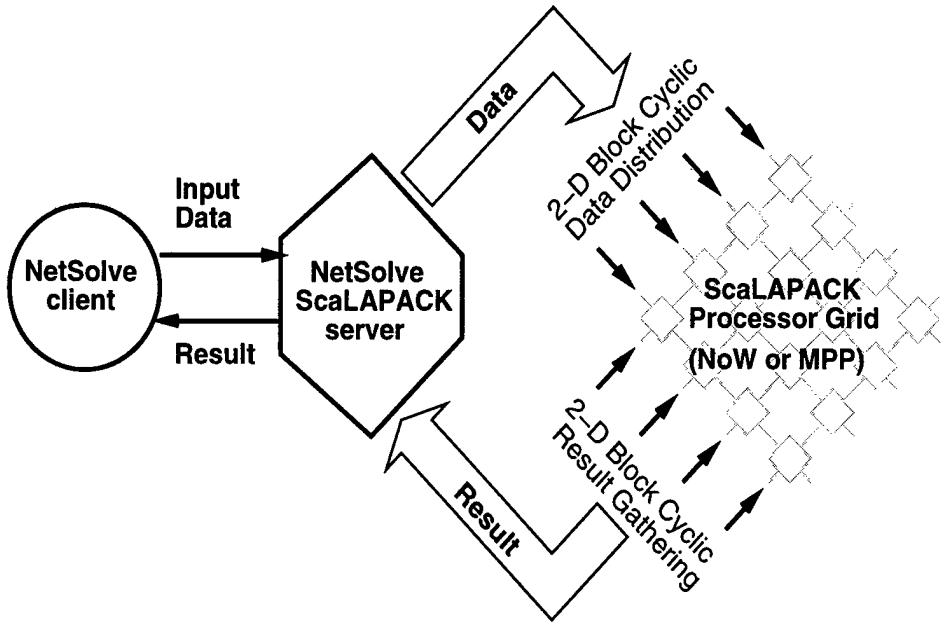


Figure 3: **The ScaLAPACK NetSolve Server Paradigm**

transparency and network-wide parallelism, the Ninf MetaServer maintains global resource information regarding computational server and databases. It can therefore allocate and schedule coarse-grained computations to achieve good global load balancing. Ninf also interfaces with existing network service such as the WWW for easy accessibility. More details on Ninf can be found in [21]. Clearly, NetSolve and Ninf bear strong similarities both in motivation and general design. Allowing the two systems to coexist and collaborate should lead to promising developments.

4.2 A gateway between Ninf and NetSolve

Some design issues prevent an immediate seamless integration of the two softwares (conceptual differences between the NetSolve agent and the Ninf Metaserver, problem specifications, user interfaces, data transfer protocols, etc.). In order to overcome these issues, the Ninf team started developing two *adapters*: a NetSolve-Ninf adapter and a Ninf NetSolve-adapter. Thanks to those adapters, Ninf clients can use computational resources administrated by a NetSolve system and vice-versa.

Figure 4(i) shows the Ninf-NetSolve adapter allowing access to Ninf resource from a NetSolve client. The adapter is just seen by the NetSolve agent as any other NetSolve server. When a NetSolve client sends a request to the agent, it can then be told to use the NetSolve adapter. The adapter performs protocol translation, interface translation, and data transfer, asks a Ninf server to perform the required computation and returns the result to the user.

In Figure 4(ii), the NetSolve-Ninf adapter can be seen by the Ninf MetaServer as a Ninf server, but in fact plays the role of a NetSolve client. This is a little different from the Ninf-NetSolve adapter because the NetSolve agent is a resource broker whereas the Ninf MetaServer is a proxy server. Once the adapter receives the result of the computation from some NetSolve server, it transfers that result back to the Ninf client.

There are several advantages of using such adapters. Updating the adapters to reflects the evolutions of NetSolve or Ninf seems to be an easy task. Some early implementation evaluations tend to show that using either system via an adapter causes acceptable overheads, mainly due to additional data transfers. Those first experiments appear encouraging and will definitely be extended to effectively enable an integration of NetSolve and Ninf.

5 Extending ImageVision by the use of NetSolve

In this section, we describe how NetSolve can be used as a building block for a general purpose framework for basic image processing.

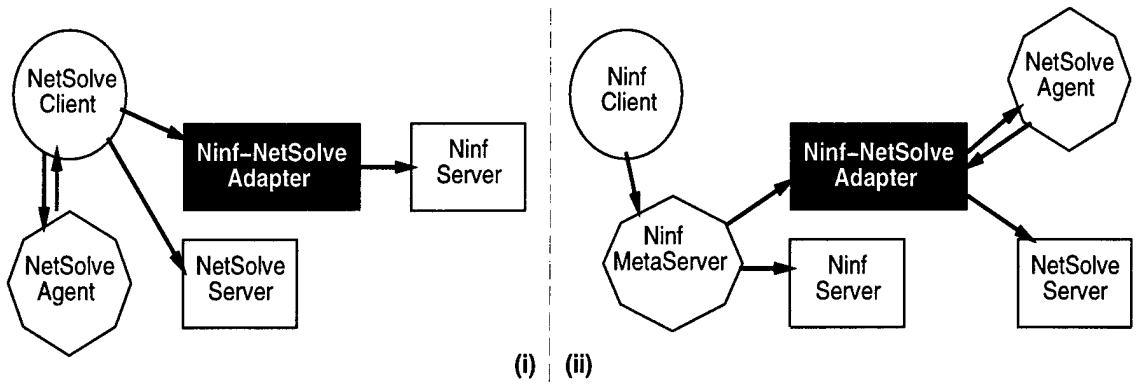


Figure 4: **Going (i) from NetSolve to Ninf and (ii) from Ninf to NetSolve**

5.1 Integrating the ImageVision library into NetSolve

This project is under development at the ICG institute at Graz University of Technology, Austria. The scope of the project is to make basic image processing functions available for remote execution over a network. The goals of the project include two objectives that can be leveraged by NetSolve. First, the resulting software should prevent the user from having to install complicated image processing libraries. Second, the functionalities should be available via Java-based applications. The ImageVision Library (IL) [22] is an object-oriented library written in C++ by Silicon Graphics, Inc. (SGI) and shipped with newer workstations. It contains typical image processing routines to efficiently access, manipulate, display, and store image data. ImageVision has been judged quite complete and mature by the research team at ICG and seems therefore a good choice as an "engine" for building a remote access image processing framework. Such a framework will make IL accessible from any platform (and not only from SGI workstations) and is described in [23].

5.2 NetSolve as an operating environment for ImageVision

The reasons why NetSolve has been a first choice for such a project are diverse. First, NetSolve is easy to understand, use, and extend. Second, NetSolve is freely available. Third, NetSolve provides language binding to Fortran, C, and Java. And finally, NetSolve's agent-based design allows load monitoring and balancing among the available servers. New NetSolve computational modules corresponding to the desired image processing functionalities will be created and integrated into the NetSolve servers. A big part of the project at ICG is to build a Java GUI to IL.

Figure 5 shows a simple example of how ImageVision can be accessed via NetSolve. A Java GUI can be built on top of the NetSolve Java API. As shown on the figure, this GUI could offer visualization capabilities. For computations, it uses an embedded NetSolve client and contacts SGI servers that have access to IL. The user of the Java GUI does not realize that NetSolve is the back end of the system, or that he uses a SGI library without running the GUI on a SGI machine! The protocol depicted on Figure 5 is of course simplistic. In order to obtain acceptable levels of performance, the network traffic needs to be minimized. There are several ways of attacking this problem. For instance, the servers could "keep a state", meaning that some data can be cached on the server for future use. Several issues are involved in the design of such a mechanism as the cache needs proper invalidation mechanisms, replacement policies, etc. Another possibility would be to combine requests to avoid retransmitting redundant data. Such a change can be already emulated my designing appropriate problem description file for the NetSolve servers. However, it may become preferable to include request combination as a standard feature of the NetSolve protocol. The current version of the Java API in GUI in NetSolve allows to reference objects (e.g. images) via URLs. This may prove useful for some applications, and in particular to the ImageVision/NetSolve integration.

6 Conclusion

The scientific community has long used the Internet for communication of email, software, and documentation. Until recently there has been little use of the network for actual computations. This situation is changing rapidly and will have an enormous impact on the future.

We have discussed throughout this article how Net-

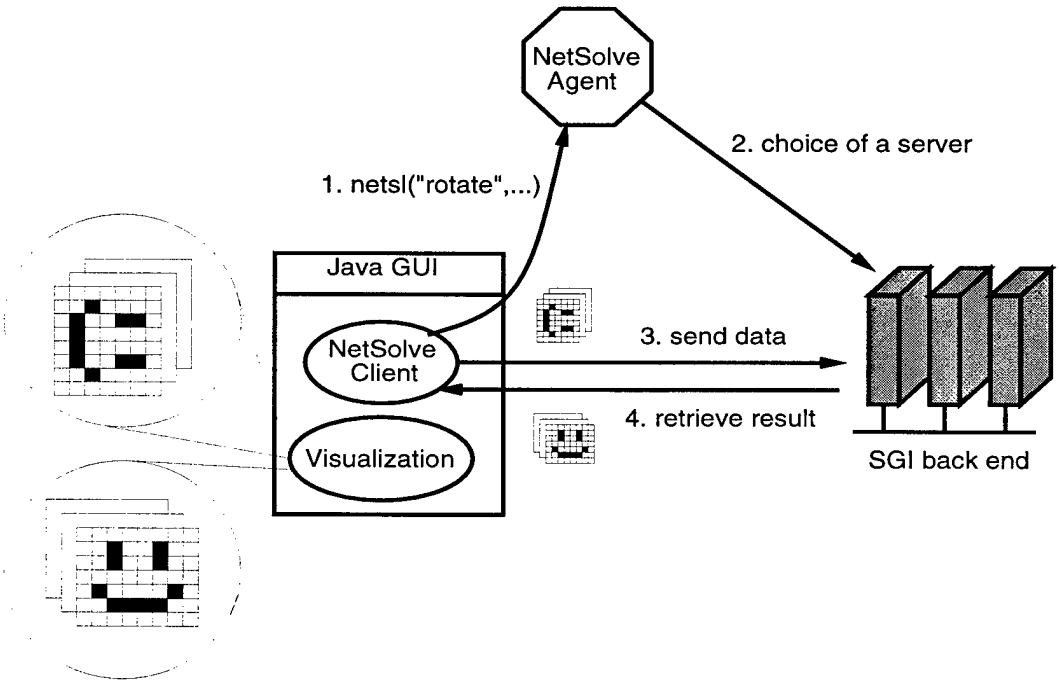


Figure 5: **ImageVision and NetSolve**

Solve can be customized, extended, and used for a variety of purposes. We first described in Sections 2 and 3 how NetSolve can encompass new types of computing resources, resulting in a more powerful and flexible environment and raising new research issues. We next discussed in Section 4 how NetSolve and Ninf can be merged into a single metacomputing environment. Finally, in Section 5, we gave an example of an entire application that uses NetSolve as an operating environment to build general image processing framework. All these developments take place at different levels in the NetSolve project and have had and will continue to have an impact on the project itself, causing it to improve and expand.

References

[1] R. Leboucq, D. Sorensen, and C. Yang. *ARPACK Users Guide*. 1997.

[2] A. Cline. Scalar- and Planar-Valued Curve Fitting Using Splines Under Tension. *Communications of the ACM*, 17:218–220, 1974.

[3] D. Young, D. Kincaid, J. Respess, and R. Grimes. Itpack2c: a FORTRAN package for solving large sparse linear systems by adaptive accelerated iterative methods. Technical report, University of Texas at Austin, Boeing Computer Services Company, 1996.

[4] J. Moré, B. Garbow, and K. Hillstrom. Minpack : Documentation file accessible at: "http://www.netlib.org/minpack/readme".

[5] P. Swarztrauber. FFTPACK : Documentation file accessible at: "ftp://ftp.ucar.edu/ftp/dsl/lib/fftpack/readme".

[6] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen. *LAPACK Users' Guide, Second Edition*. SIAM, Philadelphia, PA, 1995.

[7] C. Lawson, R. Hanson, D. Kincaid, and F. Krogh. Basic Linear Algebra Subprograms for Fortran Usage. *ACM Transactions on Mathematical Software*, 5:308–325, 1979.

[8] J. Dongarra, J. Du Croz, S Hammarling, and R. Hanson. An Extended Set of Fortran Basic Linear Algebra Subprograms. *ACM Transactions on Mathematical Software*, 14(1):1–32, 1988.

[9] J. Dongarra, J. Du Croz, I. Duff, and S. Hammarling. A Set of Level 3 Basic Linear Algebra Subprograms. *ACM Transactions on Mathematical Software*, 16(1):1–17, 1990.

[10] R.W. Freund and N.M. Nachtigal. QMR: A quasi-minimal residual method for non-Hermitian linear systems. *Numer. Math.*, 60:315–339, 1991.

[11] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. *ScaLAPACK Users' Guide.* Society for Industrial and Applied Mathematics, Philadelphia, PA, 1997.

[12] H. Casanova, J. Dongarra, and K. Seymour. Client User's Guide to Netsolve. Technical Report CS-96-343, Department of Computer Science, University of Tennessee, 1996.

[13] H Casanova and J. Dongarra. NetSolve: A Network Server for Solving Computational Science Problems. *The International Journal of Supercomputer Applications and High Performance Computing*, 11(3):212–223, 1997.

[14] M. Litzkow, M. Livny, and M.W. Mutka. Condor - A Hunter of Idle Workstations. In *Proc. of the 8th International Conference of Distributed Computing Systems*, pages 104–111. Department of Computer Science, University of Winsconsin, Madison, June 1988.

[15] M. Litzkow and M. Livny. Experience with the Condor Distributed Batch System. In *Proc. of IEEE Workshop on Experimental Distributed Systems.* Department of Computer Science, University of Winsconsin, Madison, 1990.

[16] J. Pruyne and M. Livny. A Worldwide Flock of Condors : Load Sharing among Workstation Clusters. *Journal on Future Generations of Computer Systems*, 12, 1996.

[17] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM : Parallel Virtual Machine. A Users' Guide and Tutorial for Networked Parallel Computing.* The MIT Press Cambridge, Massachusetts, 1994.

[18] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra. *MPI : The Complete Reference.* The MIT Press Cambridge, Massachusetts, 1996.

[19] J. Dongarra and R. van de Geijn. Two dimensional basic linear algebra communication subprograms. Technical Report CS-91-138, Computer Science Dept, University of Tennessee, Knoxville, TN, 1991. Also LAPACK Working Note #37.

[20] R.C. Whaley and J. Dongarra. A user's guide to the BLACS v1.1. Technical Report CS-95-281, Computer Science Dept, University of Tennessee, Knoxville, TN, 1995. Also LAPACK Working Note #118.

[21] 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*, 1996.

[22] G. Eckel, J. Neider, and E. Bassler. *ImageVision Library Programming Guide.* Silicon Graphics, Inc., Mountain View, CA, 1996.

[23] M. Oberhuber. Integrating ImageVision into NetSolve. Available at http://www.icg.tu-graz.ac.at/mober/pub, October 1997.

Biographies

Jack J. Dongarra holds a joint appointment as Distinguished Professor of Computer Science in the Computer Science Department at the University of Tennessee (UT) and as Distinguished Scientist in the Mathematical Sciences Section at Oak Ridge National Laboratory (ORNL) under the UT/ORNL Science Alliance Program. He specializes in numerical algorithms in linear algebra, parallel computing, use of advanced-computer architectures, programming methodology, and tools for parallel computers. Other current research involves the development, testing and documentation of high quality mathematical software. He was involved in the design and implementation of the software packages EISPACK, LINPACK, the BLAS, LAPACK, ScaLAPACK, Netlib, PVM, MPI, the National High-Performance Software Exchange and NetSolve; and is currently involved in the design of algorithms and techniques for high performance computer architectures.

Professional activities include membership in the Society for Industrial and Applied Mathematics (SIAM), the Institute of Electrical and Electronics Engineers (IEEE), the Association for Computing Machinery (ACM), and a Fellow of the American Association for the Advancement of Science (AAAS).

He has published numerous articles, papers, reports and technical memoranda, and has given many presentations on his research interests.

http://www.netlib.org/utk/people/JackDongarra/

Henri Casanova earned the Applied Mathematics and Computer Science Engineer degree from

the Ecole Nationale Supérieure d'Electrotechnique, d'Electronique, d'Informatique et d'Hydraulique de Toulouse (ENSEEIHT), Toulouse, France in 1993, as well as the Diplôme d'Etudes Approfondies in Parallel Architectures and Software Engineering from the Université Paul Sabatier, Toulouse, France. In 1992-1993 he was a trainee at the Institut de Recherche en Informatique de Toulouse (IRIT), in Toulouse, France. From November 1993 until November 1994 he did his military service working from the French Ministry of Defense (DGA) as Advisor for the Computer Science Division of the DPAG. In January 1995, he entered the PhD program in Computer Science at the University of Tennessee, Knoxville and has been working as a Graduate Research Assistant since then.

His research interests are diverse and include metacomputing, Internet-based computing, parallel and scientific computing, stochastic modeling (Markov chains, Large Deviation Theory) and performance prediction for distributed computing applications. Casanova is the main developer of the NetSolve project and still maintains the major part of the software.

Implementing Distributed Synthetic Forces Simulations in Metacomputing Environments

Sharon Brunett, Dan Davis, Thomas Gottschalk, Paul Messina
Center for Advanced Computing Research
California Institute of Technology
Pasadena, California 91125

Carl Kesselman
University of Southern California
Information Sciences Institute
Marina del Rey, CA 90292

Abstract

A distributed, parallel implementation of the widely used Modular Semi-Automated Forces (ModSAF) Distributed Interactive Simulation (DIS) is presented, with Scalable Parallel Processors (SPPs) used to simulate more than 50,000 individual vehicles. The single-SPP code is portable and has been used on a variety of different SPP architectures for simulations with up to 15,000 vehicles. A general metacomputing framework for DIS on multiple SPPs is discussed and results are presented for an initial system using explicit Gateway processes to manage communications among the SPPs. These 50K-vehicle simulations utilized 1,904 processors at six sites across seven time zones, including platforms from three manufacturers. Ongoing activities to both simplify and enhance the metacomputing system using Globus are described.

1 The Large-Scale DIS Problem

Over the past few years, Distributed Interactive Simulation (DIS) [1] has become an increasingly essential tool for training, system acquisition, test and evaluation within the Department of Defense. Key components of DIS include: high-fidelity computer-simulated individual entities (tanks, trucks, aircraft, ...); interactions among entities hosted on different computers through network messages; and support for Human In Loop (HIL) interactions. Using DIS, it is possible to create large-scale virtual representations of real operational environments that are inexpensive enough to be used repeatedly.

ModSAF is a particularly important example of DIS which is routinely used for cost-effective training throughout the armed forces. Generally, it is run using an ensemble of workstations communicating over a network, typically a LAN. Each workstation is responsible for simulating some modest number (30–100) of entities. These computer-generated Semi-Automated Forces (SAF) are intended to mimic realistically the behaviors of opposing or support forces within an exercise. The entity and environment models are accordingly quite detailed.

Individual simulators (workstations) interact through the exchange of data messages called PDUs (Protocol Data Units) [2]. These PDUs are used in ModSAF to describe the state of individual entities, weapons firing, detonations, environmental phenomenon, command and control orders, etc. In standard ModSAF, the PDUs are sent as UDP datagrams. Due to this unreliable message-delivery mechanism, each entity state PDU typically contains a complete summary of the vehicle's current state, and PDUs are (re)transmitted at frequent, regular "heartbeat" intervals to compensate for dropped data packets.

Independent of the nature of the PDU communications mechanism, this simplest picture of ModSAF is not scalable in that it (implicitly) assumes each simulator receives and responds to all PDUs from all other simulators—a model that clearly fails as the number of simulators and simulated entities increases. Moreover, in many realistic large scale simulations, it is invariably the case that most system-wide PDU traffic is irrelevant for the limited set of entities hosted on an individual simulator (e.g., tanks separated by tens of kilometers generally do not interact).

The DIS community encountered these issues in their STOW-E exercise (Synthetic Theater of War-Europe [3]) and ED-1A Engineering Demonstration [4], in which ModSAF was used to simulate 5,371 vehicles hosted at 12 separate sites in the USA and Eu-

0-8186-8365-1/98 $10.00

rope. Increasing the simulated entity count could not be achieved by simply adding more workstations to the network. Addition of a PDU screening mechanism ('Interest Management') helped but did not eliminate all scaling hurdles.

This paper describes a new approach to truly large-scale DIS, using multiple Scalable Parallel Processors (SPPs) to solve the scaling problems observed in STOW-E. After a short summary of project goals and accomplishments in Sections 1.1 and 1.2, Section 2 presents the general method used for porting ModSAF to run on an SPP. Sections 3-5 contain, respectively, a (long-term) vision for an effective STOW metacomputing model, an analysis of initial multi-SPP ModSAF accomplishments, and an overview of ongoing activities to enhance and extend the existing software using elements from the Globus metacomputing toolkit [7], [8].

1.1 SF Express Project Overview

The Synthetic Forces Express project (SF Express) [9] began in 1996 to explore the utility of Scalable Parallel Processors (SPPs) as a solution to the communications bottlenecks of conventional ModSAF. The SF Express team consists of researchers from the California Institute of Technology (Caltech), the Jet Propulsion Laboratory (JPL), and the Space and Naval Warfare Systems Center San Diego (SPAWARSYSCEN, formerly known as NRaD). The SF Express charter was to demonstrate a scalable communications architecture simulating 50K vehicles on multiple SPPs—an order-of-magnitude increase over the size of the STOW-E simulation.

SPPs provide a natural, attractive alternative to networked workstations for large-scale ModSAF runs. Most of the processors on an SPP can be devoted to independent executions of "SAFSim," the basic ModSAF simulator code. The reliable high-speed communications fabric between processors on an SPP provides significantly increased bandwidth over standard dataflows among networked workstations. A scalable communications scheme was constructed in three main steps:

Interest Specification Procedures: Individual data messages were associated with specific interest class indices, and procedures were developed for evaluating the total interest state of an individual simulation processor.

Intra-SPP Communications: Within an individual SPP, certain processors were designated as message routers; the number of processors used as routers can be selected for each run. These processors receive and store interest declarations from the simulator nodes and move simulation data packets according to the interest declarations.

Inter-SPP Communications: Additional interest-restricted data exchange procedures were developed to support SF Express execution across multiple SPPs.

The primary technical challenge in porting ModSAF to run efficiently on SPPs lies in constructing a suitable network of message-passing router nodes/processors. SF Express uses point-to-point SPP communications (implemented using the MPI Message Passing Interface [10]) to replace the UDP socket calls of standard ModSAF. The network of routers manage SPP message traffic, effecting interest-restricted communications among simulator nodes. This strategy allows considerable freedom in constructing the router node network. This paper describes a model based on statically-allocated communication channels among specific subsets of processors within an SPP. This Router Network Architecture (RNA) was developed at Caltech [11],[12].

As the simulation problem size increases beyond the capabilities of any single SPP, additional interest-restricted communications procedures are needed to enable "Metacomputed ModSAF" runs on multiple SPPs. After a number of options were considered, an implementation using dedicated Gateway processors to manage inter-SPP communications was selected.

1.2 Simulations of 50K+ Vehicles

On 11 August 1997, the SF Express project performed two separate simulation runs, each with more than 50,000 individually simulated vehicles. The runs used three different types of Scalable Parallel Processors (SPPs) at six separate sites spanning seven time zones, as shown in Fig.(1). These sites were linked by a variety of wide-area networks. Specifics for each site are listed in Table 1. The majority of the SPPs used the RNA communications scheme, while NASA Ames and CEWES applied an alternative approach developed at JPL [13].

The $N(P)$ entries in the table indicate the number of processors used at each site. The $N(V)_j$ columns indicate the number of locally simulated vehicles in each of the two runs. The 50K-vehicle simulation scenarios were created by the ExInit software [14] and featured immediate intense interactions among the simulated entities, causing high communications levels both within and among SPPs.

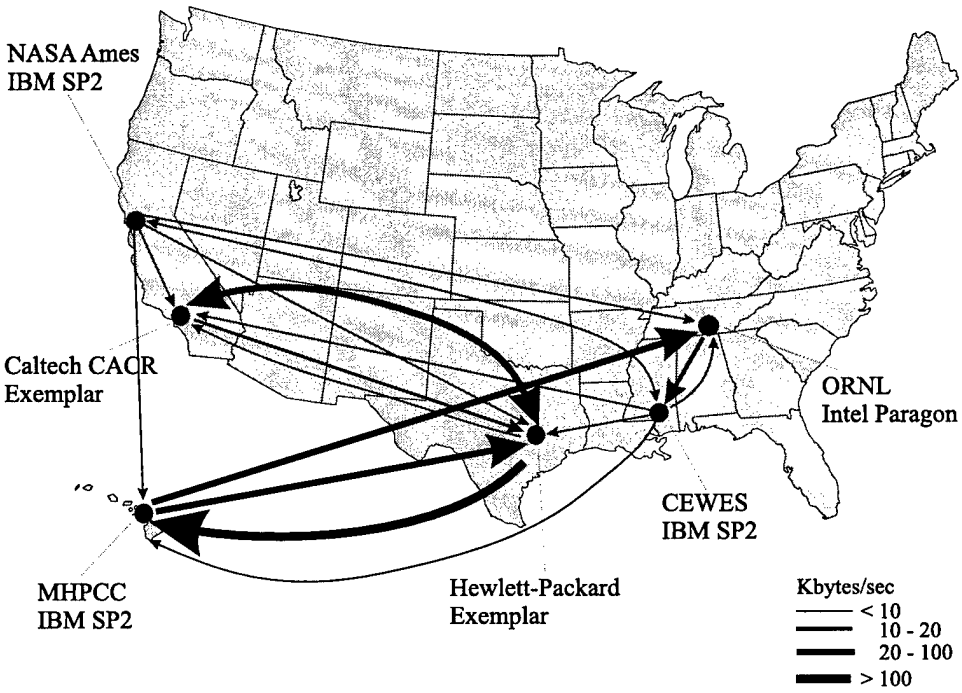


Figure 1: SPP sites and message rates in the 50K SF Express runs

Table 1: Participating Sites and Simulated Entity Counts for the 50,000 Vehicle SF Express Runs

Site	Hardware	N(P)	$N(V)_1$	$N(V)_2$
Caltech, Pasadena CA	HP Exemplar	256	13,095	12,182
ORNL, Oak Ridge TN	Intel Paragon	1024	16,695	15,996
NASA Ames CA	IBM SP2	139	5,464	5,637
CEWES, Vicksburg MS	IBM SP2	229	9,739	9,607
MHPCC, Maui HI	IBM SP2	128	5,056	7,027
HP/Convex, Richardson TX	HP Exemplar	128	5,348	6,733
Total		1,904	55,397	57,182

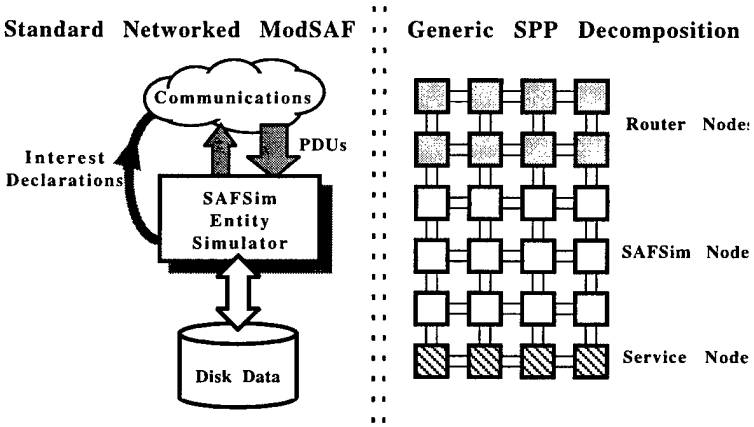


Figure 2: Schematic illustration of a networked ModSAF simulator and notional mapping of the simulator tasks onto an SPP

2 Porting ModSAF to a Scalable Parallel Processor

The basic strategy used in porting ModSAF to an SPP is a heterogeneous assignment of tasks to processors, as illustrated in Fig.(2). The processors are divided into three classes:

Entity Simulators: Most of the SPP's processors execute a minimally modified version of SAFSim, the standard simulator code.

Data Servers: A small number of nodes read and store simulation data, forwarding it to the SAFSim nodes through SPP messages.

Routers: The movement of data among the SAFSim nodes is managed by a number of dedicated router nodes. The broadcast or multicast socket calls of standard ModSAF are replaced by point-to-point communications directed by this router network.

Neither side of Fig.(2) is scalable without the imposition of additional interest management logic, which limits the number of incoming data for an individual SAFSim. Since interest management is an active research area, it is important that the SPP implementation not depend on specifics of any one interest management scheme. RNA makes only two minimal assumptions in this regard: each PDU can be associated with an interest value (an "interest class"), and each SAFSim can compute its own interest state (the set of all relevant interest values for locally simulated vehicles). The communications network must deliver to the SAFSim only those PDUs that overlap the SAFSim's declared interest state.

2.1 The Router Network Architecture

The basic building block of Router Network Architecture is a fixed set of SAFSim nodes communicating with single "Primary Router" node, as illustrated in Fig.(3). There are only two essential modifications to the standard ModSAF code, as run of the SAFSim nodes of Fig.(3):

1. The usual (broadcast) network reads and writes in the ModSAF network communications library are replaced by SPP communications with the router node.

2. Each SAFSim node periodically recomputes its collective interest state (union of interest states for all locally simulated vehicles) and sends this information to its router.

The Primary Router in Fig.(3) receives and (temporarily) stores PDUs and interest declarations from the attached SAFSims and subsequently forwards those PDUs that match the SAFSim interest states. These tasks are implemented using three straightforward constructs:

1. A large circular buffer that stores active data elements.

2. A client list that maintains the current interest declaration of the individual attached SAFSims and pointers to the next outgoing PDU for each client.

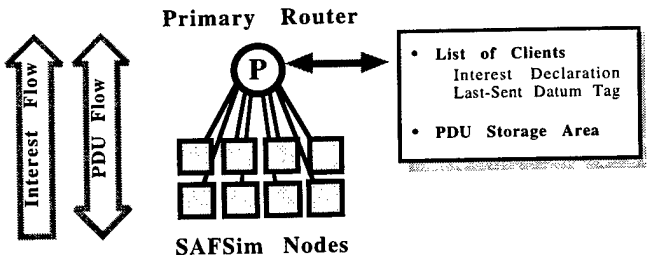


Figure 3: A Primary Router with its associated SAFSims

3. A simple interest assessment function that determines whether a PDU matches a client's declared interest.

The Primary Router in Fig.(3) is a pure data server that waits for and processes requests from SAFSim clients. For efficiency, the actual data messages exchanged between SAFSims and Routers are PDU bundles.

It has been found that a single Primary Router can comfortably manage the communications for a set of client SAFSims in Fig.(3) simulating 1K-2K total vehicles. Multiple replicas of the Primary Router Cluster are required once the overall simulation size exceeds this limit. In such cases, the basic unit of Fig.(3) is first augmented by the addition of two new Router nodes (referred to as "Pop-Up" and "Pull Down"). This enhanced routing "triad" is replicated, and additional communications links between Pop-Up and Pull-Down routers are enabled, giving rise to the full router network shown in Fig.(4).

Communications within the full architecture of Fig.(4) are also straightforward. In addition to its normal communications with the SAFSim nodes, each Primary Router forwards all SAFSim PDUs to its associated Pop-Up Router and also sends its collective interest state (the union of the SAFSim interest states) to its Pull-Down Router. Each Pull-Down router subsequently collects interest-filtered PDUs from the full Pop-Up layer, and delivers these data to the Primary Router. Message passing within the router network follows a strict set of hierarchical rules. In particular, all data exchanges are flow-controlled, being initiated by small request packets sent from one node to a router in a higher layer within Fig.(4). This approach is used to prevent both communications deadlocks and the arrival of large unanticipated messages that could exceed available system buffer space.

The Pop-Up layer in Fig.(4) provides a distributed repository for active messages within the simulation (making the Pop-Ups a perfect place to attach data loggers for subsequent replays or statistics gathering). Note that the data collection activities of the Pull-Down routers occur in parallel with the Primary SAFSim communications. This parallelism minimizes the additional time delays for PDUs that must travel through the full router network.

2.2 Performance of the Single-SPP ModSAF Implementation

Detailed studies of the RNA model are contained in Refs.[11],[12]. Highlights of these analyses are as follows:

1. The RNA approach has been run successfully on a variety of SPP architectures, including the Intel Paragon, IBM SP2, HP Exemplar, Silicon Graphics Origin 2000, and "Beowulf" PC Cluster [15].

2. These single-SPP runs have included simulations involving up to 18,000 vehicles.

3. The scaling behavior of RNA as problem size increases is well-understood, with "theoretical" expectations validated by the measured performance results.

4. The effective inter-processor communications within an SPP reduce PDU communication overhead significantly for an individual SAFSim (relative to standard ModSAF performance on a LAN/WAN network).

3 Anatomy of a DIS Metacomputer

"Metacomputing" can be defined as the concurrent use of multiple network-linked resources for solving very large computational problems. However,

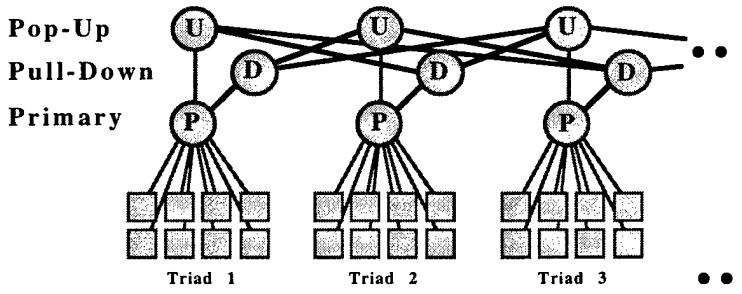


Figure 4: The full RNA router network

computing in networked environments has both advantages and drawbacks. The state and structure of networked resources are often dynamic and quite heterogeneous. Performance and portability may be compromised when trying to deal with heterogeneity. Alternatively, linking large numbers of diverse resources allows access to processing power and unique capabilities beyond the resources at any one site. It also enables applications to be solved with a mix of systems, assigning appropriate and available assets to specific parts of the overall problem.

For many classes of large distributed applications, the aggregate computational power in a collection of SPPs is only part of the metacomputing solution. A full system would link computational engines, storage systems, scientific instruments, advanced display devices, and human resources, as illustrated in Fig.(5), with "HIL" representing some sort of 'Human In Loop' interface and "Idesk" ("Immersa-desk") representing a typical advanced display device. Data may be gathered from a remote source (for example, a satellite downlink) and streamed into a collection of SPPs for real-time simulation processing. During the course of the simulation, mechanisms for logging, filtering, or compressing data may be employed for subsequent post-processing (e.g., visualization, querying, and persistent storage).

Distributed heterogeneous computing immediately implies diversity in terms of hardware architectures and performance, operating systems, administrative domains, network protocols, etc. As the size and complexity of the distributed system increases, operational issues (e.g., resource scheduling, allocation, and data staging) become increasingly important components of the metacomputing model.

The next two sections describe two initial steps toward the seamless metacomputing picture of Fig.(5). Section 4 presents the Gateway model used by the initial 50K-vehicle runs outlined in Table 1. Section 5 describes subsequent multi-SPP experiments to integrate parts of the Globus metacomputing toolkit [7], [8] in order to remove many operational difficulties encountered during initial large simulations.

4 SF Express on Multiple SPPs using Explicit Gateways

For large runs on multiple SPPs, some portions of the entity state information from each SPP will, in general, be relevant for entities simulated on other SPPs. Extensions of the single-SPP architecture must effect interest-restricted PDU exchanges among the SPPs. Dedicated Gateway processors provide a straightforward mechanism for this task.

The Gateway processors are generalizations of the intra-SPP routers from Section 2.1, and can be viewed as communications servers for two distinct classes of clients:

Local Clients: Router nodes on the same SPP as the Gateway that hold the continually changing collective PDU and Interest State of the local SPP. Local Clients send (internal) interest declarations and simulation data to the Gateway for subsequent delivery to remote resources.

External Clients: Processes on remote machines that receive and process interest declarations and PDU bundles from the local SPP. An external client could be a standard ModSAF workstation or GUI. For inter-SPP links, an External Client is essentially a mirror image of a Local Client that resides on the external SPP.

Gateways manage interest-selected data flow in two directions by way of four basic operations:

1. The collective interest state of the Local SPP is sent out to each of the external SPPs.

2. The corresponding interest declarations are received from the remote SPPs, defining standard client interests. The union of these external interest states defines the collective (external) gateway

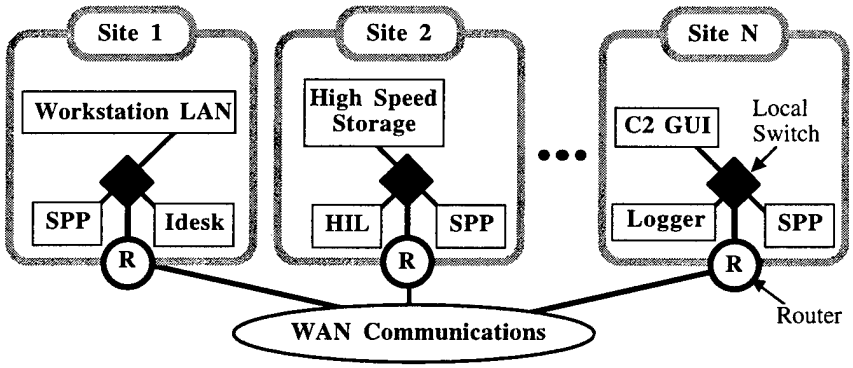


Figure 5: Schematic of a DIS metacomputing environment

interest, which is sent up to the local attached routers.

3. The Gateway receives interest-screened data from the local routers in the usual manner, and forwards these to the appropriate external hosts.

4. The Gateway receives data from the external SPPs and sends it to the attached local routers for subsequent distribution within the local SPP.

Aside from the fact that a Gateway node has two important global interest states (the attached SPP and the external world), the overall operation of Gateways is extremely similar to that of the router nodes from Section 2.1.

4.1 Gateway Specifics for the Initial 50K-Vehicle Runs

The first metacomputing experiments within the SF Express project involved a number of simplifying assumptions and restrictions on the nature of the Gateway processes, in particular:

1. The communications network among the participating SPPs is implemented as a fully connected set of links between pairs of SPPs, with each SPP dedicating a Gateway processor for each external SPP.

2. Messages between SPPs are sent as UDP/IP datagrams.

3. Interest declaration messages are retransmitted at regular intervals ("heartbeats") to accommodate the unreliable nature of the UDP messages.

In Fig.(6), the schematic diagram of the multi-SPP environment illustrates the dedicated Gateway links.

The Gateways in Fig.(6) operate as pure communications servers, whose task is to manage the flow of requested PDUs and interest states between SPPs. Details can be found in Ref. [12]. Timing results for Gateway operations in the 50K-vehicle runs are examined in Section 4.4.

The complete connectivity among Gateways in SF Express (as in Fig.(6)) should be viewed as a provisional expediency on the road to a 50K-vehicle simulation. With one exception noted below, this model easily handled the inter-SPP traffic at rates up to 1,000 PDUs/sec. However, this initial model does not scale well as the number of sites in Fig.(6) increases, and has the additional defect that Gateway processors associated with low-activity links are a wasted resource. Movement towards an architecture linking individual SPPs by some form of multicasting (possibly ATM) network should be explored.

4.2 The 50K-Vehicle Scenarios

The scenarios used by SF Express involve Blue and Red forces laid down on the 300 km by 350 km SAKI (Saudi Arabia, Kuwait, Iraq) terrain database. The full complement of vehicles is organized into a number of opposing force groups. The relative populations of vehicles types (tanks, trucks, helicopters, ...) and the actual laydowns of units and vehicles were designed according to standard military doctrine [16] including, for example, a roughly 2:1 superiority in numbers for the attacking Blue forces.

Fig.(7) presents a schematic of the force deployments in one of the two scenarios used in the 50K-vehicle runs. This "Version 2.1" laydown has about 42K Blue Vehicles and 21K Red Vehicles. Most of the vehicles (about 85%) are trucks, as is realistic for many actual military campaigns.

The large boxed areas in Fig.(7) show the assignments of scenario elements to SPP platforms. The evolution of the scenario over time is fairly simple: all of the Blue forces move east and attack while the Red forces sit and defend. This gives rise to intense interactions along the dashed "Front Line" in Fig.(7). For the

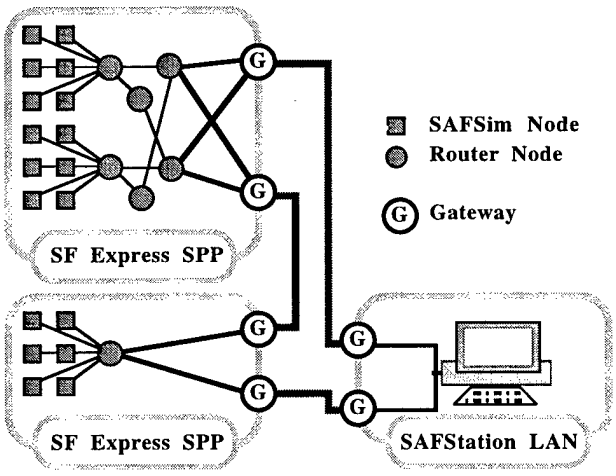


Figure 6: Schematic multi-SPP SF Express using explicit Gateways

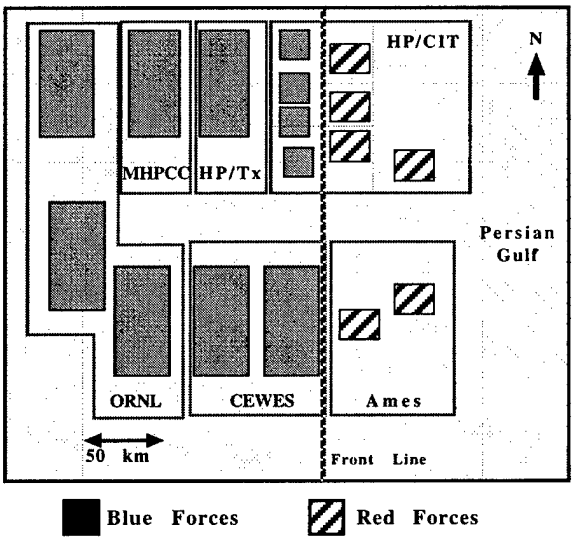


Figure 7: Version 2.1 scenario and assignments of force groups to specific SPPs

given Force⇔SPP assignments, this yields significant data exchanges between the Ames and CEWES SP2s and among the four 64-processor components of the Caltech HP Exemplar. Additional non-fighting interactions occur between some sets of adjacent Blue force groups.

4.3 Porting and Practical Issues

Initially, SF Express was ported to the Intel Paragons at Caltech. Extensive single-node runs were required to begin understanding and assessing optimization possibilities for the very large ModSAF code base. Small multiple node runs identified key communications libraries that would need modification. Numerous problems were encountered (system call assumptions, inadequate bounds checking, ...). Solutions developed during the single-node Paragon work simplified subsequent ports to other platforms, although OS-specific assumptions, awkward build procedures, and occasional cross-compilation issues required case-by-case treatments.

Once the SF Express code had matured to the point where simulations with 1K-10K vehicles were becoming routine, initial heterogeneous multi-SPP tests began. Coordination and synchronization of simulation startup was quickly identified as a key issue, along with management of the extensive scenario data files. A number of intermediate-sized runs, involving 20K-30K simulated vehicles at two or three sites, were critical first steps before attempting to commandeer six SPPs for a block of intersecting dedicated time necessary for the proposed 50K-vehicle exercise.

The large, 50K-vehicle runs with six SPPs spread across the country involved substantial administrative and operational issues. Various sites had different disk policies, accounting mechanisms, usage models, and schedulers. Ultimately, the success of the large runs resulted from moderate to significant system administration intervention, competent system support personnel, and numerous phone calls. While this was acceptable for a demonstration, it is clearly inadequate for a production model. Many of the initial Globus activities described in Section 5 focus on these operational issues.

4.4 Inter-SPP Highlights of the 50K Runs

The performance issues for the metacomputing model of Fig.(6) center on data movement through the Gateway nodes. The results presented in this section demonstrate that communications levels were easily managed, with one (essentially expected) Paragon exception.

A word on the configuration of the HP Exemplar machines is in order here. At the time of the 50K runs, the Caltech machine was available only as four independent 64-processor machines (labelled "HP/Cj" below); it is now a single 256-processor system. In contrast, the 128-processor Exemplar at the Convex site ("HP/Tx") was configured as a single system.

4.4.1 Results from the Version 2.1 Scenarios

Table 2 summarizes inter-SPP data rates for the V2.1 scenario run. The rows and columns are labelled by SPP site; entries are in Kbytes/sec. Blank entries represent links with rates of less than 0.5 Kbytes/sec.

Many of the RNA⇔RNA communication links have no appreciable activity. This is due to the geographic separation of the Force groups in Fig.(7) and additional restrictions on broadcast PDUs, as discussed in Refs.[11],[12]. The communication model running on Ames and CEWES retains a significant level of simulation-wide broadcast PDUs, giving rise to the constant "background" data rates evident in the bottom two rows of Table 2. The values in Table 2 show the rates at which data are sent from the "Row SPP" to the "Column SPP." Due to dropped packets, these are not the same as the rate at which data are received by the Column SPPs, but they are generally close. The exceptions involved links to ORNL, where packet loss was often severe. In the worst case,

MHPCC	Sends	63.9 Kbytes/sec to ORNL
ORNL	Receives	7.9 Kbytes/sec from MHPCC

With the exception of communications to ORNL, the number of dropped UDP packets within the Version 2.1 runs is small, and well within the tolerable range for ModSAF.

Table 3 contains a detailed look at three of the more active inter-SPP links from Table 2:

HP/Tx ⇔ MHPCC: Successful, moderately high bandwidth communications between machines on a Wide-Area Network.

MHPCC ⇔ ORNL: Saturated/Failed communications between machines on a Wide-Area Network.

HP/C1 ⇔ HP/C2: Successful communications between machines on a Local-Area Network.

The "PDU Busy" rows list the fraction of (wall clock) time spent in PDU communications within the SPP and through the Gateway to the remote SPP. The last two rows give the mean times for PDU bundle communications across the network. The UDP-ethernet

Table 2: Inter-SPP Communications Rates for the V2.1 Scenario Large-Scale Metacomputing Runs

	HP/C0	HP/C1	HP/C2	HP/C3	ORNL	MHPCC	HP/Tx
HP/C0	-	20.7					35.1
HP/C1	23.9	-	15.9	14.8			16.1
HP/C2		64.2	-	15.3			
HP/C3		18.4	5.8	-			
ORNL					-	63.9	
MHPCC					22.2	-	67.2
HP/Tx	3.7	21.0				169.0	-
AMES	3.3	3.3	3.3	3.3	3.3	3.3	3.3
CEWES	6.3	6.3	6.3	6.3	17.6	6.3	6.2

Table 3: Details of Gateway Performance on Three Busy Links of the V2.1 SF Express Run

Local SPP Remote SPP	HP/Tx MHPCC	MHPCC HP/Tx	ORNL MHPCC	MHPCC ORNL	HP/C1 HP/C2	HP/C2 HP/C1
Local PDU Busy	0.168	0.074	0.041	0.050	0.023	0.014
Remote PDU Busy	0.147	0.080	0.926	0.032	0.031	0.030
Read Time [msec]	0.60	0.63	22.26	0.77	0.61	0.60
Write Time [msec]	0.97	0.28	27.52	0.26	0.45	0.34

reads and writes on the ORNL Paragon are about 30 times slower than on the other platforms, leading to an overwhelmed Gateway and the significant data losses noted above.

It should be stressed that no attempts were made to optimize network communications in these initial 50K runs. Networks used included ESnet, LosNettos, NREN, DREN, ANSnet, and commodity providers. Fig.(8) shows a partial network map of communications links to the Caltech site, with shaded ellipses representing the various network domains. A message from Caltech to MHPCC visits 14 routers, while a return message travels through 12. Clearly, the SF Express 50K-vehicle runs did not use an overly optimized network.

The results in Tables 2 and 3 indicate that something more aggressive than simple UDP/IP ethernet will be needed to use successfully the ORNL Paragon in a large scale, distributed simulation. As was noted in Section 4.1, the RNA Gateway strategy can accommodate various transport mechanisms.

5 An Integrated Metacomputing Environment Using Globus

The Globus Project [7], [8] is developing a basic software infrastructure to support applications that need and/or are capable of using geographically distributed computational and information resources. A key element of Globus is the design and implementation of a distributed supercomputing infrastructure toolkit that provides an integrated set of services in five key areas:

1. **Communications:** The Nexus communications library provides message-delivery services for a variety of communications models in a manner that is cognizant of network quality of service parameters.
2. **Information:** The Metacomputing Directory Service (MDS) offers a uniform method for obtaining real-time information on system status and structure.
3. **Resource Location/Allocation:** The Global Resource Allocation Manager (GRAM) provides mechanisms for declaring application resource requirements, identifying and scheduling appropriate resources, as well as initiating and managing the application on these resources. The GRAM can be thought of as a low-level scheduler Application Program Interface (API).
4. **Security:** The Globus system includes a number of basic security services (e.g., authentication and authorization), enabling sophisticated application specific security mechanisms and single sign-on functionality.

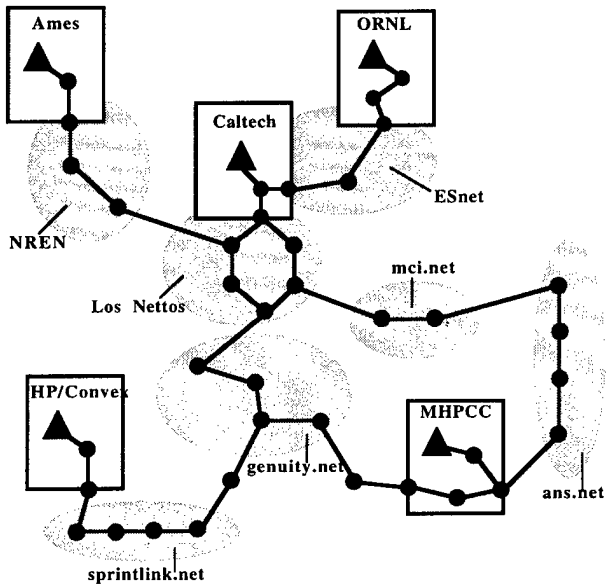


Figure 8: Partial network connectivity map for the 50K-vehicle simulations

5. **Data Access:** Mechanisms are provided for high-speed remote access to persistent storage.

5.1 Benefits of the Globus Toolkit

The modules within the Globus Toolkit directly address a number of problems uncovered during the initial, manually operated SF Express metacomputing runs.

The Nexus library provides a "resource aware" implementation of communication tasks (e.g., data exchanges between the Gateway nodes of Fig.(6)), using the best available communications mechanism (UDP over IP, HIPPI, ATM, etc.). Simple automatic selection rules or user-guided directives determine the appropriate communications method, with selections dynamically dependent on the status of the available network services. These features are particularly useful for the communications links between Gateway processors, in order to avoid bandwidth saturation, as was observed in Section 3 for the ORNL⇔MHPCC link. The communications layer provides efficient implementations of native communication methods, including message passing, multicast, distributed shared memory, remote procedure calls, etc. The selected method must be aware of Quality of Service (QoS) parameters, such as reliability, bandwidth, and latency. Intelligent, performance-based, application configuration choices can be made to match the "currently available" execution environment, enabling the user to better utilize shared resources and attain higher throughput.

The MDS and GRAM elements of the Globus toolkit address the broad problem of resource identification, allocation, and task execution within the grid of available assets. The MDS provides an automated, "information-rich" approach to system configuration, enabling intelligent automated resource allocations. MDS includes a data model to represent dynamically changing capabilities of various parallel computers and networks, so that tools and applications do not have to rely on stale or programmer-supplied knowledge (e.g., use a dedicated HIPPI or ATM node instead of garden variety UDP/IP).

Once the desired distributed assets have been identified, GRAM provides a simple, uniform interface to local resource allocations. In essence, GRAM enables the coordinated startup of a metacomputing run by a single "go" script that drives the participating SPPs, attached displays, etc. This represents a significant improvement over the existing environment, in which the non-static differences among operating systems and resource schedulers on various platforms are coordinated by hand-crafted scripts (and prayers) based on detailed knowledge of resource-specific usage models. Globus services also provides periodic health and status information for each job instantiation and allows application-specific tools to hook into generic health and status monitor services. This capability

would be an improvement over the existing SF Express method using separate monitoring tools on each SPP.

Not all startup and job management concerns are addressed with the use of a single script that starts program execution on all resources. Determining and staging required datasets is another concern. Staging of data automatically and efficiently just prior to simulation time avoids a number of difficulties associated with site-specific disk usage policies. For example, the 50K scenario datasets could not permanently reside on the file systems of the SPPs used in the SF Express runs, due to various quota limits and disk policies. This situation necessitated tedious (and somewhat error prone) manual staging prior to the large runs.

Simulations to date have involved static assignments of scenarios to SPPs, such that configuration file preparation and data staging could occur prior to SPP resource allocation. This approach typically wastes disk space and does not allow the application to take best advantage of the resources available. The Data Access services (remote I/O calls) within Globus allow high-speed remote access to persistent storage, such as simulation scenarios and behavior files, potentially saving vast amounts of disk space and frequent user-intervention required to move large data sets both before and after runs (possibly scheduled arbitrarily). The more resource-aware an application can become, the larger the window for adaptive and optimal choices.

5.2 Initial Experiments with Globus

The coordinated startup capabilities of Globus were successfully tested during two live demonstrations at the November 1997 High Performance Networking and Computing Conference (SC97) in San Jose. These experiments involved 824 processors on SPPs at six sites, as shown in Fig.(9), simultaneously displaying parts of the simulation on an Immersa-desk in the Argonne National Laboratory booth on the conference floor. Unlike the fairly conservative force group assignments of the initial 50K-vehicle simulations, these runs involved a more "interleaved" assignment of scenario files to SPPs, as shown in Fig.(9). This was done in order to provide more stressing tests of inter-SPP communications. The overall simulation involved about 40K vehicles (about 50K ModSAF entities). The important new aspects using GRAM specifications to drive the simulation were successfully demonstrated.

6 Accomplishments and Future Directions

The multi-SPP runs in August 1997 surpassed the project goal of a 50,000-vehicle simulation on a heterogeneous collection of SPPs and validated the overall SF Express concept. The ExInit team generated a collection of sound military scenarios featuring intense, quick interactions (and fighting) within the one-hour time frame of the runs.

Problem areas in the single-SPP SF Express code seemed to center on, not surprisingly, the ModSAF simulation engine itself. Of the hundreds of thousands of lines of ModSAF source code, less than five percent of the libraries were modified to accommodate RNA. The core simulation code was purposely left mostly alone, due not only to project scope, but also to decouple performance of the communications architecture from the driving simulation engine. Among other issues, simple profiling determined that ModSAF vehicle table manipulations consumed a substantial fraction of total CPU time. Possible solutions for expensive ordered list operations are noted in Ref.[17].

Problems in the multi-SPP runs of Section 4 were largely operational, arising from the differing environments at the six SPP sites. The Globus experiments described in Section 5 can be viewed as the first steps toward a more user-friendly robust system.

An attractive near-term direction involves a greater exploitation of the unified resource information services, resource location and allocation services, and data access modules within Globus to eliminate much of the configuration file mechanisms within SF Express and optimize runtime parameters. Using the currently deployed Globus services, initialization and execution of a large simulation would proceed roughly as follows:

1. The user specifies the location of the simulation data and the desired simulation size from a single place (e.g., console or file).

2. MDS evaluates the request and locates appropriate resources (with the MDS databases augmented to understand information on the inherent simulation capabilities of the individual platforms).

3. Once the appropriate computational assets are allocated, GRAM is used to start the distributed simulation and to exchange runtime system configuration information among the participants.

4. Using the system configuration information from GRAM, each SPP takes responsibility for a specific subset of the simulation scenario files, retriev-

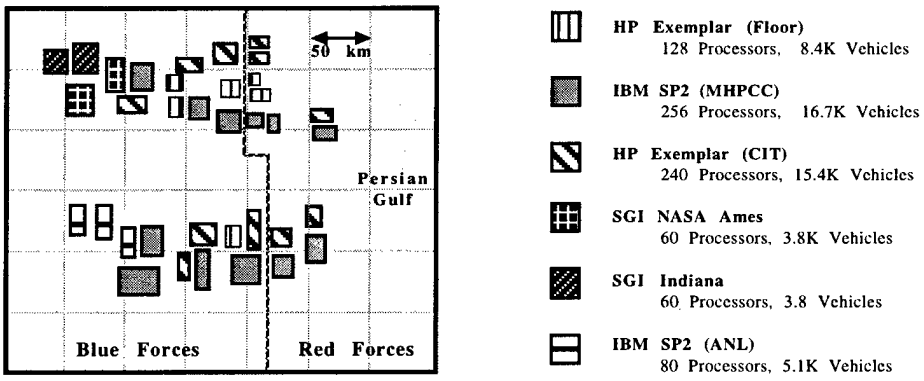


Figure 9: Version 2.1 scenario element assignments for the initial tests with Globus

ing these data automatically from the staging area using the Globus Data Access services.

In this model, user input is largely restricted to the high-level specification of the problem itself (i.e., the simulation scenarios), with Globus managing all pragmatic issues of resource allocation, data staging, job management, and network connectivity needed in order to meet the user specified requirements (which could well include additional constraints, such as required network bandwidths).

The construction of this Globus-directed metacomputing model is a realistic near-term goal. Modifications within the existing RNA code base of Ref.[12] would largely involve generalizations of the Gateway communications procedures to use portable Nexus routines in place of socket calls. Additional new logic would be needed within the single-SPP initialization sequence to support runtime assignments of scenarios to SPPs, based on configuration data from GRAM. (Neither of these tasks is seen as being particularly difficult.) This system would become the next-generation SF Express proof-of-concept demonstration, with intelligent resource allocation, simulation startup, and data management all done in a simple, user-friendly manner.

Acknowledgments

Support for this research was provided by the Information Technology Office, DARPA, with contract and technical monitoring via Naval Research and Development Laboratory (NRaD).

Access to various computational facilities and significant system support were essential for this work. The 1024-node Intel Paragon was made available by the Oak Ridge Center for Computational Sciences. The smaller Intel Paragon and 256-processor HP Exemplar were made available by Caltech/CACR. The IBM SP2s were provided by the Maui High Performance Computing Center, U.S. Army Corps of Engineers Waterways Experiment Station Information Technology Laboratory, and the Numerical Aerodynamic Simulation Systems Division at NASA Ames Research Center. Indiana University and NASA Ames provided access to the Silicon Graphics machines. A 128-CPU HP Exemplar was provided by HP/Convex Division Headquarters. We thank the system administrators and support staff at all these sites.

Globus experiments and integration were made possible by the dedicated team at Argonne National Laboratory (led by Ian Foster) and USC Information Sciences Institute, including Karl Czajkowski, Mei-Hui Sue, and Marcus Thiebaux.

Author Biographies

Sharon Brunett is a Computing Analyst for Caltech's Center for Advanced Computing Research. She received her B.S. in Computer Science and Applied Mathematics from the University of California, Riverside in 1983. Current interests include scalable I/O methodologies for SPPs, characterizations of performance on MTA architectures, and integration of multidisciplinary applications.

Dan Davis, Assistant Director at CACR, has a B.A. in Psychology and a J.D., both from the University of Colorado. With a background in Naval cryptology and intelligence, he and Dr. Gottschalk are also pursuing technology for K-12 education.

Thomas Gottschalk, CACR Senior Research Scientist, is a Member of the Professional Staff and Lecturer in Theoretical Physics at Caltech. He received a B.S. in Astrophysics from Michigan State

University in 1974 and a Ph.D. in Theoretical Physics from the University of Wisconsin in 1978. Gottschalk spent several years designing simulation models for High Energy Physics interactions. He has written large parallel codes for multi-target tracking and for physical design validation of VLSI chips.

Paul Messina is Assistant Vice President for Scientific Computing at Caltech, Faculty Associate in Scientific Computing, Director of Caltech's Center for Advanced Computing Research, and serves on the executive committee of the Center for Research on Parallel Computation. His recent interests focus on advanced computer architectures, especially their application to large-scale computations in science and engineering. He also is interested in high-speed networks and computer performance evaluation. He heads the Scalable I/O Initiative, a multi-institution, multi-agency project aimed at making I/O scalable for high-performance computing environments. Messina has a joint appointment at the Jet Propulsion Laboratory as Manager of High-Performance Computing. Messina received a Ph.D. in mathematics in 1972 and a M.S. in Applied Mathematics in 1967, both from the University of Cincinnati, and his BA in mathematics in 1965 from the College of Wooster.

Carl Kesselman is a Project Leader at the Information Sciences Institute and a Research Associate Professor in Computer Science at the University of Southern California. He received a Ph.D. in Computer Science at the University of California at Los Angeles. He co-leads the Globus project, along with Ian Foster, at Argonne National Laboratory. Dr. Kesselman's research interests include high-performance distributed computing, parallel computing, and parallel programming languages.

References

[1] J. S. Dahmann and D. C. Wood, "Scanning the Special Issue on Distributed Interactive Simulations," *Proceedings of the IEEE*, Volume 83 (1995) 1111, and references therein.

[2] R. C. Hofer and M. L. Loper, "DIS Today," *Proceedings of the IEEE*, Volume 83 (1995) 1124.

[3] C. M. Keune and D. Coppock, "Synthetic Theater of War-Europe (STOW-E) Technical Analysis," NRaD Technical Report 1703 (1995).

[4] K. Boner, C. Keune, and J. Carlson, "Synthetic Theater of War (STOW) Engineering Demonstration-1A (ED-1A) Analysis Report," NRaD Report, May, 1996.

[5] S. Rak, M. Salisbury, and R. MacDonald, "HLA/RTI Data Distribution Management in the Synthetic Theater of War," Proceedings of the Fall 1997 DIS Workshop on Simulation Standards (97F-SIW-119).

[6] R. Cole and B. Root, "Network Technology for Stow 97," Naval Research Laboratory briefing, and private communication.

[7] The documents link on the Globus Project WWW site (http://www.globus.org) points to a number of technical papers and interface specifications.

[8] I. Foster and C. Kesselman, "Globus: A Metacomputing Infrastructure Toolkit," to be published in *International Journal of Supercomputer Applications.*

[9] P. Messina *et al.*, "Distributed Interactive Simulation for Synthetic Forces," Proceedings of the International Parallel Processing Symposium, Geneva (1997).

[10] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra, "MPI, The Complete Reference," MIT Press (1996).

[11] S. Brunett and T. Gottschalk, *Pathfinder: A Scalable Implementation of ModSAF on SPPs*, CACR Report in preparation.

[12] S. Brunett and T. Gottschalk, *Large-Scale Metacomputing Framework for ModSAF*, Technical Report CACR-152, January 1998.

[13] L. Craymer and C. Lawson, "A Scalable, RTI-Compatible Interest Manager for Parallel Processors," Proceedings of the Spring 1997 DIS Workshop on Simulation Standards.

[14] J. Kohler, S. Narasimhan, J. Pittman, A. Whitlock. "ExGen Software Design Document." Naval Command, Control and Ocean Surveillance Center (NCCOSC), RDT&E DIV, June 18, 1996.

[15] http://cesdis.gsfc.nasa.gov/beowulf, the Beowulf WWW site, and references/links therein.

[16] L. Mengel, "Scenario Modifications for SF Express 50,000 Vehicle Scenario," NCCOSC RDTE DIV Report N66001-97-M-1531 (1997).

[17] L. Craymer and L. Ekroot, "Characterization and Scalability," Proceedings of the 1998 Spring Simulation Interoperability Workshop.

Session II

Resource Management, Matching, and Scheduling

Session Chair

Dan Watson
Utah State University, Logan, UT, USA

CCS Resource Management in Networked HPC Systems

Axel Keller and Alexander Reinefeld
PC^2 – Paderborn Center for Parallel Computing
Universität Paderborn, D-33102 Paderborn, Germany

Abstract

CCS is a resource management system for parallel high-performance computers. At the user level, CCS provides vendor-independent access to parallel systems. At the system administrator level, CCS offers tools for controlling (i.e. specifying, configuring and scheduling) the system components that are operated in a computing center. Hence the name "Computing Center Software". CCS provides:

- *hardware-independent scheduling of interactive and batch jobs,*
- *partitioning of exclusive and non-exclusive resources,*
- *open, extensible interfaces to other resource management systems,*
- *a high degree of reliability (e.g. automatic restart of crashed daemons),*
- *fault tolerance in the case of network breakdowns.*

In this paper, we describe CCS as one important component for the access, job distribution, and administration of networked HPC systems in a metacomputing environment.

1 Introduction

With the increasing availability of fast interconnection networks high-performance computing has undergone a metamorphosis from the use of local computing facilities towards a distributed, network-centered computing paradigm. The motivation is to better utilize the available hardware by linking LAN/WAN connected supercomputers to a virtual metacomputer [33]. While at the time being, there are only few multi-site applications that fully exploit the computational power of distributed nodes, metacomputing is already used on a broader scale for job load sharing and fault tolerance purposes.

Distributed high-performance computing environments usually comprise a wide spectrum of resources with different capabilities. Here, a resource management system must be able to cope with unreliable networks and with heterogeneity at multiple levels (e.g. administrative domains, scheduling policies, operating systems, protocols etc.). Because of the dynamic nature of the metacomputing components, the available resources should be identified at runtime. As an example, the performance of shared media (networks) varies over time, requiring constant updates on the global system state and structure.

From the user's point of view, a metacomputer should be as easy to use as the workstation on his/her desk. This means that users need a vendor-independent access interface and their applications should be transparently mapped onto a set of suitable target platforms.

In this paper, we present the architecture of our *CCS Computing Center Software*. Having started in 1992 with a comprehensive system that manages all computers in a site but gives users only access to a single system at a time, CCS was continuously upgraded. It now supports multi-site applications and has an open interface to metacomputer management services. As a long-term goal we want to integrate CCS—among other resource management systems—into an open global metacomputing environment.

In the 4^{th} release, three new concepts have been introduced: *'CCS Islands'* provide management facilities for administrating single HPC systems in a local site. At the next higher level, a *'Center Resource Manager'* coordinates the cooperative administration and use of all systems in a computing center, whereas the *'Center Information Server'* provides an active directory service for metacomputer access from the outside world.

All modules build on the generic *'Resource and Service Description'* that is used for specifying hardware and software components.

The contents of this paper are as follows: First, we put CCS in the context of the Globus project. Then we present the architecture of the CCS islands and their technical implementation. Thereafter, we introduce the concepts required for metacomputing (Center Resource Manager and Center Information Service) and how they work together. Our tools for re-

0-8186-8365-1/98 $10.00

source and service description are described in Section 5 and some preliminary results on the use of CCS in an industrial metacomputing setting are discussed in Section 6. Section 7 gives a brief review on related projects and Section 8 presents a summary.

2 CCS – A Link to Globus?

While CCS may be seen as "just another resource management software" we have always put it in a much broader context. In fact, our primary design goal was to provide a resource management system that can be integrated into metacomputer environments like our *Metacomputer Online* toolbox [32].

Because *Globus* [15], as part of the National Computational Science Alliance [34], is certainly the most well-known metacomputing project throughout the world, we now put CCS in relation to Globus. The following list gives the most important similarities and differences between the two projects.

The Globus project regards a metacomputer as a networked virtual supercomputer constructed dynamically from geographically distributed resources that are linked by high-speed networks. It aims at a vertically integrated treatment of application, middleware and network and it provides a basic infrastructure of tools building on each other:

- resource (al)location:
  Globus Resource Manager GRM [23]
- communication layer:
  Nexus [13]
- unified resource information service:
  Metacomputing Directory Service MDS [12]
- authentication interface:
  Generic Security System GSS [26]
- data access:
  Remote IO Facility RIO [16].

In the Globus metacomputer, applications are expected to configure themselves to fit the execution environment delivered by the metacomputing system, and then adapt their behavior to subsequent changes in the resource characteristics. This concept has been named 'Adaptive Wide Area Resource Environment' AWARE.

The Computing Center Software CCS was primarily designed to manage the resources in a *single* site. They may be geographically distributed but operate in a single NFS/NIS domain. It provides an open interface so that several sites may be joined by higher-level tools—a modular approach that proved useful in several industrial projects. CCS has a hierarchical structure with autonomous software layers that interact only via message passing: The lowest level is a self-sufficient 'island' controlling a single machine or cluster which can be operated stand-alone. The next higher level consists of the Center Resource Manager (CRM) and the Center-Information Server (CIS) which build the interfaces of a site to the 'outside world'.

CCS does not only provide a comfortable user interface, but it also offers a versatile, almost system-independent interface for the administrator. Its open framework architecture allows to integrate all kinds of HPC systems. Compared to Globus, CCS

- does not support metacomputing by itself, but it provides one important component,
- has not yet an API,
- does not support remote I/O,
- has no dedicated authentication interface.

3 Computing Center Software

When the CCS project [9, 30, 31] started in 1992, only few competitive systems were available. With our background in the operation of *massively* parallel computing systems, we aimed at providing

- concurrent user access to exclusively owned resources
- interactive and batch processing at the same time,
- optimal system utilization by dynamical partitioning and scheduling,
- maximum fault tolerance for remote access via WANs.

CCS became first operationable on a 1024-node transputer system and was later adapted to PowerPlus systems from Parsytec and also to workstation clusters. Aiming at portability, we designed CCS to run on UNIX systems, such as Linux, SunOS, Solaris, AIX and others. The architecture with its modular frame structure allows to integrate a great variety of other systems, cf. Sec. 3.2.

3.1 Architecture

Island Concept. With its distributed nature, fault tolerance is a basic prerequisite for CCS working correctly. In earlier versions, the machines of a computing center were managed by a single (but physically distributed) CCS software. This caused bottlenecks

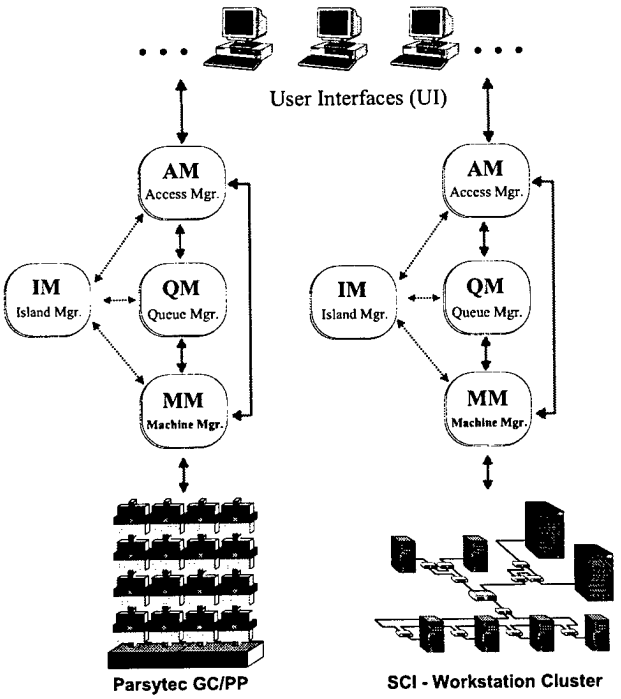


Figure 1: Architecture of CCS 'islands'

at the single scheduler serving all machines and resulted in poor fault tolerance due to the one central request handler. With the 4^{th} release, each machine is now managed by a dedicated CCS, resulting in set of comprehensive, self-sufficient *'islands'* shown in Figure 1. Each island has six components, which will be described in more detail later in the text:

- The *User Interface (UI)* offers X- or ASCII-access to all capabilities of a machine. It encapsulates the physical and technical characteristics for a homogeneous access to single or multiple heterogeneous systems.
- The *Access Manager (AM)* manages the user interfaces and is responsible for authorization and accounting.
- The *Queue Manager (QM)* schedules the user requests.
- The *Machine Manager (MM)* manages the parallel system.
- The *Island Manager (IM)* provides name services and watchdog functionalities for reliability.
- The *Operator Shell (OS)*, not shown in Figure 1, allows system administrators to control CCS, e.g. by connecting to the single daemons.

With the island concept scalability, reliability and error recovery have been improved by separating the management of different machines into different islands. Each machine uses a dedicated scheduling strategy and can therefore be operated in a different mode (batch, shared, mixed etc.). Specific user interfaces can be used to reflect special system features.

Reliability. In heterogeneous distributed environments, reliability is of prime importance. For example, a message-passing program that does not receive an answer from its partner in time, does not know

- whether the network is down,
- or whether it temporarily has a low bandwidth,
- or whether the communication partner has died.

This is because the necessary information is not available at OSI level 7. We therefore need an instance with global and up-to-date information on the status of all system components. This instance should be always accessible and it should have little or no dependencies on other modules.

In CCS, this instance is the *Island Manager (IM)*. At startup and shutdown time all CCS daemons notify the IM. Hence the IM has a consistent view on the current status of the processes in an island. The IM is authorized to stop erroneous daemons or to restart crashed ones.

In its second task, the IM provides name services. It maintains an address translation table that matches symbolic names to the daemons' physical network address (host ID and port number). This gives a level of indirection, allowing the IM to migrate daemons to other hosts in the case of overloads or system crashes. Symbolic names are given by the triple `<center, island, process>`. As a side effect, this allows to run several CCS islands on a single host concurrently.

User Management. The *User Interface (UI)* runs in a standard UNIX shell environment like tcsh, bsh, or ssh. Common UNIX mechanisms for IO-redirection, piping and shell scripts can be used and all job control signals (ctl-z, ctl-c, ...) are supported. Five CCS commands are available:

- *ccsalloc* for allocating and/or reserving resources,
- *ccsbind* for re-connecting to a lost interactive application/session,
- *ccsinfo* for displaying information on the job schedule, users, job status etc.,
- *ccsrun* for starting jobs on previously reserved resources,
- *ccskill* for resetting or killing jobs and/or for releasing resources.

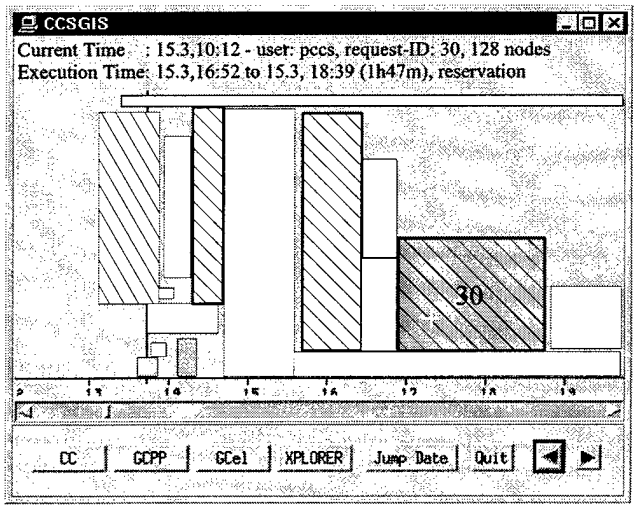


Figure 2: Scheduler GUI displaying the scheduled nodes (vertical axis) over the time axis.

The *Access Manager (AM)* analyzes user requests and is responsible for authentication, authorization and accounting.

CCS supports project specific user management. Privileges can be granted to either a whole project or to specific project members, for example

- access rights per machine
- allowed time of usage (day, night, weekend, ...)
- maximum number of concurrently used resources
- accounting per machine (product of CPU-time and #PEs)
- machine access rights (batch, interactive, the right for reserving resources)

User requests are sent to the *Queue Manager (QM)* which schedules the jobs according to the current scheduling policy. CCS provides several scheduling modules (FCFS, FFIH, FFDH, IVS) that can be plugged in by the system administrator, cf. Sec. 3.2 [18].

Job Scheduling. The first CCS release was capable of managing exclusive (non-timeshared) resources only. With release 4.0, CCS has been upgraded to support time-shared resources as well. As in Condor, Codine, or LSF, the system administrator may specify a maximum load factor that is allowed on the single nodes.

In their resource requests, users must also specify the expected finishing time of their jobs. Based on this information, CCS determines a fair and deterministic schedule. Both, batch and interactive requests are processed in the same scheduler queue. The request scheduling problem is modeled as an n-dimensional bin packing problem, where the one dimension corresponds to the continuous time flow, and the other $n-1$ dimensions represent system characteristics, such as the number of processor elements. Currently, CCS uses an enhanced first-come-first-serve (FCFS) scheduler, which fits best to the request profile in our center. The waiting times are reduced by first checking whether a newly incoming request may fit into a gap of the current schedule. The current schedule is displayed in an X-window as illustrated in Figure 2.

CCS allows to reserve resources for a given time in the future. This is a convenient feature when planning interactive sessions or online events. As an example, consider a user wants to run a parallel application with 64 processors of the Parsytec GCel from 9 to 11 am at 13.2.1999. This resource allocation is done with the command:

```
ccsalloc -m GCel -p 64 -s 9:13.2.99 -t 2h.
```

'Deadline scheduling' is another useful feature. Here, CCS guarantees the job to be completed at (or before) the specified time. A typical scenario for this feature is an overnight run that must be finished when the user comes back to his/her office in the next morning. Deadline scheduling gives CCS the flexibility to improve the system utilization by scheduling batch jobs at the earliest convenient and at the latest possible time.

The CCS scheduler is able to a handle two kinds of requests, those that are fixed in time and the variable ones. A resource that has been reserved for a given time frame is fixed: It cannot be shifted on the time axis (see the hatched rectangles in Fig. 2). Interactive requests, in contrast, can be scheduled earlier but not later than asked for. Such shifts on the time axis might occur when resources are released before their estimated finishing time.

System Partitioning. For metacomputing, we need a scheduler that computes deterministic schedules. Additional design objectives were optimal system utilization combined with a high degree of system independence. To deal with these conflicting requirements we have split the scheduler software into two parts, one of them (QM) being completely independent of the underlying hardware architecture. With this separation, the scheduler daemon has no information on the mapping constraints such as the minimum cluster size, or the amount/location of the link entries.

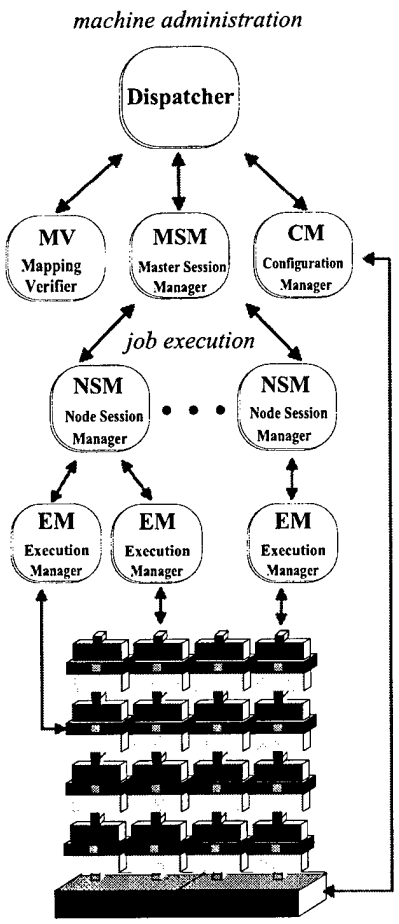


Figure 3: Detailed view of the machine manager (MM)

These machine dependent tasks are performed by a separate instance, the *Machine Manager (MM)*. The MM verifies whether a schedule given by the QM can be mapped onto the hardware at the specified time, now also taking concurrent use by other applications into account. If the schedule cannot be mapped onto the machine, the MM returns an alternative schedule to the QM.

The separation between the hardware-independent QM and the system-specific MM also allows to employ system-dependent mapping heuristics that are implemented in small system-specific modules. Special requests for IO-nodes, partition shapes, memory constraints, etc. are taken into consideration in the verifying process. Moreover, with the machine-specific information encapsulated in the MM, CCS islands can be easily adapted to other architectures.

Process Creation and Control. At configuration time, the QM sends the user request to the MM. The MM then allocates the compute nodes, loads and starts the application code and releases the resources after the run.

Because the MM also verifies the schedule, which is a polynomial or NP-hard problem, a single MM daemon might become a computational bottleneck. We have therefore split the MM into two parts, one for the machine administration and one for the job execution (see Figure 3). Each part contains a number of modules and/or daemons.

The machine administration part consists of three separate daemons (MV, MSM, CM) that execute asynchronously as shown in Figure 3. A small *Dispatcher* coordinates the lower-level components.

The *Machine Verifier (MV)* checks whether the schedule given by the QM can be realized at the specified time with the specified resources. Based on its more detailed information on the machine structure (hardware and software) it runs system-specific scheduling and partitioning schemes. The resulting schedule is then returned to the QM.

The *Configuration Manager (CM)* provides the interface to the hardware. It is responsible for booting, partitioning, and shutting down the operating system software. Depending on the system's capabilities, the CM may gather subsequent requests and re-organize or combine them for improving the throughput—analogously to a hard disk controller.

The *Master Session Manager (MSM)* interfaces to the job execution level. It sets up the session, including application-specific pre- or post-processing, and it maintains information on the status of the application.

It allocates and synchronizes the system entries of the user partition with the help of the *Node Session Manager (NSM)*, that is run on each specified entry node. The NSM starts and stops jobs and it controls the processes. When receiving a command from the MSM, the NSM starts an *Execution Manager (EM)* which establishes the user environment (UID, shell settings, environment variables, etc.) and starts the user application.

On time-sharing systems, the NSM invokes as many EMs as needed. It also gathers dynamic load data and sends it to the MM and QM where it is used for scheduling and mapping purposes.

Virtual Terminal Concept. With the increasing use of supercomputers for *interactive* simulation and design, the support of remote access via WANs becomes more and more important. Unpredictable behavior and even temporary breakdowns of the network should (ideally) be hidden from the user.

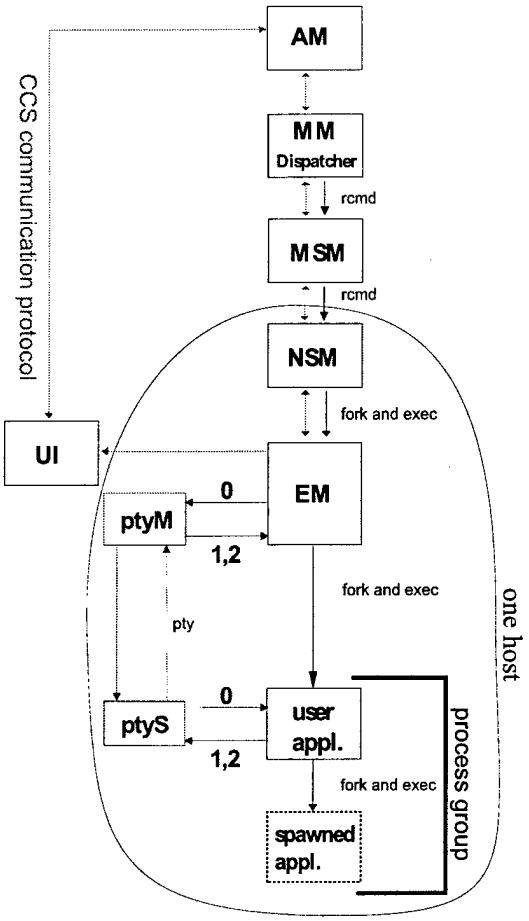


Figure 4: Control and data flow in CCS

In CCS, this is done by the EM which buffers the standard IO streams (stdin, stdout, stderr) of the user application. In case of a network break down, all open output streams are sent by e-mail to the user or they are written into a file when specified by the user. Users can re-bind to interrupted sessions, provided that the application is still running. CCS guarantees that no data is lost in the meantime.

In summary, Figure 4 gives a overview on the control and data flow in a CCS island.

3.2 Implementation Aspects

CCS consists of about 180,000 lines of C code. The code is—as far as possible—ANSI compliant and POSIX 1003.1-1990 conform. It follows a 'programming frame approach' by splitting most of the modules into two parts, a generic and a system-specific one.

As an example, Figure 5 shows the MM frame. Here, only the mapping module is machine dependent, all other parts are generic and can be re-used.

The daemons are driven by events from incoming

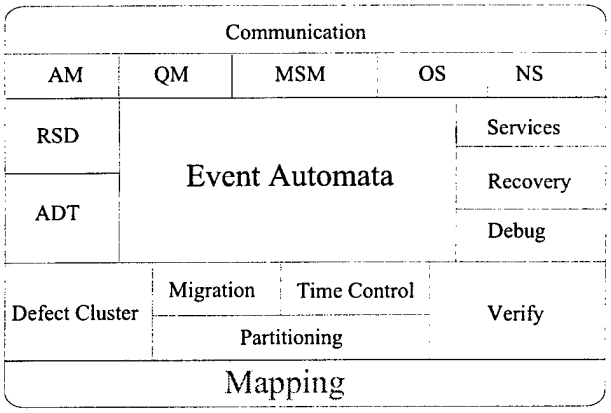


Figure 5: The MM frame

requests or timeouts. Each daemon has a watchdog which checks whether the daemon is still alive. If not, the watchdog shuts its daemon down. This is recognized by the IM, which in turn restarts the dead daemon and informs its communication partners to re-bind to the new instance.

All daemons include a wrapped timer that creates clock ticks for debugging purposes and for simulating incoming requests on a variable time scale.

Even though CCS is POSIX-conform, we implemented a *Runtime Environment (RTE)* layer that wraps system calls. This allows for easy porting to new operating systems. Currently, the RTE provides interfaces for

- the management of dynamic memory, including debugging and usage logging,
- signal handling,
- file I/O including filter routines for ASCII-files,
- manipulation of the process environment,
- terminal handling (e.g. pty),
- sending e-mails,
- logging of warnings and error messages.

The integration of new schedulers is easy, because the QM has an API to plug in new modules. This also allows the QM to use several schedulers. At runtime, the QM takes the decision which scheduler to use, thereby adjusting to specific operating modes (e.g. interactive use only or mixed time sharing and space sharing).

Communication Layer. The communication layer separates a daemon's code from the communication network, allowing to change communication protocols without the need to change the source code of the

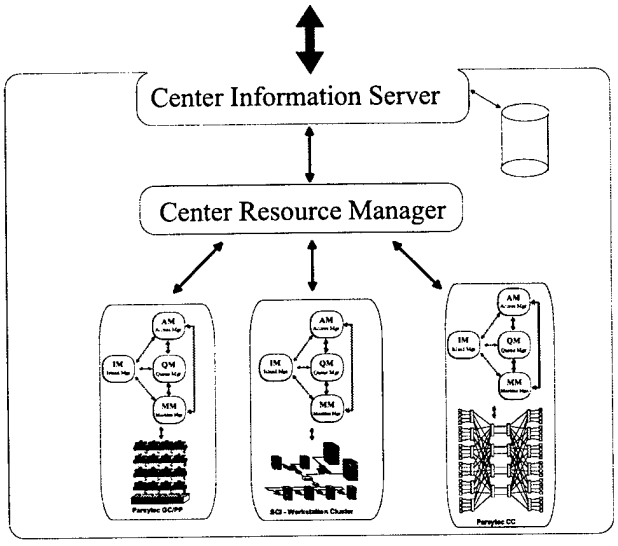


Figure 6: A site managed by CCS

daemons. The communication layer performs the following tasks:

- it provides a reliable and hardware-independent exchange of data,
- it allows to dynamically connect/disconnect to communication partners,
- it checks the availability of communication partners (in cooperation with the IM),
- it translates symbolic module names (in cooperation with the IM).

In our current implementation the daemons communicate through remote procedure calls (RPC). Compared to the faster TCP/IP sockets, asynchronous RPCs provide a more high-level method for process interaction—closely related to the client-server model of distributed computing. Also, they support data conversion with the XDR-library.

The binding of incoming RPC calls (events) to the corresponding callback-functions is done during runtime to allow to dynamically add new events by registering the corresponding event handler.

4 CCS Interface to Metacomputing

With the autonomous islands described in the last section we have one important component for metacomputer management. The three other components are:

- A passive instance that maintains up-to-date information on the system structure and state.
- An active instance that is responsible for the location and allocation of resources within a center. It also coordinates the concurrent use of several systems, which may be administered by different local resource management systems.
- A powerful but user-friendly tool that allows system administrators and users to specify classes of resources.

These three components are: The center information manager CIS, the center resource manager CRM, and the resource and services description RSD.

Center Information Server (CIS). The CIS is the 'big brother' of the island manager (IM) at the next higher level, the metacomputer level. Like the UNIX Network Information Service NIS or the Globus Metacomputing Directory Service MDS, our CIS provides up-to-date information on the resources in a site. Compared to the active IM in the islands, CIS is a passive component.

At startup time, or when the configuration has been changed, an island signs on at the CIS and informs it about the topology of its machines, the available system software, the features of the user-interfaces, the communication interfaces and so on. The CIS maintains a database on the network protocols, the system software (programming models, libraries, etc.) and the time constraints (for specific connections, etc.). The CIS also plays the role of a 'docking station' for mobile-agent software or external users.

For the higher level metacomputer components, the CIS data must be compatible or easily convertible to the formats used by other resource management systems.

The Center Resource Manager (CRM). Like the Globus resource manager, the CRM is a high-level but independent tool that lies on top of the CCS islands. It supports the set-up and execution of multi-site applications running concurrently on several platforms. The term multi-site application can be understood in two ways: It could be just one application that runs on several machines without explicitly being programmed for that execution mode [17], or it could comprise different modules, each of them executing on a machine that is best suited for running that specific piece of code. In the latter case the modules can be implemented in different programming languages using different message passing libraries (e.g. PVM, MPI, PARIX, MPL etc.). Multi-communication tools like

PLUS [6] are necessary to make this kind of multiple-site application possible.

For executing multi-site applications three tasks need to be done:

- locating the resources,
- allocating the resources,
- starting and terminating the modules.

For locating the resources, the CRM maps the user request (given in RSD notation) against the static and dynamic information on the available system resources.

The static information (e.g. topology of a single machine or the site) has been specified by the system administrator, while the dynamic information (e.g. state of an individual machine, network characteristics etc.) is gathered at runtime. All this information is provided by the CIS. Since our resource description language is able to describe dependency graphs, a user may additionally specify the required communication bandwidth for his/her application. In the mapping and migration process, the communication pattern should also be taken into account. Data on the previous runtime behavior can be gathered and condensed in an execution profile as described in [19].

After the target resources have been located, they must be allocated. This can be done in analogy to the two-phase-commit protocol in distributed database management systems: The CRM requests the allocation of all required resources at all involved islands. If not all resources were available, it either reschedules the job or it denies the user request. Otherwise the job can now be started in a synchronized way. Here, machine-specific preprocessing tasks or inter-machine specific initializations (e.g. starting of special daemons) must be initialized.

Analogously to the islands level, the CRM is able to migrate user resources between machines to achieve a better utilization. Accounting and authorization at the metacomputer level can also be negotiated at this layer.

The CRM can be implemented in several ways. As an example, it could be implemented as a single daemon or in the form of distributed instances like the QM-MM complex at the islands level.

5 Resource and Service Description

CCS includes a versatile resource description facility, named *RSD* for *Resource and Service Description.* RSD is used

- at the administrator level for describing type and topology of the available resources, and

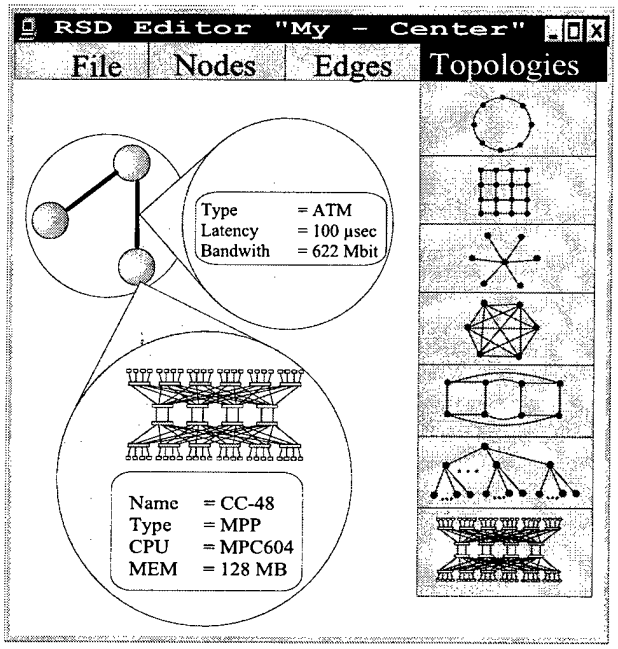


Figure 7: Graphical RSD editor

- at the user level for specifying the required system configuration for a given application.

The predecessor of RSD, the resource description language RDL [3] served in earlier releases of CCS. In general, it was regarded as too complex. Especially industrial users did not want to take the burden of typing in a textual resource specification when just wanting to run a code on a machine with a simple, regular topology. It seems, that it was too early for the user community to appreciate the full descriptive power of a versatile description language. Hence, we hided the language interface by easy-to-use command line options. But of course, RDL was still used behind.

With the current trend to distributed computing, resource description tools become important again. Based on our experiences with RDL, we now provide a more generic approach with three interfaces:

- a graphical interface (GUI) for specifying simple topologies and attributes,
- a language interface for specifying more complex and repetitive graphs (mainly intended for the system administrator), and
- an application programming interface (API) for access from within an application program.

The graphical editor stores the graphical and textual data in an internal data representation. This data

is bundled with the API access methods and sent as an attributed object to the target systems, where it is matched against other hard- or software descriptions.

The internal data description can only be accessed through the API. For later modifications it is retranslated into its original form of graphic primitives and textual components. This is possible, because the internal data representation also contains a description of the component's graphical layout. In the following, we describe the core components of RSD in more detail.

Graphical Interface. The graphical editor provides a set of simple modules that can be edited and linked together to build a dependency graph of the requested resources or a system description.

At the *administrator level*, the GUI is used to describe a center's resource components in a top down manner, starting at the outermost interconnection topology, see Figure 7. With drag and drop techniques, the administrator specifies the available machines, their links and the interconnection to the outside world.

In the next step, the machines are specified in more detail by clicking on a node. The editor then opens a window to display detailed information on the machine, if available. The GUI offers a set of standard machine layouts and some generic topologies like tree, grid or hypercube. The size and shape is defined according to the available hardware. For a single node, detailed attributes like network interface cards, disk sizes, I/O throughput, or the automatic start of daemons may be specified.

Language Interface. From a system administrator's point of view, graphical user interfaces are not powerful enough for describing complex metacomputing environments with a large number of services and resources. Administrators need an additional tool for specifying irregularly interconnected, attributed structures.

Hence, we devised a language interface that is used to specify arbitrary topologies. The hierarchical concept allows different dependency graphs to be grouped for building even more complex nodes, that is hypernodes. For a complete formal definition of the language interface see [7].

Figures 9 and 10 illustrate a resource specification for a metacomputer application **Meta** running on two systems as shown in Figure 8. The metacomputer comprises an SCI workstation cluster and

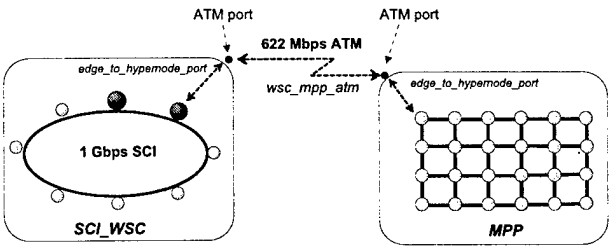


Figure 8: RSD example for multi-site application

```
NODE Meta {
  DEFINITION:
     PORT[] = (SCI, ATM, FDDI);    -- multi-valued attribute

  DECLARATION:                     -- include the two hyper nodes
     INCLUDE "SCI_WSC";
     INCLUDE "MPP";

  CONNECTION:-- of the MPP with SCI workstation cluster
        EDGE wsc_mpp_atm {
        NODE SCI_WSC PORT ATM <=> NODE MPP PORT ATM;};
                     BANDWIDTH = 622 Mbps;};
};
```

Figure 9: RSD specification of Fig. 8

a massively parallel system, interconnected by a bidirectional ATM network.

The definition of **Meta** is straight-forward, see Figure 9. Figure 10 shows the specification of the SCI cluster component, consisting of 8 nodes, two of them with quad-processor systems. For each node, the following attributes are specified: CPU type, memory per node, operating system, and the port of the SCI link. All nodes are interconnected by a uni-directional SCI ring with 1.0 Gbps. In the example, the first node is the gateway to the workstation cluster. It presents its ATM port to the next higher node level (see ASSIGN statement in Fig. 10) to allow for remote connections.

Internal Data Representation. The abstract data type establishes the link between the graphical and the text based representation. It is also used to store descriptions on disk and to exchange them across networks. The internal data representation must be capable of describing the following properties:

- arbitrary graph structures
- hierarchical systems or organizations
- nodes and edges with arbitrary sets of valued attributes

```
NODE SCI_WSC {
  DEFINITION:
  CONST N = 8;        -- number of nodes
  SHARED;             -- allocate resources for shared use

  DECLARATION:
  -- we have 2 SMP nodes (gateways), each with 4 processors
  -- each gateway provides one SCI and one ATM port
  FOR i=0 TO 1 DO
     NODE i {
     DECLARATION:
     CPU=ALPHA; MEMORY=512; MULTI_PROC=4; PORT=[SCI,ATM];
     };
  OD

  -- the others are single processor nodes
  -- each with one SCI port
  FOR i=2 TO N-1 DO
     NODE i {
     DECLARATION:
     CPU=ALPHA; MEMORY=256; OS=SOLARIS;PORT=SCI;
     };
  OD

  CONNECTION:
  -- build the 1.0 Gbps unidirectional ring
  FOR i=0 TO N-1 DO
     EDGE edge_$i_to_$((i+1) MOD N)
     { NODE i PORT SCI => NODE ((i+1) MOD N) PORT SCI;
     BANDWIDTH = 1.0 Gbps;
     };
  OD

  -- establish a special virtual edge from node 0 to the
  -- port of the hyper node SCI_WSC (=outside world)
  ASSIGN edge_to_hypernode_port
  { NODE 0 PORT ATM <=> PORT ATM;};
};
```

Figure 10: RSD specification of the SCI part

Furthermore it should be possible to reconstruct the original representation, either graphical or text based. This facilitates the maintenance of large descriptions (e.g. a complex HPC center) and allows visualization at remote sites.

In order to use RSD in a distributed environment, a common format for exchanging RSD data structures is needed. The traditional approach would be to use a data stream format. However, this would involve two additional transformation steps whenever RSD data is to be exchanged (internal representation into data stream and back). Since the RSD internal representation has been defined in an object oriented way, this overhead can be avoided, when the complete object is sent across the network.

Today there exists a variety of standards for transmitting objects over the Internet, e.g. *CORBA*, *JavaBeans*, or *Component Object Module COM+*. Since we do not want to commit on either of these, we only define the interfaces of the RSD object class but not its private implementation. This allows others to choose an implementation that fits best to their own data structures. Interoperability between different implementations can be improved by defining translating constructors, i.e. constructors that take an RSD object as an argument and create a copy of it using another internal representation.

6 CCS in Practice

CCS was first used in an industrial setting in the Europort project [8] where large industrial codes were ported to PVM, PARMACS and MPI to run them on massively parallel systems. While we got much positive feedback from the users who praised the stability and versatility of CCS, our resource description language RDL (a predecessor of the RSD described here) was regarded as 'too complex'. Users complained about the tedious task of typing in a long RDL description when they just wanted to run a program on a simple target architecture.

Hence, with the 2^{nd} release of CCS we concealed RDL, giving the user only a simple command line interface. But with the advent of metacomputing, resource description became important again.

Application-Centric Metacomputing. The concept of CCS 4.0 proved useful in two ESPRIT projects [20], both with the goal to provide easy access to industrial applications that are run on Internet or Intranet-connected HPC systems. In both cases, a virtual user access point was implemented in Java that schedules incoming jobs to the temporarily best suited compute server in the Internet. Small and medium enterprises are expected to benefit most by the use of the distributed HPC services for running their most compute-intensive simulation applications – a service they could otherwise not afford due to expensive hardware, maintenance and education cost.

The keyword to these projects is *'application centric metacomputing'*: We do not simply provide raw computing time—as done in several other metacomputing projects—but we rather give access to specific pre-registered applications on a pay-per-use basis. The reasons are twofold:

- First, compute-intensive applications are typically also data-intensive, some of them repetitively running queries against very large databases. Clearly, the databases should be installed prior to access time and updated at night time.

- Second, industrial users are typically not willing to learn about vendor-specific HPC access just to

run their code; they rather prefer to see the machine through their application code's interface.

This scheme was proven in industrial projects running CPU-time intensive CFD simulations on servers in France, Germany and Great Britain. Because CFD simulations produce a large amount of output for visualizing flows and pressures, the server includes a caching facility, allowing the user to specify only that portion of data that is actually needed.

In the second project, we implemented a distributed pharmaceutical application server that allows truely *interactive* design of drug targets. The distributed server contains codes for the prediction of protein functions from sequences, for sensitive sequence searches, for 3D structure generation and for structure comparison. A virtual user access point has been implemented in Java with a job load balancing scheme based on the CIS concept. Security is ensured by data encryption, firewalls and Kerberos authentication. In addition, the server can be installed on in-house LANs for running the most sensitive drug design projects.

7 Related Work

Resource management systems emerged from the need for a better utilization of expensive HPC systems. The *Network Queuing System NQS* [25], developed by NASA Ames for the Cray2 and Cray Y-MP, might be regarded as the ancestor of many modern queuing systems like the Cray *Network Queuing Environment NQE* and the *Portable Batch System PBS*.

Following another path in the line of ancestors, the *IBM Load Leveler* is a direct descendant of *Condor* [27], whereas *Codine* [21] has its roots in *Condor* and *DQS*. They have been developed to support 'high-throughput computing' on UNIX workstation clusters. In contrast to high-performance computing, the goal is here to run a large number of (mostly sequential) batch jobs on workstation clusters without affecting interactive use. The *Load Sharing Facility LSF* [28] is another popular software for utilize LAN-connected workstations for high-throughput computing. For more detailed information on cluster managing software, the reader is referred to [2, 24].

These systems have been extended for supporting the coordinated execution of parallel applications, mostly based on PVM. A multitude of schemes have been devised for high-throughput computing on a somewhat larger scale, including the Iowa State University's *Batrun* [35], the CORBA-based *Piranha* [29], the Dutch *Polder* initiative [11], the *Nimrod* project [1], and the object-oriented *Legion* [22] which proved useful in a nation-wide cluster. While these schemes emphasize mostly on the application support on homogeneous systems, the *AppLeS* project [5] provides application-level scheduling agents on heterogeneous systems, taking into account their actual resource performance.

For the research presented in this paper, the already mentioned *Globus* project [15] is most important. Based on the lessons learned in the I-WAY experiment [14], the National Computational Science Alliance [34] implements a framework of an adaptive wide area metacomputer environment, where Globus, among Condor and *Symbio* (for clustering WindowsNT systems), plays a key role in establishing a national distributed computing infrastructure.

Globus aims at building an adaptive wide area resource environment (AWARE) with a set of tools that enables applications to adapt to heterogeneous and dynamically changing metacomputing environments. Similar to our CIS, a *metacomputing directory service (MDS)* [12] has been proposed to address the need for efficient and scalable access to diverse, dynamic, and distributed information. The API is vendor-independent. MDS is able to handle static and dynamic information. Like our MARS system [19], MDS is intended to manage application specific information that has been found useful in previous program runs (e.g. memory requirements, program structure, communication patterns).

8 Summary

We have presented history, presence and future development of the resource management software CCS. The current release 4.0 has the following features:

- It is modular and autonomous on each layer. New machines, networks, protocols, schedulers, system software, and meta-layers can be added at any point—some of them even without the need to re-boot the system.
- It is reliable. There is no single point of failure. Recovery is done at the machine layer. The center information manager (CIS) is passive and can be restarted or mirrored.
- It is scalable. There exists no central instance. The hierarchical approach allows to connect to other centers' resources. This concept has been found useful in several industrial projects.
- It is extensible. Other resource management systems (e.g. Codine, LSF, Condor) can be linked to CCS without the need to adjust their internal control regime.

From a software engineering view, each module can be implemented in another way, regardless of earlier implementations. From an administrators point of view, the system is easy to administer (by means of RSD and the operator shell), it is reliable, dynamic, and offers customized control on each level.

Compared to the Globus project, there are some similarities, but on a somewhat lower level. With this respect, CCS may be seen as a testbed for gaining valuable experiences with an existing resource management system that provides some important features required for practical metacomputing.

Current Status. Not all of the features have been fully implemented yet. We are currently in the transition phase between the previous RDL language and the here described, more general RSD description tool. Both, the CRM and the CIS have not yet been implemented completely. Furthermore, we plan to change the communication layer from RPCs to MPI-2 or Nexus. These communication layers support the use of multi-threaded daemons, thereby improving the performance of CCS under heavy load.

References

[1] D. Abramson, R. Sosic, J. Giddy, B. Hall. *Nimrod: A Tool for Performing Parameterized Simulations using Distributed Workstations.* 4th IEEE Symp. High Perf. and Distr. Comp. August 1995.

[2] M. Baker, G. Fox, H. Yau. *Cluster Computing Review.* Northeast Parallel Architectures Center, Syracuse University, Nov 1995, New York. http://www.npar.syr.edu/techreports/index.html

[3] B. Bauer, F. Ramme. *A General Purpose Resource Description Language.* Grebe, Baumann (eds), Parallel Datenverarbeitung mit dem Transputer, Springer-Verlag, Berlin, 1991, 68-75.

[4] R. Baraglia, G. Faieta, M. Formica, D. Laforenza. *Experiences with a Wide Area Network Metacomputing Management Tool using IBM SP-2 Parallel Systems.* Concurrency: Practice and Experience, John Wiley & Sons, Ltd., Vol. 8, 1996.

[5] F. Berman, R. Wolski, S. Figueira, J. Schopf, G. Shao. *Application-Level Scheduling on Distributed Heterogeneous Networks.* Supercomputing 96, Nov. 1996.

[6] M. Brune, J. Gehring, A. Reinefeld. *Heterogeneous Message Passing and a Link to Resource Management.* J. Supercomputing, Kluwer Acad. Publ., Vol 11, 355–369 (1997).

[7] M. Brune, J. Gehring, A. Keller, A. Reinefeld. *RSD - Resource and Service Description.* Tech. Rep., Paderborn Center for Parallel Computing, 1998. Also submitted to HPCS'98.

[8] A. Colbrook, M. Lemke, H. Mierendorff, K. Stueben, C.-A. Thole, O. Thomas. *Europort - ESPRIT European Porting Projects.* Int. Conf on High-Perf. Comp. and Netw., Springer LNCS 796 (1994), 46–54.

[9] *Computing Center Software CCS.* Paderborn Center for Parallel Computing. http://www.uni-paderborn.de/pc2/projects/ccs.

[10] T. DeFanti, I. Foster, M. Papka, R. Stevens, T. Kuhfuss. *Overview of the I-WAY: Wide Area Visual Supercomputing.* International Journal of Supercomputer Applications, 10(2):123-130, 1996.

[11] D. Epema, M. Livny, R. van Dantzig, X. Evers, J. Pruyne. *A Worldwide Flock of Condors: Load Sharing among Workstation Clusters.* FGCS, vol. 12, 1996, 53–66.

[12] S. Fitzgerald, I. Foster, C. Kesselman, G. von Laszewski, W. Smith, S. Tuecke. *A Directory Service for Configuring High-Performance Distributed Computations.* Proc. 6th IEEE Symp. on High-Performance Distributed Computing 1997

[13] I. Foster, J. Geisler, C. Kesselman, S. Tuecke. *Managing Multiple Communication Methods in High-Performance Networked Computing Systems.* J. Parallel and Distributed Computing, 40:35-48, 1997.

[14] I. Foster, J. Geisler, W. Nickless, W. Smith, S. Tuecke. *Software Infrastructure for the I-WAY High-Performance Distributed Computing Experiment.* Proc. 5th IEEE Symp. on High Performance Distributed Computing, 562-570, 1996.

[15] I. Foster, C. Kesselman. *Globus: A Metacomputing Infrastructure Toolkit.* Journal of Supercomputer Applications.

[16] I. Foster, D. Kohr, R. Krishnaiyer, J. Mogill. *Remote I/O: Fast Access to Distant Storage.* ANL Technical Report.

[17] E. Gabriel, T. Beisel, M. Resch. *PACX (PArallel Computer eXtension), An Installation Guide.* Installation guide for PACX Version 2.0, RUS Tech. Rep., 1997.

[18] J. Gehring, F. Ramme. *Architecture-Independent Request-Scheduling with Tight Waiting-Time Estimations.* IPPS'96 Workshop on Scheduling Strategies for Parallel Processing, 1996, Hawaii, Springer LNCS 1162, 41–54.

[19] J. Gehring, A. Reinefeld. *MARS - A Framework for Minimizing the Job Execution Time in a Metacomputing Environment.* Future Generation Computer Systems, Vol. 12, 87-99 (1996).

[20] J. Gehring, A. Reinefeld, A. Weber, *PHASE and MICA: Application Specific Metacomputing.* Europar'97, Passau, Germany, 1997.

[21] GENIAS Software GmbH. *Codine: Computing in Distributed Networked Environments.* http://www.genias.de/products/codine/.

[22] A. Grimshaw, J.B. Weissman, E.A. West, E.C. Loyot. *Metasystems: An Approach Combining Parallel Processing and Heterogeneous Distributed Computing Systems.* J. Par. Distr. Comp. 21 (1994), 257–270.

[23] GRM. *Resource Manager Specification v0.3.* http://www.globus.org/scheduler/grm_spec.html

[24] J.P. Jones, C. Brickell. *Second Evaluation of Job Queueing/Scheduling Software: Phase 1 Report.* Nasa Ames Research Center, NAS Tech. Rep. NAS-97-013, June 1997.

[25] B.A. Kinsbury. *The Network Queuing System.* Cosmic Software, NASA Ames Research Center, 1986.

[26] J. Linn. *Generic Security Service Applications Programming Interface.* Internet RFC 1508, (1993).

[27] M.J. Litzkow, M. Livny. *Condor–A Hunter of Idle Workstations.* Procs. 8th IEEE Int. Conf. Distr. Computing Systems, June 1988, 104–111.

[28] LSF. *Product Overview.* http://www.platform-.com/products/, July 1997.

[29] S. Maffeis. *Piranha: A CORBA Tool for High Availability.* IEEE Computer, April 1997, 59–66.

[30] F. Ramme, T. Römke, K. Kremer. *A Distributed Computing Center Software for the Efficient Use of Parallel Computer Systems.* HPCN Europe, Springer LNCS 797, Vol. II, 129-136 (1994).

[31] F. Ramme. *Transparente und effiziente Nutzung partitionierbarer Parallelrechner.* PhD Dissertation (in German), Paderborn Center for Parallel Computing, 1997.

[32] A. Reinefeld, R. Baraglia, T. Decker, J. Gehring, D. Laforenza, F. Ramme, T. Römke, J. Simon. *The MOL Project: An Open Extensible Metacomputer.* Proceedings HCW'97, Vienna, IEEE Computer Society Press, 17–31.

[33] L. Smarr, C.E. Catlett. *Metacomputing.* Communications of the ACM 35,6(1992), 45–52.

[34] R. Stevens, P. Woodward, T. DeFanti, C. Catlett. *From I-WAY to the National Technology Grid.* Communiations of the ACM 11(1997), 51–60.

[35] F. Tandiary, S.C. Kothari, A. Dixit, E.W. Anderson. *Batrun: Utilizing Idle Workstations for Large-Scale Computing.* IEEE Parallel and Distr. Techn., Summer 1996, 41–48.

Acknowledgments

Thanks to the members of the CCS-team, who have spent a tremendous effort on the development, implementation, and debugging since the project start in 1992: *Bernard Bauer, Matthias Brune, Harald Dunkel, Jörn Gehring, Oliver Geisser, Christian Hellmann, Axel Keller, Achim Koberstein, Rainer Kottenhoff, Karim Kremers, Fru Ndenge, Friedhelm Ramme, Thomas Römke, Helmut Salmen, Dirk Schirmer, Volker Schnecke, Jörg Varnholt, Leonard Voos, Anke Weber.*

Author Biographies

Axel Keller received his diploma in computer science from the University of Paderborn in 1993. As a staff member of the Paderborn Center for Parallel Computing he spent much time in designing and implementing CCS releases 2, 3, and 4.

Alexander Reinefeld received his CS diploma and PhD from the University of Hamburg in 1982 and 1987, respectively. In 1984/85 and 1987/88, he was awarded a DAAD scholarship and a Sir Walton Killam Post Doctoral fellowship for a two years study at the University of Alberta. He worked as a software consultant and as an assistant professor at the University of Hamburg. Since 1992, he manages the Paderborn Center for Parallel Computing.

A Dynamic Matching and Scheduling Algorithm for Heterogeneous Computing Systems

Muthucumaru Maheswaran and Howard Jay Siegel

Parallel Processing Laboratory
School of Electrical and Computer Engineering
Purdue University
West Lafayette, IN 47907-1285 USA
{maheswar, hj}@ecn.purdue.edu

Abstract

A heterogeneous computing system provides a variety of different machines, orchestrated to perform an application whose subtasks have diverse execution requirements. The subtasks must be assigned to machines (matching) and ordered for execution (scheduling) such that the overall application execution time is minimized. A new dynamic mapping (matching and scheduling) heuristic called the hybrid remapper is presented here. The hybrid remapper is based on a centralized policy and improves a statically obtained initial matching and scheduling by remapping to reduce the overall execution time. The remapping is non-preemptive and the execution of the hybrid remapper can be overlapped with the execution of the subtasks. During application execution, the hybrid remapper uses run-time values for the subtask completion times and machine availability times whenever possible. Therefore, the hybrid remapper bases its decisions on a mixture of run-time and expected values. The potential of the hybrid remapper to improve the performance of initial static mappings is demonstrated using simulation studies.

Keywords: dynamic scheduling, heterogeneous computing, list scheduling, mapping, matching, parallel processing, scheduling.

This work was supported by the DARPA/ITO Quorum Program under the NPS subcontract numbers N62271-97-M-0900 and N62271-98-M-0217.

1. Introduction

Different portions of a computationally intensive application often require different types of computations. In general, a given machine architecture with its associated compiler, operating system, and programming environment does not satisfy the computational requirements of all portions of an application equally well. However, a heterogeneous computing (HC) environment that consists of a heterogeneous suite of machines and high-speed interconnections provides a variety of architectural capabilities, which can be orchestrated to perform an application that has diverse computational requirements [2, 10, 14, 15]. The performance criterion for HC used in this paper is to minimize the completion time, i.e., the overall execution time of the application on the machine suite.

One way to exploit an HC environment is to decompose an application task into subtasks, where each subtask is computationally well suited to a single machine architecture. Different subtasks may be best suited for different machines. The subtasks may have data dependencies among them, which could result in the need for inter-machine communications. Once the subtasks are obtained, each subtask is assigned to a machine (matching). The subtasks and inter-machine data transfers are ordered (scheduling) such that the overall completion time of the application is minimized. It is well known that such a matching and scheduling (mapping) problem is, in general, NP-complete [3]. Therefore, many heuristics have been developed to

0-8186-8365-1/98 $10.00

obtain near-optimal solutions to the mapping problem. The heuristics can be either static (matching and scheduling decisions are made prior to application execution) or dynamic (matching and scheduling decisions are made during application execution).

Most static mapping heuristics assume that accurate estimates are available for (a) subtask computation times on various machines and (b) inter-machine data transfer times. Often, it is difficult to accurately estimate the above parameters prior to application execution. Therefore, this paper proposes a new dynamic algorithm, called the hybrid remapper, for improving the initial static matching and scheduling. The hybrid remapper uses the run-time values that become available for subtask completion times and machine availabilities during application execution time. It is called the hybrid remapper because it uses some results based on an initial static mapping in conjunction with information available only at execution time.

The hybrid remapper heuristics presented here are based on the list scheduling class of algorithms (e.g., [1, 9]). An initial, statically obtained mapping is provided as input to the hybrid remapper. If the initial mapping is not provided, it should be obtained before running the hybrid remapper by executing a static mapping algorithm such as the baseline [18], genetic-algorithm-based mapper [18], or Levelized Min Time [9].

The hybrid remapper executes in two phases. The first phase of the hybrid remapper is executed prior to application execution. The set of subtasks is partitioned into blocks such that the subtasks in a block do not have any data dependencies among them. However, the order among the blocks is determined by the data dependencies that are present among the subtasks of the entire application. The second phase of the hybrid remapper, executed during application run time, involves remapping the subtasks. The remapping of a subtask is performed in an overlapped fashion with the execution of other subtasks. As the execution of the application proceeds, run-time values for some subtask completion times and machine availability times can be obtained. The hybrid remapper attempts to improve the initial matching and scheduling by using the run-time information that becomes available during application execution and the information that was obtained prior to the execution of the application. Thus, hybrid remapper's decisions are based on a mixture of run-time and expected values.

This research is part of a DARPA/ITO Quorum Program project called MSHN (Management System for Heterogeneous Networks). MSHN is a collaborative research effort that includes NPS (Naval Postgraduate School), NRaD (a Naval Laboratory), Purdue, and USC (University of Southern California). It builds on SmartNet, an operational scheduling framework and system for managing resources in a heterogeneous environment developed at NRaD [6]. 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 organization of this paper is as follows. The matching and scheduling problem and the associated assumptions are defined in Section 2. Three variants of the hybrid remapper heuristics are described in Section 3. Section 4 examines the data obtained from the simulation studies conducted to evaluate the performance of the hybrid remapper heuristic. In Section 5, related work is discussed. Finally, Section 6 gives some future research directions.

2. Problem Definition

The following assumptions are made regarding the application. The application is decomposed into multiple subtasks and the data dependencies among them are known and are represented by a directed acyclic graph (DAG). That is, the nodes in the DAG represent the subtasks and the links represent the data dependencies. An estimate of the expected computation time of each subtask on each machine in the HC suite is known a priori. This assumption is typically made when conducting mapping research (e.g., [4, 7, 13, 16]). Finding the expected computation time is another research problem. Approaches based on analytical benchmarking and task profiling are surveyed in [14, 15]. Any loops and data conditionals are assumed to be contained inside a subtask.

It is assumed that the hybrid remapper is running on a dedicated workstation and all mapping decisions are centralized. Once a subtask is mapped onto a machine it is inserted into a local job queue on that particular machine. The execution of the subtask is managed by the job control environment of the local machine. The subtask executions are non-preemptive. All input data items of a subtask must be received before its execution can begin, and none of its output data items are available until its execution is completed. These assumptions make the matching and scheduling problem in HC systems more manageable. Nevertheless, solving the mapping problem with these assumptions is a significant step toward solving the more general problem.

An application task is decomposed into a set of subtasks $\underline{S}$, where s_i is the i-th subtask. Let the HC environment consist of a set of machines $\underline{M}$, where m_j be the j-th machine. The estimated expected computation time of subtask s_i on machine m_j is given by $e_{i,j}$. The earliest time at which machine m_j is available is given by $\underline{A}[j]$, where $|A| = |M|$.

The data communication time between two machines has two components: a fixed message latency for the first byte to arrive and a per byte message transfer time. An $|M|\times|M|$ communication matrix is used to hold these values for the HC suite. Similar matrices are used by other researchers in HC (e.g., [7, 13, 16]).

To facilitate the discussion in Section 3, a hypothetical node called an exit node is defined for the DAG as follows. An exit node (subtask) is a node with 0 computation time that is appended to the DAG such that there is a 0 data transfer time communication link to this node from every node in the DAG that does not have an output edge. The critical path for a node in the DAG is defined as the longest path from the given node to the exit node.

3. The Hybrid Remapper Algorithm

3.1. Overview

The notion behind most dynamic mapping algorithms is that due to the dynamic nature of the mapping problem, it is not efficient to use a fixed mapping computed statically. Therefore, most dynamic mappers regularly either generate the mapping or refine an existing mapping at various times during task execution. That is, dynamic mapping algorithms solve the mapping problem by solving a series of partial mapping problems (consisting of only a subset of the original set of subtasks). The partial mapping problem is usually solved by a static mapping heuristic. Because the mapping is performed in real time, it is necessary to use a fast algorithm to avoid any machine idle times that occur from having to wait for the mapper to complete its execution. In the hybrid remapper algorithm presented here, the partial mapping problem is solved using a list-based scheduling algorithm.

In the following subsections, three variants of the hybrid remapper algorithm are described. The first phase, common for all three variants of the hybrid remapper, involves partitioning the subtasks into blocks and assigning ranks to each subtask (where the rank indicates the subtask's priority for being mapped, as defined below). The variants of the hybrid remapper differ in the second phase by the minimization criteria they use and by the way they order the subtasks examined by the partial mapping problem. One variant of the hybrid remapper attempts to minimize the expected partial completion time at each remapping step, and the others attempt to minimize the overall expected completion time. Two variants of the hybrid remapper order the subtasks at each remapping step using ranks computed at compile time, and the other using a parameter computed at run time.

3.2. Partitioning and Rank Assignment

This first phase uses the initial static mapping, expected subtask computation times, and expected data transfer times to preprocess the DAG that represents the application. Initially, the DAG is partitioned into B blocks numbered consecutively from 0 to $B-1$. The partitioning is done such that the subtasks within a block are independent, i.e., there are no data dependencies among the subtasks in a block. All subtasks that send data to a subtask s_j in block k must be in any of blocks 0 to $k-1$. Furthermore, for each subtask s_j in block k there exists at least one incident edge (data dependency) such that the source subtask is in block $k-1$, i.e., an incident edge from some s_i. The $(B-1)$-th block includes the subtasks without any successors and the 0-th block includes only those subtasks without any predecessors. The exit node is not included in any block in the DAG partitioning. The three blocks obtained using this partitioning algorithm for an example seven node DAG is shown in Figure 1(a).

Once the subtasks in the DAG are partitioned, each subtask is assigned a rank by examining the subtasks from block $B-1$ to block 0. The rank of each subtask in the $(B-1)$-th block is set to its expected computation time on the machine to which it was assigned by the initial static matching. Now consider the k-th block, $0 \le k < B-1$. Recall $e_{i,x}$ is the expected computation time of the subtask s_i on machine m_x. Let $c_{i,j}$ be the data transfer time for a descendent s_j of s_i to get all the relevant data items from s_i. The value of $c_{i,j}$ will be dependent on the machines assigned to subtasks s_i and s_j by the initial mapping, and the information in the communication matrix. Let $\mathrm{iss}(s_i)$ be the immediate successor set of subtask s_i such that there is an arc from s_i to each member of $\mathrm{iss}(s_i)$ in the DAG. In the equation below, each $e_{i,x}$ implies subtask s_i is assigned to machine m_x by the initial mapping. With these definitions, the rank of a subtask s_i is given by:

$$\mathrm{rank}(s_i) = e_{i,x} + \max_{s_j \in \mathrm{iss}(s_i)} (c_{i,j} + \mathrm{rank}(s_j))$$

Figure 1(b) illustrates the rank assignment process for the subtask s_i. The rank of a subtask can be interpreted as the length of the critical path from the point the given subtask is located on the DAG to the exit node, i.e., the time until the end of the execution of all its descendents. Two variants of the hybrid remapper described here are based on the heuristic idea that by executing the subtasks with higher ranks as quickly as possible, the overall expected completion time for the application can

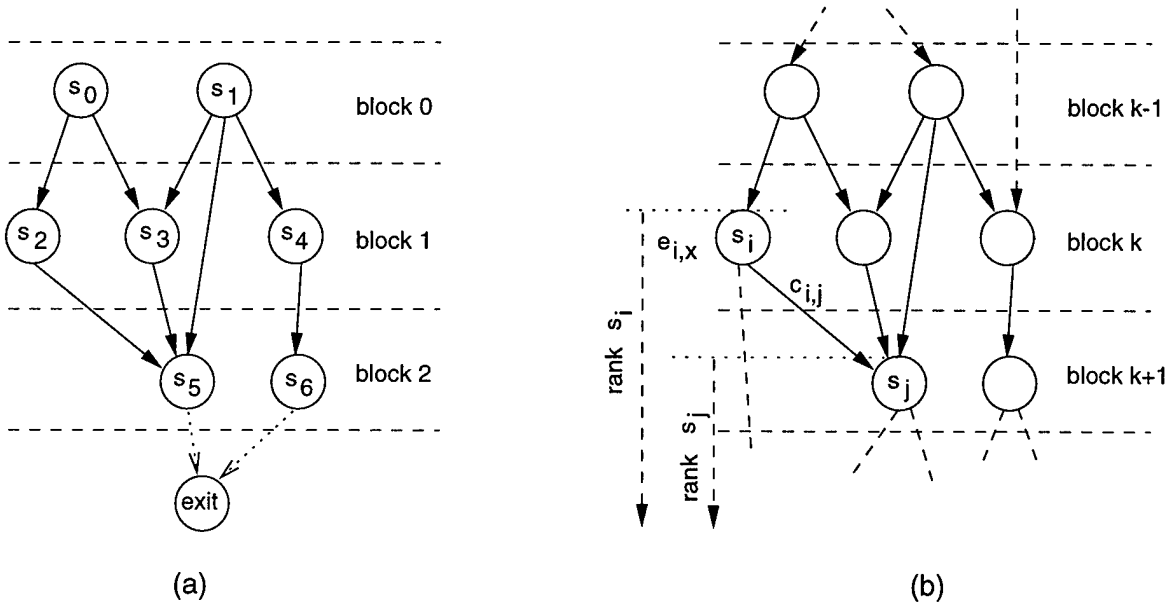


Figure 1: (a) Partitioning a DAG into blocks and (b) assigning ranks to the nodes of a DAG.

be minimized.

3.3. Common Portion of the Run-Time Phase

In all three variants of the hybrid remapper, the execution of the subtasks proceeds from block 0 to block $B-1$. A block k is considered to be executing if at least one subtask from block k is running. Also, the execution of several blocks can overlap with each other in time, i.e., subtasks from different blocks could be running at the same time.

The hybrid remapper changes the matching and scheduling of the subtasks in block k while the subtasks in block $(k-1)$ or before are being executed. The hybrid remapper starts examining the block k subtasks when the first block $(k-1)$ subtask begins its execution. When block k subtasks are being mapped, it is highly likely that run-time completion time information can be used for many subtasks from blocks 0 to $k-2$. There may be some subtasks from blocks 0 to $k-2$ that are still running or waiting execution when subtasks from block k are being considered for remapping. For such subtasks, expected completion times are used.

3.4. Minimum Partial Completion Time Static Priority (PS) Algorithm

As mentioned earlier, the hybrid remapper uses a list-scheduling type of algorithm to recompute the matching and scheduling for the subtasks in each block. In a list-scheduling type of algorithm, the subtasks are first ordered based on some priority. Then, each subtask is mapped to a machine by examining the list of subtasks from the highest priority subtask to the lowest priority subtask. The machine to which each subtask is assigned depends on the matching criterion used by the particular algorithm.

In this variant of the hybrid remapper, the priority of a subtask is equal to the rank of that subtask that was computed statically in the first phase (Subsection 3.2). The matching criterion used for subtask s_i is the minimization of the partial completion time, defined below. Thus, this variation is referred to as the minimum partial completion time static priority (PS) algorithm.

Let m_x be the machine on which s_i is being considered for execution. Then let $\mathrm{pct}(s_i,x)$ denote the partial completion time of the subtask s_i on machine m_x, $\mathrm{dr}(s_i)$ be the time at which the last data item required by s_i to begin its execution arrives at m_x, and $\mathrm{ips}(s_i)$ be the immediate predecessor set for subtask s_i such that there is an arc to s_i from each member of $\mathrm{ips}(s_i)$ in the DAG. For any subtask s_i in block 0, $\mathrm{pct}(s_i,x) = e_{i,x}$. For any subtask s_i not in block 0, where $s_j \in \mathrm{ips}(s_i)$, and s_j is currently mapped onto machine m_y,

$$\mathrm{dr}(s_i) = \max_{s_j \in \mathrm{ips}(s_i)} (c_{j,i} + \mathrm{pct}(s_j,y))$$

$$\mathrm{pct}(s_i,x) = e_{i,x} + \max(A[x], dr(s_i))$$

In the computation of $\mathrm{pct}(s_i,x)$, the above equation is recursively used until subtask s_j is such that its run-time

completion time on machine m_y is available or subtask s_j is in block 0. The subtask s_i is remapped onto the machine m_x that gives the minimum $\text{pct}(s_i,x)$, and $A[x]$ is updated using $\text{pct}(s_i,x)$. Then the next subtask from the list is considered for remapping.

3.5. Minimum Completion Time Static Priority (CS) Algorithm

The notion behind the PS algorithm was that by remapping the highest rank subtask s_i to execute on the machine that will result in the smallest expected partial completion time, the overall completion time of the application may be minimized. Instead of this approach, the variant of the hybrid remapper described here attempts to minimize the overall completion time by remapping each subtask s_i in block k such that the length of the critical path through subtask s_i is reduced. Thus, this variation is referred to as the minimum completion time static priority (CS) algorithm. The reason for considering both PS and CS is that in PS the remapping is faster but CS attempts to derive a better mapping because it considers the whole critical path through s_i.

Let m_x be the machine on which s_i is being considered for execution. Then let the longest completion time path from a block 0 subtask to the exit node through the subtask s_i be $\text{ct}(s_i,x)$. The overall completion time of the application task is determined by one such longest path through a block k subtask. Consider the subtask s_i in Figure 2. Assume that the longest path through s_i is shown by bold edges in Figure 2. For any subtask s_i,

$$\begin{aligned}\text{ct}(s_i,x) &= \max_{s_j \in \text{iss}(s_i)}(\text{pct}(s_i,x)+c_{i,j}+\text{rank}(s_j)) \\ &= \text{pct}(s_i,x)+\max_{s_j \in \text{iss}(s_i)}(c_{i,j}+\text{rank}(s_j))\end{aligned}$$

The subtask s_i is remapped onto the machine m_x that gives the minimum $\text{ct}(s_i,x)$, and $A[x]$ is updated using $\text{pct}(s_i,x)$. Then the next subtask in the list is considered for remapping.

3.6. Minimum Completion Time Dynamic Priority (CD) Algorithm

The rank of a subtask s_i is computed prior to application execution. Therefore, if s_i is remapped to a machine other than the one it was assigned to by the initial static mapping, the rank of s_i may not give the length of the critical path from s_i to the exit node.

The algorithm presented here is same as the CS algorithm, except ranks are no longer used in ordering the subtasks within a block. Instead of using the statically computed ranks, this algorithm uses the value of $\text{ct}(s_i,x)$, where m_x is the machine assigned to s_i in the initial mapping, to order the subtasks within a block. Thus, this variation is referred to as the minimum completion time dynamic priority (CD) algorithm.

The example shown in Figure 3 illustrates why using ranks computed at compile time to order the subtasks within a block may not lead to the best overall completion time. In the given example, the DAG shown in Figure 3(a) is mapped onto two machines m_0 and m_1. Figure 3(b) shows the subtask computation time matrix, which gives the computation time of a subtask on different machines. The initial static mapping is shown in Figure 3(c). The numbers inside each bar correspond to the subtask index and the execution time of the subtask, in "subtask index/execution time" notation. The times are given in seconds. The data transfer times are negligible if the source and destination machines are the same, otherwise, for this example there is a fixed time of two seconds for the data transfer. In Figure 3(a), the number outside each node indicates the rank of that subtask derived using the initial mapping.

When block 2 is considered for remapping by either the PS or CS algorithm, s_5 is mapped first and then s_4 is mapped. Suppose s_0 finishes its execution in 20 seconds instead of 10 seconds and s_1 finishes in 10 seconds. This causes the subtask s_4 to become critical and s_5 to become non-critical, i.e., s_5 is not part of the critical path anymore. By using the rank numbers that were statically computed, the PS and CS algorithms map s_5 before s_4. Thus, s_5 will be mapped to the best machine and this can delay the completion of s_4. Instead of using the statically computed ranks, the CD algorithm considers $\text{ct}(s_i,x)$, where subtask s_i is assigned to m_x in the initial mapping. For this example, subtask s_4 is assigned to machine m_0 and subtask s_5 is assigned to machine m_1. Therefore, the CD algorithm considers $\text{ct}(s_4,0)$ and $\text{ct}(s_5,1)$ to determine the remapping order.

$$\text{ct}(s_4,0) = 20+15+10+10 = 55$$

$$\text{ct}(s_5,1) = 10+20+10+2+10 = 52$$

Because the value of $\text{ct}(s_5,1)$ is less than the value of $\text{ct}(s_4,0)$, s_4 is considered for remapping before s_5 by the CD algorithm. This example illustrates that using $\text{ct}(s_i, x)$, where m_x is the machine that is assigned to s_i in the initial mapping, enables the remapping algorithm to track the critical path better than using the static ranks.

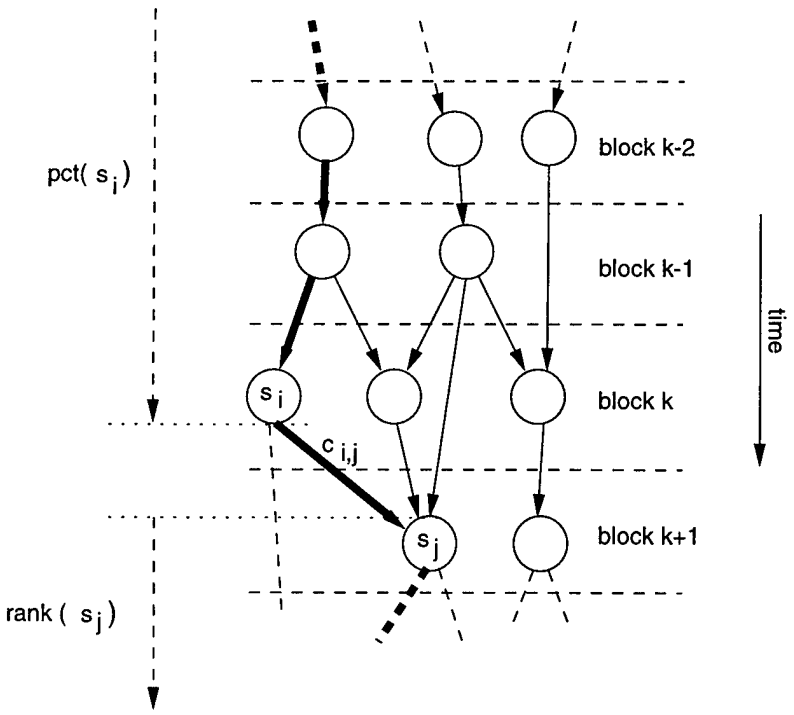


Figure 2: Estimating the completion time by considering the longest path through s_i.

4. Experimental Results and Discussion

4.1. Simulation Parameters

A simulator was implemented to evaluate the performance of the hybrid remapper variants. Various parameters are given as input to the simulator. Some parameters are specified as fixed values, e.g., number of machines, and others as a range of values with a maximum and a minimum value, e.g., subtask computation time. When a range is specified, the actual value is set to a random value within the specified range. Each data point in the results presented in this section is an average of 100 simulation runs. The experiments were performed on a Sun Ultra with a SPARC processor running at 165 MHz.

To generate a DAG that represents an application, the number of subtasks, maximum out degree of a node, number of data items to be transferred among different subtasks, range for subtask computation times, and range for data item sizes are provided as input to the simulator. Using these input parameters the simulator creates a table with the subtasks along the columns and data items along the rows. If a subtask s_j produces a data item d_i then the cell (i,j) has the label PRODUCER and if the subtask s_j consumes a data item d_i then the cell (i,j) has the label CONSUMER. A data item has one and only one producer, but may have zero or more consumers. For a given data item, a producer is randomly picked and then consumers are picked such that the resulting graph is acyclic and the maximum out degree constraints are satisfied.

To define the HC suite, the number of machines is provided as input. The simulator randomly generates valid subtask computation times to fill a table that determines the subtask computation times on each machine in the HC suite. For these experiments it is assumed that a fully connected, contention-free communication network is used. The inter-machine communication times are source and destination dependent. Communication times are specified by a range value. The run-time value of a parameter such as the subtask execution time or inter-subtask data communication time can be different from the expected value of the parameter. The variation is modeled by generating simulated run-time values by sampling a probability distribution function (PDF) that has the expected value of the parameter as the mean.

4.2. Generational Scheduling

In this subsection, the generational scheduling (GS) algorithm [5] is briefly described. The performance of the hybrid remapper is compared with the performance of the GS algorithm in the next subsection. The GS algorithm is a dynamic mapping heuristic for HC systems.

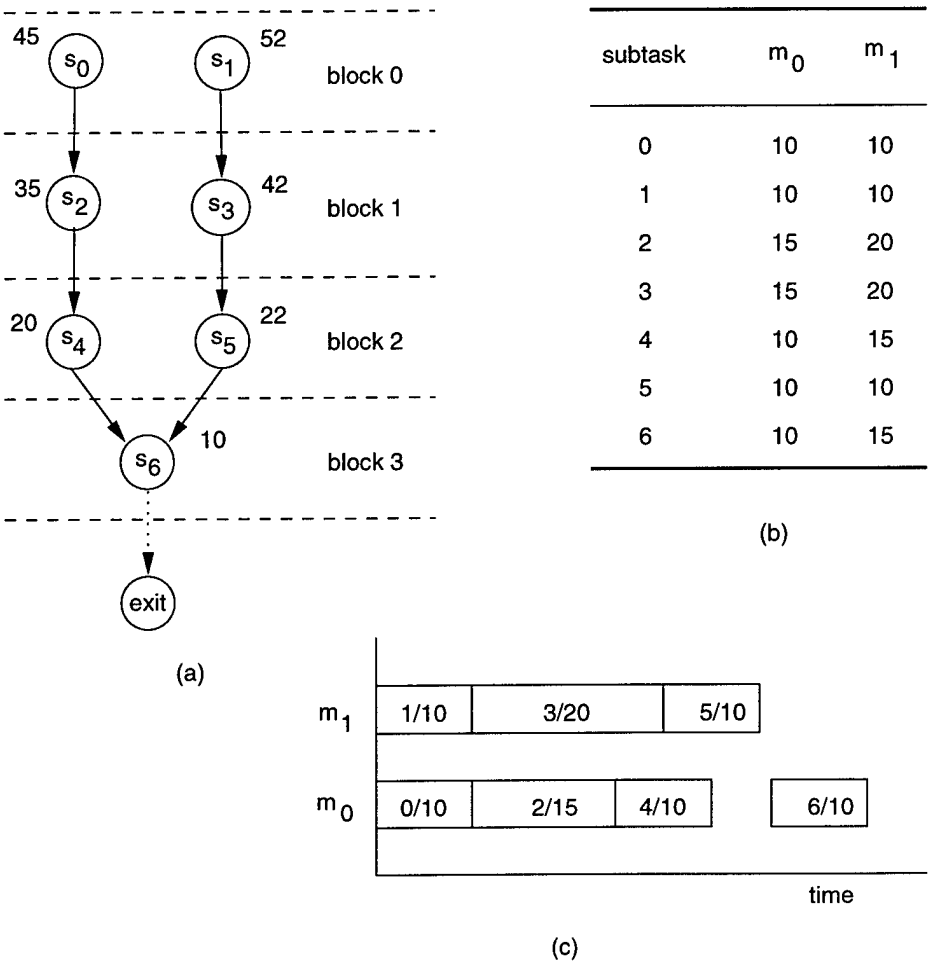


Figure 3: An example mapping to illustrate the benefit of the CD algorithm: (a) the partitioned DAG, (b) the subtask computation time matrix, and (c) the initial mapping.

Initially, the GS forms a partial scheduling problem by pruning all the subtasks with unsatisfied precedence constraints from the initial DAG that represents the application. The initial partial scheduling problem consists of subtasks that correspond to those in block 0 of the hybrid remapper approach. The subtasks in the initial partial scheduling problem are then mapped onto the machines using an auxiliary scheduler. The auxiliary scheduler considers the subtasks for assignment in a first come first serve order. A subtask is assigned to a machine that minimizes the completion time of that particular subtask.

When a subtask from the initial partial scheduling problem completes its execution, the GS algorithm performs a remapping. During the remapping, the GS revises the partial scheduling problem by adding and removing subtasks from it. The completion of the subtask that triggered the remapping event may have satisfied the precedence constraints of some subtasks. These subtasks are added to the initial partial scheduling problem. The subtasks that have already started execution are removed from the initial partial scheduling problem. Once the revised partial scheduling problem is obtained, the subtasks in it are mapped onto the HC machine suite using the auxiliary scheduler. This procedure is iteratively performed until the completion of all subtasks.

4.3. Hybrid Remapper

From the discussions in Section 3, it can be noted that the hybrid remapper is provided with an initial mapping that is derived prior to application execution using a static matching and scheduling algorithm. The simulator generates a random DAG, using the parameters it receives as input, at the beginning of each simulation run. An initial static mapping for this DAG is obtained by matching and scheduling this DAG onto the HC suite using the baseline algorithm [18].

The baseline algorithm that is used to derive the initial mapping is a fast static matching and scheduling algorithm. It partitions the subtasks into blocks using an algorithm similar to the one described in Subsection 3.2. Once the subtasks are partitioned into blocks, they are ordered such that a subtask in block k comes before a subtask in block l, where $k<l$. The subtasks in the same block are arranged in descending order based on the number of descendents of each subtask (ties are broken arbitrarily). The subtasks are considered for assignment by traversing the list, beginning with block 0 subtasks. A subtask is assigned to the machine that gives the shortest time for that particular subtask to complete.

In this simulator, three different PDFs (a) Erlang(2) [12], (b) uniform, and (c) skewed uniform are used to generate the simulated run-time values. For Erlang(2), the expected values are provided as the mean and the PDF is sampled to obtain a simulated run-time value. In Figure 4, 10,000 consecutive random numbers generated by the Erlang(2) random number generator with mean ten is shown using a 200-bin histogram. For the skewed uniform PDF, the following rule is used to generate the simulated run-time value. Let α_1 be the negative percentage deviation, α_2 be the positive percentage deviation, and u be a random number that is uniformly distributed in [0,1]. Then, the simulated run-time value of a parameter τ can be modeled as $\tau \times (100-\alpha_1+(\alpha_1+\alpha_2)u)/100$. For the uniform PDF, $\alpha_1 = \alpha_2 = \alpha$. For the simulation results presented here, Erlang(2) is used unless otherwise noted.

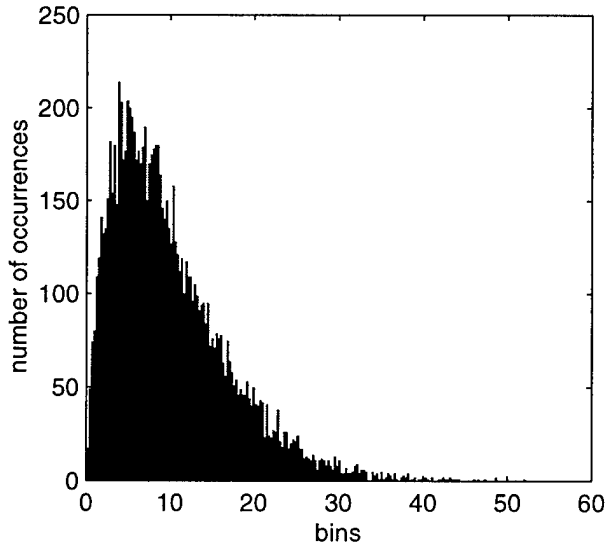


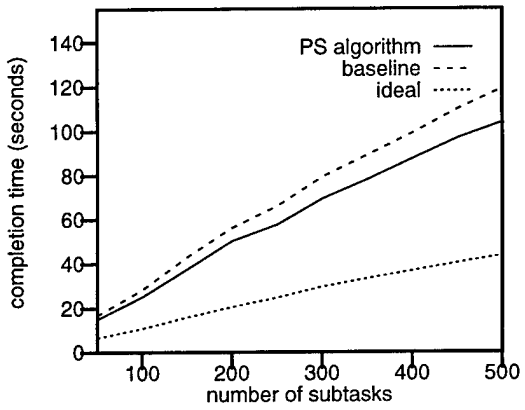
Figure 4: A 200-bin histogram for 10,000 consecutive samples of the Erlang(2) random number generator with mean equal to ten.

In these experiments, baseline refers to first deriving a static mapping using the baseline algorithm and expected subtask computation and communication times, and then, using this mapping, computing the total application execution time based on the simulated run-time values for computation and communication times. Also, in these experiments, ideal refers to deriving a static mapping using the baseline algorithm and simulated run-time values (instead of the expected values) for subtask computation and communication times. Note that this ideal is used for comparison purposes only, and cannot be implemented in practical environments. Also note that the ideal is not necessarily the optimal mapping. These simulated run-time values are also used to evaluate the application task completion time with the hybrid remapper variants.

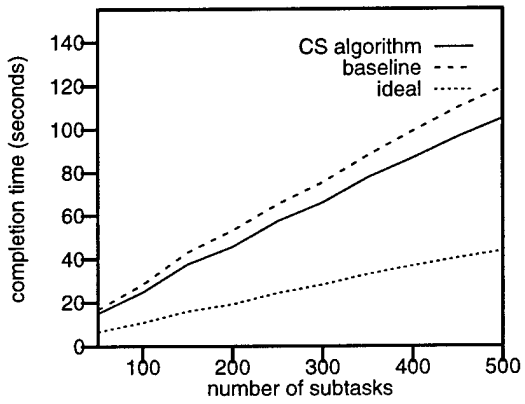
In Figure 5(a), the performance of the PS algorithm is compared to the mapping that is obtained using the baseline algorithm for ten machines. Figure 5(b) shows a similar comparison for the CS algorithm for ten machines. The performance of the CD algorithm is shown in Figures 6(a) and 6(b). Figure 6(a) compares CD and the baseline for varying numbers of subtasks and ten machines. Figure 6(b) compares the two approaches for varying numbers of machines and 200 subtasks.

From Figures 5(a), 5(b), and 6(a) it can be observed that the performance difference among the three variants is almost negligible. The heuristic improvements performed to obtain the CS and CD variants from the PS variant of the hybrid remapper make the CS and CD use more initial matching and scheduling derived information. That is, while CS and CD use more information in an attempt to derive a better mapping than the PS, the information is based on expected values, rather than run-time values. Thus, there is no significant improvement. Also, in these simulation studies, the initial mapping is obtained using a simple baseline algorithm. The performance of CS and CD may improve if a higher quality initial assignment is used, e.g., if a genetic algorithm based mapper [18] is used for the initial matching and scheduling.

As the number of subtasks increase, the performance difference between each hybrid remapper variant and the baseline increases. This increase in performance can be attributed to two factors: (a) increased number of remapping events and (b) increased average number of subtasks per block. Increasing the number of remapping events provides the hybrid remapper with more opportunities to exploit the run-time values of parameters that are available during application execution. Also, with the increased number of subtasks per block the hybrid remapper can derive schedules that are very different from the initial schedule. Therefore, the average performance of the hybrid remapper increases with increasing number of subtasks.



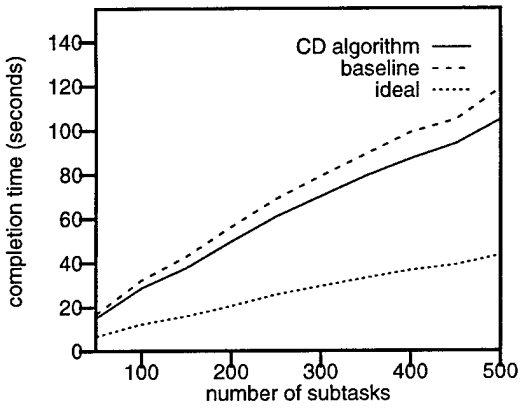
(a)



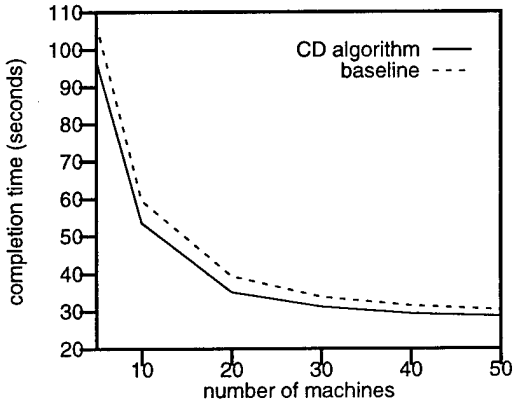
(b)

Figure 5: Performance of the hybrid remapper versus the baseline for (a) the PS algorithm and (b) the CS algorithm.

Ten machines and 100 subtasks were used in Figure 7. In Figure 7(a), the performance of the CD algorithm is compared with the baseline for varying computation/communication ratios and Figure 7(b) shows the performance comparison of the CD algorithm with the baseline for varying average number of subtasks per block. Figure 7(a) shows that the hybrid remapper performs better as the computation/communication ratio increases. The computation/communication ratio is the average subtask execution time divided by the average inter-subtask communication time. In Figure 7(a), the low computation/communication ratio denotes the range 1.0-10.0, medium computation/communication ratio denotes the range 10.0-200.0, and high computation/communication ratio denotes the range 200.0-4000.0. With increasing computation/communication ratio, the data transfer times become less significant compared to the subtask computation times. The reason for the hybrid remapper not performing well with a low computation/communication ratio is currently under investigation.



(a)



(b)

Figure 6: Performance of the CD algorithm versus the baseline for (a) varying the subtasks and (b) varying the machines.

From Figure 7(b) it can be noted that the relative performance of the CD algorithm increases with increasing the average number of subtasks per block. When there are more subtasks per block, it is possible for the hybrid remapper to derive mappings that are very different from the initial mapping.

Figure 8(a) compares the performance of the CD algorithm with the baseline algorithm for a uniform distribution PDF, 20 machines, and 200 subtasks. Figure 8(b) performs the same comparison for a skewed uniform distribution PDF, 20 machines, and 200 subtasks. In the skewed uniform distribution the negative percentage deviation is half of the positive percentage deviation.

As noted earlier, one of the features of the hybrid remapper algorithm that is presented here is overlapping its operation with the execution of the subtasks. To

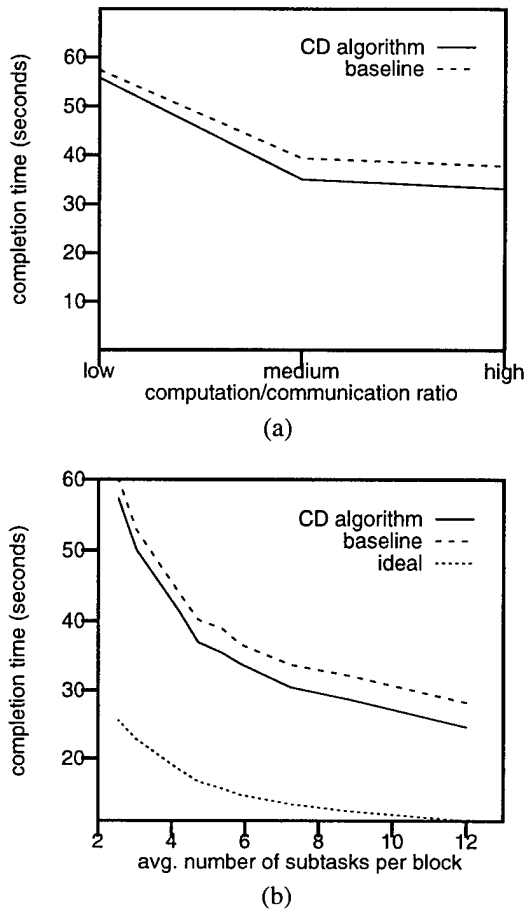


Figure 7: Performance of the CD algorithm versus the baseline for (a) varying the computation/communication ratio and (b) varying the average number of subtasks per block.

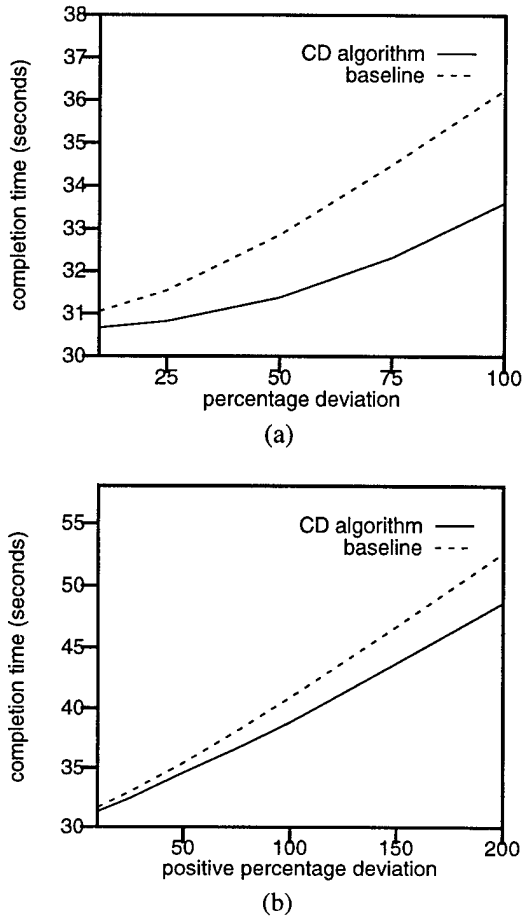


Figure 8: Performance of the CD algorithm versus the baseline for (a) using the uniform distribution for parameter modeling and (b) skewed uniform for parameter modeling.

obtain complete overlap, in the worst case, the remapping time for a block of subtasks should be less than the execution time of the smallest subtask in the previous block. More precisely, the time available for remapping block k is equal to the difference between the time the first block $k-1$ subtask begins execution and the time the first block k subtask can begin execution. Figure 9(a) shows the per block remapping time for the CD algorithm for varying numbers of subtasks and ten machines. In Figure 9(b), the per block remapping time for the CD algorithm is shown for varying numbers of machines and 200 subtasks.

In Figure 10, the performance of the CD algorithm is compared with the GS algorithm for varying numbers of subtasks. From the simulation results it can be observed that the CD algorithm is slightly outperforming the GS algorithm. From the discussions in Section 3.6, it can be noted that the CD algorithm attempts to minimize the length of the critical path at each remapping step. In the GS algorithm, the critical path through the DAG is not considered when the subtasks are remapped. This is one reason for the better performance of the CD algorithm. The GS algorithm has more remapping events compared to the hybrid remapper. The number of remapping events is equal to the number of subtasks in the GS algorithm and equal to the number of blocks in the CD algorithm. The increased number of remappings allows the GS algorithm to base its assignment decisions on more current values. This may be why the GS is performing only three to four percent worst than the CD algorithm even though GS does not consider the critical path through the DAG. In the GS algorithm, at least one machine may be waiting on the scheduler to finish the mapping process. This scheduler induced wait time on the HC suite was not included in the GS versus CD com-

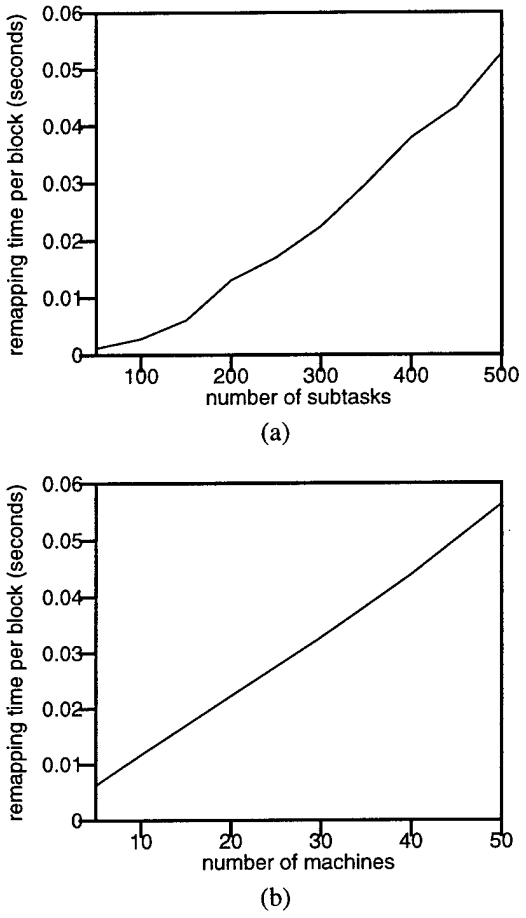


Figure 9: Per block remapping time of CD for (a) varying subtasks and (b) varying machines.

parison.

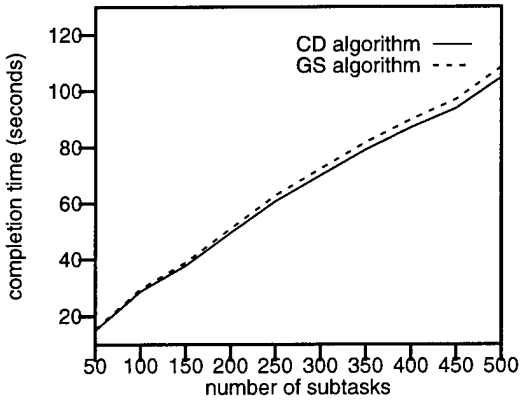


Figure 10: Performance of the CD algorithm versus the Generational Scheduling algorithm for varying numbers of subtasks.

5. Related Work

Other groups have also studied dynamic mapping heuristics for HC systems (e.g., [5, 8, 11]). A brief description of the GS algorithm and an experimental comparison of the hybrid remapper with the GS algorithm were presented in Section 4. The Self-Adjusting Scheduling for Heterogeneous Systems (SASH) algorithm is presented in [8]. One of the differences between the hybrid remapper and the SASH algorithm is that the hybrid remapper uses a list-scheduling based algorithm to perform the remappings at run time, whereas the SASH algorithm uses a variation of the branch and bound algorithm to generate the partial mappings at each remapping event. Also, unlike the GS and SASH algorithms, the hybrid remapper presented here can use any initial mapping to guide its remapping decisions, i.e., the initial mapping is used to compute the ranks and completion time estimates in the hybrid remapper. It is necessary to experimentally determine how the quality of the initial mapping impacts the overall performance of the hybrid remapper.

In [11], two mapping algorithms are presented. One is based on a distributed model and the other is based on a centralized model. The distributed mapping algorithm is different from the algorithms presented in [5, 8], and the hybrid remapper presented here, which are all centralized algorithms. The centralized mapping algorithm is based on a global queue equalization algorithm.

6. Conclusions and Future Work

The simulation results indicate that the performance of a statically obtained initial mapping can be improved by the hybrid remapper. From the simulation results obtained, performance improvement can be as much as 15% for some cases. The timings also indicate that the remapping time needed per block of subtasks is in the order hundreds of milliseconds for up to 50 machines and 500 subtasks. In the worst case situation, to obtain complete overlap, the computation time for the shortest running subtask must be greater than the per block remapping time.

The experimental studies revealed that the hybrid remapper performs better than the generational scheduling, but the margin of difference was only three to four percent. The hybrid remapper has a better machine utilization compared to the generational scheduling algorithm, because in the hybrid remapper the mapping operations are overlapped with the application execution. Further research is necessary to develop ways to improve the hybrid remapper's performance. This include examining the use of different schemes for partitioning the DAG into blocks, exploring the use of different ways of

ordering subtasks within a block, and investigating the use of different criteria for determining subtask to machine assignments.

The partitioning scheme that is currently used in the hybrid remapper does not consider the usage pattern of the data items produced by a subtask. The partitioning is solely based on the data dependencies. This could cause a subtask with low rank value in a block k to be mapped before a subtask with high rank value in a block l, where $l > k$. Various alternate partitioning schemes need to be explored and evaluated to examine different criteria for forming blocks.

One of the features of the hybrid remapper algorithm presented here is the overlap of the execution of the hybrid remapper algorithm with the execution of the subtasks. In the hybrid remapper developed in this research, the remapping event for block k is the readiness to execute of the first block $k-1$ subtask. Hence, the number of remapping events is equal to the number of blocks. In other algorithms, such as the Generational Scheduling algorithm [5], the number of remapping events is equal to the number of subtasks. It is necessary to study the trade-offs of increasing the number of remapping events on the performance of the algorithms and the amount of machine idle time from having to wait for a mapping decision. Also, the interaction of varying the amount of uncertainty in the parameter values and increasing the number of remapping events needs further research.

In this paper, the performance of the hybrid remapper is compared with the performance of the static baseline, and the dynamic generational scheduling algorithm [5]. Further simulation studies are necessary to compare the performance of the hybrid remapper with other dynamic mapping algorithms, such as the queue equalization algorithm [11].

The hybrid remapper developed in this research assumed a fully connected, contention-free communication model. This model needs to be improved to accommodate message contention and restricted inter-machine network topologies that occur in practical situations. Also, enhancements are necessary to support cases where a subtask can have multiple sources (machines) for a needed data item [17].

The performance of the hybrid remapper has been studied using simulations in this research. Exploring the possibility of obtaining performance bounds using analytical methods is yet another possible area of future research.

Another future area of study is to evaluate the performance of the hybrid remapper when the initial mapping is generated by a genetic algorithm (GA) based mapper [18]. Also, it would be interesting to compare the relative performance of the hybrid remapper and the mapping obtained by a static GA-based mapper as the run-time values of the parameters deviate from their expected values.

In summary, a new dynamic mapping algorithm called the hybrid remapper was presented in this paper. The hybrid remapper uses novel heuristic approaches to dynamically improve a statically obtained initial mapping. The potential of the hybrid remapper to improve the performance of initial static mappings was demonstrated using simulation studies.

Acknowledgments -- The authors thank Robert Armstrong, Tracy Braun, Debra Hensgen, Taylor Kidd, Yu-Kwong Kwok, and Viktor Prasanna for their comments and useful discussions.

References

[1] T. L. Adam, K. M. Chandy, and J. R. Dickson, "A comparison of list schedules for parallel processing systems," *Comm. of the ACM,* Vol. 17, No. 12, Dec. 1974, pp. 685-690.

[2] M. M. Eshaghian, ed., *Heterogeneous Computing,* Artech House, Norwood, MA, 1996.

[3] D. Fernandez-Baca, "Allocating modules to processors in a distributed system," *IEEE Trans. on Software Engineering,* Vol. SE-15, No. 11, Nov. 1989, pp. 1427-1436.

[4] R. F. Freund, "The challenges of heterogeneous computing," *Parallel Systems Fair at the 8th Int'l Parallel Processing Symp.,* Apr. 1994, pp. 84-91.

[5] R. F. Freund, B. R. Carter, D. Watson, E. Keith, and F. Mirabile, "Generational scheduling for heterogeneous computing systems," *Int'l Conf. Parallel and Distributed Processing Techniques and Applications (PDPTA '96),* Aug. 1996, pp. 769-778.

[6] R. F. Freund, T. Kidd, D. Hensgen, and L. Moore, "SmartNet: A scheduling framework for metacomputing," *2nd Int'l Symp. Parallel Architectures, Algorithms, and Networks (ISPAN '96),* June 1996, pp. 514-521.

[7] A. Ghafoor and J. Yang, "Distributed heterogeneous supercomputing management system," *IEEE Computer,* Vol. 26, No. 6, June 1993, pp. 78-86.

[8] B. Hamidzadeh, D. J. Lilja, and Y. Atif, "Dynamic scheduling techniques for heterogeneous computing systems," *Concurrency: Practice and Experience,* Vol. 7, No. 7, Oct. 1995, pp. 633-652.

[9] M. A. Iverson, F. Ozguner, and G. J. Follen, "Parallelizing existing applications in a distributed heterogeneous environment," *4th Heterogeneous Computing Workshop (HCW '95),* Apr. 1995, pp. 93-100.

[10] A. Khokhar, V. K. Prasanna, M. Shaaban, and C. L. Wang, "Heterogeneous computing: Challenges and opportunities," *IEEE Computer,* Vol. 26, No. 6, June 1993, pp. 18-27.

[11] C. Leangsuksun, J. Potter, and S. Scott, "Dynamic task mapping algorithms for a distributed heterogeneous computing environment," *4th Heterogeneous Computing Workshop (HCW '95),* Apr. 1995, pp. 30-34.

[12] A. Papoulis, *Probability, Random Variables, and Stochastic Processes, Second Edition,* McGraw-Hill, New York, NY, 1984.

[13] P. Shroff, D. W. Watson, N. S. Flann, and R. F. Freund, "Genetic simulated annealing for scheduling data-dependent tasks in heterogeneous environments," *5th Heterogeneous Computing Workshop (HCW '96),* Apr. 1996, pp. 98-117.

[14] H. J. Siegel, J. K. Antonio, R. C. Metzger, M. Tan, and Y. A. Li, "Heterogeneous computing," in *Parallel and Distributed Computing Handbook,* A. Y. Zomaya, ed., McGraw-Hill, New York, NY, 1996, pp. 725-761.

[15] H. J. Siegel, H. G. Dietz, and J. K. Antonio, "Software support for heterogeneous computing," in *The Computer Science and Engineering Handbook,* A. B. Tucker, Jr., ed., CRC Press, Boca Raton, FL, 1997, pp. 1886-1909.

[16] H. Singh and A. Youssef, "Mapping and scheduling heterogeneous task graphs using genetic algorithms," *5th Heterogeneous Computing Workshop (HCW '96),* Apr. 1996, pp. 86-97.

[17] 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.

[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," *J. of Parallel and Distributed Computing,* Special Issue on Parallel Evolutionary Computing, accepted and scheduled to appear.

Biographies

Muthucumaru Maheswaran is a PhD candidate in the School of Electrical and Computer Engineering at Purdue University, West Lafayette, Indiana, USA. His research interests include task matching and scheduling in heterogeneous computing environments, parallel languages, data parallel algorithms, distributed computing, scientific computation, and world wide web systems. He has authored or coauthored two journal papers, seven conference papers, two book chapters, and one technical report.

Mr. Maheswaran received a BScEng degree in electrical engineering from the University of Peradeniya, Sri Lanka, in 1990 and a MSEE degree from Purdue University in 1994. Mr. Maheswaran is a member of the Eta Kappu Nu honorary society.

Howard Jay Siegel is a Professor in the School of Electrical and Computer Engineering at Purdue University. He is a Fellow of the IEEE (1990) and a Fellow of the ACM (1998). He received BS degrees in both electrical engineering and management (1972) from MIT, and the MA (1974), MSE (1974), and PhD degrees (1977) 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* (second edition 1990). He was a Coeditor-in-Chief of the *Journal of Parallel and Distributed Computing* (1989-1991), and was on the Editorial Boards of the *IEEE Transactions on Parallel and Distributed Systems* (1993-1996) and the *IEEE Transactions on Computers* (1993-1996). 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.

Prof. Siegel's heterogeneous computing research includes modeling, mapping heuristics, and minimization of inter-machine communication. He is an Investigator on the Management System for Heterogeneous Networks (MSHN) project, supported by the DARPA/ITO Quorum program to create a management system for a heterogeneous network of machines. He is the Principal Investigator of a joint ONR-DARPA/ISO grant to design efficient methodologies for communication in the heterogeneous environment of the Battlefield Awareness and Data Dissemination (BADD) program.

Prof. Siegel's other research interests include parallel algorithms, interconnection networks, and the PASM reconfigurable parallel machine. His algorithm work involves minimizing execution time by exploiting architectural features of parallel machines. Topological properties and fault tolerance are the focus of his research on interconnection networks for parallel machines. He is investigating the utility of the dynamic reconfigurability and mixed-mode parallelism supported by the PASM design ideas and the small-scale prototype.

Dynamic, Competitive Scheduling of Multiple DAGs in a Distributed Heterogeneous Environment

Michael Iverson and Füsun Özgüner

The Department of Electrical Engineering
The Ohio State University
Columbus, OH 43210
{iverson,ozguner}@ee.eng.ohio-state.edu

Abstract

With the advent of large scale heterogeneous environments, there is a need for matching and scheduling algorithms which can allow multiple DAG-structured applications to share the computational resources of the network. This paper presents a matching and scheduling framework where multiple applications compete for the computational resources on the network. In this environment, each application makes its own scheduling decisions. Thus, no centralized scheduling resource is required. Applications do not need direct knowledge of the other applications. The only knowledge of other applications arrives indirectly through load estimates (like queue lengths). This paper also presents algorithms for each portion of this scheduling framework. One of these algorithms is modification of a static scheduling algorithm, the DLS algorithm, first presented by Sih and Lee [1]. Other algorithms attempt to predict the future task arrivals by modeling the task arrivals as Poisson random processes. A series of simulations are presented to examine the performance of these algorithms in this environment. These simulations also compare the performance of this environment to a more conventional, single user environment.

Keywords: Matching and Scheduling, DAG, Multiuser, Poisson Random Process, List Scheduling.

1 Introduction

Heterogeneous computing has a number of distinct advantages [2, 3, 4], centering around the ability to utilize the features of different machine architectures. A central theme of heterogeneous computing is the ability to construct a single computational entity from a network of heterogeneous machines. As advanced networking technologies become available, the practical size of these heterogeneous environments is growing to a point where it is possible to create a single computational resource from a set of high performance computers distributed across the globe. In such a system, multiple users will be able to simultaneously utilize the computational resources of this network to execute a variety of large parallel applications. The primary challenge of using such a computing environment is to obtain a near-optimal solution to the matching and scheduling problem. To accomplish this task, there are several unique characteristics of this environment which must be considered: the dynamic nature of the machine and network loads, the size of the network, and the need for multiple users to fairly compete for the computational resources.

Given these issues, this paper presents a framework for executing multiple applications in this heterogeneous environment. These applications have a directed, acyclic graph (DAG) structure. In this framework, each application is responsible for scheduling its own tasks. Thus there is no centralized scheduling authority. This paper also presents a series of algorithms to operate within the framework. One of these algorithms is based upon a static matching and scheduling algorithm, called the DLS algorithm, first presented by Sih and Lee [1]. These algorithms attempt to predict the future loads of the machines, by modeling task arrivals as a Poisson random process. A series of simulations are presented to demonstrate these methods. In the next section, relevant background material is examined, and, in Section 3, an overview of the execution environment is presented. Section 4 gives a detailed presentation of the algorithms used within this environment. The results of a simulation study are discussed in Section 5, and conclusions from these results are offered in Section 6.

2 Background

The majority of the interest in DAG scheduling has been restricted to static environments. One simple and efficient type of heterogeneous scheduling method is the level-based algorithm, which schedules a task based upon that task's depth in the DAG. Some methods which fall into

0-8186-8365-1/98 $10.00

this category include those presented by Leangsuksun and Potter [5], who study a variety of simple, heterogeneous scheduling heuristics, and a method called the LMT algorithm, which assigns all of the tasks at a particular depth in the DAG at one time [6]. Kim and Browne [7] present a static scheduling technique called linear clustering, where tasks are clustered into chains of tasks, and these clusters are mapped onto the physical machines. This heterogeneous scheduling method has limited application to the proposed problem, since it assumes that the individual processors perform uniformly for all code types (i.e. the performance of a task on each heterogeneous processor varies only by a scale factor). A more complex static method, called the MH algorithm, is presented by El-Rewini and Lewis [8]. Again, this method is limited in that it uses the same simple model of a heterogeneous system as in [7]. Another method is the Cluster-M technique introduced by Eshaghian and Wu [9], which clusters tasks together based upon architectural compatibility. Of interest to this research is the method presented by Sih and Lee [1]. This static technique, called Dynamic Level Scheduling, schedules tasks by using a series of changing priorities. The DLS algorithm has been shown by Sih and Lee to be superior to other static DAG scheduling algorithms for heterogeneous systems, and will be discussed in more complete detail below.

While most of the DAG scheduling algorithms are static, there are a few algorithms that examine the problem of scheduling DAGs in a dynamic environment. For heterogeneous systems, Haddad [10, 11] presents a dynamic load balancing scheme for DAGs. This scheme differs from a conventional scheduling algorithm, in that it does not look at the exact structure of a given application. It instead uses a number of metrics that characterize the tasks and the task graph, to balance the computational load. For homogeneous MIMD systems, Rost et al. [12] present a scheduling model called agency scheduling. This model supports decentralized scheduling decisions by giving a set of distributed scheduling tasks control over a local set of processors. Neither of these methods explicitly consider the problem of scheduling multiple applications in a distributed environment. This paper presents a new dynamic scheduling method, which is designed for some of the unique features of this environment. In the next section, these features are examined in detail.

3 Definitions

As stated above, in this environment, multiple applications are competing for the computational resources of the network. Each application is represented by a set of communicating tasks. These tasks are organized using a DAG, $G = (V, E)$, where the set of vertices $V = \{v_1, v_2, \ldots, v_n\}$ represents the set of tasks to be executed, and the set of weighted, directed edges E represents communication between tasks. Thus, $e_{ij} = (v_i, v_j) \in E$ indicates communication from task v_i to v_j, and $|e_{ij}|$ represents the volume of data sent between these tasks. The execution environment consists of a set of heterogeneous machines, which can be represented by the set $M = \{m_1, m_2, \ldots, m_q\}$. The computation cost function, $C : V \times M \rightarrow \Re$, represents the execution cost of each task on each available machine. Thus, the cost of executing task v_i on machine m_j will be denoted by $C(v_i, m_j)$. If a particular task cannot be executed on a given machine, the function will evaluate to infinity.

In this environment, each machine is limited to executing one task at a time. There are several compelling reasons to adopt this organization.

1. In DAG scheduling, when a task is scheduled, it is desirable to know the time at which the task will complete execution. In a system where multiple tasks can simultaneously execute on a machine, the completion time of a given task depends upon the other tasks which are also executing on the machine. Since tasks from other applications may arrive at any time, it is not possible to determine the completion time *a priori*.

2. When system loads are able to change after a task has been assigned, task migration is necessary to balance machine loads. Task migration can be difficult in a heterogeneous environment, since there is no guarantee that there is another machine available to execute a given task.

To ensure that each machine can execute only one task at a time, each machine will have a FIFO queue. Tasks wanting to execute on a machine must wait in the machine's queue until the machine is available. However, it is possible for a task to receive data from predecessors while another task is executing, and likewise send data to successor tasks. Since such communication is not processor intensive, it should have little effect upon the execution of the running task.

The above organization does not include any resources for making scheduling decisions. Therefore, there will be another set machines in the network: scheduling machines. Each application will execute a scheduling task (on a scheduling machine), which is responsible for making all of the scheduling decisions for that application. Ideally, these scheduling machines would be general purpose workstations (possibly even the user's workstation). A conceptual model of this environment is shown in Figure 1.

Since each application is self-scheduling, an application only has direct knowledge of its own tasks. The

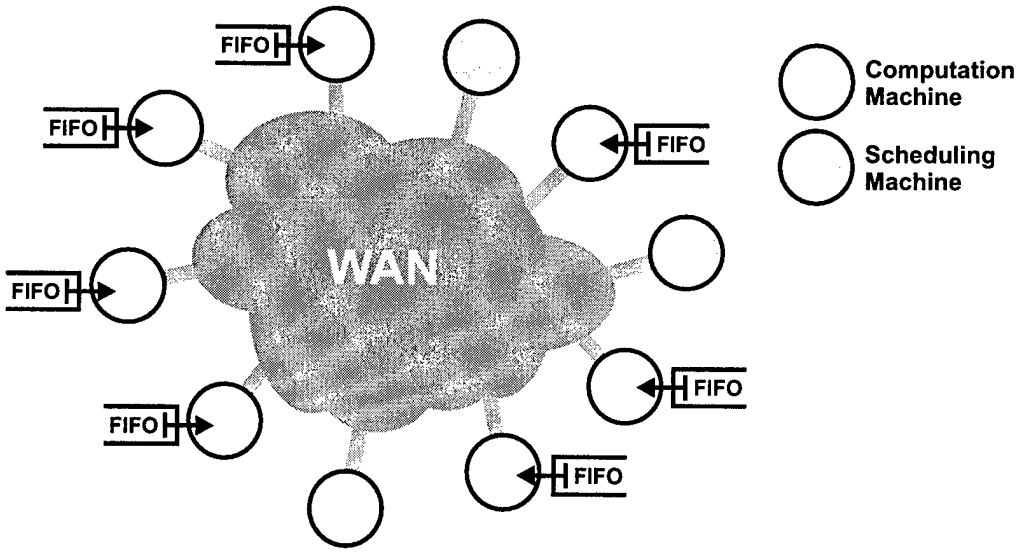


Figure 1: Conceptual model of execution environment.

only information it has about other applications is in the form of machine and network load estimates. The scheduling machines are responsible for maintaining this dynamic state information. The load of a particular machine m_j is characterized by the queue length Q_{m_j}, the task arrival rate λ_{m_j}, and the average size l_{m_j} of the arriving tasks, in terms of execution time. Although beyond the scope of this paper, techniques will be needed to acquire and estimate this loading data. One such method is presented by Hou and Shin [13], who use Bayesian decision theory to predict queue lengths using observations which may be out-of-date (due to the network latency).

The communication costs are represented using the function $D : E \times M \times M \rightarrow \Re$. Thus, the cost of sending the message from task v_r to task v_s (represented by edge e_{rs}) from machine m_i to machine m_j will be $D(e_{rs}, m_i, m_j)$. Communication between tasks complicates the matching and scheduling process in this environment. In order for a task to begin execution, two conditions must be satisfied: (1) data from previous tasks must be available, and (2) the task must be at the head of the queue. Ideally, both of these conditions would be satisfied at the same time. Achieving this goal is not likely, however. Thus, there are two possible scenarios:

1. *Data is available before the task reaches the head of the queue.* In this case, the task will have to wait to begin execution (the task should have been placed in the queue earlier). This queuing delay can potentially increase the completion time of the application. Other applications are unaffected.
2. *Task reaches the head of the queue before data is available.* In this case, the task was placed in the queue too early, and the task will have to wait for the data. As it waits, the machine will be idle, and the other tasks in the queue will have their progress blocked.

There is a significant problem with having a task block the queue. When a task is allowed to block the queue for an arbitrary amount of time, it is no longer possible to accurately determine the completion time of a task when it is placed in the queue. However, there is a means of limiting the effects of this behavior. It is possible to require the scheduling algorithm to estimate the amount of blocking time that a task is likely to incur, based upon the queue length and the time at which the data is expected to arrive. This estimate can then be explicitly included as part of the queue length estimates, thus minimizing the effects of any blocking time. Even with this modification, blocking time still represents a waste of machine resources.

Some additional definitions are needed by the matching and scheduling algorithms defined below. The *static level* of task v_i, $L_{\text{static}}(v_i)$, is defined to be the largest sum of the median execution times of the tasks along any directed path from task v_i to an end node of the graph. Since the environment is heterogeneous, the median execution time of a task v_i, denoted $\bar{C}(v_i)$, is used to characterize the overall behavior of that task. If the actual median is infinite, the median value will be replaced with the largest finite execution time. In order to differentiate between different

machines, the term

$$\Delta(v_i, m_j) = \bar{C}(v_i) - C(v_i, m_j) \quad (1)$$

is defined to indicate the speed that machine m_j executes task v_i, relative to the median value. A large value of Δ implies a fast machine. Given this set of definitions, the details of matching and scheduling in this environment will be presented in the next section.

4 Matching and Scheduling Algorithm

In this section, we will define a framework for solving the matching and scheduling problem in the environment defined above, and then present specific algorithms for each portion of the framework. The framework will be defined as a series of sets containing the tasks of the application, and a series of heuristics to move tasks between these sets. The first set will be the set of unscheduled tasks, denoted U. As its name implies, the set U will contain tasks which have not been scheduled to execute on a particular machine. At the beginning of the algorithm, all tasks are members of the set U.

In list scheduling methods, no task can be scheduled until all of its predecessors have been scheduled. Using this ordering, we can define the set of ready tasks R to be the set of tasks whose predecessors have been scheduled, and thus can be assigned to a machine. From the definition above, it is clear that $R \subset U$. Below, a heuristic will be defined that will choose the best-suited task (and the machine to execute it on) from this set of ready tasks R. This will be called the *matching decision policy*. Since much of the information used by the matching decision policy is dynamic in nature, it is important to consider the time at which the matching and scheduling decisions are made. This time will be determined by a heuristic called the *scheduling time policy*. The combination of these two policies will move tasks from set R to a new set P: the set of pending tasks. The set of pending tasks P is the set of tasks which have been assigned to execute on a particular machine, but have yet to be placed in the appropriate machine's queue. This set is partitioned into q subsets $P_1, P_2, \ldots, P_q$. Each subset P_j contains the tasks which have been assigned to machine m_j. When a task has been placed in the queue, it is moved into a final set S, the set of scheduled tasks. The time at which a task is placed in the appropriate queue is determined by a heuristic called the *queuing time policy*. Figure 2 illustrates the sets defined above, and the flow of tasks from one set to another.

Given these definitions, the process of making a decision within this framework can be broken down into three steps: determining how to make a matching and scheduling decision (matching decision policy), determining when to make a matching and scheduling decision (scheduling time policy), and determining when to place a scheduled task in the chosen queue (queuing time policy). Figure 3 shows the algorithm which governs how each of these steps is executed. Every T time units, this algorithm will use the policies discussed above to move tasks between the various sets. The value of T is called the *examination interval*. In the subsections below, algorithms will be presented for each of these policies. The matching decision policy will be a dynamic adaptation of the DLS algorithm presented by Sih and Lee [1]. The scheduling time and queuing time policies will use probabilistic methods to determine the appropriate time to schedule or queue tasks.

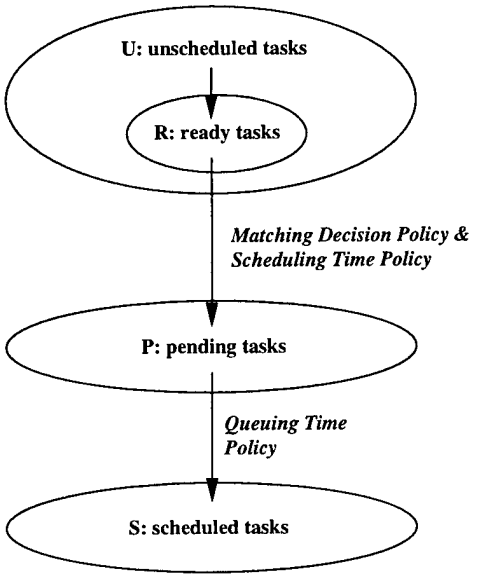


Figure 2: Dynamic DAG Scheduling Framework.

4.1 Matching Decision Policy

As mentioned above, the matching decision algorithm is based upon a dynamic adaptation of the DLS algorithm. Therefore, a brief overview of the relevant portions of the original static version of the DLS algorithm will be presented first.

4.1.1 Static DLS Algorithm

In [1], Sih and Lee present the DLS algorithm: a compile time, static algorithm for scheduling a DAG onto a set of heterogeneous machines. This algorithm, can be categorized as a list scheduling algorithm, where the tasks are assigned to the machines in topological order. As with nearly all list scheduling algorithms, the DLS algorithm operates by assigning a priority, called a level, to each task in that graph. This priority is then used to choose among the set of tasks which are ready to be scheduled at that time. The the DLS algorithm differs from previous algorithms in that the level of a task depends upon the tasks which have already

While $U \neq \phi, do$:
begin
  While the scheduling time policy indicates that tasks should be scheduled:
  begin
    Use matching decision policy to choose a task-machine pair (choose a task to move from set R to set P).
    Check precedence constraints of tasks (find any tasks which can be moved into set R).
  end
  Use queuing time policy to examine pending tasks (move tasks ready to be queued from set P to set S).
  Wait T time units.
end

Figure 3: Generic matching and scheduling algorithm for the framework

been assigned. This concept is called the *dynamic level* of a task, and is defined to be

$$L_{\text{dynamic}}(v_i, m_j) = L_{\text{static}}(v_i) - \max[t_{\text{data}}(v_i, m_j), t_{\text{free}}(m_j)] + \Delta(v_i, m_j), \quad (2)$$

where $t_{\text{free}}(m_j)$ denotes the time at which machine m_j will be idle, and $t_{\text{data}}(v_i, m_j)$ denotes the time when task v_i's data will be available on machine m_j. The first term of the expression is the static level of the task. This term indicates the path length to the end of the graph. Since a long path is more likely to be the critical path of the graph, a large value of L_{static} will increase the scheduling priority. The second term indicates when the task can begin on the machine, based upon the time when the data is available on the machine, and the time at which the machine is able to execute the task. An earlier starting time will imply a higher priority. The third term indicates how fast the machine m_j will execute the task, relative to the other machines in the system. With this dynamic level expression, making a matching decision is equivalent to finding the ready task and machine which maximize the above expression. Another advantage to this approach is that both the task and processor are chosen at the same time. Sih and Lee show that this policy is superior to independently selecting either the task or the processor.

4.1.2 Dynamic Matching Decision Policy

As mentioned above, the purpose of the matching decision policy, given a set R of tasks ready to be scheduled, is to find the "best" task-machine pair from the set of ready tasks R. This decision policy can be constructed using a modified version of the dynamic level equation presented above.

In order to operate in a dynamic environment, the terms of the dynamic level expression, shown in equation 2, will require modification. Examining the individual terms of this expression, it can be seen that the first and last terms do not depend upon any data which is dynamic in nature. However, the middle term, $\max[t_{\text{data}}(v_i, m_j), t_{\text{free}}(m_j)]$, which denotes when the task will be able to begin execution on the specified machine, does depend upon dynamic information. The two arguments of the max operator represent the two independent events which need to be satisfied in order for a task to begin executing on a machine. For the environment discussed above, the two terms needed are the time at which the data is available on the chosen machine and the time at which the machine is idle, and thus able to execute the task. For this environment, this second quantity is defined to be

$$t_{\text{free}}(m_j) = t + Q_{m_j} + \sum_{\forall v_k \in P_j} C(v_k, m_j), \quad (3)$$

where t is the current time. The second term in the expression is the execution time of the tasks in machine m_j's queue at this time, and the last term represents the total execution time of the tasks in set P_j (i.e. waiting to be placed in machine m_j's queue). With this modification, this portion of the DLS algorithm can be used as the matching decision policy. The next issue of interest is the process of determining when to make a scheduling decision.

4.2 Scheduling Time Policy

As shown by Sih and Lee [1], it is better to choose the task and machine simultaneously, rather than choose either independently of the other. Thus, the purpose of the scheduling time policy is to determine when it is appropriate to make matching and scheduling decisions, not when to schedule a particular task. There is a tradeoff inherent in deciding when to schedule a task. Since the loading information used by matching decision policy will change with time, if a task is scheduled early, the information used to make the decision could be too inaccurate to be of use. However, if the algorithm waits too long to schedule the task, it is possible that a desired machine will be unavailable (due to a long queue) and the task will be forced to execute on a suboptimal machine.

In this paper, the following heuristic is proposed to determine when scheduling decisions should be made. A

scheduling decision will be made if the probability that any ready task will experience queuing delay on its most desired machines exceeds a predefined threshold value. To simplify the derivation of this probability, consider a single machine m_j. Using the above information, we can derive an expression defining the probability that a task will experience queuing delay on this machine if it is not assigned to the machine now (if it is not assigned now, it would have to wait a minimum of T time units until the scheduler examines the situation again). In other words, we would like to find the probability that a sufficient number of tasks from other applications will arrive (and be placed in the queue) in the next T time units such that the task will be forced to experience queuing delay.

To determine an expression for this probability, it is necessary to define a "critical number" of tasks—the minimum number of average-sized tasks which would have to arrive within the examination interval T, such that any task assigned at time $(T + t)$ would experience queuing delay. Assuming that the arriving tasks are of average size l_{m_j}, the critical number will lie within the interval

$$\left(\left\lceil \frac{t_{\text{slack}}}{l_{m_j}} \right\rceil, \left\lceil \frac{T + t_{\text{slack}}}{l_{m_j}} \right\rceil \right), \tag{4}$$

where t_{slack} denotes the difference between the time at which the data will be available and the time at which the queue will be empty. This term, called the slack time, is defined to be

$$t_{\text{slack}} = \begin{cases} t_{\text{data}} - (Q_{m_j} + t) & \text{if } t_{\text{data}} > (Q_{m_j} + t) \\ 0 & \text{otherwise} \end{cases}. \tag{5}$$

While it is more likely that the critical number will be closer to the upper bound of this interval, it is better to use the lower bound, due to the behavior of the method in the boundary condition (which will be explained below). Therefore, in this paper, the critical number of tasks z will be defined to be

$$z = \left\lceil \frac{t_{\text{slack}}}{l_{m_j}} \right\rceil. \tag{6}$$

Now, assuming that the task arrivals can be reasonably approximated using a Poisson random variable, the probability that the number of arrivals k within the interval T will be greater than or equal to z can be defined to be

$$\begin{aligned} P_{m_j}[k \geq z] &= 1 - P[k < z] \\ &= 1 - \sum_{k=0}^{z-1} \frac{(\lambda_{m_j} T)^k}{k!} e^{-\lambda_{m_j} T}. \end{aligned} \tag{7}$$

The reason for choosing to use the lower boundary in equation 5 is due to the case when t_{slack} is equal to zero. By defining the quantity z in this manner, equation 7 will evaluate to one.

This expression only considers the probability of experiencing queuing delay on a single machine. In reality, it is likely that there will be more than one machine available to execute the task. It is therefore desirable to expand the above expression to consider the possibility of a task experiencing queuing delay on more than one of its best performing machines. Thus, the scheduling time policy will be defined to schedule tasks if there is a ready task which has a probability of experiencing queuing delay on its *three* fastest machines that is greater than a predefined threshold β. So, to determine if a task is "in danger" of not getting a desired machine, the algorithm finds the machines m_i, m_j, and m_k on which the task executes the fastest. Then, the algorithm computes the critical number z for each of the above machines, (denoted z_{m_i}, z_{m_j}, and z_{m_k}) and the probability of experiencing queuing delay on each of these machines, using equation 7 above. With these values, the overall probability of not getting one of these three machines is

$$\begin{aligned} P_{\text{queue}} = (P_{m_i}[k \geq z_{m_i}]) \cdot \\ (P_{m_j}[k \geq z_{m_j}]) \cdot (P_{m_k}[k \geq z_{m_k}]). \end{aligned} \tag{8}$$

The choice of three is clearly a heuristic. The advantage of using a fixed number, like three, is primarily computational: the algorithm will be more efficient, which is important in a dynamic environment. If there are fewer than three machines on which a task can execute, the probability will be computed using this lesser number of machines. The choice of the value β is also a heuristic. A series of experiments are performed to evaluate the choice of a value for the parameter β. These results will be presented in Section 5.

4.3 Queuing Time Policy

As discussed above, when placing a task in a queue, there are two possible scenarios: the task is placed in the queue too early and experiences blocking delay, or the task is placed in the queue too late, and experiences queuing delay. The ideal time to place a task in the queue lies between these two extremes. The formulation of the queuing time policy will use a probabilistic formulation similar to the scheduling time policy described above. To construct a heuristic to attempt to place a task in the queue at the appropriate time, a pair of cost functions will be defined. The first cost function, C_{block}, will indicate the blocking cost the task will experience if it is placed in the queue now. The second cost function, C_{queue}, will indicate the probable queuing cost the task will experience if the queuing algorithm waits another T time units to assign the task. The goal of the queuing time policy is to place each task in

the queue in a manner which minimizes both the queuing and blocking cost.

The blocking cost function is defined to be the amount of blocking time the machine will experience. Given the time at which the data is available, t_{data}, and the queue length at time t, Q_{m_j}, the blocking cost is

$$C_{\text{block}} = t_{\text{data}} - (Q_{m_j} - t) = t_{\text{slack}}. \tag{9}$$

This value is equal to the t_{slack} term defined above.

The definition of the queuing cost function is more elaborate. While the blocking cost is a deterministic quantity (provided that t_{data} and Q_{m_j} are accurate) the queuing cost is stochastic, in that it will depend upon the probability of future arrivals in the queue. Like the probable queuing cost definition from the scheduling time policy, this cost function will depend upon a critical number z. This number represents the minimum number of averaged-sized tasks which have to arrive in order for the task to experience queuing time. As before, z is defined to be

$$z = \left\lceil \frac{t_{\text{slack}}}{l_{m_j}} \right\rceil. \tag{10}$$

However, unlike the scheduling time case, we are not interested in the probability of experiencing queuing delay, but the probable amount of queuing delay. To determine this quantity, first consider a simpler task of finding how much queuing time a task would experience if exactly k tasks were to arrive in the time interval T. In this case, the queuing cost is

$$t_{\text{queue}}(k) = \begin{cases} kl_{m_j} - (t_{\text{slack}} + T) & \text{if } kl_{m_j} > (t_{\text{slack}} + T) \\ 0 & \text{otherwise} \end{cases}. \tag{11}$$

This expression can then be used to determine the probable queuing cost, by multiplying by the discrete probability of exactly k arrivals, and summing over all possible values of k:

$$\begin{aligned} C_{\text{queue}} &= \sum_{k=0}^{\infty} t_{\text{queue}}(k) P[n = k] \\ &= \sum_{k=z}^{\infty} (kl_{m_j} - (t_{\text{slack}} + T)) P[n = k] \\ &= \sum_{k=z}^{\infty} (kl_{m_j} - (t_{\text{slack}} + T)) \frac{(\lambda_{m_j} T)^k}{k!} e^{-\lambda_{m_j} T}. \end{aligned} \tag{12}$$

Since it is undesirable to compute an infinite summation, the expression can be rearranged to become

$$\begin{aligned} C_{\text{queue}} &= (kl_{m_j} - (t_{\text{slack}} + T)) \\ &\quad - \sum_{k=0}^{z-1} (kl_{m_j} - (t_{\text{slack}} + T)) \frac{(\lambda_{m_j} T)^k}{k!} e^{-\lambda_{m_j} T}. \end{aligned} \tag{13}$$

Now, given these two cost functions C_{queue} and C_{block}, the queuing time policy will place a task in its queue when the blocking cost is greater than the queuing cost. However, as mentioned previously, the queuing cost and blocking cost may not have the same effect upon the system. Therefore, an additional parameter γ will be introduced, in order to modify the relative weight of the two cost functions. Thus, to decide whether or not to queue a particular task, the algorithm will compute the quantity $C_{\text{queue}} - \gamma C_{\text{block}}$. Every T time units, the algorithm will compute this quantity for each of these pending tasks. If, for a given task, the quantity is negative, the machine will not place the task in the queue at this time. Otherwise, if the quantity is positive, the task is placed in the queue. As was the case for the scheduling time policy, the choice of the parameter γ will be examined experimentally in Sections 5.

5 Results

To evaluate these methods, a series of simulations were performed, using a custom, event-based simulator. These simulations examine the effects of the parameters β and γ, and compare the performance of the algorithms to the static DLS algorithm, which uses a more conventional environment where each user has exclusive use of the machines for a period of time. A representative set of results are shown in the figures 4 and 5. In this case, eight 64-task applications are scheduled on a 16 machine heterogeneous system. The execution times, task graphs, and computation costs were randomly generated, and it is possible for a task not to be able to execute on every machine. The graphs were generated such that they were capable of using about 8 machines in parallel.

The starting time of each of the applications was chosen over a random interval between 0 and 200 time units, to limit any artificial effects from starting all the applications at the same time. The examination interval T was chosen to be 1 time unit. The results shown in the graphs are an average of 5 separate simulations, to minimize any effects caused by specific graph structures. For each simulation, the schedule length of the applications was recorded, and the efficiency of the computation was determined. The efficiency measures the amount of blocking time in the system, relative to the amount of computation. For example, an efficiency of 0.75 would indicate that 75% of the total CPU time used was useful computation, and the remaining 25% was blocking time.

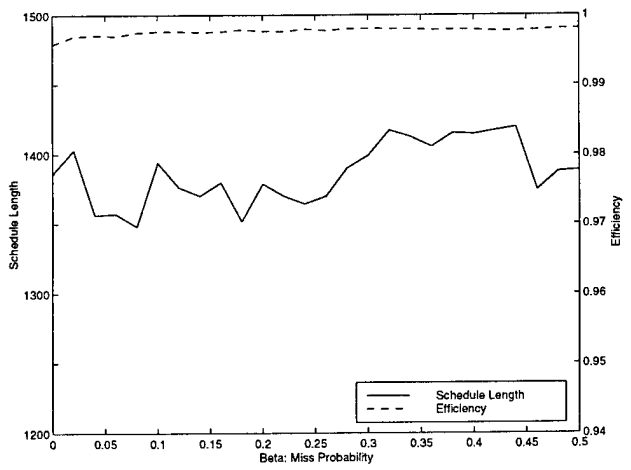


Figure 4: Average Schedule Length and Efficiency vs. Miss Probability (β).

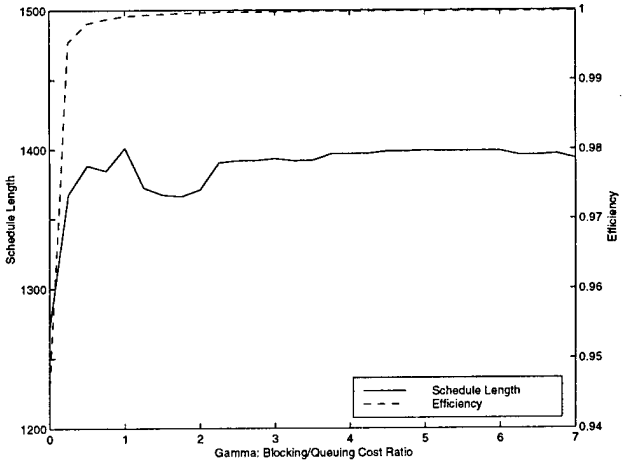


Figure 5: Average Schedule Length and Efficiency vs. Blocking/Queuing Cost Ratio (γ).

Figure 4 shows the schedule length and efficiency versus the parameter β (probability of missing the three fastest resources). These values are averaged over all of the values of γ. Overall, the parameter β has a relatively small effect upon the schedule length, which is likely due to the good quality of the loading information presented to the algorithm. However, for larger values of β, the schedule length tends to be long, since there is a higher probability that a task will have to execute on a suboptimal processor. Likewise, for very small values of β, the schedule length is also long, due to the fact that the scheduling decision was made with less accurate loading information. The efficiency is more or less constant with respect to β, since this parameter has no real effect upon the blocking time of the system.

Figure 5 shows the other case: the schedule length and efficiency versus the parameter γ (queuing/blocking cost ratio), averaged over all of the values of β. These results show the negative effect of blocking time upon the system. Using small values of γ, it is possible to get lower schedule lengths by placing the tasks in the queue early, and incurring blocking time. However, this has an adverse effect upon the efficiency, and, for slightly larger values of γ, tends to have a negative effect upon the schedule length (since processor resources are wasted). For simulations with more applications, the blocking time has an even greater impact upon the schedule length. For larger values of γ, there is a distinct minimum in the graph, representing the best trade-off between blocking time and queuing time (for this simulation). At this point, the best schedule lengths can be obtained without incurring large amounts of blocking time.

It is also desirable to compare the performance of this method to a static scheduling paradigm, where each application has exclusive use of all (or a portion) of the machines in the network. To accomplish this, each task graph was also scheduled using the static DLS algorithm. Using this data, the speedup was computed to be the total time needed to execute all eight applications sequentially, divided by the total time needed to execute all eight applications in the dynamic environment. Given that each application used in this experiment is, on average, capable of utilizing eight machines, the closest comparison of these two environments is for an eight machine system. In this case, the maximum speedup over all parameter values was found to be 1.21, a 21% improvement over the static environment. The speedup in the 16 machine system is considerably higher, since, on average, half of the machines will remain idle in the static environment (where all 16 machines are dedicated to the application). In this case, the maximum speedup was found to be 2.36. As expected, the dynamic scheduling method outperforms the static method, since it allows other applications to use computational resources which would be left idle in a static scheduling paradigm.

6 Conclusions

In this paper, a means of competitively scheduling multiple DAG-structured applications in a distributed heterogeneous environment is presented. Initial results show that this type of scheduling is practical, and confirm the assumptions made about the behavior of the scheduling environment. Currently, the authors are working on implementing these algorithms on an actual distributed network, to better evaluate and refine the techniques presented here.

References

[1] 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, pp. 175–187, Feb. 1993.

[2] J. B. Andrews and C. D. Polychronopoulos, "An analytical approach to performance/cost modeling of parallel computers," *J. Parallel Distributed Computing*, vol. 12, pp. 345–356, 1991.

[3] R. F. Freund and H. J. Siegel, "Heterogeneous processing," *IEEE Computer*, vol. 26, pp. 13–17, June 1993.

[4] A. Khokhar, V. Prasanna, M. Shaaban, and C.-L. Wang, "Heterogeneous supercomputing: Problems and issues," in *Proc. of the 1992 Workshop on Heterogeneous Processing*, pp. 3–12, IEEE Computer Society Press, Mar. 1992.

[5] C. Leangsuksun and J. Potter, "Designs and experiments on heterogeneous mapping heuristics," in *Proc. of the 1994 Heterogeneous Computing Workshop*, (Cancún, Mexico), pp. 17–22, IEEE Computer Society Press, Apr. 1994.

[6] M. A. Iverson, F. Özgüner, and G. Follen, "Parallelizing existing applications in a distributed heterogeneous environment," in *Proc. of the 1995 Heterogeneous Computing Workshop*, (Santa Barbara, CA), pp. 93–100, IEEE Computer Society Press, Apr. 1995.

[7] S. J. Kim and J. C. Browne, "A general approach to mapping parallel computations upon multiprocessor architectures," in *the 1988 Inter. Conf. on Parallel Processing*, vol. 3, pp. 1–8, CRC Press, 1988.

[8] H. El-Rewini and T. G. Lewis, "Scheduling parallel program tasks onto arbitrary target machines," *J. Parallel Distributed Computing*, vol. 9, pp. 138–153, 1990.

[9] M. M. Eshaghian and Y.-C. Wu, "Mapping and resource estimation in network heterogeneous computing," in *Heterogeneous Computing* (M. M. Eshaghian, ed.), pp. 197–223, Artech House, 1996.

[10] E. Haddad, "Load distribution optimization in heterogeneous multiple processor systems," in *Proc. of the 1993 Workshop on Heterogeneous Processing*, pp. 42–47, IEEE Computer Society Press, 1993.

[11] E. Haddad, "Dynamic optimization of load distribution in heterogeneous systems," in *Proc. of the 1994 Heterogeneous Computing Workshop*, (Cancún, Mexico), pp. 29–34, IEEE Computer Society Press, Apr. 1994.

[12] J. Rost, F.-J. Markus, and L. Yan-Hua, "Agency scheduling: A model for dynamic task scheduling," in *Proc. of the 1st Inter. EURO-PAR Conf.*, (Stockholm), pp. 93–100, Springer-Verlag: Lecture Notes in Computer Science, Aug. 1995.

[13] C.-J. Hou and K. G. Shin, "Load sharing with consideration of future task arrivals in heterogeneous distributed real-time systems," *IEEE Trans. Computers*, vol. 43, pp. 1076–90, Sept. 1994.

Michael Iverson received the B.S. degree in Computer Engineering at Michigan State University in 1992, and the M.S. degree in Electrical Engineering at The Ohio State University in 1994. He is currently researching topics in heterogeneous distributed computing for his Ph.D. dissertation. In addition to his research, Mr. Iverson is building Internet video conferencing systems and wireless networking systems for University Technology Services at Ohio State.

Füsun Özgüner received the M.S. degree in electrical engineering from the Istanbul Technical University in 1972, and the Ph.D. degree in electrical engineering from the University of Illinois, Urbana-Champaign, in 1975. She worked at the I.B.M. T.J. Watson Research Center with the Design Automation group for one year and joined the faculty at the Department of Electrical Engineering, Istanbul Technical University in 1976. Since January 1981 she has been with The Ohio State University, where she is presently a Professor of Electrical Engineering. Her current research interests are parallel and fault-tolerant architectures, heterogeneous computing, reconfiguration and communication in parallel architectures, real-time parallel computing and parallel algorithm design. She has served as an associate editor of the IEEE Transactions on Computers and is the Program Vice-Chair for Fault Tolerance and Reliability for the 1998 International Conference on Parallel Processing.

The Relative Performance of Various Mapping Algorithms is Independent of Sizable Variances in Run-time Predictions *

Robert Armstrong
Debra Hensgen
Taylor Kidd

Computer Science Department
Naval Postgraduate School
Monterey, CA 93940

Abstract

In this paper we study the performance of four mapping algorithms. The four algorithms include two naive ones: Opportunistic Load Balancing (OLB), and Limited Best Assignment (LBA), and two intelligent greedy algorithms: an O(nm) greedy algorithm, and an $O(n^2m)$ greedy algorithm. All of these algorithms, except OLB, use expected run-times to assign jobs to machines. As expected run-times are rarely deterministic in modern networked and server based systems, we first use experimentation to determine some plausible run-time distributions. Using these distributions, we next execute simulations to determine how the mapping algorithms perform. Performance comparisons show that the greedy algorithms produce schedules that, when executed, perform better than naive algorithms, even though the exact run-times are not available to the schedulers. We conclude that the use of intelligent mapping algorithms is beneficial, even when the expected time for completion of a job is not deterministic.

1 Introduction

This paper describes the experiments and simulations that we executed to determine the relative performance of certain mapping algorithms in different heterogeneous environments. In this paper we assume that all jobs are independent of one another. That is, they do not communicate or synchronize with one another. This type of architecture is common in today's LAN-based distributed server environment.

Our goal was to determine whether using intelligent mapping algorithms would be beneficial, even if the jobs did not run for exactly the amount of time expected. Intelligent mapping algorithms utilize the expected run-times of each job on each different machine to attempt to minimize some scalar performance metric. For our experiments, this metric is the time at which the last job completes. In particular, we were concerned about whether it would still be beneficial to use intelligent mapping if one or several jobs run for a substantially different amount of time than expected, but are still accurately characterized statistically. Because determining a perfect mapping is an NP-complete problem, we examined the performance of several different (polynomial) heuristics. The algorithms we chose are listed below.

- A naive $O(n)$ algorithm known as Opportunistic Load Balancing (OLB). This algorithm simply places each job, in order of arrival, on the next available machine.
- A simple $O(nm)$ algorithm known as Limited Best Assignment (LBA). This algorithm uses the expected run-time of each job on each machine. It assigns each job to the machine on which it has the least expected run-time, ignoring any other loads on the machines, including that produced by the jobs that it has assigned.

  This algorithm, though easily implementable in a scheduling framework that automatically assigns jobs to machines, is very similar to the algorithm used by many users who remotely start their jobs by hand at supercomputer centers without examining queue lengths.
- Two greedy algorithms, one of order $O(nm)$ and the other of order $O(n^2m)$. Both of these algorithms make use of the expected run-time of each job on each machine as well as the expected loads on each machine. These algorithms will be more fully described in Section 2.

*This research was supported by DARPA under contract number E583. Additional support was provided by the Naval Postgraduate School and the Institute for Joint Warfare Analysis.

0-8186-8365-1/98 $10.00 © 1998 IEEE

The primary reasons for our study are both that jobs rarely execute for exactly the expected run-time and often the expected run-times are not exactly known. In a system where each job has exclusive use of a machine, differences between actual and predicted run-times occur either because (1) all of the compute characteristics [10] are not known or enumerated by the designer of the program, or (2) because the time to access memory and disk is stochastic and not deterministic. Of course, in many environments, additional non-determinism is due to other jobs running on the machine or simultaneously using a shared network or a shared file server. This paper focuses on those cases where one or more of the jobs being scheduled have run-times that could differ substantially from the expected run-time. For those cases, we seek to determine whether there is still an advantage to using an algorithm that makes use of expected run-times or whether a computationally simpler algorithm that does not require estimating run-times, such as Opportunistic Load Balancing (OLB), might not yield equivalently good performance.

In the next section, we describe the two greedy algorithms that we used in our experiments and simulations. We then describe our experiments concerning the non-determinism of expected run-times and examine, using the derived distributions in simulations, the performance of the intelligent algorithms. That is, we collect run-times for various jobs on various machines, analyze their distributions, and extrapolate these distributions for use in our simulations. We conclude the paper with a short summary and comparison to related work.

2 The Greedy Algorithms

In addition to the simple OLB and LBA algorithms described in the previous section, our experiments used two greedy algorithms. We now describe those algorithms in detail.

The first algorithm is an $O(nm)$ algorithm, where n is the number of jobs and m is the number of machines, and the second algorithm is of order $O(n^2m)$. Each algorithm first estimates the expected run-time of each job on each machine, assuming that if a job cannot execute on a particular machine, the estimation will be set to some very large number. As we describe these algorithms we will consider these expected run-times as elements of a 2-dimensional, n by m matrix called A. That is, $A[i,j]$ is the expected run-time of job i on machine j.

The $O(nm)$ algorithm, which, like in the SmartNet documentation [6], we will call Fast Greedy, considers the jobs in the order requested[1]. It first determines the value $A_{1,j}$, such that $A_{1,j} \leq A_{1,k}\ \forall\ k \in \{1..m\}$. It then assigns job 1 to machine j. Following this, it adds $A_{1,j}$ to all $A_{i,j}\ \forall\ i \in \{2..n\}$. Then, for each remaining job, $p \in \{2..m\}$, it determines the value $A_{p,j}$, such that $A_{p,j} \leq A_{p,k}\ \forall\ k \in \{1..m\}$. It then assigns job p to machine j. Following this, it adds $A_{p,j}$ to all $A_{i,j}\ \forall\ i \in \{p+1..n\}$. At each step, then, it is assigning each job to its best machine, given the previous assignments. We note that the jobs are assigned in the order in which they were requested.

The $O(n^2m)$ algorithm, which again borrowing from SmartNet nomenclature we call simply Greedy, actually computes two mappings using two different sub-algorithms and then chooses the mapping that gives the smallest sum of the predicted run-times, minimized over all machines. The two sub-algorithms are similar to the first greedy algorithm above, differing only in the order in which they assign jobs to machines. We first enumerate the steps of the first sub-algorithm.

1. Initialize the set $\{RemainingJobs\}$ to contain all jobs.
2. $\forall\ i \in \{RemainingJobs\}$, find $A_{i,j} \leq A_{i,k}\ \forall\ k \in \{Machines\}$. Call such an $A_{i,j}$, A_{i,min_i}.
3. Determine p such that $A_{p,min_p} \leq A_{i,min_i}\ \forall\ i \in \{RemainingJobs\}$.
4. Remove p from $\{RemainingJobs\}$, scheduling job p on machine min_p.
5. Add A_{p,min_p} to $A_{i,min_p}\ \forall\ i \in \{RemainingJobs\}$.
6. If $\{RemainingJobs\}$ is not empty, return to step 2.

The idea behind this first sub-algorithm is that, at each step, we attempt to minimize the time at which the last job, which has been thus far scheduled, finishes.

The second sub-algorithm differs from the first sub-algorithm in that, at the third step, it finds p such that $A_{p,min_p} \geq A_{i,min_i}\ \forall\ i \in \{RemainingJobs\}$. This algorithm, then tries to minimize the worst case time at each step.

3 Effect of Non-Determinism on Algorithm Performance

We now examine the effect of non-determinism on the performance of the greedy and LBA algorithms that we described above. Our reason for studying this

[1] In describing these algorithms, we use the term *order requested* to mean the order in which the job requests have been placed prior to invocation of the algorithm. We also investigated the performance of these algorithms if jobs are first sorted before these algorithms are invoked.

is because both the LBA and the greedy algorithms use the expected run-time to produce their mappings. One of our major motivations for this work is to determine whether such intelligent algorithms are still useful if the actual run-time is non-deterministic, that is, essentially sampled from a distribution around the expected run-time. In order to determine what distributions we should sample our run-times from in our simulation, we first conducted some experiments with actual programs to try to determine what types of distributions characterize their run-times.

3.1 Job Run-time Distributions

We have already explained why job-machine run-times are typically not constant, but rather vary according to some distribution. To test the performance of our algorithms, it is essential to draw samples of the run-times of jobs from a particular distribution; but first we need to determine some realistic distributions that we can use in our simulations. Therefore, we repeatedly executed some parallel and sequential programs, gathered run-time statistics, and analyzed them.

We performed several experiments using the NAS Benchmarks [3]. These benchmarks were used to determine the types of run-time distributions that may be typical for at least some jobs on some machines. We needed to determine sample parameters for these run-time distributions so that they could be reproduced by our simulator. While performing our tests, we controlled the following environmental characteristics: server location, network and server load, number of processors, amount of memory, and processor speed. Table 1 summarizes the configurations of our machines `caesar` and `elvis` upon which we ran our experiments.

	`caesar`	`elvis`
Type SGI	Challenge L	Onyx
Proc Speed (MHz)	200	150
Proc Type (MIPS)	R4400	R4400
# of Processors	4	4
Memory (Mbytes)	64	192
Secondary Unified Cache	4 Mb	1 Mb

Table 1: Configuration of SGI machines `caesar` and `elvis`, both running IRIX64 v6.2.

The jobs that we used throughout these experiments were from two sources: NASA's reference implementation for some of the NAS Benchmarks, and our own implementations of other NAS Benchmarks that met the required criteria. Four of the experiments use some version of the NAS Integer Sort (IS) Benchmark, implemented either in parallel on four processors, or in single processor mode. Two other experiments used the NAS Embarrassingly Parallel (EP) Benchmark run on a single processor. We now explain our experiments and their results.

3.1.1 Integer Sort, Executed on Four Processors

This experiment examined the run-time distribution of a version of the NAS Integer Sort Benchmark executed on four processors. We implemented the integer sort using a counting sort [5, pages 175–178] algorithm. We used Silicon Graphic's light weight process (thread) support functions, including `mfork()`, to implement our version of this benchmark.

We ran this sort across a heavily loaded network, obtaining both the executable and the data from a file server that was also heavily loaded. When run on `caesar`, the run-time distribution, for 100 executions, appears Gaussian.[2] Figure 1 shows a histogram of this distribution. When run on `elvis`, the run-time distribution, again for 100 executions, appears exponential and is shown in Figure 2. We note that the origin of the exponential distribution shown in Figure 2 is translated to approximately 3.0. That means that the sort had to run for *at least* 3.0 seconds before stopping. The distribution that we see very closely matches an exponential distribution with a mean of around 0.20, translated 3.0 seconds to the right. We expect that many jobs would have a distribution similar to this, because all jobs must run at least some amount of time[3].

In these experiments, we also see that memory size, and so, the need to swap to local disk, can have a definite effect upon the run-time distribution of a job. The integer sort on `elvis` completes, on average, 30% sooner than the same job on `caesar`. We note that, in this case, the amount of memory has more influence

[2] The form of the distributions were determined by carefully selecting the bin size and then curve fitting. The authors are familar with both visual and analytical tests for normality, but analytical tests were not used given the strong visual similarity of the frequency plots to that of a Normal curve. (The fact that some sample point frequencies lie above and below the selected Normal distribution is due to the number of samples being finite. Such phenomena would have appeared even if 100 data points had been sampled from a known Normal run-time distribution.)

[3] An exponential distribution is defined to start at 0.0. If applied, without translation, in this case, that would mean there is a strong possibility of near-zero run-times.

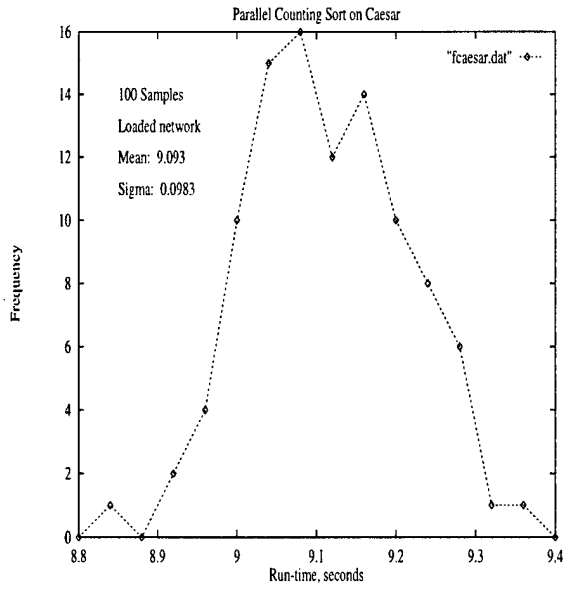


Figure 1: Forked counting sort, `caesar`.

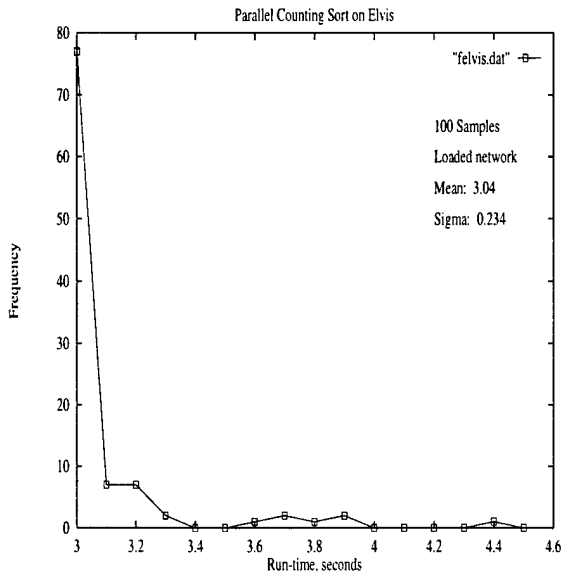


Figure 2: Forked counting sort, `elvis`.

on the run-time of the job than does the speed of the processor. Of primary importance, however, is the observation indicating that the same job, running on two different machines, not only has different mean run-times, but the distribution of run-times is different, yielding a Gaussian-like distribution on one machine and an exponential-like distribution on the other.

3.1.2 Integer Sort, Single Processor

This experiment is the same as that discussed in the last section, with the exception of being run on a single processor instead of being distributed across four processors. Although a slightly different C++ implementation was used, we again based our program on the counting sort.

When the integer sort was run on `caesar` and `elvis`, the run-time distribution was not easily characterized; however, it appears related to a Gaussian distribution. Histograms of the distributions, similar to that shown in Figure 4, are possibly multimodal, which indicates that multiple distributions may be present. While this experiment does not provide us with definitive results, it does point to the fact that run-time distributions can be quite complex. We suspect that these conditions are related to changes in the network and server loads.

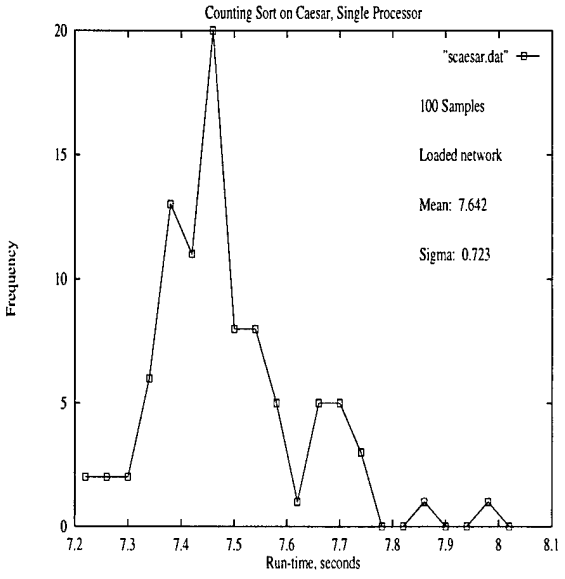


Figure 3: Counting sort, `caesar`, single processor.

Once again, this set of experiments showed us that additional memory can greatly enhance run-time performance. The tests on `elvis` ran 7 times faster than those run on `caesar`, which has the faster processors. The tests also show that run-time distributions can be very complex, and may be difficult to reproduce in a simulation. Although our simulations did not use such complex distributions, they should be modeled in future work.

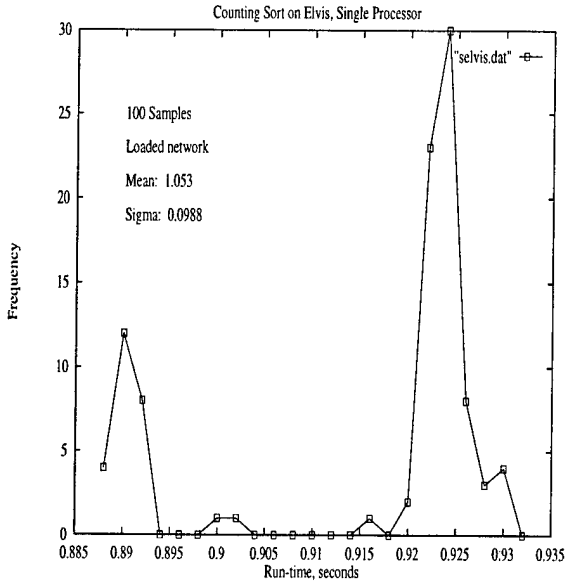


Figure 4: Counting sort, `elvis`, single processor.

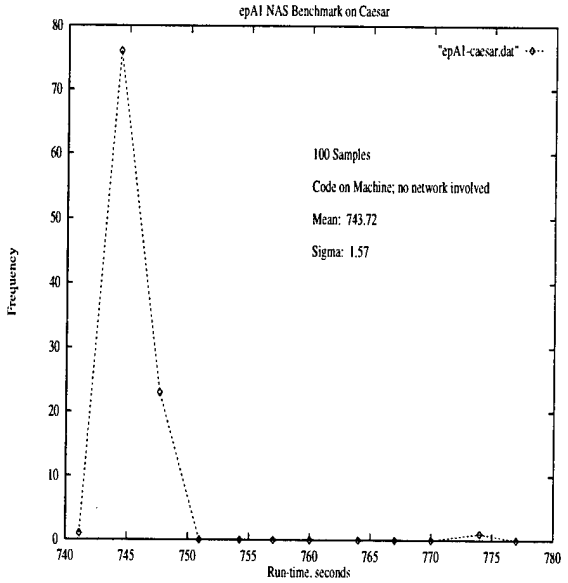


Figure 5: epA1 NAS Benchmark, with executable residing on local disk.

3.1.3 Embarrassingly Parallel NAS Benchmark

The next set of experiments that we describe compared the run-time distributions of compute intensive jobs run from local disk to those run across the network from a file server. The tests that we describe in this section were executed only on `caesar` because `elvis` did not have a sufficiently large local disk available. We used the reference implementation [3], from NASA, of the NAS Embarrassingly Parallel (EP) Benchmark. This implementation uses the portable Message Passing Interface (MPI) [12] to parallelize the code. The tests we ran, however, were compiled to be executed on a single processor[4]. The EP Benchmark was run 100 times for each test. See Figures 5 and 6.

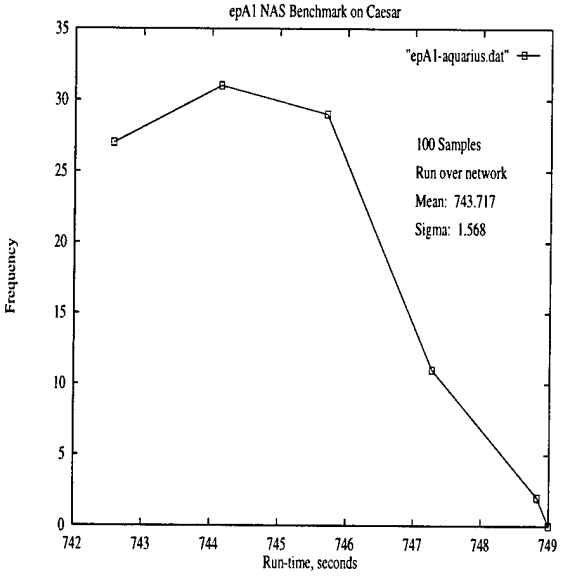


Figure 6: epA1 NAS Benchmark, files obtained over a lightly loaded network.

3.2 Simulation Experiments

We now describe our simulation experiments that are aimed at examining how well the mapping algorithms performed when the jobs scheduled did not execute for exactly the mean run-time. The matrices that we refer to in the description below have rows indexed by the job and columns indexed by the machine.

- Matrix Format. We used different matrices containing jobs and machines of varying characteristics. Each matrix contained mean run-times for each of five different jobs on each of ten different machines. The average means of the corresponding columns and rows were the same for all matrices and the jobs themselves were quite heterogeneous.

- Job Request Sets. In order to obtain different results for each matrix, we generated two random sequences of 125 job requests, which we will call 125-1 and 125-2, where each individual request was chosen according to a uniform random distribution from among five different jobs. We also generated two more ran-

[4]The MPI mechanism is still utilized in the EP Benchmark when it is compiled for a single processor.

dom sets, this time of 500 job requests, calling them 500-3 and 500-4. We did this to look at performance variations between job request orderings, as well as to examine any performance differences that might occur because fewer or more jobs were requested.

- Job Request Format. We generated each of the 5 jobs, for each request, at random. Thus, in these experiments, the jobs were requested in random order. This was done because the order of job request affects the schedule. The Fast Greedy Algorithm maps and schedules the jobs on machines in the order in which they are submitted. The Greedy Algorithm uses the order to break ties. We chose to execute these randomly ordered requests both because they more closely mimic a real environment where different jobs are submitted by different users and because we wished to examine whether these algorithms performed better or worse when unsorted, as opposed to sorted, requests were submitted.

- Run-time Generation for Simulations. We executed each simulation 15 times. In each run, a different value was used to seed the random number generator that was used to generate the simulated "actual" run-time duration. The total time required to execute each schedule was summed and the average was computed. Multiple seeds were used to ensure that our results were not skewed[5].

- Baseline Calculations. In addition to simulations where we generated simulated run-times from particular distributions, we performed some **baseline calculations**. These baseline calculations provided results that were, in effect, equivalent to running the simulation where the run-time of a job on a given machine was always exactly its expected run-time.

- Actual Run-time Distributions. When we generated run-times that were different from the mean predicted run-times, we ran experiments for both Gaussian and exponential distributions. Based upon our experiments with the NAS IS and EP Benchmarks above, we chose to implement a translated exponential distribution.

  Again, based upon our earlier experiments described in Section 3.1, we chose to use a truncated Gaussian distribution in our simulation experiments to mimic the Gamma distribution that best fit our data. We chose to truncate left of the mean at $\mu - \sigma$.

3.3 Results of Simulation Experiments where Jobs Ran for Times Different from the Predicted Run-times

This set of experiments examined the performance of intelligent mapping algorithms when job run-times differed from the expected run-times that were used to develop the mappings. Using the distributions identified in the previous experiments, we instantiated specific parameters in order to simulate some typical jobs. We simulated jobs with both exponential and truncated Gaussian run-time distributions. In this paper we summarize results; individual results from additional individual experiments, which are consistent with the conclusions that we make in this paper, can be found in Armstrong's thesis [2].

The graphs in this section compare the final completion times of the jobs under the various mappings. We use the label **Baseline** to mean that the value represented would be the completion time if all of the jobs ran for exactly their predicted mean run-times. In order to emphasize the differences between the values that we plot in the graph, we do not include the OLB run-times. The OLB run-times, for the exponential and Gaussian distribution simulations that we discuss below, averaged around 10,000 seconds in all cases shown, i.e., 500 requests.

3.3.1 Exponential Distribution Experiments

The results of these experiments compare the performance of the various mapping algorithms when all jobs have an exponential run-time distribution. We recall that the sample run-times from those experiments closely fit a shifted exponential distribution with mean of 3.0.

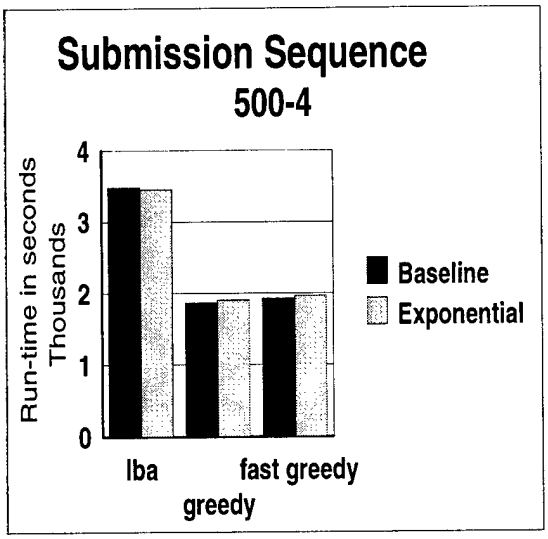


Figure 7: Exponential run-time distribution results, 500-4.

[5] This is a common method to reduce the influence of a single random number generation sequence that may be biased.

We now compare the time at which the last job finishes if executed according to each of the mappings, assuming that a job is not started on a machine until the last job completes. The figures in this section show both the expected completion time assuming deterministic run-times as well as under the assumption that the run-times are exponentially distributed, shifted to the right such that its mean matches the expected run-time.

Figure 7 shows these comparisons for some matrices that we used in our simulations. This figure shows that the schedules built by the intelligent mapping algorithms are still effective even though the actual run-time of a given job on a given machine can differ greatly from its expected run-time.

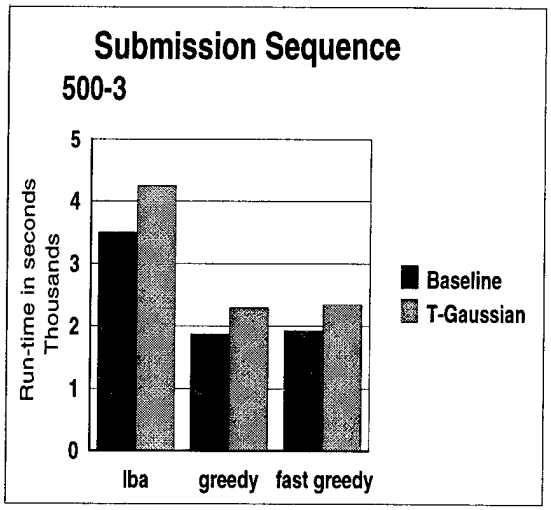


Figure 8: Truncated Gaussian run-time distribution results, 500-4.

3.3.2 Truncated Gaussian Experiments

We then performed additional simulations to examine the performance of the the intelligent mapping algorithms when all jobs had approximately Gamma run-time distributions. We determined from our experiments that we could approximate such a distribution by truncating a Gaussian distribution to the left of the mean at roughly $\mu - \sigma$. Throughout this experiment, the mean, μ, was the expected run-time for the individual job/machine pair, and σ^2 was set to 300% of μ. Therefore, these experiments are useful in determining whether, when the variance is very large for all jobs, the greedy algorithms still performed much better than both the LBA and OLB algorithms. No negative run-times were generated in our experiments because the truncation value was always positive.

The results in Figure 8 show that the schedules are finishing up to 25% later than in the previous experiments. This not unexpected, as truncation will shift the mean of the resulting distribution to the right. In the next section we provide a theoretical discussion as to why we would expect the times to be at least 20% later. The results also show that the greedy algorithms still perform better than the OLB and LBA algorithms when job run-time distributions are truncated Gaussian with very large variances. Our experiments, and the theoretical explanation below, imply that it may be worthwhile to update the mapping as the jobs are being executed, to minimize the effect of the large job variances.

3.3.3 Theoretical Explanation for Longer Run-times shown in Gaussian Experiments

To theoretically predict the new mean of the truncated distribution described in the last section, we can use simple Gaussian statistics [1]. Without loss of generality, our explanation uses a standard Gaussian distribution with a mean of 0 and a standard deviation of 1. If $A(z_1)$ is the area under the distribution from the mean, $z = 0$, to $z = z_1$, then it can be easily shown that the new mean, μ_{new}, for our truncated distribution is

$$\mu_{new} = A^{-1}\left[\frac{.5 - A(1)}{2}\right] \qquad (1)$$

Using this, we see that the new mean should be $\mu_{new} = .20\sigma$.

Unfortunately, the truncation of the Gaussian distribution only accounts for a 20% increase in the mean. Therefore, this explanation alone leaves some 5% unaccounted for. The remaining 5% is due to two factors. The first can be traced to the fact that we are using a truncated Gaussian instead of a Gamma distribution. The second is the fact that the expected value of the maximum of several Gaussian distributions is not the maximum of the expected values. The application of this well-known probability result to quality of service metrics is documented elsewhere [9].

3.3.4 Comparison of the Two Greedy Algorithms

We note that in our results, presented both here and in Armstrong's thesis, the Greedy and Fast Greedy algorithms appeared to perform similarly. Over all of our experiments we only saw the Greedy Algorithm performing up to 15% better than the Fast Greedy Algorithm. Other work has suggested that the improvement should be much higher. However, the other work, to our knowledge, was based upon presenting *sorted* requests to these mapping algorithms. The theoretical explanation for these results is beyond the scope of this paper and is discussed in another paper [7].

4 Related Work

To our knowledge, no one else has studied the performance of intelligent heterogeneous mapping algorithms when the run-times of jobs are non-deterministic, by using the distributions of run-times for actual programs determined under different resource loadings.

Ibarra and Kim [8] were the first to study the performance of the algorithms upon which we concentrated. Their analytical study centered around determining the worst-case performance of the algorithms. Weissman [15] used simulation to study interference-based policies; that is, policies that take into account the fact that as you increase the load on any shared resource, the rate of execution of other jobs decreases. Our policies, and simulations, assumed that the jobs were executed on a first-come, first-served basis. Although we did not study their performance here, genetic algorithms have been proposed as a good way to schedule tasks on heterogeneous resources, particularly when communication or synchronization is needed between tasks [13], [14]. Many systems have followed the lead of SmartNet [6] in implementing intelligent schedulers, such as those we describe here, in their resource management systems [11], [4], [16].

5 Summary

In this paper, we experimented with several applications on resources with differing loads and fitted their run-times to distributions. We then used these distributions to determine via simulation whether, when the run-times are non-deterministic, it is still beneficial to use intelligent algorithms that make use of the expected run-times to compute a mapping. We found that it continues to be beneficial even when the expected run-time distributions have large variances. As the distributions in our simulations were derived from the execution of actual programs, our distributions are realistic. However, there are additional distributions that are also realistic that we have not yet examined. We intend to pursue these in future work.

References

[1] ALDER, H. L., AND ROESSLER, E. B. *Introduction to Probability and Statistics*, third ed. Freeman, London, England, 1964.

[2] ARMSTRONG, R. K. Investigation of Effect of Different Run-time Distributions on SmartNet Performance. Master's thesis, U.S. Naval Postgraduate School, September 1997.

[3] BAILEY, D., ET AL. The NAS Parallel Benchmarks 2.0. Tech. Rep. NAS-95-020, NASA Ames Research Center, December 1995.

[4] BEGUELIN, A., ET AL. *HeNCE: A User' Guide.* Oak Ridge National Laboratory and University of Tennessee, December 1992. The document itself is available on the web at cs.utk.edu.

[5] CORMEN, T. H., LEISERSON, C. E., AND RIVEST, R. L. *Introduction to Algorithms.* The MIT Press, Cambridge, Massachusetts, 1990.

[6] FREUND, R., KIDD, T., HENSGEN, D., AND MOORE, L. Smartnet: A Scheduling Framework for Heterogeneous Computing. *Proceedings of the International Symposium on Parallel Architectures, Algorithms and Networks* (1996).

[7] HENSGEN, D., KIDD, T., AND ARMSTRONG, R. Comparison of greedy algorithms for scheduling jobs in a heterogeneous environments. In progress.

[8] IBARRA, AND KIM. Heuristic Algorithms for Scheduling Independent Tasks on Nonidentical Processors. *Journal of the ACM* (1977).

[9] KIDD, T., AND HENSGEN, D. Why the mean is inadequate for accurate scheduling decisions. In progress.

[10] KIDD, T., HENSGEN, D., FREUND, R., KUSSOW, M., AND CAMPBELL, M. Compute Characteristics: A Useful Characterization of Job Runtimes. In preparation for submission (1998).

[11] NEUMAN, B. C., AND RAO, S. The Prospero Resource Manager: A Scalable Framework for Processor Allocation in Distributed Systems. *Concurrency: Practice and Experience* (1994).

[12] PACHECO, P. A User's Guide to MPI. Tech. rep., Department of Mathematics, University of San Francisco, March 1995.

[13] SINGH, H., AND YOUSSEF, A. Mapping and Scheduling Heterogeneous Task Graphs using Genetic Algorithms. *Proceedings of the Heterogeneous Computing Workshop* (1996).

[14] WANG, L., SIEGEL, H. J., AND ROYCHOWDHURY, V. P. A Genetic-Algorithm-Based Approach for Task Matching and Scheduling in Heterogeneous Computing Environments. *Proceedings of the Heterogeneous Computing Workshop* (1996).

[15] WEISSMAN, J. B. The Interference Paradigm for Network Job Scheduling. *Proceedings of the Heterogeneous Computing Workshop* (1996).

[16] ZHOU, ZHENG, WANG, AND DELISLE. Utopia: A load sharing facility for lage heterogeneous distributed computer systems. *Software: Practice and Experience* (1993).

Biographies

Major Robert K. Armstrong is currently in charge of the Modeling and Simulation Laboratory for the Marine Corps Air Ground Combat Center, Twentynine Palms, California. He received his BS in Engineering from the United States Naval Academy in 1985, is a graduate of the Amphibious Warfare School in Quantico, Virginia, and has earned an MS in Computer Science from the Naval Postgraduate School, Monterey, California in 1997. Major Armstrong has served in the capacity of Artillery Officer with the 1st Marine Division in Korea, Somalia, and Kuwait. His interests include computer architecture, distributed systems, and modeling and simulation for training.

Debra Hensgen received her Ph.D. in Computer Science, in the area of Distributed Operating Systems from the University of Kentucky in 1989. She is currently an Associate Professor of Computer Science at the Naval Postgraduate School in Monterey, California. She moved to Monterey from the University of Cincinnati three years ago where she was first appointed as an Assistant Professor and then a tenured Associate Professor of Electrical and Computer Engineering. Her research interests include resource management and allocation systems and tools for concurrent programming. She has authored numerous papers in these areas. She is currently a Subject Area Editor for the Journal of Parallel and Distributed Computing and is the chief architect and a co-Principal Investigator for the DARPA-funded MSHN project which is part of DARPA's larger QUORUM program.

Taylor Kidd is an Associate Professor of Computer Science at the Naval Postgraduate School (NPS) in Monterey, California. He received his Ph.D. in Electrical and Computer Engineering from the University of California at San Diego (UCSD) in 1991. He received his MS and BS, also in Electrical and Computer Engineering, from UCSD in 1986 and 1985 respectively. Prior to accepting a position at the NPS, he was a researcher at the Navy's NRaD laboratory in San Diego, California. His current interests include distributed computing and the application of stochastic filtering and estimation theory to distributed systems. He is a co-Principal Investigator, along with Debra Hensgen, for the DARPA-funded MSHN project which is part of DARPA's larger QUORUM program.

Session III

Modeling Issues and Group Communications

Session Chair

David J. Lilja
University of Minnesota, Minneapolis, MN, USA

Modeling the Slowdown of Data-Parallel Applications in Homogeneous and Heterogeneous Clusters of Workstations

Silvia M. Figueira* and Francine Berman**

Department of Computer Science and Engineering
University of California, San Diego
La Jolla, CA 92093-0114
{silvia,berman}@cs.ucsd.edu
http://www-cse.ucsd.edu/users/{silvia,berman}

Abstract

Data-parallel applications executing in multi-user clustered environments share resources with other applications. Since this sharing of resources dramatically affects the performance of individual applications, it is critical to estimate its effect, i.e., the application slowdown, in order to predict application behavior. In this paper, we develop a new approach for predicting the slowdown imposed on data-parallel applications executing on homogeneous and heterogeneous clusters of workstations. Our model synthesizes the slowdown on each machine used by an application into a contention measure – the aggregate slowdown factor – used to adjust the execution time of the application to account for the aggregate load. The model is parameterized by the work (or data) partitioning policy employed by the targeted application, the local slowdown (due to contention from other users) present in each node of the cluster, and the relative weight (capacity) associated with each node in the cluster. This model provides a basis for predicting realistic execution times for distributed data-parallel applications in production clustered environments.

1. Introduction

Clusters of workstations have been used effectively as parallel machines for large, data-parallel, scientific applications [3, 4, 6, 16]. Workstations in such clusters are typically time-shared, causing competing applications to split the CPU and memory capacity on each node. The result of such sharing is that the fraction of resources available for a single distributed parallel application is reduced, causing a *slowdown* in the overall application performance. In [1], Arpaci et al. have shown that this slowdown can be significant. When slowdown can be predicted accurately, this information can be used to improve scheduling decisions in shared distributed systems [9].

A number of researchers have investigated how slowdown might be calculated and used to improve performance. Slowdown on a single machine has been used for task scheduling (as shown in [3] and [6]) and to predict performance (as shown in [10], [13] and [22]). Co-scheduling algorithms developed for networks of workstations have taken the slowdown on each machine into account, as shown in [2] and [7]. Weissman [19] has proposed a scheduling model based on resource contention using an *interference paradigm*. The interference measure determines how much slower an application will compute or a parallel application will communicate.

In this paper, we develop a new model for predicting the slowdown imposed on data-parallel applications executing on homogeneous and heterogeneous clusters of workstations. Our model synthesizes the slowdown on each machine used by an application into a contention measure – the *aggregate slowdown factor* – used to adjust the execution time of the application to account for the aggregate load. This factor can be combined with the time to execute the application in dedicated mode (on the same cluster) to provide an estimate of application performance in the presence of contention.

The aggregate slowdown factor is based on both application-specific resource usage and node computational capacity. Node capacity depends both on the dedicated capacity (e.g., CPU speed) of the node and on the load experienced by the node. Note that the dedicated capacity of nodes in a cluster may be non-uniform when the nodes are heterogeneous.

This paper is organized as follows. Section 2 introduces our model and the aggregate slowdown measure. We focus on aggregate slowdown factors for two common work partitioning policies (load-dependent and constraint-based) in Section 3. Section 4 discusses the local

* Supported by a scholarship from CAPES (Brazil).
** Supported in part by NSF contract number ASC-9301788 and ASC-9701333.

0-8186-8365-1/98 $10.00 © 1998 IEEE

slowdown in each node of the cluster. In Sections 5 and 6, we describe our experiments and evaluate the accuracy of the model presented. Section 7 concludes with a summary.

2. The Environment

We define a contention measure, the *aggregate slowdown factor*, that determines how load in the cluster will retard the performance of an individual application. Using this factor, the time X to execute a data-parallel application on a cluster is given by

$$X = X_{ded} \times sd, \quad (1)$$

where sd is the aggregate slowdown factor (dependent on the aggregate load in the cluster), and X_{ded} is the time to execute the targeted application on the cluster without interference from competing applications.

For this paper, we focus on loosely synchronous CPU-bound applications as defined by Fox [12]. These applications are coarse-grained data-parallel scientific applications in which computation and communication phases alternate. Such applications can profit from execution in clustered environments.

The computational environment is a cluster of homogeneous or heterogeneous workstations. Note that the model is independent of the number of nodes in the cluster. We assume that the cluster is shared by CPU-bound serial and/or data-parallel distributed applications, which execute for the entire duration of the targeted application. Each node in the cluster has one CPU. We assume that contention due to CPU-bound applications sharing resources provides the major source of slowdown. Developing an accurate contention model for CPU-bound applications provides a fundamental building block for contention models for a more heterogeneous job mix in which competing applications may also be non-CPU bound.

For this model, we consider the effects due to system overhead as well as application communication costs to be minimal. Although considering communication costs seemed essential at first, our experiments showed that for representative CPU-bound data-parallel applications the amount of communication was not significant.

In our experiments, we assumed that each workstation schedules its processes locally and independently using a round-robin mechanism. Note that priority-based mechanisms, usually employed by workstations' operating systems, reduce to round-robin when the competing applications are CPU-bound [8].

Finally, we assume that the memory in each node fits the working set of all the applications executing on the node and that no delay is imposed by swapping. The model can be extended to include more varied memory access costs in a straightforward manner.

3. Aggregate Slowdown

Aggregate slowdown is defined to be the performance slowdown of an application due to other applications sharing the cluster. Aggregate slowdown factors must be calculated based both on the work partitioning policy and on the capacity of each node. Node capacity is given by two parameters, each determined for each node in the cluster:

- sd_a = the local slowdown at node a and
- w_a = the weight of node a.

The *local slowdown* at node a, sd_a, is the delay imposed on an application running on node a as a function of other applications that share a's CPU. We show how to calculate the local slowdown in Section 4.

The *weight* of a node a, w_a, reflects its dedicated capacity relative to the other nodes in the cluster. The value for w_a can be calculated as the ratio of the time to execute a task on the slowest node s to the time to execute the same task on node a (as described in [7]). In a homogeneous cluster, $w_a = 1$, for all a. To calculate weights for nodes in a heterogeneous cluster, we execute a serial benchmark in dedicated mode on the different nodes of the cluster and calculate the weight for each node $a \neq s$ as $w_a = t_s / t_a$, such that the machine with the longest time t_s, has weight $w_s = 1$. Note that weights are dependent on the benchmark chosen [7, 9].

The following subsections develop aggregate slowdown factors for two work partitioning policies commonly used by high-performance data-parallel applications:

- *load-dependent partitioning*, in which the amount of work allocated to each node is calculated based on its computational capacity, and
- *constraint-based partitioning*, in which work is partitioned among the nodes according to a set of constraints (e.g., memory availability).

Note that we base our model on work partitioning (and not on data partitioning) since the computational effort required by a node is not always proportional to the amount of data the node is assigned. Note also that the model developed does not cover applications that have a dynamic work partitioning independent of the load, i.e., applications for which work partitioning varies throughout their execution (e.g., particle simulation). The model does cover these applications when a dynamic rebalance strategy, which takes the load of each node into account, is implemented *as part of the code*.

3.1 Load-Dependent Partitioning

In this partitioning, work is allocated according to the available computational capacity of each node in the

cluster. In this case, work is partitioned so that all nodes will finish together (ideally), assuming they all started at the same time. For this partitioning, we determine the aggregate slowdown in terms of the *aggregate capacity* available on the cluster. We define aggregate capacity as the sum of the available computational capacities of the nodes. We define $capacity_a$ as the available computational capacity of node a. It is important to note that the available capacity of each node depends on both its local load and its weight. For instance, for an unloaded node k, $capacity_k = w_k$, and for a loaded node k, $capacity_k = w_k / sd_k$.

The aggregate slowdown is given by the ratio of the aggregate capacity available in the unloaded (or dedicated) cluster to the aggregate capacity available in the loaded cluster. We calculate the aggregate slowdown for a cluster formed by n nodes as

$$sd = \frac{aggregate\ capacity_{ded}}{aggregate\ capacity} = \frac{\left(\sum_{a=1}^{n} w_a\right)}{\left(\sum_{a=1}^{n} \frac{w_a}{sd_a}\right)}. \quad (2)$$

Figure 1 illustrates a load-dependent partitioning. The work for application A is partitioned according to the node capacities of a set of heterogeneous nodes, as shown in the figure. If node 4 is twice as slow as the others (i.e., $t_1 = t_2 = t_3 = t_4 / 2$), then $sd = (2 + 2 + 2 + 1) / (2 / 2 + 2 / 2 + 2 / 3 + 1 / 1) = 7 / 3.67 = 1.9$ according to equation (2).

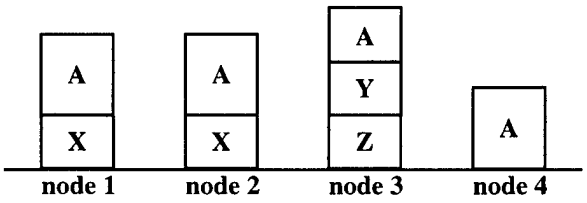


Figure 1: Cluster formed by 4 heterogeneous nodes and shared by three applications: A, X, and Y. Application A's work is divided among the nodes according to their computational capacity. Each box represents the amount of work associated with the corresponding node.

3.2 Constraint-Based Partitioning

When work is partitioned based on a set of constraints, such as memory availability or data locality, the aggregate slowdown can be calculated from the extra amount of work (relative to the work assigned in a *uniform* partitioning, in which the work is partitioned evenly among the nodes) assigned to each node. That is, the time to execute the part of the application assigned to each node is formed by two components: the amount of work that would be performed if the work partitioning were uniform, and the extra work that must be executed by a node because the work partitioning is not uniform. Using this approach, the time to execute the part of the application assigned to node a in the presence of contention is

$$time_a = (time_{a,ded} \times sd_a) + (time_{a,ded} \times ew_a \times sd_a) = time_{a,ded} \times sd_a \times (1 + ew_a), \quad (3)$$

where

- $time_{a,ded}$ = time to execute the part of the application assigned to node a, in dedicated mode,
- ew_a = extra work executed by node a, calculated as

$$ew_a = \frac{f_a - (1/n)}{1/n} = ((f_a \times n) - 1),$$

- f_a = fraction of work assigned to node a, and
- n is the number of nodes in the cluster.

Note that node a can be assigned less work than it would be if the partitioning were uniform. In this case, $f_a < 1/n$ and $ew_a < 0$.

If all nodes begin execution concurrently, the slowest node will determine the time to execute the application, which is given by equation (1). From (1) and (3), we can calculate the aggregate slowdown as

$$sd = \frac{X}{X_{ded}} = \frac{\max_a \left\{ \frac{(1 + ew_a) \times sd_a}{w_a} \right\}}{\max_a \left\{ \frac{(1 + ew_a)}{w_a} \right\}}, \quad (4)$$

where the numerator represents the time it takes to execute the application in the presence of contention, and the denominator represents the time it takes to execute the application in dedicated mode.

Note that the uniform partitioning policy, in which each node is assigned an equal amount of work, is characterized by $ew_a = 0$ for all a, and the aggregate slowdown for this policy can also be calculated by equation (4).

Equation (4) assumes that the targeted application will use the same constraint-based work partitioning (given by ew_a) in determining both X and X_{ded}. This may not be always the case, since the work partitionings used for X and X_{ded} may be determined by different constraints. For example, if the partitioning depends on the memory available on each workstation, which varies according to the load, the partitioning used for the dedicated and non-

dedicated executions may not be the same. Therefore, we must allow for different constraint-based partitionings to be used for X and X_{ded}. Call these constraint-based partitionings d_1 and d_2. To accommodate for d_1 (for X) and d_2 (for X_{ded}), equation (4) is modified to

$$sd = \frac{\max_a \left\{ \frac{(1 + ew_{a,1}) \times sd_a}{w_a} \right\}}{\max_a \left\{ \frac{(1 + ew_{a,2})}{w_a} \right\}}, \quad (5)$$

where

- $ew_{a,y}$ = extra work executed by node a with partitioning d_y ($y = 1, 2$), calculated as

$$ew_{a,y} = \frac{f_{a,y} - (1/n)}{1/n} = ((f_{a,y} \times n) - 1), \text{ and}$$

- $f_{a,y}$ = fraction of work assigned to node a for partitioning d_y ($y = 1, 2$).

In the setting where the dedicated time X_{ded} is given for a uniform work partitioning, equation (4) can be reduced to

$$sd = \frac{\max_a \left\{ \frac{(1 + ew_a) \times sd_a}{w_a} \right\}}{\max_a \{1/w_a\}}, \quad (6)$$

where the numerator represents the time to execute in the presence of contention with a constraint-based partitioning (characterized by ew_a), and the denominator represents the time to execute in dedicated mode with a uniform partitioning.

Since $\max_a \{1/w_a\} = 1$, for any set of nodes, equation (6) can be reduced to

$$sd = \max_a \left\{ \frac{(1 + ew_a) \times sd_a}{w_a} \right\}. \quad (7)$$

Figure 2 illustrates the constraint-based partitioning. It shows application A's work partitioning in a four-node cluster. The application has 12 work units, which are partitioned among the nodes according to memory availability. The dotted lines in the figure determine application A's work units. According to the figure, $f_1 = 1/12$, $f_2 = 3/12$, $f_3 = 3/12$, and $f_4 = 5/12$. If the nodes are homogeneous, nodes 2 and 3 determine the time to execute application A. This happens because nodes 2 and 3 are both assigned 3 work units, which are delayed by a factor of 2, whereas node 4 is assigned 5 work units, which execute in full speed. In this case, according to equation (7), $sd = (1 + 0) \times 2 / 1 = 2$. However, if the nodes are heterogeneous and node 4 is twice as slow as the others, the time to execute application A is determined by node 4, and $sd = (1 + 0.67) \times 1 / 1 = 1.67$, also according to equation (7).

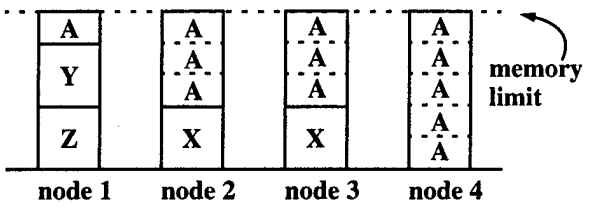


Figure 2: Cluster formed by 4 nodes and shared by three applications: A, X, and Y. Application A's 12 units of work (which are determined by the dotted lines) are divided among the nodes according to memory availability. Each box represents the amount of work associated with the corresponding node.

4. Local Slowdown

The aggregate slowdown sd is calculated based on the local slowdown sd_a on each workstation a of the cluster. The local slowdown on node a is the delay imposed on an application running on node a as a function of other applications that share a's CPU. In [10], we presented a model to calculate the local slowdown based on information (such as the computation/communication ratio) about the applications executing on the system. Since this information may not be always available, in this paper we present a different way of calculating local slowdown based on information provided by the system.

If the scheduling policy implemented by the local scheduler executed on each workstation is reduced to round-robin (which is generally the case when all tasks are CPU-bound and have the same priority), the slowdown imposed by contention on each node a can be calculated simply as

$$sd_a = p_a + 1, \quad (8)$$

where p_a is the number of extra processes executing on node a.

Equation (8) - with some slight variations - has been used for slowdown predictions in [3, 4, 6, 13, 19, 22]. However, it assumes that the competing applications executing on the nodes of the cluster are well balanced and executing at full speed (without idle intervals). This may not be always the case.

When the load is not well-balanced and idle cycles occur in a node, the local slowdown used in the aggregate slowdown calculation must be a function of the fraction of the *busy* (as opposed to idle) time of the competing applications. In this case, the slowdown imposed by contention on node a should be modeled pessimistically as

$$sd_a = 1 + \sum_{i=1}^{p_a} busy_{a,i}, \quad (9)$$

where $busy_{a,i}$ is the fraction of time in which application i is busy on node a, i.e., application i is either computing within its own time slice, communicating, or waiting for its time slice.

The value for $busy_{a,i}$ depends on the effects of the load in the cluster, including the targeted application. To calculate $busy_{a,i}$, we use the CPU-usage associated with application i on node a, and we have

$$busy_{a,i} \approx \min\{1,\ CPU_{a,i} \times (p_a + 1)\}, \quad (10)$$

where $CPU_{a,i}$ is the fraction of a's CPU used by application i. Note that $CPU_{a,i}$ is a fraction provided by the operating system. This fraction can also be predicted by tools such as the Network Weather Service [21].

According to equation (10), application i is busy all the time ($busy_{a,i} = 1$) when its CPU-usage corresponds to at least $1 / (p_a + 1)$. When application i's CPU-usage is less than $1 / (p_a + 1)$, it does not use the CPU all the time, and the product of its CPU-usage and the number of applications in the system gives the fraction of time in which application i is busy. Note that, since the targeted application will affect the amount of time application i is busy, $p_a + 1$ (not just p_a) is used.

Equation (10) is actually an upper bound on the time that application i is busy. It considers the worst case in which applications' computation cycles are synchronized (i.e., all applications compete for the CPU at the same time) and the targeted application neither has nor induces idle cycles in the competing applications. When computation cycles are not synchronized, applications will compete for the CPU less often. Also, when the targeted application either has or induces idle cycles in the competing applications, these applications may stay busy less time. In these cases, the slowdown on the node will decrease.

5. Experiments

The formulas presented in Section 3 provide a model for the aggregate slowdown. To assess the accuracy of this model, we performed a large collection of experiments on a wide range of CPU-bound benchmarks commonly found in high-performance scientific applications. Some examples of the applications used were: Jacobi2D [5], Jacobi3D [5], Red-Black SOR [5], Multigrid [5], Genetic Algorithm [20], and LU Solver [15]. In these experiments, we compared actual execution times with modeled times (dedicated time factored by aggregate slowdown) for applications executing on a cluster of workstations with CPU-bound synthetic loads. The synthetic loads were formed by a combination of CPU-bound serial and/or data-parallel applications distributed over the cluster. In this paper, we show representative experiments for each scheduling policy. On all graphs, we show actual times in contentious (multi-user) environment, predicted times using our slowdown model, and dedicated (single-user) actual times for comparison. A more complete list of the experiments can be found in [9].

All of our experiments show the modeled execution times to be within 15% of actual execution times. This demonstrates that, for reasonably accurate dedicated time performance estimates, aggregate slowdown captures contention delays in multi-user systems correctly and can provide an accurate model for performance predictions. The discrepancy in error between modeled and actual times is due to a variety of factors. For example, the fact that the round-robin scheduling policy assumed on each workstation is not a "perfect" round-robin contributes to the error.

Our experiments were performed on two platforms: a cluster of homogeneous nodes, represented by a DEC Alpha-Farm located at the San Diego Supercomputer Center, and a cluster of heterogeneous nodes formed by two DEC Alphas (located in the Computer Systems Laboratory at UCSD) and two IBM RS-6000s (located in the Parallel Computation Laboratory, also at UCSD). In the DEC Alpha-Farm, the nodes are connected by a GIGAswitch, which is dedicated to the nodes. In the heterogeneous cluster, the nodes are part of different Ethernet networks.

In this section, we show a representative subset of the experiments performed on the homogeneous and heterogeneous clusters described above.

5.1 Experiments on the Homogeneous Cluster

In the first set of experiments, we targeted clusters of uniform workstations.

Figure 3 illustrates two representative experiments with the load-dependent work partitioning policy. Shown is a distributed SOR application [5] developed using PVM [18] and executed on 4 nodes of the DEC Alpha-Farm with two different loads. The parameters for the experiments are shown in Table 1. In experiment 1 (*exp 1*), there is a CPU-bound data-parallel application executing on two of the nodes. In this case, *aggregate capacity* = 1 / 2 + 1 / 2 + 1 + 1 = 3, and *sd* = 4 / 3 = 1.33. In experiment 2 (*exp 2*), there is a CPU-bound data-parallel application executing on two of the nodes, and two serial CPU-bound applications executing on another node, as shown in Figure 1. In this case, *aggregate capacity* = 1 / 2 + 1 / 2 + 1 / 3 + 1 = 2.33, and the SOR algorithm was slowed by a

factor of 4 / 2.33 = 1.72.

Table 1: Parameters for the Experiments in Figure 3

Node	*sd* for exp 1	*sd* for exp 2
1	2	2
2	2	2
3	1	3
4	1	1

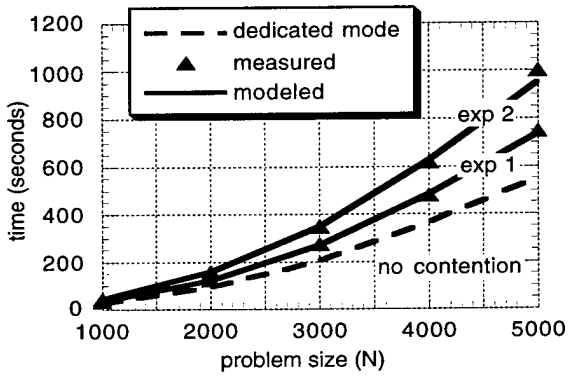


Figure 3: Time to execute the SOR algorithm on 4 nodes of the DEC Alpha-Farm in dedicated mode and with 2 different loads.

Figure 4 represents experiments using the constraint-based work partitioning policy. Shown is a Multigrid application [5] (developed using KeLP [11] and MPI [14]) executed for different problem sizes (given by $N \times N$) on 4 nodes of the DEC Alpha-farm. Two of the nodes (nodes 2 and 3) also host a CPU-bound data-parallel application, and one of the nodes (node 1) also hosts two serial, CPU-bound applications, as shown in Figure 2. The blocks of data were divided among the nodes according to a set of constraints resulting in the partitioning shown in Table 2 (column *f*). Table 2 also shows the other parameters (*sd* and *ew*) used to calculate the aggregate slowdown (3.0). Note that the dedicated time is given for a uniform work partitioning.

Table 2: Parameters for the Experiment in Figure 4

Node	*sd*	*f*(%)	*ew*	$(1 + ew) \times sd$
1	3	25	0	3.00
2	2	17	- 0.33	1.34
3	2	25	0	2.00
4	1	33	0.33	1.33

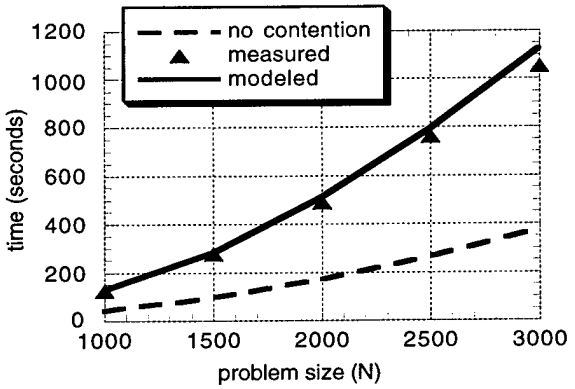


Figure 4: Time to execute the Multigrid application on 4 nodes of the DEC Alpha-Farm, with constraint-based partitioning, in dedicated mode and with contention on the nodes.

Figure 5 and Figure 6 also represent experiments using the constraint-based work partitioning policy. They present examples of a Jacobi3D algorithm [5] (developed using KeLP [11] and MPI [14]) executing for different problem sizes (given by $N \times N$). The dedicated time was given for a uniform work partitioning. Figure 5 shows modeled and measured times for execution on 4 nodes where other applications are also executing. In experiment 1 (*exp 1*), there was a CPU-bound parallel application executing on two of the nodes as shown in Table 3, imposing a slowdown of 2 on the execution of the Jacobi3D algorithm. In experiment 2 (*exp 2*), there was a CPU-bound parallel application executing on two of the nodes, one of which was also shared by a serial CPU-bound application, as shown in Table 4. In this case, the Jacobi3D algorithm was slowed by a factor of 3.

Table 3: Parameters for Experiment 1 in Figure 5

Node	*sd*	*f*(%)	*ew*	$(1 + ew) \times sd$
1	2	25	0	2
2	2	25	0	2
3	1	25	0	1
4	1	25	0	1

Table 4: Parameters for Experiment 2 in Figure 5

Node	*sd*	*f*(%)	*ew*	$(1 + ew) \times sd$
1	3	25	0	3
2	2	25	0	2
3	1	25	0	1
4	1	25	0	1

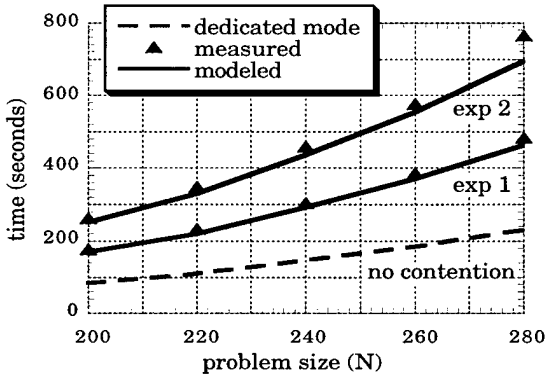


Figure 5: Time to execute the Jacobi3D algorithm on 4 nodes of the DEC Alpha-Farm in dedicated mode and with 2 different loads.

Figure 6 shows an example of the execution of the Jacobi3D benchmark for different problem sizes (given by $N \times N$) on 4 nodes, in which the CPU-bound applications described by Table 5 are also executing. The fractions in the table represent the amount of time the application is busy on the respective node. The work partitioning was constraint-based. The aggregate slowdown factor is 2.375, even though there are two applications executing on node 4. This is explained by the imbalance due to the work partitioning of the competing applications. In particular, this imbalance causes application 2 to be idle part of the time, increasing node 4's computational capacity. Note that the dedicated time parameter is given for a uniform work partitioning.

Table 5: Busy Fractions for the Experiment in Figure 6

Node	Parallel Application 1	Parallel Application 2
1		
2	2 / 8	
3		1
4	1	3 / 8

Table 6: Parameters for the Experiment in Figure 6

Node	sd	$f(\%)$	ew	$(1+ew) \times sd$
1	1.000	25	0	1.000
2	1.250	25	0	1.250
3	2.000	25	0	2.000
4	2.375	25	0	2.375

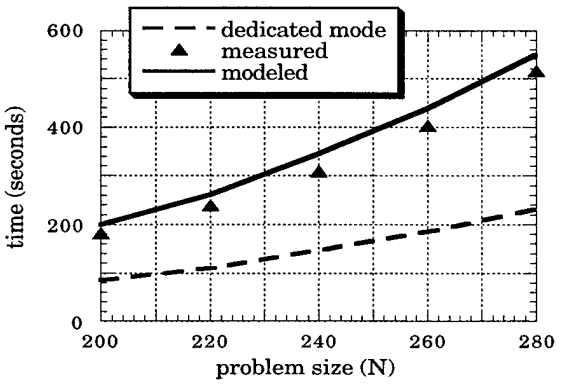


Figure 6: Time to execute the Jacobi3D algorithm on 4 nodes of the DEC Alpha-Farm in dedicated mode and together with the load described in Table 5.

5.2 Experiments on the Heterogeneous Cluster

We now relax the constraints that all the nodes in the cluster are uniform and communication links within the cluster are dedicated to the cluster itself.

Figure 7 shows an example of the execution of the SOR benchmark [5] (developed using PVM [18]) for different problem sizes (given by $N \times N$) on the 4-node heterogeneous cluster. The work was divided among the machines according to their capacity and loads. The ambient load was formed by one CPU-bound data-parallel application executing on the DEC Alphas. Table 7 describes the values used to calculate the aggregate slowdown for this situation, which is 8.14 / 5.08 = 1.6. Note that the time to execute the SOR benchmark for a 4000x4000 matrix is shorter than the time estimated by the model because, for this problem size, the round-robin scheduling policy (assumed on each workstation) is not a "perfect" round-robin, and the SOR benchmark gets higher priority.

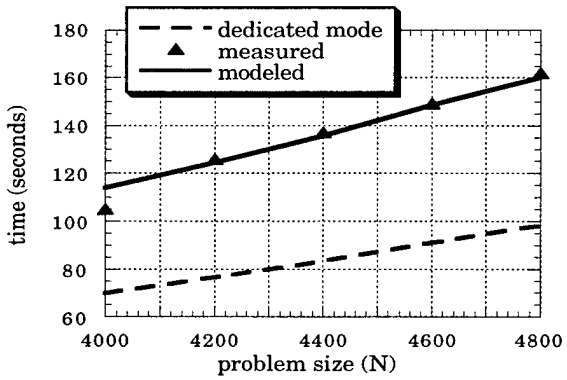


Figure 7: Times for the SOR algorithm executing with load-dependent work partitioning on a heterogeneous cluster in dedicated mode and under contention.

Table 7: Parameters for the Experiment in Figure 7

Node	w	sd
$alpha_1$	3.07	2
$alpha_2$	3.07	2
rs_1	1.00	1
rs_2	1.00	1

Figure 8 represents experiments using a load-dependent work partitioning policy with a Genetic Algorithm application [20] developed using PVM [18]. Shown are modeled and measured times for execution with different problem sizes (given by population size) on four nodes of the heterogeneous cluster. In this experiment, the IBM RS-6000s are also executing a CPU-bound data-parallel application. The parameters for the experiment are shown in Table 7. The aggregate slowdown is 1.18.

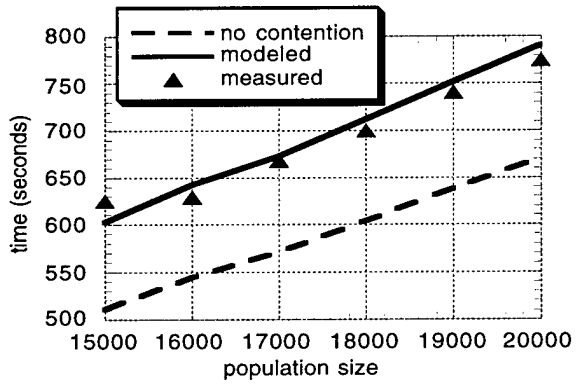


Figure 8: Times for the Genetic Algorithm executing with load-dependent work partitioning on a heterogeneous cluster in dedicated mode and under contention.

Table 8: Parameters for the Experiment in Figure 8

Node	w	sd
$alpha_1$	2.3	1
$alpha_2$	2.3	1
rs_1	1.0	2
rs_2	1.0	2

Figure 9 represents experiments using a constraint-based work partitioning policy. Shown is a representative SOR application [5] developed using PVM [18] and executed for different problem sizes (given by $N \times N$). The platform is a heterogeneous cluster consisting of two DEC Alphas and two IBM RS-6000s. Data for the dedicated run was partitioned according to a set of constraints, resulting in the partitioning shown in Table 9 (column f).

Table 9: Parameters for Experiments in Figure 9

Node	w	$f(\%)$	$(1+ew_i)/w_i$
$alpha_1$	3.07	17	0.22
$alpha_2$	3.07	33	0.43
rs_1	1.00	17	0.66
rs_2	1.00	33	1.33

In experiment 1 (*exp 1*) there was an additional CPU-bound data-parallel application executing on the 2 IBM RS-6000s. Table 10 shows the work partitioning (column f) and slowdown (sd) parameters used in this experiment. According to Table 9 and Table 10, the aggregate slowdown for this case is 2.67 / 1.33 = 2.01.

Table 10: Parameters for Experiment 1 in Figure 9

Node	w	sd	$f(\%)$	$(1+ew_i) \times sd_i/w_i$
$alpha_1$	3.07	1	17	0.22
$alpha_2$	3.07	1	33	0.43
rs_1	1.00	2	33	2.67
rs_2	1.00	2	17	1.33

In experiment 2 (*exp 2*) there was a CPU-bound data-parallel application executing on the 2 IBM RS-6000s, another executing on one IBM RS-6000 and one DEC Alpha, and three more CPU-bound serial applications executing on the other Alpha. The fractions in Table 11 represent the amount of time the application is busy on the respective node. Table 12 shows the work partitioning (column f) and slowdown (column sd) used in this experiment. According to Table 9 and Table 12, the aggregate slowdown for this case is 3.27 / 1.33 = 2.46. Note that, even though there is one application executing on $alpha_2$, its local slowdown due to load imbalance is 1.33.

Table 11: Busy Fractions for Experiment 2 in Figure 9

Node	Parallel Appl. 1	Parallel Appl. 2	Serial Appl. 3	Serial Appl. 4	Serial Appl. 5
$alpha_1$			1	1	1
$alpha_2$		1 / 3			
rs_1	1	1			
rs_2	1				

Table 12: Parameters for Experiment 2 in Figure 9

Node	w	sd	$f(\%)$	$(1+ew_i) \times sd_i/w_i$
$alpha_1$	3.07	4.00	27	1.42
$alpha_2$	3.07	1.33	27	0.47
rs_1	1.00	3.00	27	3.27
rs_2	1.00	2.00	19	1.45

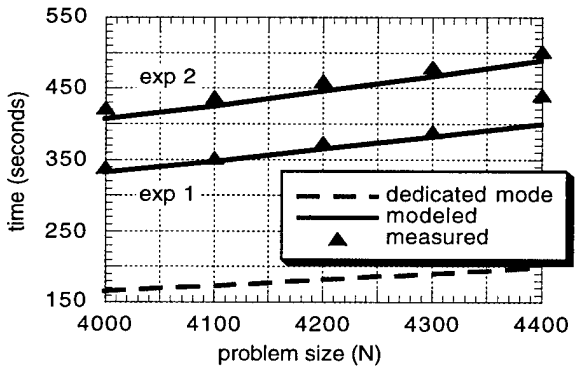


Figure 9: Times for two experiments with the SOR algorithm executing with constraint-based work partitioning on a heterogeneous cluster in dedicated mode and with contention.

Figure 10 also represents experiments using the constraint-based work partitioning policy. It presents examples of a Jacobi2D benchmark [5] (developed using KeLP [11] and MPI [14]) executing for different problem sizes (given by $N \times N$). The graph shows modeled and measured times for execution on the 4 nodes. One of the DEC Alphas and one of the IBM RS-6000s are also used to execute a well-balanced CPU-bound data-parallel task, as shown in Table 13, causing the aggregate slowdown to be 2. Note that the time estimated by the model is a little higher than the measured time because the round-robin scheduling policy (assumed on each workstation) is not a "perfect" round-robin and, in this case, Jacobi2D gets higher priority and executes faster than expected by the model.

Table 13: Parameters for the Experiment in Figure 10

Node	w	sd	$f(\%)$	$(1+ew_i) \times sd_i/w_i$
$alpha_1$	1.00	1	25	1.00
$alpha_2$	1.00	2	25	2.00
rs_1	1.84	2	25	1.09
rs_2	1.84	1	25	0.54

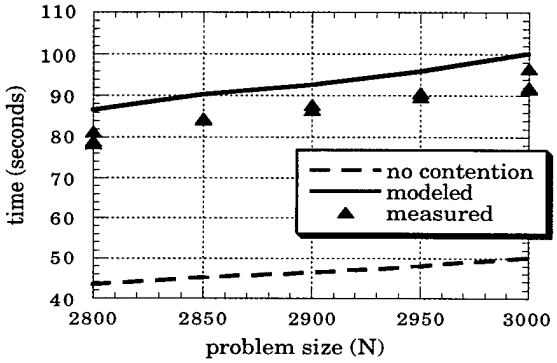


Figure 10: Times for the Jacobi2D benchmark executing with constraint-based work partitioning on the heterogeneous cluster in dedicated mode and with contention.

Figure 11 also presents examples of the Jacobi2D benchmark executing for different problem sizes (given by $N \times N$). The graph shows modeled and measured times for execution on the 4 nodes. The work partitioning was constraint-based. The contention is generated by one CPU-bound parallel application executing on the IBM RS6000s, a second one executing on the DEC Alphas, and a third one executing on one IBM RS-6000 and one DEC Alpha. This scenario is represented in Table 14. The aggregate slowdown is 3 due to the load in the most heavily loaded DEC Alpha, as shown in Table 15.

Table 14: Node Usage for the Experiment in Figure 11

Node	Parallel Application 1	Parallel Application 2	Parallel Application 3
$alpha_1$		✔	
$alpha_2$		✔	✔
rs_1	✔		✔
rs_2	✔		

Table 15: Parameters for the Experiment in Figure 11

Node	w	sd	$f(\%)$	$(1+ew_i) \times sd_i/w_i$
$alpha_1$	1.00	2	25	2.00
$alpha_2$	1.00	3	25	3.00
rs_1	1.84	3	25	1.63
rs_2	1.84	2	25	1.09

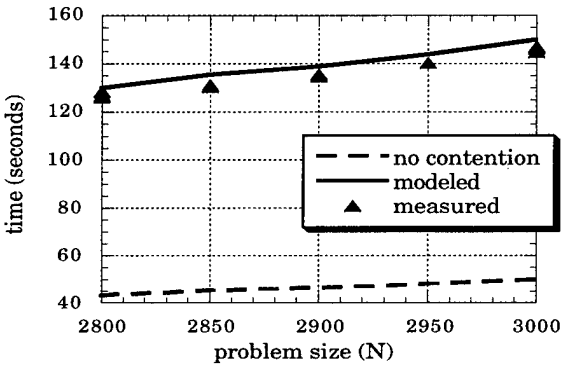


Figure 11: Times for the Jacobi2D benchmark executing with constraint-based work partitioning on the heterogeneous cluster in dedicated mode and with contention.

6. Evaluation of the Model

The models presented were developed based on the amount of busy time of the competing applications. They assume that the time to execute the targeted application is longer than the busy/idle cycles of the competing applications. If the target application is fast in comparison with the duration of these cycles, i.e., the time to execute the application is close to (or smaller than) the duration of one busy/idle cycle, the accuracy of the models decreases.

Figure 12 and Figure 13 illustrate the difference in accuracy obtained in the prediction of the time to execute the SOR benchmark on the heterogeneous cluster in two situations. In both situations, the SOR competes for the cluster with one CPU-bound data-parallel application that executes on one DEC Alpha and one IBM RS-6000. In Figure 12, the time to execute one busy/idle cycle of the competing application was 15.82 seconds. In this case, the time to execute the algorithm was longer than one busy/idle cycle of the competing application, and the average error was 2%. In Figure 13, the time to execute one busy/idle cycle of the competing application was 223.76 seconds. In this case, the time to execute the same algorithm was shorter than one busy/idle cycle of the competing application, and the variation of the actual times to execute the algorithm causes the average error to be within 12%.

Note that in Figure 13, for problem size 4400x4400, one execution of the benchmark with contention was faster than the execution in dedicated mode. This phenomenon is explained by a variation in the execution time in dedicated mode [17] caused by traffic in the network, which is nondedicated in the heterogeneous cluster. In our experiments, this variation is not significant because the amount of communication is small.

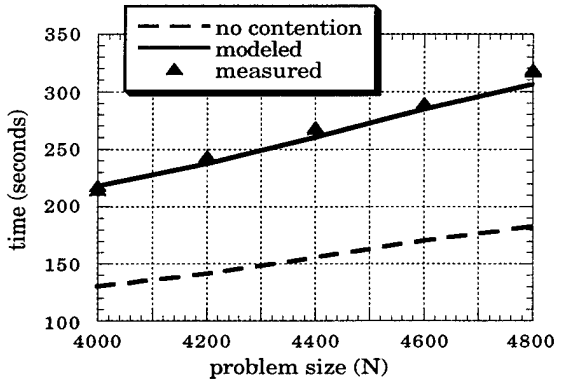


Figure 12: Time to execute the SOR benchmark in dedicated mode and competing with one application that has a 15.82-second busy/idle cycle.

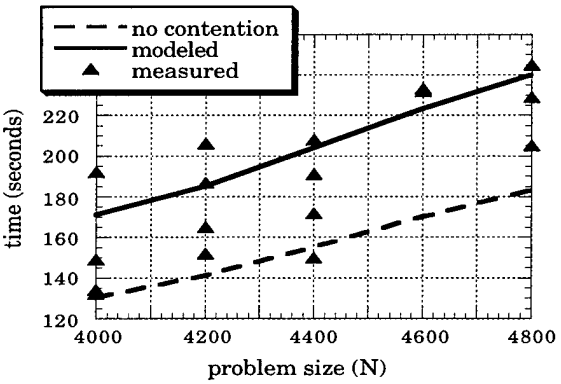


Figure 13: Time to execute the SOR benchmark in dedicated mode and competing with one application that has a 223.76-second busy/idle cycle.

7. Summary

In this paper, we have presented a model to predict contention effects in clustered environments. This model provides a basis for predicting realistic execution times for applications on clusters of workstations, a fundamental component of performance-efficient scheduling. The model is parameterized by the policy used to partition application work, the local slowdown present in each node of the cluster, and the relative weight of each node in the

cluster, all of which contribute substantively to application performance.

To determine the effects of contention, we developed a measure of **aggregate slowdown** – the delay on an individual application due to contention from other applications sharing the cluster. The determination of aggregate slowdown varies with the work partitioning policy and was developed here for two common work partitioning policies (load-dependent and constraint-based). We performed a set of experiments comparing modeled and actual times on a dedicated system with synthetic load for a set of benchmarks commonly found in high-performance scientific applications. The experiments showed our models to predict relatively accurately – on average within 15% – the delay imposed on an individual application due to contention on the cluster. Since the effect caused by contention in a time-shared environment can be large, as shown by our experiments, the aggregate slowdown provides a critical component in the accurate prediction of performance for data-parallel programs on multi-user clusters of workstations.

Acknowledgments

We would like to thank our colleagues in the UCSD Parallel Computation Laboratory and in the San Diego Supercomputer Center for their support, specially Stephen Fink, Karan Bhatia, Mike Vildibill, Ken Steube, Cindy Zheng, and Victor Hazlewood. We would also like to thank Professor Joseph Pasquale for the usage of the machines in the Computer Systems Laboratory at UCSD.

References

[1] R. H. Arpaci, A. C. Dusseau, A. M. Vahdat, L. T. Liu, T. E. Anderson, and D. A. Patterson, "The Interaction of Parallel and Sequential Workloads on a Network of Workstations", in *Proceedings of SIGMETRICS'95/PERFORMANCE'95 Joint International Conference on Measurement and Modeling of Computer Systems*, pp. 267-278, May 1995.

[2] M. Atallah, C. Black, D. Marinescu, H. Siegel, and T. Casavant, "Models and Algorithms for Coscheduling Compute-Intensive Tasks on a Network of Workstations", *Journal of Parallel and Distributed Computing*, vol. 16, pp. 319-327, 1992.

[3] A. Beguelin, J. Dongarra, G. Geist, R. Manchek, and V. Sunderam, "Graphical Development Tools for Network-Based Concurrent Supercomputing", in *Proceedings of Supercomputing 91*, pp. 435–444.

[4] A. Bricker, M. Litzkow, and M. Livny, "Condor Technical Summary", Technical Report #1069, University of Wisconsin, Computer Science Department, May 1992.

[5] W. L. Briggs, "A Multigrid Tutorial", Society for Industrial and Applied Mathematics, Philadelphia, Pennsylvania, 1987.

[6] H. Dietz, W. Cohen, and B. Grant, "Would you run it here...or there? (AHS: Automatic Heterogeneous Supercomputing)", in *Proceedings of the International Conference on Parallel Processing*, vol. II, pp. 217–221, August 1993.

[7] X. Du and X. Zhang, "Coordinating Parallel Processes on Networks of Workstations", Technical Report, High Performance Computing and Software Lab, University of Texas at San Antonio, August 1996.

[8] A. C. Dusseau, R. H. Arpaci, and D. E. Culler, "Effective Distributed Scheduling of Parallel Workloads, in *Proceedings of ACM SIGMETRICS'96*, pp. 25-36, May 1996.

[9] S. M. Figueira, "Modeling the Effects of Contention on Application Performance in Multi-User Environments," Ph.D. Dissertation, CSE Department, UCSD, December 1996.

[10] S. M. Figueira and F. Berman, "Predicting Slowdown for Networked Workstations," in *Proceedings of the Sixth International Symposium on High-Performance Distributed Computing*, August 1997.

[11] S. J. Fink, S. B. Baden, and S. R. Kohn, "Flexible Communication Mechanisms for Dynamic Structured Applications", in *Proceedings of the Third International Workshop on Parallel Algorithms for Irregularly Structured Problems*, Santa Barbara, CA, August 1996.

[12] G. Fox, "Hardware and Software Architectures for Irregular Problem Architectures," in *Unstructured Scientific Computation on Scalable Multiprocessors*, P. Mehrotra, J. Saltz, and R. Voigt, The MIT Press, Cambridge, MA, pp. 125–160, 1992.

[13] S. Leutenegger and X. Sun, "Distributed Computing Feasibility in a Non-Dedicated Homogeneous Distributed System", NASA - ICASE Technical Report 93-65, September 1993.

[14] Message-Passing Interface Forum, "MPI: A Message-Passing Interface Standard", University of Tennessee, Knoxville, TN, June 1995.

[15] NAS Parallel Benchmarks, http://www.nas.nasa.gov/NAS/NPB.

[16] NOW, http://now.cs.berkeley.edu.

[17] J. Schopf and F. Berman, "Performance Prediction in Production Environments," in *Proceedings of IPPS/SPDP '98*, to appear.

[18] V. Sunderam, "PVM: A Framework for Parallel Distributed Computing", *Concurrency: Practice and Experience*, vol. 2, n. 4, pp. 315–339, December 1990.

[19] J. Weissman, "The Interference Paradigm for Network Job Scheduling", in *Proceedings of the Heterogeneous Computing Workshop*, pp. 38-45, April 1996.

[20] D. Whitley, T. Starkweather, and DUAnn Fuquay, "Scheduling Problems and Traveling Salesman: The Genetic Edge Recombination Operator," in *Proceedings of International Conference on Genetic Algorithms*, 1989.

[21] R. Wolski, "Dynamically Forecasting Network Performance to Support Dynamic Scheduling Using the Network Weather Service," in *Proceedings of the 6th High-Performance Distributed Computing Conference*, August 1997.

[22] X. Zhang and Y. Yan, "A Framework of Performance Prediction of Parallel Computing on Non-dedicated Heterogeneous Networks of Workstations", in *Proceedings of 1995 International Conference of Parallel Processing*, vol. I, pp. 163-167, 1995.

Biographies

Silvia M. Figueira was born in Rio de Janeiro, Brazil. She received both her B.S. and M.Sc. degrees in Computer Science from the Federal University of Rio de Janeiro, Brazil, in 1988 and 1991, respectively, and her Ph.D. degree in Computer Science from the University of California, San Diego in 1997. She was involved in research as a member of the technical staff of NCE/UFRJ (The Computing Center of the Federal University of Rio de Janeiro) from 1985 to 1991. Currently, she is a Postdoctoral Fellow at the Computer Science Department of the University of California, San Diego. Her current research interests are in high-performance distributed computing.

Francine Berman is Professor of Computer Science and Engineering at U. C. San Diego and a Senior Fellow at the San Diego Supercomputer Center. She received her Ph.D. in 1979 from the University of Washington. Her research focuses on application scheduling in metacomputing, and programming environments, models and tools which support high-performance computing on distributed networks. Dr. Berman has participated on numerous Program and Conference Committees and will serve as co-Chair of the 1999 High Performance Distributed Computing Conference. She currently serves on the editorial boards of IEEE Transactions on Parallel and Distributed Computing, the Journal of Parallel and Distributed Computing, SIAM Review, and as Area Editor for Metacomputing at The Journal of Supercomputing.

Specification and Control of Cooperative Work in a Heterogeneous Computing Environment

Guillermo J. Hoyos-Rivera[1], Esther Martínez-González[1], Homero V. Ríos-Figueroa[2]
Víctor G. Sánchez-Arias[2], Héctor G. Acosta-Mesa[3] and Noé López-Benítez[4]

[1]Maestría en Inteligencia Artificial
Universidad Veracruzana
Sebastián Camacho # 5
Xalapa, Veracruz 91000 MEXICO
Ph. (52 28) 17 29 57 Fax. (52 28) 17 28 55
{ghoyos, emartine}@mia.uv.mx

[2]Laboratorio Nacional de Informática Avanzada, LANIA, A.C.
Enrique C. Rébsamen # 80, colonia Isleta
Xalapa, Veracruz 91090 MEXICO
Ph. (52 28) 18 13 02 Fax. (52 28) 18 15 08
{hrios, vsanchez}@xalapa.lania.mx

[4]Texas Tech University
Computer Science Dept.
College of Engineering
Lubbock, Texas 79410
Ph. (806) 742 1194 Fax (806) 742 3519
nlb@ttu.edu

Abstract

The implementation of an interface to support cooperative work in a heterogeneous computing environment is based on previously proposed definitions referred to as Cooperative Work Model (CWM) and Cooperative Work Language (CWL). The Interface for Cooperative Work (ICW) and the Graphical Interface for Cooperative Work (GICW) are the main two components of a tool useful in the set up and control of a cooperative working environment in a general purpose heterogeneous computing platform. This tool is described in this paper as well as some desired characteristics to improve its effectiveness. The specification and control of a virtual parallel machine are illustrated with an algorithm for 3D-reconstruction from two stereoscopic images. Test results on this application are also reported.

1. Introduction

Cooperative work involves the coordination of several tasks during their execution. All tasks share a common goal and cooperation rules coordinate their actions that in turn use communication primitives to make their interaction possible. There are two important factors behind the motivation of this work. The first one is that many problems can be organized as a set of cooperative modules that could be executed in parallel. The second

[3] *Acosta Mesa is now with the Universidad Tecnológica de la Mixteca, Huajuapan de León, Oaxaca, Méx ico*

0-8186-8365-1/98 $10.00 © 1998 IEEE

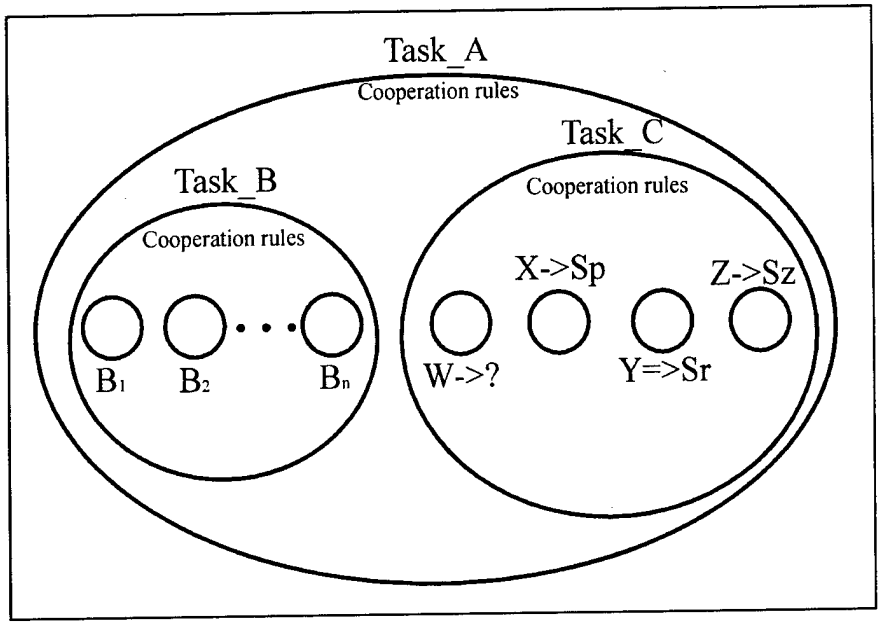


Figure 1. Hierarchical Layout of Task Processor Assignments

factor is the recognition that message passing computation is becoming more accessible today. It is no longer necessary to buy expensive devices to gain access to large computational power. Existing general purpose networks can now be used for cooperative work. These factors lead to the definition of a *Cooperative Work Model* (*CWM*) [18, 23], and a *Cooperative Work Language* (*CWL*) [19] with the purpose of making the specification of cooperative work, its parallelism and distribution easier. *CWM* and *CWL* are inspired upon the definition of *Communicating Sequential Processes* (*CSP*) [6], and the basic notions of *processes* and *pipes* used in *Unix* [17]. The *Interface for Cooperative Work* (*ICW*) and the *Graphical Interface for Cooperative Work (GICW)* are the main two components of the tool implemented based on *CWM* and *CWL*. *ICW* is useful in the set up and control of a cooperative working environment in a heterogeneous computing platform. A hierarchical specification of processes makes *ICW* different from other schemes such as *Cluster-M* [3] and *HeNCE* [2]. The main objective of *Cluster-M* is an efficient mapping of tasks into a set of processors. *HeNCE*, on the other hand, seeks the specification of tasks given in the form of a task graph such that parallelism is exploited. However, the nodes of the task graph refer to lower level specification such as procedures or routines. *ICW* allows a recursive refinement such that lower granularity is also possible when the application so requires it. Other tool that can also be compared with *ICW* is the *Wisconsin Wind Tunnel* (*WWT*) [16]. Unlike *ICW*, *WWT* is targeted to shared-memory oriented applications and can be used to simulate the behavior of hardware systems under design.

This paper describes the implementation of *ICW* and *GICW*. First, in section 2 we review the basic elements of the *CWM* model, i.e., tasks, cooperation rules, and intertask communication. In section 3 and 4, implementation issues are discussed. Section 5 deals briefly with the stereoscopic image reconstruction and section 6 reports some results obtained exploring task distribution schemes using the tool presented in this paper.

2. The Cooperative Work Model

The notion of cooperative work as a set of interrelated tasks arranged in a hierarchical structure was behind the proposed *CWM* model. In a distributed heterogeneous computing environment, some if not all tasks in the model can be executed in parallel and have the capability to communicate with each other by explicit message passing. The execution environment of tasks will be governed by predefined *cooperation rules*. Interaction among tasks will take place by using some established communication primitives.

Tasks may be assigned to any suitable host in the system. Figure 1 describes a possible hierarchical arrangement of sets of tasks. *Task_A* consists of its own execution environment and the execution environment of *Task_B* and *Task_C*. *Task_B* consists of its own execution environment and that of n replicated tasks denoted *as* B_i. Finally *Task_C* consists of its own execution environment and that of tasks X in host S_p, Y in any host of architecture

S_r, Z in host S_z and W. Tasks X, Y, Z, W and the n copies B_i do not contain other tasks inside them. Task W is a special case. It should be executed in a particular host that is not yet known. A search is required to determine the location of such a task. Note that this specification of tasks is different from that used for *Cluster-M* [3]. *CWL* specifies a hierarchical order governed by the cooperation rules between tasks and without regard at this point to any allocation scheme.

2.1. Tasks

The minimal work unit is the task. It is considered a completely executable program (a process) following the binary format of the operating system under which it was created.

By definition the tasks specified have the following characteristics:

- Each task starts and ends execution at some point in time,
- When a task starts execution, optionally receives some input parameters,
- No task shares memory with any other task, and
- The only way to share information with other tasks is by explicit message passing.

Any task may be classified according to the four different criteria described in the next paragraphs.

Types of tasks. A task can be a generic task or a CW task. A generic task is any general-purpose program that can be executed by writing the command name in the operating system prompt, like /bin/ls, /bin/cp, $HOME/bin/print, etc. This kind of task does not have the need to communicate with other tasks. A CW task is a compiled executable program explicitly written for our interface.

Level in the hierarchy. Under these criteria, any task can be of any two types: atomic or composed. An atomic task will be any executable program. It can be either, a CW task or a generic task. Atomic tasks will consist only of its own execution environment. A composed task can only be a CW task. A composed task consists of its own execution environment and that of one or more atomic or composed tasks spawned by it. A composed task has its own executable code. The tasks executed by a composed task will be called members of a composed task, and the composed task executing other tasks will be referred to as the caller task.

Place of execution. According to the possible places where tasks can be executed, they can be classified as explicitly located or not explicitly located tasks. A task not explicitly located is executed in any host of a virtual parallel computer. The place where these kinds of tasks are to be executed will be determined dynamically at runtime. An explicitly located task will be executed always in the same host, or a set of hosts of the same architecture. This is due to any of three reasons: (1) the executable program exists only in one host of the system, (2) the executable program was compiled for a particular architecture or (3) it is desirable to execute the program in some particular host because it may be the most suitable. A special case occurs when a task can be explicitly located but the host where the corresponding executable program resides is unknown. In this case a locator dynamically finds the host where the executable program resides.

Number of copies in a concurrent execution. Any replicated task can be explicitly located or not explicitly located. If they are explicitly located, then all copies of the task will be executed concurrently in the same host or in a subset of hosts of a specified architecture. Otherwise, each copy will be executed in any host of the system.

2.2. Cooperation Rules

Every member task will have associated with it a cooperation rule to control its execution. Three basic cooperation rules are defined: *SYNCP*, *ASYNCP* and *SEQ*.

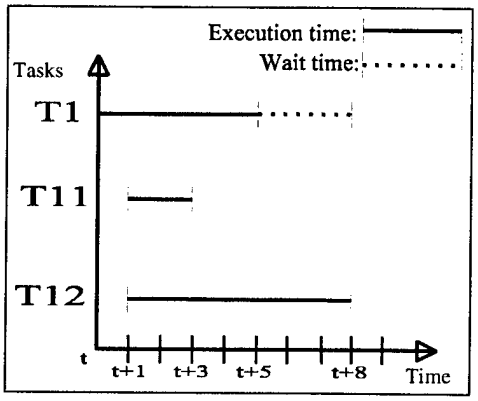


Figure 2. Behavior of SYNCP

SYNCP stands for *Synchronous Parallelism.* When some tasks are ruled by *SYNCP* all of them are started at the same time and keep working concurrently. The caller task will not end until all the member tasks have terminated. However, any member task does not depend on its caller task, nor on the tasks that were spawned at the same time. As an example, consider the expression:

T1:SYNCP[T11, T12]

where *T1* is the caller task and the member tasks will be *T11* and *T12*. Task *T1* will not end until both *T11* and *T12* have terminated. Figure 2 describes *SYNCP*.

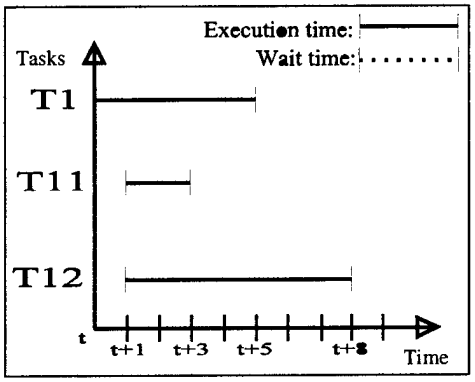


Figure 3. Behavior of *ASYNCP*

ASYNCP stands for *Asynchronous Parallelism*. This rule operates almost in the same way as *SYNCP*, but in this case the caller task does not have to wait for all the member tasks to terminate. Any task, including the caller task can terminate without having to wait for the termination of any other task working under this rule. The same example posed for the last rule is useful for this one, but with the difference that *T1* will be able to terminate independently of the termination time of tasks *T11* and *T12*. Figure 3 describes the behavior of this rule.

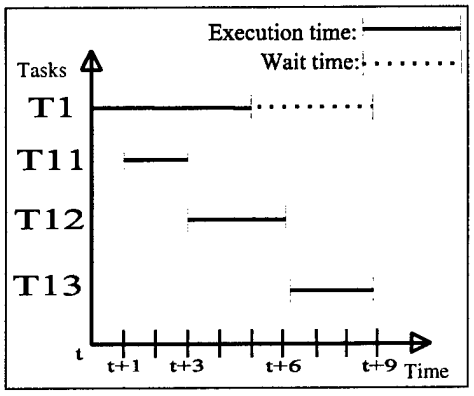


Figure 4. Behavior of *SEQ*

SEQ denotes a sequential execution. When using this rule on a series of member tasks, the next task to be executed will not be started until the previous one has finished. Figure 4 depicts its operation.

2.3. Communications

Each cooperating task in a *CWL* program has a *communication rank* that specifies those tasks that communicate with it. The communication rank of a task *T* running in *ICW* is formed by:

- The composed task that spawned *T*
- All the tasks spawned at the same time as *T*
- All the tasks spawned by *T*

Messages sent are saved in a buffer that is accessed by the destination process when it is ready. If the message is not yet in the buffer, the receiving process must wait until it arrives.

3. Implementation

ICW maps the cooperative work defined by the model into a specific distributed heterogeneous environment. The interface provides all the necessary mechanisms to distribute, replicate and locate any task to be executed. The execution of the tasks will be monitored and controlled to guarantee a complete execution of all the programs or, in case of failure the interface will report it. The goal of this feature is to make the debugging process easier. To decide which tools to use in the *ICW* implementation, several alternatives were analyzed. Two of these alternatives include *LAM* (*Local Area Multicomputer*) [14] and *PVM* (*Parallel Virtual Machine*) [4]. *LAM* is a full extension of the *MPI* [13] (*Message Passing Interface*) standard. We initiated the implementation of *ICW* using *PVM* mainly because it offered several desirable features and it was available. Currently the interface is being updated, through *GICW*, to work with both, *PVM* and *LAM* at the user's choice.

3.1. Cooperative Work Language

A typical program written in *CWL* is composed of six sections:

- Architectures declaration (`ARCHS`)
- Hosts declaration (`HOSTS`)
- Tasks declaration (`TASKS`)
- Generic tasks declaration (`GTASKS`)
- Root task declaration (`ROOT`)
- Cooperative Work declaration (`CW`)

The first two sections (`ARCHS` and `HOSTS`) contain the list of names of architectures and hosts respectively. These sections can be omitted if there are not explicitly located tasks.

The `TASKS` section contains the declaration of all the tasks that are involved in the execution. This section cannot be omitted.

The `GTASKS` section lists all the generic tasks. The interface will not accept any declared generic task to be used as a composed task. The existence verification of this type of tasks is carried out at runtime. This section is optional.

In both sections `TASKS` and `GTASKS` it is possible to indicate that a task is going to be executed in a particular

host or a particular architecture. The operator → indicates that a task is to be executed in a particular host and the operator ⇒ indicates a particular architecture.

The ROOT declaration indicates which task contains, directly or indirectly, other tasks. In other words, the task declared as ROOT is the superset in the hierarchy. The ROOT task must be declared previously in the TASKS declaration. If not, or if it is declared in the GTASKS declaration, it will not be accepted.

Finally, the *CW* declaration indicates the structure of the execution, and the rules that control the relation between tasks. Every task listed in this declaration will be validated against the tasks declared in the TASKS and GTASKS sections. A typical *CWL* program is shown in Figure 5.

3.2. Writing *CW* tasks

Programs for the interface are generic *C* programs with the only peculiarity that they must be compiled with the preprocessor directive `#include "ICW.h"`, which contains all the necessary definitions to control the execution and all the functions to communicate between tasks. This file includes two special functions: void `ICW_init(int argc, char *argv[])` and void `ICW_end(void)`.

```
ARCHS: SUN4,LINUX;

HOSTS: afrodita, hefestos,
 cronos;
TASKS:
 test,test1->cronos,test2,
 test3,test4,test5,
 test6=>LINUX;
GTASKS: test7;
ROOT: test;
CW {
 test: SYNCP[test1,test2];
 test2: SEQ[test3,test4];
 test3: ASYNCP[test5(3)];
 test4: SYNCP[test6(10),test7];

}
```

Figure 5. Typical CWL program

`ICW_init()` must be called as the first executable instruction of the function main() in every *CW task*. This function will receive from the caller the hierarchy of tasks and then execute them, sending to every task the appropriate information to trigger execution. `ICW_init()` must be called with the standard arguments received by the *C* program because that information is used by the interface to determine the operating environment.

The last instruction of an *ICW* program must be `ICW_end()`. This function will test the correct termination of all the tasks spawned by the current task and will terminate with the appropriate exit code. To avoid conflicts and unpredictable behavior every internal function and variable of the interface start with the characters "ICW_". A minimal expression of an *ICW* program is shown in Figure 6.

Communication functions. All communication primitives to be used in CW programs will be limited by the communication rank of the tasks. To establish a communication with any task outside the rank it will suffice to either use some tasks as intermediaries to send messages or use directly the PVM identification and communication functions to determine the identity of the target task, and to send messages, respectively. For example, referring to Figure 1, task X may need to send a message to task B_1 but this task is outside the communication rank of X.

The *ICW* communication functions have been defined for each possible data type to be transmitted and all of them follow the same standard.

```
#include <stdio.h>
#include "ICW.h"

void main(int argc, char *argv[]){
  ICW_init(argc, argv);
  ICW_end();
}
```

Figure 6. A minimal expression of a ICW task program

Typical functions to send and receive data have the following prototypes:

- *ICW_send_<datatype>(char *dest, int copy, int id, <datatype> *buffer, int len)* where <datatype> is a valid simple datatype in the programming language.

- *ICW_send_string(char *dest, int copy, int id, char *buffer).* This function is mainly used to send strings.

- *ICW_receive_<datatype> (char *orig, int copy, int id, <datatype> *buffer, int len, long tout).*

• *ICW_receive_string (char *orig, int copy, int id, <datatype> *buffer, long tout).*

Parameters. The parameters **char *dest** or **char *orig** indicates with an identifier of a program name, the task sending or receiving messages. The valid identifiers are `ICW_member`, `ICW_caller`, `ICW_next`, `ICW_previous` or a program name. If a program name is used, *ICW* will attempt to find a program within the communication rank of the task that calls the communication function with the specified name.

In the case of a replicated task with a copy parameter different from zero, the message will be sent to or received from the *n*th copy of the replicated task.

If the `ICW_member` is used there are three possible results. To send a message and if the copy parameter is zero, the message will be sent to all the member tasks. To send or receive a message, and if the copy parameter is say *n* (different from zero), the message will be sent to or received from the *n*th task of the member tasks. To receive a message and if the copy parameter is zero, a message from any of the member tasks will be accepted.

If `ICW_caller` identifier is used the message will be sent to, or accepted from the caller task.

With `ICW_next` the message will be sent to or accepted from the next task in the same level in the hierarchy. The same happens with `ICW_previous`, but in this case, it will be sent to or accepted from the previous task in the same level in the hierarchy.

int copy indicates the number of copies of a replicated task, or the number of member tasks a message is sent to, or received from.

int id is an integer number that must match the sending and receiving processes. It is used as a validation of the message. A value of *-1* in the receiver tells the process to receive a message with any id number.

<type> *buffer is a pointer to the buffer that contains the data to be sent or where it will be received. Its type must match the type of the data in transit.

int len indicates the length of the data buffer.

Receive functions have an additional parameter:

long tout indicates how long a process should wait for a message to arrive. If it is zero the waiting time defaults to 300 seconds.

3.3. Execution of *CWL* programs

CW tasks must be executed through a *CWL* program. This program, although compiled, does not generate any executable code. If the execution of all tasks is successful the execution of the *CWL* program will be successful. If only one of the tasks fails the overall execution environment fails. The execution process is divided into two stages. One stage is the compilation of the *CWL* program, and the other stage is the execution of all the tasks involved in the cooperative work specified.

Compilation stage. The first step of the compilation stage attempts to contact the *PVM* daemon. If it is not possible to do it, the interface attempts to start it up. If this is not possible the program will not compile and the interface exits with an appropriate error code. *PVM* uses a hosts file to know which hosts will be included in the parallel virtual machine. The hosts file name is `.icwhosts` and resides in the user home directory. The compiler will check that all the declared architectures in the `ARCHS` declaration and hosts declared in the `HOSTS` declaration really exist in the *PVM* environment, otherwise, the interface will exit with an error. Next, the compiler will compile the `TASKS` and `GTASKS` declarations. The existence of executable programs in the hosts of the virtual computer is verified at runtime. However, the compiler will check the consistency of the declarations. The compiler will also check that all the architectures and hosts used in the declaration of the explicitly located tasks had been previously declared in the corresponding sections. Finally, it will check that the `ROOT` task has been declared in the `TASKS` declaration as well as all the tasks referenced in the `CW` declaration. The result of compiling the `CW` section will be an internal representation of the hierarchy followed during the execution of tasks. This hierarchy is used at runtime to determine the behavior of every part of the execution process.

Execution stage. This stage consists of the execution of the entire hierarchy of tasks involved in the cooperative problem. The first to be executed is the `ROOT` task, which will be forked and enrolled as a *PVM* process. The interface will wait for the end of the execution of the `ROOT` task. After three unsuccessful execution attempts the interface will exit with an error. As previously mentioned, the complete execution will be successful only if all its components are executed successfully. If at least one task fails, the overall execution process fails.

4. The Graphical Interface

An option to build and execute a *CWL* program is through the *Graphical Interface for Cooperative Work* (*GICW*). With the *GICW* is possible to create an efficient grouping for the objective Cooperative Work in an interactive and dynamic way. Figure 7 shows a view of *GICW*.

The *GICW* offers two operation modes. The first mode manages information elements for the Cooperative

Work of an application via a set of windows. It integrates the tasks (proper and generic), hosts and architectures.

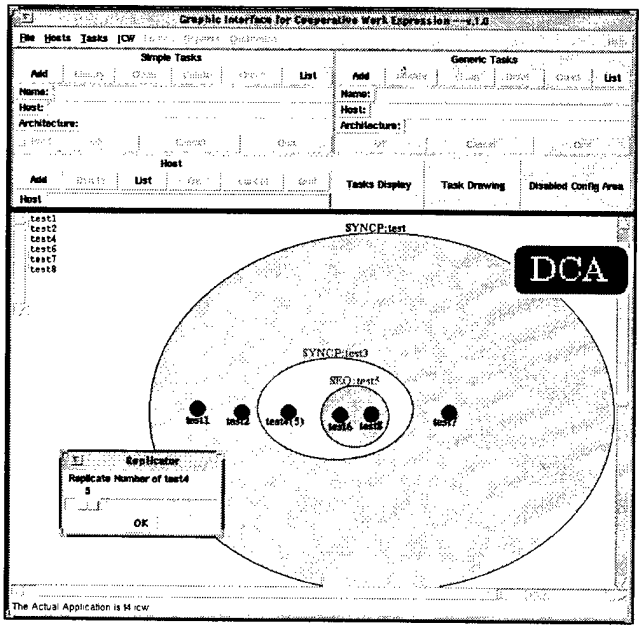


Figure 7. Graphic Interface for Cooperative Work

The second is the Graphic Configuration mode. It integrates all operations adding a *Dynamic Configuration Area* (*DCA*). The *DCA* shows the task representation looking like the graphs in the *CWM* shown in Figure 1. In the central part of this area appears the root task represented by an oval with its name and its cooperative rule. Within this oval is possible to integrate composed and atomic tasks required in the cooperative work of the current application.

A composed task is also represented with a circle with the name of the task and its cooperation rule. Small circles are used to represent atomic tasks. To add one task in the configuration it is selected from the box list located in the left side of the *DCA*. The selected task is then dragged into the root task area or into the area of a previously created composed task.

To create or modify tasks, hosts, or architectures, it is important to use the corresponding entries that appear in the upper side of the configuration area. These windows are useful to specify directly weather a task will execute in a particular host or architecture.

To integrate the number of copies of an atomic task is necessary to select it from de *DCA* and adjust the number of copies in the window that appears for this purpose. Code generation is performed according to the objects appearing in the *DCA* and their grouping. The output file generated is identified with the application name and the extension `.icw`. This file contains the *CWL* specification of the cooperative work.

The option *ICW Execution* is selected from the menu and a window appears to display execution results.

The implementation is based on the scripting language *Tcl* and its graphical toolkit *Tk* (version 8.0) which are widely portable and allow easy GUI programming [15].

The *GICW* has no validation mechanism for the existence of tasks. A parser is used when a configuration file is loaded. The *GICW* extensions are based on the *MPI* implementation of *ICW*. The implementation integrates 1) a state monitor mechanism of the tasks that compound the current cooperative work application, 2) the search in alternative paths that are not in the *PATH* environment variable, and 3) the dynamic configuration of the host in the virtual machine.

5. Application: 3D Reconstruction

One of the most important features of human vision is its capacity for perceiving a three dimensional world. This perceptual capacity is achieved through a highly evolved visual system composed of cooperative visual modules, which are able to recognize objects and describe the layout, and motion of our surroundings [21]. One visual module that is most relevant for the perception of depth is *stereopsis* [10]. This visual module takes as input two images of a scene taken from different locations (for example, one taken by the left eye and the other by the right eye) and computes the correspondence of features which most likely originate from the same 3D surface patch (Figure 8). From the features correspondence is possible to obtain the depth at these points [7].

We describe a distributed implementation of the Pollard, Mayhew & Frisby (PMF) algorithm for stereoscopic reconstruction. Our implementation has been coded in *ICW* and runs on a network of *SUN* and *Silicon Graphics* workstations [1]. The main stages of the PMF algorithm for stereoscopic reconstruction [10] are the following:

- Edge detection. The points with highest changes in intensity are detected in each digital image. This can be achieved with a variety of edge detectors, such as the Marr-Hildreth operator, Canny edge detector or Sobel's [5]. For simplicity, we have taken this last option.
- Stereoscopic correspondence. This is the core part of the whole algorithm that finds the most likely correspondences between edges in the left and right images.
- Reconstruction. Once the correspondences have been found it is possible to evaluate simple arithmetic expressions to find the depth at these locations.

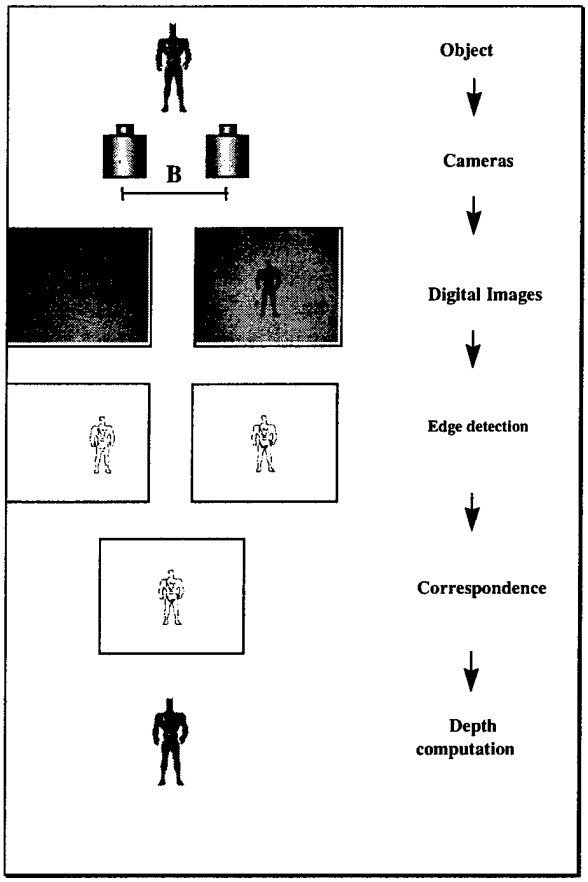


Figure 8. Stages in stereoscopic reconstruction

We take images from two cameras with parallel viewing directions to simplify the problem of stereo correspondence, because in this case, the corresponding features lie in epipolar lines. This means, that they are approximately in the same raster line in the two images.

Using the constraint that corresponding features usually have a *disparity gradient DG* less than one, we apply a search process to find the best matches. The disparity gradient is defined for a pair of matches, where each match associates one feature in the left image with one feature in the right image.

If feature $Pl = (plx,ply)$ in the left image is matched with feature $Pr = (prx,pry)$ in the right one, and feature $Ql = (qlx,qly)$ in the left is matched with feature $Qr = (qrx,qry)$ in the right as shown in Figure 9, then the *disparity D* for the match *(Pl, Pr)* is obtained as follows:

$$D(Pl,Pr) = plx - prx.$$

The *disparity difference DD* for the pair of matches *(Pl,Pr)* and *(Ql,Qr)* is just the disparity for the *P* match minus the disparity for the *Q* match. That is:

$$DD((Pl,Pr),(Ql,Qr)) = d(Pl,Pr) - d(Ql,Qr)$$

Now imagine the two images superimposed. The *cyclopean separation CS* is the distance from the mid-point of the line joining *Pl* and *Pr* to the mid-point of the line joining *Ql* and *Qr*. The gradient *DG* is the absolute value of the disparity difference divided by the cyclopean separation.

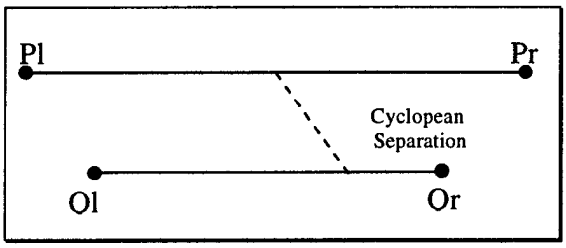


Figure 9. Geometry of the disparity gradient

In Figure 9, the disparity difference is the difference in length between the two horizontal lines. The cyclopean separation is the length of the slanting line. The *DG* can be expressed as follows:

$$DG = DD/SC \leq 1$$

If a point *(X,Y,Z)* projects at *(xl,y)* and *(xr,y)* in the left and right image respectively, we can find its position in space, in terms of the disparity xl – xr, using the formulas [7]:

$$X = \frac{B(xl+xr)}{2(xl-xr)}$$

$$Y = \frac{By}{xl-xr}$$

$$Z = \frac{Bf}{xl-xr}$$

where *B* is the separation between the camera's centers and *f* is the focal length.

6. Comparative results

The distributed implementation consists of dividing each image in bundles of lines and allocating a bundle to each workstation. Once a bundle is processed, the results are returned and concentrated by the host computer for display as described in Figure 10.

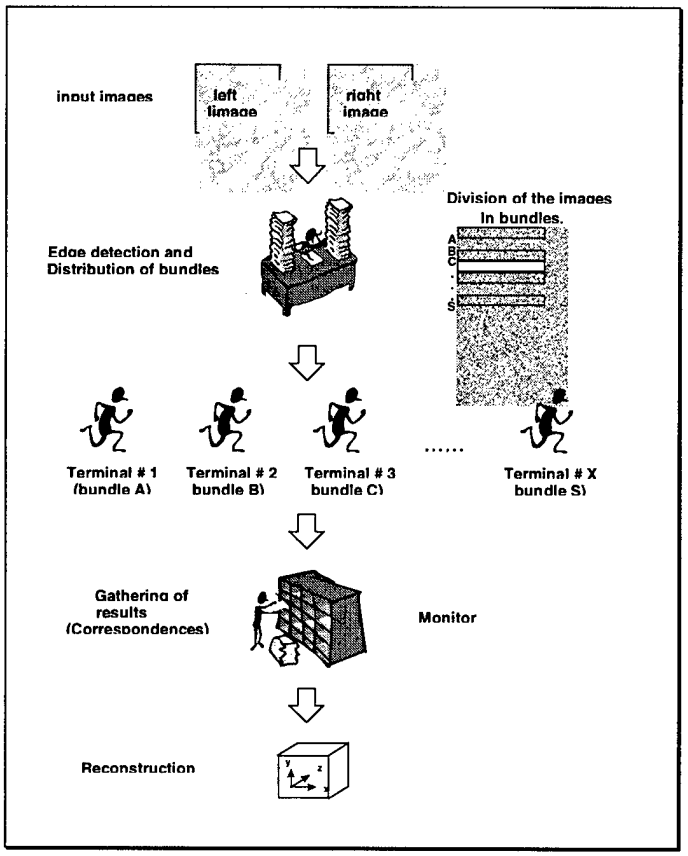


Figure 10. Distribution of tasks for stereoscopic correspondence

Each processed image is composed of 240 rows and 320 columns. The size of each bundle is obtained as the number of rows divided by the number of available workstations. In our implementation the size of the bundle can vary up to *240/n* lines, where $n = 1, 2, \ldots, 12$ is the number of workstations. For a uniform number of features in each bundle, the lines that compose each bundle are not taken consecutively but every *240/n* lines. Some results of the reconstruction are shown in Figure 11.

Two sets of tests were carried out. In the first set only 12 SUN units (models ELC, ILC and IPC with 24 Mbytes of RAM) were used. For the second set of tests Silicon Graphics (SGI) machines (Indy R4600 with 32 Mbytes of RAM) where introduced. Under the second scheme, a SUN unit distributes tasks to SGI units (labeled 1 to 5). The results obtained are shown in Figures 12 and 13. Both figures compare results obtained under no workload conditions and normal workload conditions.

Figure 12 shows a monotonic improvement in the execution time (this behavior is more consistent under a normal workload condition) up until the number of units reaches 10. An increase in the number of workstations does not show any improvement in the overall execution time. At this point, very likely the communication costs involved with further partitioning of the application upset any gain in execution times. In this regard similar behavior can be observed with the combination SUN and SGI workstations in Figure 13. Naturally, the introduction of SGI units renders a dramatic improvement in the execution time. However, particularly in the case of no workload conditions, performance remains constant indicating again the effect on communication costs. Under normal workload conditions, improvements are noticeable with additional units.

The objective of these experiments is to demonstrate the feasibility of using *ICW* to execute distributed applications. The results highlight the need to incorporate appropriate task allocation and scheduling heuristics [3, *11,12,20*] to map the set of tasks in the application to suitable units in the system and improve execution times. The integration of these schemes will make ICW a complete and useful tool in the analysis and implementation of large-scale parallel and distributed applications.

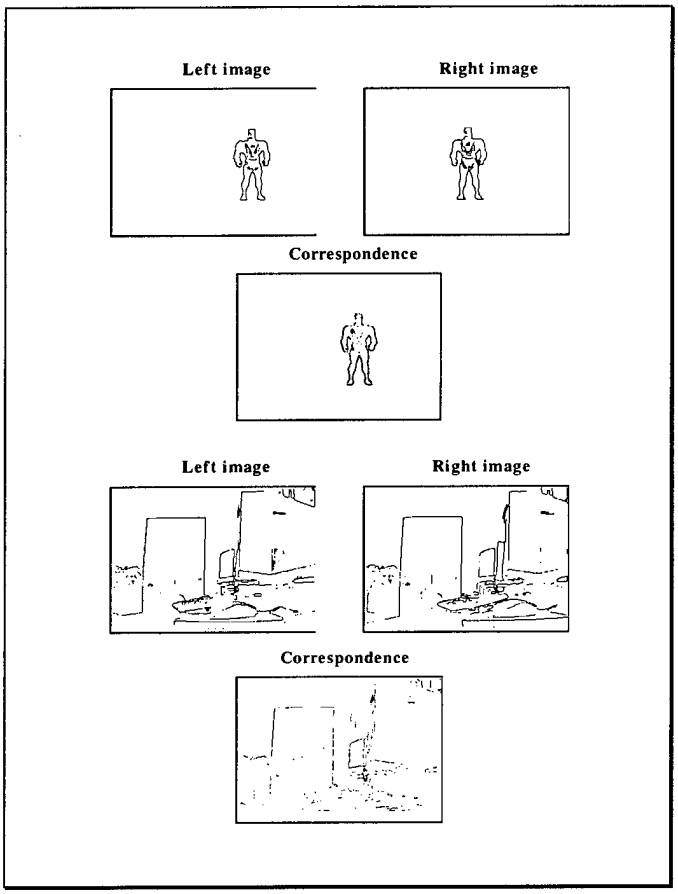


Figure 11. Results of the reconstruction

7. Conclusions and future work

At this time we have defined a model to specify cooperative work and completed the implementation of the first version of the interface (*ICW*). We are currently working on *LAM 6.1* and refining some management aspects of *CW* applications.

ICW is the first implementation of the *CWM* and the *CWL* models and although the project is at an early stage, trying to define the most desirable features has been the most time consuming endeavor. This work however, demonstrates the feasibility of the model, and will be the base for the analysis and implementation of a more complete *CWL*. The tool proposed facilitates the introduction of scientists into the world of parallel and distributed processing as it provides an easy interface to the specification of parallelism, writing, and debugging of communicating programs using installed general purpose networked resources.

However, to optimize performance the interface must be able to evaluate hosts configurations and detect the states of those processing units used in the distribution.

The tool has been written to deal with *C* programs. An upgraded version will incorporate transputers to facilitate the specification of lower level parallelism. Another expected development is to improve the mechanisms to detect and, if possible, recover from failures. Yet another important future development calls for the integration of task assignment heuristics and their evaluation to achieve a much improved task distribution in terms of execution times and resource utilization. In terms of future applications for which ICW will be used include cooperative virtual environments and gesture recognition [22] algorithms.

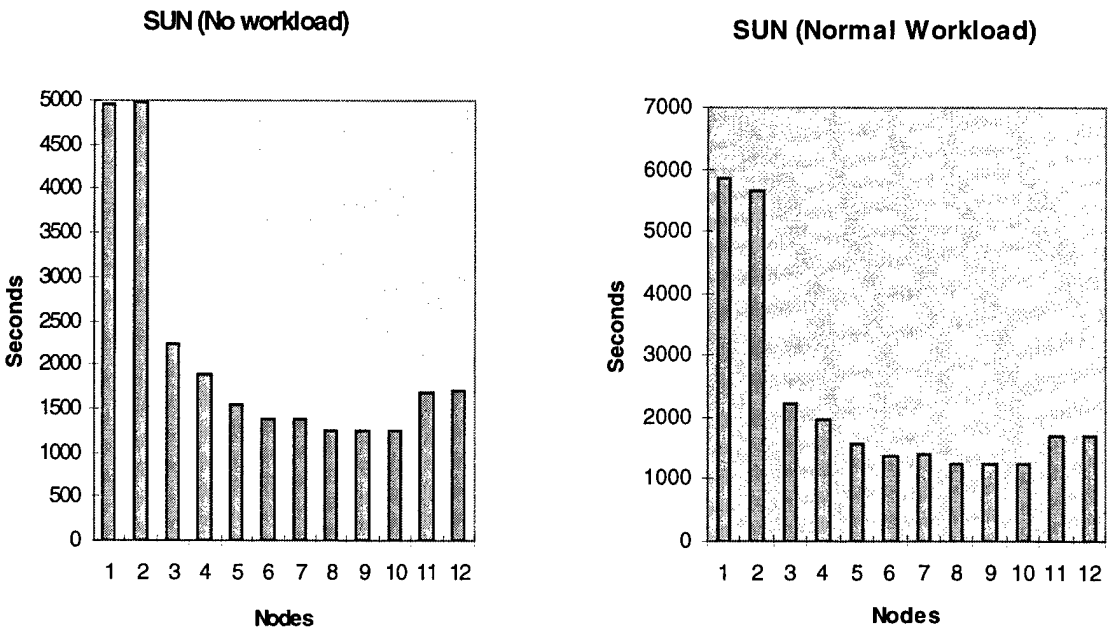


Figure 12. Execution times on a homogeneous system consisting of SUN workstations only.

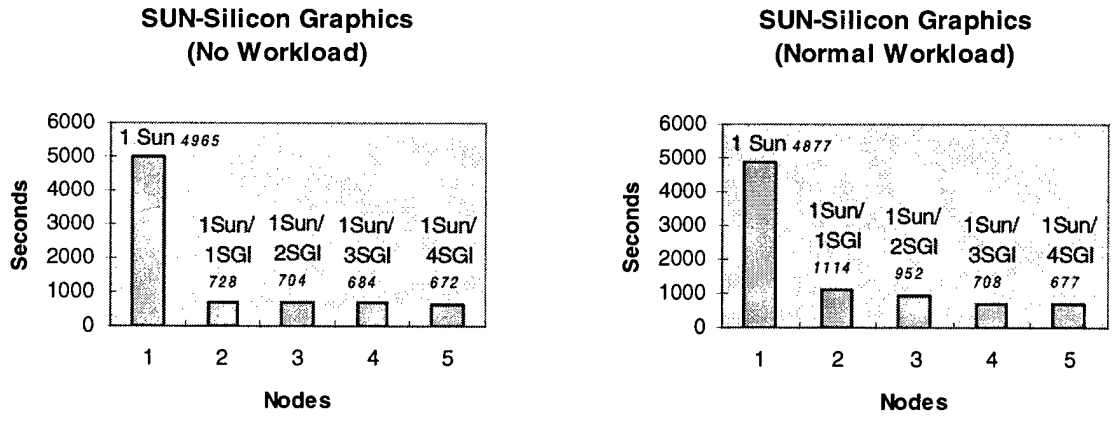


Figure 13. Execution times on a combined SUN-SGI configuration.

References

[1] H.G. Acosta-Mesa, H.V. Rios-Figueroa, *"Implementación distribuida de un módulo de reconstrucción estereoscópica"*, Maestría en Inteligencia Artificial, Universidad Veracruzana - LANIA, 1996

[2] A. L. Beguelin, J. J. Dongarra, G. A. Geist, R. Mancheck, and K. Moore "HeNCE: a Heterogeneous Network Computing Environment", University of Tennessee, Computer Science Dept. Technical Report No. 93-205, August 1993.

[3] M. M. Eshaghian and Y-C. Wu, "A Portable Programming Model for Network Heterogeneous Computing", in *Heterogeneous Computing*, Mary M. Eshagian, ed., Artech House, Inc., 1996.

[4] A. Geist, et al. "PVM: Parallel Virtual Machine. A User's Guide and Tutorial for Networked Parallel Computing", MIT Press. 1995

[5] R.C. González, R.E Woods, "Digital Image Processing", (3rd. edition) Addison-Wesley, 1992

[6] *C.A.R. Hoare,* "Communicating Sequential Processes", *Communications of the ACM*, 21(8): 666-667 1978.

[7] B.K.P, Horn, "Robot Vision", *The MIT Press*, 1987

[8] G.J, Hoyos-Rivera, V.G. Sánchez-Arias "Proposal of an Interface to Support Cooperative Work in a Distributed System Environment". I Encuentro de Computación. Taller de Sistemas Distribuidos y Paralelos. Memorias. Querétaro, Qro. México. September 1997. SMCC, SMIA, UNAM, Asoc. Filosófica de México, SMI, UAQ.

[9] G.J. Hoyos-Rivera, "Propuesta de una Interfaz para el Apoyo al Trabajo Cooperativo en un ambiente de Arquitectura Paralela y Sistemas Distribuidos", Maestría en Inteligencia Artificial, Universidad Veracruzana - LANIA, 1997

[10] J.E.W. Mayhew, J.P. Frisby, "3D Model Recognition from Stereoscopic Cues", *The MIT Press*, 1991

[11] A. R. McSpadden, N. Lopez-Benitez, "Stochastic Petri Nets Applied to the Performance Evaluation of Static Task Allocations in Heterogeneous Computing Environments", *IEEE Heterogeneous Computing Workshop, 1997*, Geneva, Switzerland.

[12] D. A. Menasce, D. Saha, S.C. Silva Porto, V.A.F. Almeida, S.K. Tripathi, "Static and Dynamic Processor Scheduling Disciplines in Heterogeneous Parallel Architectures", *Parallel and Distributed Computing,* Vol. 58, 1995, pp. 1-18.

[13] Message Passing Interface Forum, MPI: A Message-Passing Interface Standard CRPC-TR94439, Center for Research on Parallel Computation, Rice University, April, 1994, http://netlib2.cs.utk.edu/papers/mpibook/mpibook.ps

[14] N. Nervin. "The Performance of LAM 6.0 and MPICH 1.0.12 on a Workstation Cluster. Ohio Supercomputer Center. Technical Report OSC-TR-1996-4 . Columbus Ohio.

[15] J. K. Ousterhout. Tcl and the Tk Toolkit. Addison-Wesley. Professional Computing Series. September, 1995.

[16] S. K. Reinhardt, M. D. Hill, J. R. Larus, A. R. Lebeck, J.C. Lewis, and D. A. Wood, "The Wisconsin Wind Tunnel: Virtual Prototyping of Parallel Computers", ACM Sigmetrics Conference, May 1993.

[17] D.M. Ritchie, K. Thompson. "The Unix Time-Sharing System", The Bell System Technical Journal 57 No. 6 page 2. Jul-Aug, 1984

[18] V.G. Sánchez-Arias, "Arquitectura para el apoyo al trabajo cooperativo basado en una red de sistemas paralelos y distribuidos", Reporte interno LANIA, R1-1124P-A, marzo 1996.

[19] V.G. Sánchez-Arias, G.J. Hoyos-Rivera. "Using PVM to Build an Interface to Support Cooperative Work in a Distributed Systems Environment". Lecture Notes in Computer Science. Proceedings of Recent Advances in Parallel Virtual Machine and Message Passing Interface 4th European PVM/MPI Users' Group Meeting. Cracow, Poland, November 1997. pp 127-134

[20] B. Shirazi, M. Wang, G. Pathak, "Analysis and Evaluation of Heuristic Methods for Static Task Scheduling", *J. of Parallel and Distributed Computing,* Vol. 10, 1990, pp. 222-223.

[21] S. Ullman, "Analysis of Visual Motion by Biological and Computer Systems" *Computer 14*, 57-69, 1981.

[22] K. Voss, H. V. Rios-Figueroa and J. Peña. "Head tracking by glasses detection". Proceedings of the Workshop on vision and robotics, National Computer Conference, Mexico, 1997.

[23] Sánchez-Arias, V. G. "Arquitectura para el apoyo al trabajo cooperativo basado en un ambiente de architectura paralela y systemas distribuidos", Proceedings 2nd. International Workshop on Parallel Processing, DEA-IIMAS-UNAM, Mexico, July 1996, pp 8-10.

Author Biographies

Guillermo J. Hoyos-Rivera received the BSc degree in Informatics in 1992, and the MSc degree in Artificial Intelligence in 1997 from the Universidad Veracruzana with honors. He has been a Lecturer in the Universidad Veracruzana as a Lecturer since 1991 and in several other schools and universities. Currently, he is an associate full-time Researcher in the Artificial Intelligence department in the Universidad Veracruzana and the Universidad Anahuac de Xalapa. His research interest include operating systems, computer networks, telecommunications, protocols and parallel and distributed processing. He is a member of the SMIA (Mexican Society of Artificial Intelligence).

Esther Martinez-Gonzalez received the BSc degree in Informatics from the Universidad Nacional Autonoma de Mexico (UNAM) in 1994. She is about to graduate with an MSc degree in Artificial Intelligence from the Universidad Veracruzana. Since 1994 she has been a Lecturer, first in the UNAM and currently at the Universidad Veracruzana. She is now engaged with the Research Coordination of the Universidad Veracruzana. Her research interest areas are in distributed systems and cooperative work.

Homero V. Rios-Figueroa received the BSc degree in Mathematics and the MSc degree in Computer Science from the Universidad Nacional Autonoma de Mexico (UNAM) in 1987 and 1989, respectively. The PhD degree in Computer Science and Artificial Intelligence from the University of Sussex, England, in 1994. From 1988 to 1990 he was a Lecturer with the UNAM. In 1994, he joined the Laboratorio Nacional de Informatica Avanzada (LANIA) in Xalapa, Mexico, where he is now a full time Researcher. His research interests include computer vision and virtual reality. He is a member of the SMIA (Mexican Society of Artificial Intelligence).

Victor G. Sanchez-Arias received the BSc degree in Control Comunications and Electronics and the MSc degree in Computer Science from the Universad Nacional Autonoma de Mexico (UNAM) in 1974 and 1976, respectively. He obtained the Diplomé D'Etudies Approfondies (DEA) in Informatics from the École Nationale Supérieure de l'Institut de Mathèmatiques Appliquées de Grenoble (IMAG), France. In 1982, he received the PhD degree in Informatics Engineering from the IMAG. He was with BULL, Paris, France from 1985 to 1988 where he was engaged in the research and development of networks, distributed systems and applications. He was a Researcher at the IMAG (1981-1984) and at the UNAM (1988-1991). Since 1992, he is titular Researcher and consultant at the Laboratorio Nacional de Informatica Avanzada (LANIA). He is a member of the SMIA (Mexican Society of Artificial Intelligence) and SMCC (Mexican Society of Computer Science). His research interest include distibuted and cooperative systems and parallelism.

Hector G. Acosta-Mesa received the BSc degree in Computer Systems from the Instituto Technologico de Veracruz. He also received his MSc degree in Artificial Intelligence from the Universidad Veracruzana. From 1991 to 1996 he was working for CFE (Comision Federal de Electriciad) in Xalapa, Veracruz. He is now a Researcher and Professor in the Universidad Technologica de la Mixteca in Oaxaca. Areas of research interest include robotics and computer vision. He is a member of the SMIA (Mexican Society of Artificial Intelligence).

Noe Lopez-Benitez received the BSc degree in Communications and Electronics from the University of Guadalajara, Guadalajara, Mexico. The MSc degree in Electrical Engineering from the University of Kentucky, and the PhD in Electrical Engineering from Purdue University in 1989. From 1980 to 1983, he was with the IIE (Electrical Research Institute) in Cuernavaca, Mexico. From 1989 to 1993, he served in the Dept. of Electrical Engineering at Louisiana Tech University. He is now a Faculty member in the Dept. of Computer Science at Texas Tech University. His research interests include fault-tolerant computing systems, reliability and performance modeling, and distributed processing. He is a member of the IEEE, the IEEE Computer Society,.and the ACM.

A Mathematical Model, Heuristic, and Simulation Study for a Basic Data Staging Problem in a Heterogeneous Networking Environment

Min Tan*, Mitchell D. Theys†, Howard Jay Siegel†, Noah B. Beck†, and Michael Jurczyk‡

*Cisco Systems, Inc.
170 West Tasman Drive
San Jose, CA 95134-1706
mintan@cisco.com

†Parallel Processing Laboratory
School of Electrical and Computer Engineering
Purdue University
West Lafayette, IN 47907-1285, USA
{hj, theys, noah}@ecn.purdue.edu

‡Department of Computer Engineering and Computer Science
University of Missouri at Columbia
201 Engineering Building West
Columbia, MO 65211, USA
mjurczyk@cecs.missouri.edu

Abstract

Data staging is an important data management problem for a distributed heterogeneous networking environment, where each data storage location and intermediate node may have specific data available, storage limitations, and communication links. Sites in the network request data items and each item is associated with a specific deadline and priority. It is assumed that not all requests can be satisfied by their deadline. 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. A mathematical model for the basic data staging problem is introduced. Then, a multiple-source shortest-path algorithm based heuristic for finding a suboptimal schedule of the communication steps for data staging is presented. A simulation study is provided, which evaluates the performance of the proposed heuristic. The results show the advantages of the proposed heuristic over two random based scheduling techniques. This research, based on the simplified static model, serves as a necessary step toward solving the more realistic and complicated version of the data staging problem involving dynamic scheduling, fault tolerance, and determining where to stage data.

This research was supported by DARPA/ISO and the Office of Naval Research under ONR grant number N00014-97-1-0804, and by NRaD under contract number N66001-96-M-2277, M. D. Theys was also supported by a Purdue Benjamin Meisner Fellowship and an Intel Fellowship.

Keywords: BADD, data staging, data management, Dijkstra's multiple-source shortest-path algorithm, distributed heterogeneous networking environment, heterogeneous computing.

1. Introduction

The DARPA Battlefield Awareness and Data Dissemination (BADD) program [12] includes designing an information system for forwarding (staging) data to proxy servers prior to their usage by a local application, using satellite and other communication links. The network combines terrestrial cable and fiber with commercial VSAT (very small aperture terminal) internet and commercial broadcast. This provides a unique basis for information management. It will allow web-based information access and linkage as well as server-to-server information linkage. The focus is on providing the ability to operate in a 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 program. An informal description of the data staging problem in a military application is as follows. A warfighter is in a remote location with a portable computer and needs data for planning 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 item is associated with a specific priority, where larger priority value implies

higher importance. It is assumed that not all requests can be satisfied by their deadline. Data staging involves positioning data prior to its use in decision making for facilitating a faster transfer time when it is requested.

Positioning the data before it is requested 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 [13]. 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 [12, 13]. The storage situation becomes even more difficult when copies of data items are allowed to reside on different machines in the network so that there are more available sources from which the requesting sites can obtain certain data [16], and so there is an increased level of fault tolerance, in cases of links or storage locations going off-line.

The simplified data staging problem addressed in this paper requires a schedule for transmitting data between pairs of nodes in the corresponding communication system for satisfying as many of the data requests as possible, with the high priority requests given precedence. Each node in the system can be: (1) a source machine of initial data items, (2) an intermediate node for storing data temporarily (e.g., routers or switches), and/or (3) a final destination machine that requests a specific data item. This problem comes under the topic of distributed heterogeneous computing [15] because nodes may have different storage limitations, different communication links available, different data available, and different data to request. The actual data staging problem is dynamic in nature, because in reality the network configuration can change, certain communication links may become unavailable, new data requests can be submitted sporadically, priorities of existing requests can be modified, and certain nodes in the network may fail.

This paper concentrates on solving a simpler version of the data staging problem in which all parameter values for the communication system and the data request information (e.g., requesting machines and network configuration) 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 due to storage capacity and communication constraints. Also, the fault tolerance issues mentioned above are not addressed. 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 the user can easily retrieve the data from this location. A heuristic is presented and evaluated that effectively satisfies this simplified data staging problem. This research, based on the simplified model presented here, serves as a necessary step toward solving the more realistic and complicated version of the data staging problem involving dynamic scheduling, fault tolerance, and determining where to stage data.

Section 2 provides overviews of work that is related to the data staging problem. In Section 3, a mathematical model for a basic data staging problem is introduced. Section 4 presents a multiple-source shortest-path algorithm based heuristic for finding a suboptimal schedule of the communication steps for data staging. This heuristic adopts the simplified view of the data staging problem described by the mathematical model. A simulation study is discussed in Section 5, which evaluates the performance of the proposed heuristic. A BADD-like network environment has been used in developing the parameters for conducting this simulation study. Section 6 summarizes this paper, giving the current status of this research on data staging, and plans for future work. A glossary of notation is included in Section 7 for reference purposes.

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. A problem that is, at a high level, remotely similar to data staging is the facility location problem [8] in management science and operations research. Under the context of the construction of several new production facilities, a manufacturing firm needs to arrange the locations of the facilities and plants effectively, such that the total cost of transporting individual components from the inventory facilities to the manufacturing plants for assembly is minimized. It is required that the firm makes several interrelated decisions: how large and where should the plants be, what production method should be used, and where should the facilities be located? If an analogy is made between (1) the plants and the destination nodes that make the data requests, (2) the individual manufacturing components and the requested data elements to be transferred, and (3) the facilities and the source locations of requested data, then at a high level the facility location problem has features similar to those of the data staging problem (e.g., use a graph-based method to reduce the facility location problem to a shortest-path or minimum spanning tree problem).

However, when examing the relationship between the facility location problem and the data staging problem carefully, there are significant differences. First, each component that a plant requests is usually not associated with a prioritizing scheme, while in the data staging problem each data request has a priority. Also, each component request from a plant commonly does not have a corresponding deadline related factor, while in the data staging problem each data request has a deadline. For the data staging problem, the priority and deadline associated with each data request are the two most important parameters for formulating the optimization criterion. For example, the minimization of the sum of the weighted priorities of satisfiable data requests (based on their deadlines) is used as the optimization criterion in the mathematical model of a basic data staging problem presented in Section 3. But for the facility location problem, in general, researchers adopt optimization criteria that are related to the physical distances between plants and facilities in either a continuous or discrete domain without any prioritizing schemes or deadline related factors (e.g., [4, 6, 9, 10, 14]). Thus, although lessons can be drawn from the design of algorithms for different versions of the facility location problem, there are no obvious direct correlations between either the formulations or the potential solutions of those two problems.

Data management problems similar to data staging for the BADD program are studied for other communication systems. With the increasing popularity of the World Wide Web (WWW), the National Science Foundation (NSF) recently projected that new techniques for organizing cache memories and other buffering schemes are necessary to alleviate memory and network latency and to increase bandwidth [3]. More advanced approaches of directory services, data replication, application-level naming, and multicasting are being studied to improve the speed and robustness of the WWW [2]. Evidence has been shown [7], that several file caches could reduce file transfer traffic, and hence the volume of traffic on the internet backbone. In addition, distributed environments are looking for ways to increase system performance with intelligent data placement [1]. The study of data staging can potentially draw lessons from and generate positive input for the active research in these related, but not directly comparable, areas of research.

Other research exploring heuristics for use in the BADD environment is being performed [10]. This work examines methods for scheduling the ATM-like channels of the BADD environment efficiently. The work does not develop a mathematical model and does not include several parameters considered here, such as deadlines and starting times. The work does show that "greedy" heuristics are effective tools for use in the BADD environment and uses a network simulator to corroborate this statement.

3. Mathematical model

A mathematical model for a basic data staging problem is presented in this section. This model serves as an initial version of a quantitative model for data staging. It allows the heuristic introduced in Section 4 to be given formally. As stated and discussed in Section 1, this paper concentrates on solving a simpler version of the data staging problem *statically*, where all parameter values for the communication system and the data request information stay fixed throughout the scheduling process. The values of all parameters in the following model may change temporally to reflect the dynamic nature of the underlying network system when the model is extended and used in a dynamic situation. In that case, the parameter values represent the best known information collected at the given point in time (e.g., all requests for data elements include only those known at any specific time instant). All necessary parameters for specifying the communication system and the data request information are introduced as follows. The model includes information about (1) the nodes in the network, (2) the links in the network, and (3) the data requests in the network. Each machine has parameters for the storage capacity and node number. A link has an availability starting time, availability ending time, bandwidth, latency, source node and destination node. Every request has an approximate data size, list of sources, and list of destinations. Each request source consists of a node number and a time after which the data is available on that node. Each request destination contains a node number, priority, and deadline for the data request. This description of the network and associated data requests are used to formulate the mathematical model to be used in solving the basic data staging problem. A glossary of notation is included in Section 7 for the readers convenience.

A communication system M consists of m machines $\{M[0], M[1], ..., M[m-1]\}$. Each machine can be a server that stores data elements and a client that makes data requests to the system. Each machine also can be an intermediate node for storing a copy of a specific data item temporarily. $Cap[i](t)$ represents the available memory storage capacity of machine $M[i]$ ($0 \leq i < m$) at time t.

A network topology graph G_{nt} specifies the connectivity of the communication system for the machines in M with the following notation. A set of m vertices $V =$

$\{V[0], V[1], ..., V[m-1]\}$ is generated that corresponds to the m machines in the communication system. In this model, if two machines are connected by the same transmission link during v non-overlapping and discontinuous time intervals, then v different *virtual* links corresponding to the appropriate available time intervals are used to represent this situation (e.g., the availability of a satellite link for fifteen minutes each hour). Also, each transmission link is uni-directional. A bi-directional link between two machines is represented as two different virtual uni-directional links that correspond to the transmission link in each direction.

Let $Nl[i,j]$ be the total number of direct virtual communication links from $M[i]$ to $M[j]$. $L[i,j][k]$ denotes the k-th direct virtual communication link from $M[i]$ to $M[j]$, where $0 \le i,j < m$, $i \ne j$, and $0 \le k < Nl[i,j]$. For each $L[i,j][k]$, a directed edge $E[i,j][k]$ from $V[i]$ to $V[j]$ is added to G_{nt}. All the added edges constitute the set of edges E of G_{nt}. Each $L[i,j][k]$ is associated with one unique time frame during which the corresponding link is available for communication. Let $Lst[i,j][k]$ denote the link starting time when $L[i,j][k]$ becomes available and $Let[i,j][k]$ denote the link ending time when $L[i,j][k]$'s availability terminates. With the above notation, link $L[i,j][k]$ is available between $Lst[i,j][k]$ (starting time) and $Let[i,j][k]$ (ending time).

Let a data item be a block of information that can be transferred between machines. For any data item d, $|d|$ represents the size of the associated data set. Let $D[i,j][k](|d|)$ denote the communication time for transferring data item d (of size $|d|$) from machine $M[i]$ to machine $M[j]$ through their k-th dedicated virtual link during time frame $[Lst[i,j][k], Let[i,j][k]]$. $D[i,j][k](|d|)$ includes all the various hardware and software related components of the inter-machine communication overhead (e.g., network latency and the time for data format conversion between $M[i]$ and $M[j]$ when necessary). Machines $M[i]$ and/or $M[j]$ may be intermediate nodes for transferring d rather than the original source or the final destination node of d.

Suppose n is the number of data items with distinctive names (identifiers) available in the corresponding communication system M. Let $\Delta = \{\delta[0], \delta[1], ..., \delta[n-1]\}$ be the set of these data items, where each $\delta[i]$ is unique. For example, a weather map of Europe generated at 2 p.m. would have a different name than a weather map of the same region generated at 6 p.m. A data location table that specifies the initial locations of the n available data items can be constructed with the following notation. Let $N\delta[i]$ be the number of different machines that the data item $\delta[i]$ is located at initially. $Source[i,j]$ denotes the j-th initial source location of the data item $\delta[i]$ (with no implied significance for the ordering of the sources), where $0 \le i < n$, $0 \le j < N\delta[i]$, and $0 \le Source[i,j] < m$. Also, $\delta st[i,j]$ denotes the starting time at which $\delta[i]$ is available at its j-th initial source location.

Suppose ρ is the number of the requested data items with distinctive names (identifiers) in the corresponding communication system M, where $0 \le \rho \le n$. Let $Rq = \{Rq[0], Rq[1], ..., Rq[\rho-1]\}$ be the set of the requested data items. Each $Rq[j]$ $(0 \le j < \rho)$ is the name of a data item and there must exist i $(0 \le i < n)$, such that $Rq[j] = \delta[i]$. Each $Rq[j]$ must be unique. A data request table that specifies the requests of data items can be constructed with the following notation. Let $Nrq[j]$ denote the number of different requests for $Rq[j]$. $Request[j,k]$ denotes the machine from which the k-th request for data item $Rq[j]$ originates (with no implied order among the requests), where $0 \le j < \rho$, $0 \le k < Nrq[j]$, and $0 \le Request[j,k] < m$. Also, $Rft[j,k]$ denotes the finishing time (or deadline) after which the data item $Rq[j]$ on its k-th requesting location is no longer useful (e.g., data items may be needed before a specific time when certain decisions must be made). Suppose the priority of each data request is between 0 and P, where P is the highest priority possible (i.e., a member of the class of most important requests). $Priority[j,k]$ denotes the priority for the data request of the data item $Rq[j]$ on its k-th requesting location.

Assume that the scheduling procedure of the communication steps starts at time 0. Let $S = \{S_0, S_1, ..., S_{\sigma-1}\}$ denote a set of σ distinct schedules for the communication steps of transmitting requested data items. Consider a specific schedule S_h, where $0 \le h < \sigma$. The k-th request for data item $Rq[j]$ is satisfiable with respect to S_h if $Rq[j]$ can be obtained by the requesting machine, $M[Request[j,k]]$, before the deadline, $Rft[j,k]$. Let $Srq[S_h]$ denote the set of two-tuples $\{(j,k) \mid k$-th request of the data item $Rq[j]$ is satisfiable$\}$. Suppose $W[i]$ $(0 \le i \le P)$ denotes the relative weight of the i-th priority. These weightings allow system administrators to specify the relative importance of priority α data request versus priority β data request, where $0 \le \alpha,\beta \le P$. The effect, $E[S_h]$, of the scheduling scheme S_h is defined as

$$E[S_h] = -\left(\sum_{(j,k) \in Srq[S_h]} W[Priority[j,k]]\right).$$

The global optimization criterion is defined as

$$\min_{0 \le h < \sigma} E[S_h].$$

Given this mathematical model, the objective of data

staging in this paper for a specific communication system is to find an S_h such that $E[S_h]$ is minimized (i.e., the total sum of the weighted priorities of all satisfiable data requests with respect to S_h is maximized). It should be noted that an exhaustive set of schedules is not created in this research.

4. Data staging heuristic

4.1. Overview

The heuristic for solving the data staging problem introduced in this section is based on Dijkstra's algorithm for solving the multiple-source shortest-path problem on a weighted and directed graph [5]. In Subsection 4.2, background information about Dijkstra's algorithm is provided. It is summarized from the material in [5]. Only the relevant part with respect to the data staging heuristic is discussed in detail. Subsection 4.3 presents the heuristic that is used to schedule the communication steps. The definition of the shortest-path estimate for Dijkstra's algorithm and the cost function of a communication step for local optimization in the heuristic are defined. The complexity analysis of the heuristic is provided in Subsection 4.3 as well.

4.2. Dijkstra's algorithm

The multiple-source shortest-path problem is defined as follows. Let $\tilde{G} = (\tilde{V}, \tilde{E})$, be a weighted and directed graph, where $\tilde{V}$ is a set of vertices and $\tilde{E}$ is a set of edges. For a single data item, suppose $V_S \subset \tilde{V}$ denotes a set of source vertices and $V_D \subset \tilde{V}$ denotes a set of destination vertices. The goal is to find a shortest path from any source vertex $v_s \in V_S$ to every destination vertex $v_d \in V_D$. The length of a path is the sum of the weights of its constituent edges. For an established path, an immediate predecessor vertex $\pi[v]$ of each vertex $v \in \tilde{V}$ is either another vertex or NIL. Any multiple-source shortest-path algorithm needs to set $\pi[v]$ properly so that the chain of predecessors originating at a vertex $v_d \in V_D$ corresponds to a shortest path from any $v_s \in V_S$ to v_d.

The main technique used in this multiple-source shortest-path algorithm is relaxation. For each vertex $v \in \tilde{V}$, an attribute $d[v]$, which is referred to as a shortest–path estimate, is maintained. This $d[v]$ is an upper bound on the length of a shortest path from any source vertex $v_s \in V_S$ to $v_d \in V_D$. The relaxation procedure with respect to a directed edge from vertex u to vertex v consists of testing whether the length of the shortest path to v found so far can be decreased by going through u (with known $d[u]$) via that edge from u to v. If the above testing results in a positive answer, a relaxation step decreases the value of the shortest-path estimate $d[v]$ (e.g., set $d[v]$ as $d[u]$ plus the weight of the edge from u to v) and updates v's immediate predecessor vertex as u. Relaxation is the only way by which the shortest-path estimate $d[v]$ and the predecessor vertex $\pi[v]$ can change.

Algorithms for solving a multiple-source shortest-path problem may differ in the way by which the edges are relaxed. In Dijkstra's algorithm, a set of vertices V_F (set to be V_S initially), whose final shortest paths from any $v_s \in V_S$ have already been determined, is maintained. The algorithm repeatedly selects the vertex $u \in \tilde{V} - V_F$ with the minimum shortest-path estimate, inserts u into V_F, and relaxes any edge from u to v by updating $d[v]$ and $\pi[v]$ properly, for all $v \in \tilde{V} - V_F$. The algorithm terminates when $V_F = \tilde{V}$.

4.3. Heuristic and complexity analysis

All necessary communication steps are scheduled by the data staging heuristic presented in this subsection. The heuristic utilizes the following three strategies collectively:

(i) Choose an order of data transfers that results in a set of data items being available on intermediate nodes earlier. This set of data items will be chosen based on some criteria that will cause the ordering to satisfy more data requests.

(ii) Maximize the sum of the priorities of the potentially satisfiable data requests.

(iii) Consider the urgency of a request as its deadline approaches.

A data staging heuristic that has a well-balanced set of local optimization criteria using the above three strategies should intuitively perform well. The multiple-source shortest-path algorithm based heuristic presented in this subsection is built upon Dijkstra's algorithm and utilizes all of the above three strategies. The heuristic iteratively picks which data item to transfer next constrained by a cost function. Each iteration of the heuristic involves: (a) running Dijkstra's algorithm for each data request individually, (b) determining the "cost" to transfer a data item to its successor in the shortest path, (c) picking the lowest cost data request and transferring that data item, (d) updating system parameters to reflect resources used in (c), and (e) repeating (a) through (d) until there are no more satisfiable requests in the system.

1. For all k ($0 \le k < Nl[s,r]$), do the following steps.
2. if ($A_T[i,s] > Lst[s,r][k]$) {
   /* if $Rq[i]$ is obtained by $M[s]$ after $L[s,r][k]$ is available */
3. if ($[A_T[i,s] + D[s,r][k](|Rq[i]|)] \le Let[s,r][k]$)
   /* if the available time interval is long enough to transfer $Rq[i]$ via $L[s,r][k]$ */
4. if ($Cap[r](A_T[i,s]) \ge |Rq[i]|$)
   /* if $M[r]$ has enough storage capacity for $Rq[i]$ */
   $A_L[s,r][i][k] = A_T[i,s] + D[s,r][k](|Rq[i]|)$
   /* find "available time" using this link*/
5. } else { /* if $Rq[i]$ is obtained by $M[s]$ before $L[s,r][k]$ is available */
6. if ($[Lst[s,r][k] + D[s,r][k](|Rq[i]|)] \le Let[s,r][k]$)
   /* if the available time interval is long enough to transfer $Rq[i]$ via $L[s,r][k]$ */
7. if ($Cap[r](Lst[s,r][k]) \ge |Rq[i]|$)
   /* if $M[r]$ has enough storage capacity for $Rq[i]$ */
   $A_L[s,r][i][k] = Lst[s,r][k] + D[s,r][k](|Rq[i]|)$ }
   /* find "available time" using this link */
8. if ($A_T[i,r] > \min_{0 \le k < Nl[s,r]} \{A_L[s,r][i][k]\}$) {
   /* if smaller shortest-path estimate is found */
9. $A_T[i,r] = \min_{0 \le k < Nl[s,r]} \{A_L[s,r][i][k]\}$
   /* update the shortest-path estimate for $V[r]$ */
10. $k_l = k$ giving minimum in step 9 /* record the virtual link used */
    } /* $\underline{k_l}$ is the argument k that minimizes $A_L[s,r][i][k]$ */

Figure 1: Pseudocode for implementing the relaxation step.

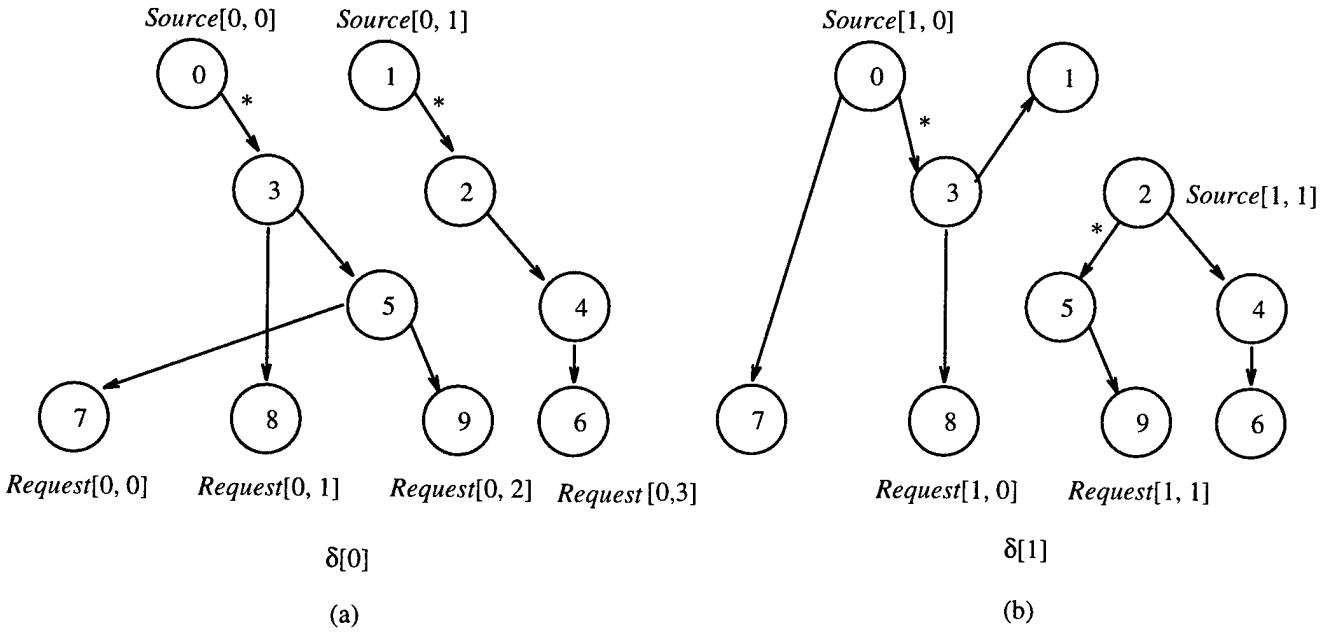


Figure 2: An example communication system that requests (a) δ[0] and (b) δ[1].

In some cases, Dijkstra's algorithm would not need to be executed each iteration for a particular data request, i.e. if the links that the request uses are not affected by updating the system parameters. Future versions of the heuristic are planned to take advantage of not having to recalculate all the shortest paths during each iteration.

For each requested data item $Rq[i_1]$ $(0 \le i_1 < \rho)$, as stated in Section 3, there exists i_2 $(0 \le i_2 < n)$, such that $Rq[i_1] = \delta[i_2]$. Suppose $V_S[i_1]$ is the set of source vertices corresponding to $Rq[i_1]$ in G_{nt}. Then $V[k] \in V_S[i_1]$ $(0 \le k < m)$ if and only if there exists j $(0 \le j < N\delta[i_2])$, such that $Source[i_2, j] = k$. This means that $V_S[i_1]$ contains all the vertices of G_{nt} that correspond to the machines that are the initial locations of $Rq[i_1]$ (i.e., $\delta[i_2]$). Suppose $V_D[i_1]$ is the set of destination vertices corresponding to $Rq[i_1]$ in G_{nt}. Then $V[k] \in V_D[i_1]$ $(0 \le k < m)$ if and only if there exists j $(0 \le j < Nrq[i_1])$, such that $Request[i_1,j] = k$. This means that $V_D[i_1]$ contains all the vertices of G_{nt} that correspond to the machines that are making data requests for $Rq[i_1]$. Let the length of a path from a source vertex $v_s \in V_S[i_1]$ to a destination vertex $v_d \in V_D[i_1]$ be defined as the difference between the earliest possible time when data item $Rq[i_1]$ can be available on the machine corresponding to v_d via the machines and the virtual communication links along the path and the time the data item is available on $v_s \in V_S[i_1]$ (this time can be calculated using the various parameters defined in Section 3). With the above defined $G_{nt} = (V,E)$, $V_S[i]$, and $V_D[i]$, where $0 \le i < \rho$, a separate multiple-source shortest-path problem is well defined for each of the ρ different requested data items in the context of the data staging problem.

Readers should notice that it may be impossible to use the individually shortest paths to all $v_d \in V_D[i]$ for each data item $\delta[i]$ $(0 \le i < \rho)$ due to possible communication link contention in the network for transferring different data items. Also, a multiple-source shortest-path algorithm for G_{nt} only attempts to minimize the time when a given requested data item is obtained by its corresponding requesting locations. But as clearly stated in Section 1 and Section 3, other criteria like request deadlines and the priorities of the satisfiable data requests must be taken into account as well.

As reviewed in Subsection 4.2, the way of defining $d[v]$ (i.e., the shortest-path estimate) based on the known $d[u]$ and information about all edges from u to v (e.g., their weights) is essential for applying Dijkstra's algorithm to derive the data staging heuristic. For $G_{nt} = (V, E)$, $V_S[i]$, and $V_D[i]$ $(0 \le i < \rho)$, let the shortest-path estimate of $V[j]$ (corresponding to machine $M[j]$) for requested data item $Rq[i]$ be defined as $A_T[i,j]$. This is the earliest possible available time found so far when the requested data item $Rq[i]$ is obtained by machine $M[j]$ with the consideration of the availability of the virtual links and the available memory capacity of machine $M[j]$. Suppose $\delta[k] = Rq[i]$ $(0 \le k < n)$. Initially, $A_T[i,Source[k,q]] = \delta st[k,q]$ for all $0 \le q < N\delta[k]$. That is, for all initial source locations $v_s \in V_S[i]$, their shortest-path estimates are the starting times when $\delta[k]$ is available at those nodes.

It is assumed that each machine can send different data items (each via a different link) to its neighboring machines in the network simultaneously. Future work will relax this assumption. Suppose $A_T[i,s]$ and $A_T[i,r]$ are known and there are virtual links $L[s,r][k]$ $(0 \le k < Nl[s,r])$ from $M[s]$ to $M[r]$. Let $A_L[s,r][i][k]$ denote the time when the requested data item $Rq[i]$ can be available on machine $M[r]$ via fetching the copy from $M[s]$ through the virtual link $L[s,r][k]$. The relaxation step with respect to the edges from $V[s]$ to $V[r]$ based on the known $A_T[i,s]$ and $A_T[i,r]$ is implemented by the C-style pseudocode in Figure 1.

As illustrated by Step 10 in the above pseudocode, the exact virtual link $L[s,r][k_l]$ used for updating the shortest-path estimate of $V[r]$ needs to be recorded, due to the existence of multiple virtual links between $M[s]$ and $M[r]$. Thus, the predecessor vertex $\pi[r]$ in the usual description of the Dijkstra's algorithm is extended as a predecessor field and is defined as a two-tuple (s,k_l) in this data staging heuristic, where s records the source machine and k_l records the virtual link used. At this stage, the information about the availability of link $L[s,r][k_l]$ does not need to be updated because each execution of Dijkstra's algorithm is for a single given data item and the transfer of any specific data item only needs to use that link once.

For each $Rq[i]$ $(0 \le i < \rho)$ individually, based on the above defined shortest-path estimate, the shortest paths for all $v_d \in V_D[i]$ can be generated with respect to G_{nt} defined in Section 3. Consider an example shown by Figure 2, with a communication system consisting of ten machines. Also, to simplify the presentation, suppose that there is at most one virtual link between any pair of machines. There are two data items $\delta[0]$ and $\delta[1]$ being requested. Machines 0 and 1 are the sources for $\delta[0]$ ($Source[0,0] = 0$, $Source[0,1] = 1$), and machines 0 and 2 are the sources for $\delta[1]$ ($Source[1,0] = 0$, $Source[1,1] = 2$). The destinations for $\delta[0]$ are machines 6, 7, 8, and 9 ($Request[0,0] = 7$, $Request[0,1] = 8$, $Request[0,2] = 9$, $Request[0,3] = 6$), and the destinations for $\delta[1]$ are machines 8 and 9 ($Request[1,0] = 8$, $Request[1,1] = 9$). Suppose that the shortest paths generated corresponding to $\delta[0]$ individually are shown in Figure 2(a), and the

shortest paths generated corresponding to $\delta[1]$ individually are shown in Figure 2(b). Note that Figure 2(a) and 2(b) correspond to the same machine suite, and that the links shown in Figure 2(a) and 2(b) collectively are a subset of all the inter-machine links. This example is used throughout the rest of this section to illustrate the proposed data staging heuristic.

A vertex $V[r]$ (corresponding to machine $M[r]$ in G_{nt}) is defined as a contingent vertex with respect to $V_S[i]$, if $V[r] \in V - V_S[i]$ and there is an edge entering into $V[r]$ that starts from a vertex in $V_S[i]$. For the example shown in Figure 2(a), $V[2]$ and $V[3]$ are contingent vertices with respect to $V_S[0] = \{V[0], V[1]\}$. For the example shown in Figure 2(b), $V[3]$, $V[4]$, $V[5]$, and $V[7]$ are contingent vertices with respect to $V_S[1] = \{V[0], V[2]\}$. $M[r]$ may be an intermediate machine along some paths from any source vertex in $V_S[i]$ to a set of destination vertices in $V_D[i]$. Suppose this set of destination (requesting) vertices associated with a data item $\delta[i]$ and a contingent vertex $V[r]$ is defined as $Drq[i,r]$. For any element $drq \in Drq[i,r]$, drq is an integer ($0 \le drq < Nrq[i]$), such that $V[Request[i,drq]] \in V_D[i]$. For the example shown in Figure 2(a), for $V[3]$ (corresponding to $M[3]$), $Drq[0,3] = \{0, 1, 2\}$ corresponding to requests for $\delta[0]$ from $M[7]$, $M[8]$, and $M[9]$, respectively. For $V[2]$ (corresponding to $M[2]$), $Drq[0,2] = \{3\}$ corresponding to the request for $\delta[0]$ from $M[6]$. Similarly, for the example shown in Figure 2(b), $Drq[1,3] = \{0\}$ corresponding to the request of $\delta[1]$ from $M[8]$, $Drq[1,5] = \{1\}$ corresponding to the request of $\delta[1]$ from $M[9]$, and $Drq[1,4] = Drq[1, 7] = \varnothing$.

After applying the multiple-source shortest-path algorithm for each requested data item $Rq[i]$ ($0 \le i < \rho$), individually, with the associated $V_S[i]$, $V_D[i]$, and G_{nt}, ρ sets of shortest paths are generated, one for each of the ρ different requested data items. Suppose the predecessor field of $V[r]$ corresponding to the shortest paths generated for $Rq[i]$ is (s,k). Each contingent vertex $V[r]$ that has an incident edge $E[s,r][k]$ from $V[s]$ in $V_S[i]$ and its associated $Drq[i,r] \neq \varnothing$ corresponds to a valid next communication step to be scheduled. This communication step is the one that specifies transferring $Rq[i]$ from $M[s]$ to $M[r]$ via link $L[s,r][k]$. For the example shown in Figure 2, there are four valid communication steps that can be scheduled (specified by asterisks). But different valid communication steps may have conflicting resource requirements (e.g., $M[0]$ cannot send $\delta[0]$ and $\delta[1]$ to $M[3]$ simultaneously due to network conflict for the example shown in Figure 2). Thus, a local optimization criterion is used to select one of the valid communication steps to be scheduled. The next paragraphs derive a cost function that is used as the basis for making this selection. The cost function will involve considerations of satisfiability, effective priority, and urgency as defined later.

Suppose $\tilde{A_T}[i,j]$ ($j \in Drq[i,r]$) denotes the time when $Rq[i]$ is received and available at its corresponding j-th requesting location. A satisfiability function $Sat[i,r](j)$, is defined as: $Sat[i,r](j) = 1$, if $\tilde{A_T}[i,j] \le Rft[i,j]$; and 0, if $\tilde{A_T}[i,j] > Rft[i,j]$. Note that this shortest path involves passing through vertex $V[r]$, so if a request in $Drq[i,r]$ is not satisfied, there is not other path that will cause it to be satisfied. That is, if $Sat[i,r](j) = 1$, then the data request of $Rq[i]$ from machine $M[Request[i,j]]$ is satisfied. Otherwise, the corresponding data request is not satisfied. As an example of the definition of $Sat[i,r](j)$, consider the shortest paths generated by selecting first the valid communication step for transferring $\delta[0]$ from $M[0]$ to $M[3]$ in the example shown by Figure 2(a). Suppose $T_0 = \tilde{A_T}[0,0] = A_T[0,7]$, $T_1 = \tilde{A_T}[0,1] = A_T[0,8]$, and $T_2 = \tilde{A_T}[0,2] = A_T[0,9]$. Also, assume that $T_0 \le Rft[0,0]$, $T_1 > Rft[0,1]$, and $T_2 \le T_f[0,2]$. Then, $Sat[0,3](0) = 1$, $Sat[0,3](1) = 0$, and $Sat[0,3](2) = 1$. T_0, T_1, and T_2 are calculated during the last execution of the Dijkstra's algorithm with respect to $\delta[0]$.

Suppose $Efp[i,j]$ denotes the effective priority for the data request of $Rq[i]$ from its j-th requesting location, where $Efp[i,j] = Sat[i,r](j) \times W[Priority[i,j]]$. Suppose $Urgency[i, j]$ denotes the urgency for the data request of $Rq[i]$ from its j-th requesting location, where $Urgency[i,j] = -Sat[i,r](j) \times (Rft[i,j] - \tilde{A_T}[i,j])$, where smaller $Urgency[i]$ implies that it is less urgent to transfer $Rq[i]$ to the j-th requesting location. Suppose $W_E \ge 0$ is the relative weight for the effective priority factor and $W_U \ge 0$ is the relative weight for the urgency factor in the scheduling. Readers should notice that (a) applying Dijkstra's algorithm to obtain $A_T[i,r]$ through shortest paths, (b) maximizing $Efp[i,j]$, and (c) maximizing $Urgency[i,j]$ follow the three strategies for designing data relocation heuristics recommended in (i), (ii), and (iii), respectively, at the beginning of this subsection. The cost, $Cost[s,r][i,j][k]$, for transferring the requested data item $Rq[i]$ from machine $M[s]$ to $M[r]$ via link $L[s,r][k]$ is defined as:

$$Cost[s,r][i,j][k] = -W_E Efp[i,j] - W_U Urgency[i,j].$$

The next chosen communication step should be the one that has the smallest associated cost among all valid next communication steps for transferring all $Rq[i]$ where $0 \le i < \rho$, and $Sat[i,r]$ is not 0 for all r. If $Sat[i,r]$ is 0 for all r, that request receives no resources and the data does

not move from its current locations. The request is not eliminated from the network. Currently the heuristic is applied to a static system, as this constraint is loosened and a dynamic system is explored, links might become available that would facilitate the delivery of an otherwise unsatisfiable request. So requests that are at one point in time unsatisfiable, might become satisfiable at a later point in time. For the valid communication step for transferring $\delta[0]$ from $M[0]$ to $M[3]$ shown in Figure 2(a), $Efp[0,0] = W[Priority[0,0]]$ and $Urgency[0,0] = -(Rft[0,0] - T_0)$.

The rationale for choosing the above cost for local optimization is as follows. First, only a valid next communication step whose associated $Sat[i,r]$ is not 0 for all r will facilitate satisfying data request(s). $Cost[s,r][i,j][k]$ attempts to maximize the total priority of the satisfiable data requests. Furthermore, in order to satisfy as many data requests as possible, intuitively it is necessary to transfer a specific data item to the requesting locations whose deadlines are close to expire. This intuition is captured by the inclusion of the urgency factor. Thus, collectively with the consideration of the total priority of the satisfiable data requests and the urgency of those data requests in this local optimization step, this data staging heuristic should generate a suboptimal S_h that reasonably achieves the global optimization criterion presented in Section 3.

Suppose that the current chosen communication step for S_h according to $Cost[s,r][i,j][k]$ is to transfer the requested data item $Rq[i]$ from $M[s]$ to $M[r]$ via link $L[s,r][k]$ during the time interval between t_0 and t_1. Before repeating the above multiple-source shortest-path based heuristic for determining the next communication step of S_h, the following four categories of information need to be updated.

(1) Update the network topology graph G_{nt} — Delete the edge $E[s,r][k]$ corresponding to link $L[s,r][k]$ in G_{nt}. Also, add up to two more edges from $V[s]$ to $V[r]$ that correspond to two more virtual links. One with the link starting time as $Lst[s,r][k]$ and link ending time as t_0, and another with link starting time as t_1 and link ending time as $Let[s,r][k]$. If $t_0 = Lst[s,r][k]$ and/or $t_1 = Let[s,r][k]$, then the above first and/or second additional virtual links are not needed.

(2) Update $Cap[r](t)$ — $Cap[r](t)$ is decremented by $|Rq[i]|$, where $|Rq[i]|$ is the size of $Rq[i]$, because machine $M[r]$ keeps a copy of the data item $Rq[i]$ (e.g., $Cap[3](t)$ is decremented by $|\delta[0]|$ for the communication step shown in Figure 2(a)).

(3) Update the set of source vertices $V_S[i]$ for $Rq[i]$ — $V_S[i] = \{\text{latest } V_S[i]\} \cup \{V[r]\}$. Also, set the starting time when $Rq[i]$ is available on $M[r]$ as $A_T[i,r]$. For the example communication step for transferring $\delta[0]$ from $M[0]$ to $M[3]$ shown in Figure 2(a), $M[3]$ becomes a source of $\delta[0]$ and its starting time is $A_T[0,3]$.

(4) Update the garbage collection related information — The copy of $Rq[i]$ on machine $M[r]$ is used as an intermediate copy for forwarding $Rq[i]$ to some other machines. Suppose this set of machines is defined as $Im[i,r]$. $Im[i,r]$ can be determined by tracing the shortest paths generated above for each $v_d \in V_D[i]$. For the example communication step of transferring $\delta[0]$ from $M[0]$ to $M[3]$ shown in Figure 2(a), by tracing the shortest-paths generated for $V[7]$, $V[8]$, and $V[9]$, $Im[0,3]$ can be determined as $\{5, 8\}$. After all those machines in $Im[i,r]$ have received the copy of $Rq[i]$ from $M[r]$, the copy of $Rq[i]$ on machine $M[r]$ can be deleted. Suppose the time for the last machine in $Im[i,r]$ to receive its copy of $Rq[i]$ from $M[r]$ is $R_T[i,r]$, then at this time, $Cap[r](t)$ is incremented by $|Rq[i]|$. This above procedure implements the garbage collection scheme in data staging.

A single iteration of this multiple-source shortest-path based heuristic starts at the step for generating shortest paths for all remaining data requests (based on the Dijkstra's algorithm and the current system status (e.g., data source locations and link availability)). The iteration ends at the step for updating the system information described above. Then with the new $Cap[r](t)$, G_{nt}, and $V_S[i]$, execute the above multiple-source shortest-path based heuristic repeatedly to determine the rest of the communication steps in S_h. The heuristic terminates when all remaining data requests are not satisfiable.

For the complexity analysis of this multiple-source shortest-path based heuristic for determining one communication step in S_h, suppose that $|E|$ is the number of edges and $|V|$ is the number of vertices in the network topology graph G_{nt}. If a Fibonacci heap [5] is used to implement the priority queue, the worst case asymptotic complexity of Dijkstra's algorithm is $O(|E| + |V|\lg|V|)$. For the network topology graph G_{nt} terminology described in (2) of Section 3, $|E| = \sum_{0 \le i \ne j < m} Nl[i,j]$ and $|V| = m$. Because in the worst case it is necessary to apply the multiple-source shortest-path algorithm to all the requested data items $Rq[i]$ $(0 \le i < \rho)$, the worst case asymptotic complexity of this heuristic for determining one communication step in S_h is

$$O[\rho(m \lg m + \sum_{0 \le i \ne j < m} Nl[i,j])],$$

where ρ is the total number of requested data items defined in Section 3.

Given the heuristic approach presented in this section uses Dijkstra's algorithm in conjunction with a minimization criteria, it is called the Dijkstra/minimization heuristic. It is evaluated in the next section.

5. Simulation study

To perform the simulation study, network topologies and data requests must be generated, values for W_E and W_U must be determined, and other scheduling schemes need to be created to compare to the Dijkstra/minimization heuristic. Rather than just choosing one network topology and set of data requests, because one can not accurately reflect the changing data requests and network availability with one case, 40 test cases were generated and the Dijkstra/minimization heuristic was executed using each of these cases and the results were averaged. The data requests and the underlying communication systems were randomly generated over a set of parameters (corresponding to the notation introduced in Section 3) as specified below. All parameters are randomly generated with uniform distributions in predefined ranges representing systems in a BADD-like environment. The sources and requesting machines for all data items are also generated randomly. The test generation program guarantees that the generated communication system is strongly connected [5], such that there is a path consisting of physical transmission links between any pair of nodes in both directions.

These randomly generated patterns of data requests and the underlying communication systems are used for three reasons: (1) it is beneficial to obtain cases that can demonstrate the performance of the Dijkstra/minimization heuristic presented over a broad range of conditions; (2) a generally accepted set of data staging benchmark tasks does not exist; and (3) it is not clear what characteristics a "typical" data staging task would exhibit. Determining a representative set of data staging benchmark tasks remains an unresolved challenge in the research field of data staging and is outside the scope of this paper.

Finding optimal solutions to data staging tasks with realistic parameter values are intractable problems. It is currently impractical to directly compare the quality of the solutions found by the above Dijkstra/minimization heuristic 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 authors' 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 Dijkstra/minimization heuristic presented in this paper.

The performance of the Dijkstra/minimization heuristic is compared with two random-search based scheduling procedures. The only difference between the first random procedure and the above Dijkstra/minimization heuristic is that, instead of choosing a valid communication step using $Cost[s,r][i,j][k]$ as discussed for the Dijkstra/minimization heuristic, the Dijkstra random heuristic randomly chooses an arbitrary valid communication step to schedule.

The second random-search based scheduling procedure performs Dijkstra once for each requested data item, assuming it is the only requested item in the network. Then the paths through the network are scheduled for each data item, finishing $Rq[i]$ before $Rq[i+1]$. If a conflict arises, i.e., the time frame in which a particular link was originally scheduled by the independent Dijkstra's for a given request is unavailable, the request is dropped and not satisfied. This approach is referred to as single Dijkstra random because Dijkstra's algorithm is only executed once for each data item.

Each test set for the results shown in Figures 3 and 4 was generated with the following parameters. The number of machines in the communication system is between ten and twenty. Each machine has between 10MB to 20GB memory storage capacity. The maximum outbound degree of a machine $M[i]$ (i.e., the number of machines that $M[i]$ can transfer data items to directly through physical transmission links) is five. There are at most two physical transmission links between any two machines (there can be none). The total number of data requests is one to ten times the number of machines in the system. There can be up to three sources and three destinations for each of the data requests. Each data item size ranges from 10KB to 1MB. The priority of each data item is 1, 5, or 10, and the relative weight of a priority is equal to the priority itself (i.e., $W[i] = i$, for all $0 \le i \le P$). Each request has its own priority. The bandwidth of each physical transmission link is between 10KB/sec and 10MB/sec. The link starting and ending times and the data item starting (available) and finishing times (or deadlines) are modeled building on information about the underlying communication infrastructures and data

request patterns in [12, 13]. These parameters were used as they capture the information about the network that is necessary to show the functionality of the heuristic over a variety of network configurations.

Let the E–U ratio be W_E/W_U. As shown by the cost function $Cost[s,r][i,j][k]$ introduced in Subsection 4.3, the E-U ratio may affect the performance of the Dijkstra/minimization heuristic. Figures 3 and 4 show the performance of the Dijkstra/minimization heuristic when the E-U ratio ranges from 0.001 to 1000 (shown by dashed-dotted lines). Figure 3 uses the average sum of the weighted priorities of the satisfiable data requests and Figure 4 uses the average number of the satisfiable data requests. Both are averaged over 40 randomly generated test cases.

In this study, the Dijkstra/random heuristic is executed ten times for each of 40 randomly generated cases (the same 40 cases as used for Dijkstra/minimization). Then, its average sum of the weighted priorities of the satisfiable data requests and its average number of the satisfiable data requests over all ten runs for all 40 cases are calculated. As shown in Figures 3 and 4, the Dijkstra/minimization heuristic consistently outperforms the Dijkstra/random heuristic (shown by the dotted lines). The difference shows the advantage of using a minimization criteria to resolve conflicting demands for a link.

Also shown in Figures 3 and 4, single Dijkstra random performs poorly (shown by the pluses) compared to the Dijkstra/minimization heuristic and the Dijkstra/random heuristic. The difference between the Dijkstra/minimization and the single Dijkstra random shows the advantage of the process of interleaving link demands from multiple data items.

Solid lines in Figures 3 and 4 show the average sum of the weighted priorities of all data requests and the average number of all data requests, respectively. Readers should notice that not all data requests for a randomly generated test case can be satisfied even with the optimal scheduling scheme for data staging. The asterisks show the average of all the requests that could be satisfied if each had exclusive use of the network, i.e., it was the only request in the network. Thus, the asterisks in Figures 3 and 4 represent a loose upper bound for the performance of any data staging heuristic. The difference between the solid line and the asterisks represent those requests that could never be satisfied due to insufficient resources of the network, i.e., links or storage. This information is useful as a prediction tool to determine changes to the network that would increase the number of satisfied requests.

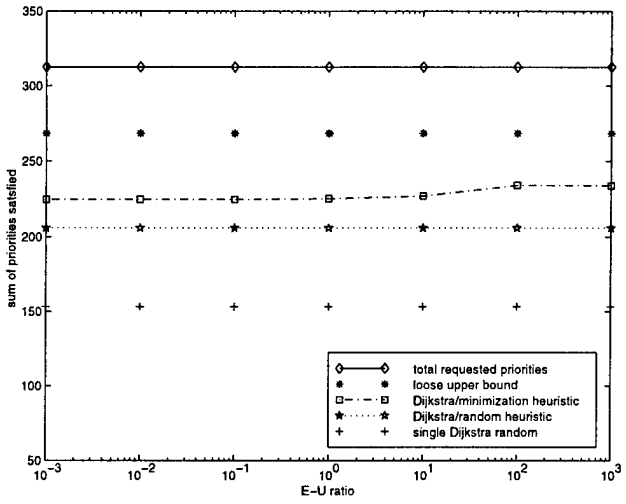


Figure 3: Comparison in terms of the average sum of the priorities of the satisfiable data requests for a lightly loaded network.

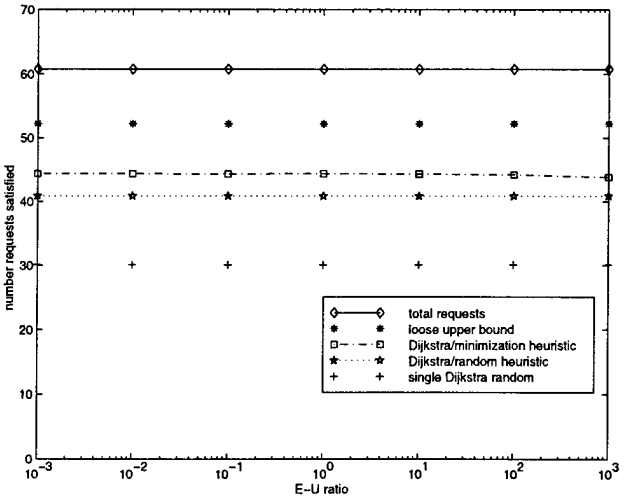


Figure 4: Comparison in terms of the average number of the satisfiable data requests for a lightly loaded network.

Figure 5 shows the average sum of the priorities satisfied for more heavily congested network topologies and Figure 6 shows the average of the number of requests satisfied. The network topologies that were used for these cases had fewer nodes in the network (ten to twelve), but had more requests (20 to 40 times the number of nodes), as well as one to five different sources and one to five different requesting machines. This

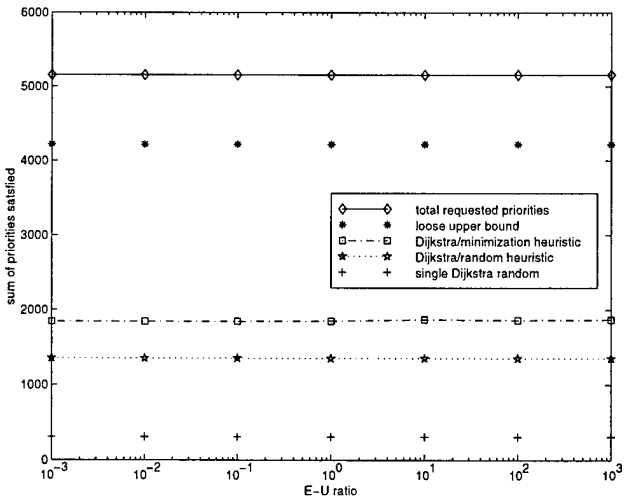


Figure 5: Comparison in terms of the average sum of the priorities of the satisfiable data requests for a heavily congested network.

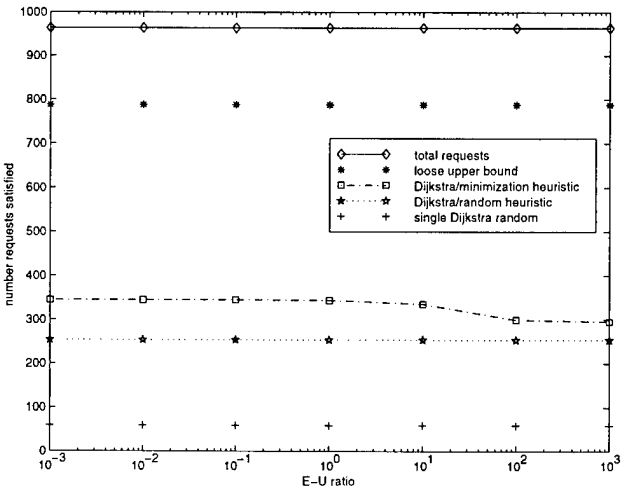


Figure 6: Comparison in terms of the average number of the satisfiable data requests for a heavily congested network.

caused the number of total requested data items, and the total priorities requested, to be at least an order of magnitude higher than in our previous experiments. The E-U ratio was again varied between 0.001 and 1000 to see what effect this would have on the results. It can be observed from Figures 3 and 5 that varying the E-U ratio has only a small impact, and that either $Efp[i,j]$ or $Urgency[i,j]$ by itself would be a sufficient criterion. As with the lightly loaded case, the scheduling the Dijkstra/minimization heuristic created for the heavily congested network is better than that of single Dijkstra random, and that of Dijkstra/random heuristic.

6. Summary and future work

Data staging is an important data management problem for information systems. It addresses the issues of distributing and storing over numerous geographically dispersed locations both repository data and continually generated data. When certain data with their corresponding priorities need to be collected together at a site with limited storage capacities in a timely fashion, a heuristic must be devised to schedule the necessary communication steps efficiently.

A rigorous mathematical model was created to describe a simplified static version of the data staging problem. This model is a first attempt at addressing this problem. The Dijkstra/minimization heuristic was introduced in Section 4 to solve this version of the data staging problem. Section 5 presented the results of simulation testing that shows the performance of the proposed Dijkstra/minimization heuristic over the Dijkstra/random heuristic and single Dijkstra random for a class of data staging tasks.

There are many issues that must be resolved before a complete heuristic for solving the entire data staging problem can be presented. The mathematical model described in this paper serves as a starting point for a rigorous model for the general data staging problem and will evolve over time. Also, when dynamic scheduling is necessary, methods need to be devised to include run-time information into the selection criterion. Fault tolerance issues must be considered as well in order to build a robust heuristic. The results of this research and its extensions may impact web management procedures, as well as the DARPA BADD program.

7. Glossary of notation

$A_L[s,r][i][k]$: time when $Rq[i]$ can be available on machine $M[r]$ via fetching the copy from $M[s]$ through the virtual link $L[s,r][k]$

$A_T[i,j]$: the earliest possible time found so far when $Rq[i]$ is available on $M[j]$

$\tilde{A}_T[i,j]$: time when $Rq[i]$ is received at its corresponding j-th requesting location with respect to the generated shortest path

$Cap[i](t)$: available memory storage capacity of machine $M[i]$ at time t

$Cost[s,r][i,j][k]$: cost for transferring the requested data item $Rq[i]$ associated with the j-th destination, from machine $M[s]$ to $M[r]$ via link $L[s,r][k]$

$|d|$: size of the associated data item d

$d[v]$: shortest-path estimate for vertex v in Dijkstra's algorithm

$D[i,j][k](|d|)$: communication time for transferring data item d with size $|d|$ from $M[i]$ to $M[j]$ through their k-th dedicated virtual communication link

Δ: set of data items available in the communication system

$\delta[i]$: i-th data item available in the communication system

$\delta st[i,j]$: starting time at which $\delta[i]$ is available at its j-th initial source location

$Drq[i,r]$: set of destination vertices associated with a data item $\delta[i]$ and a contingent vertex $V[r]$, $Drq[i,r] \subseteq \{0, 1, ..., Nrq[i] - 1\}$

E: set of edges in G_{nt} that corresponds to all virtual communication links among machines

$\tilde{E}$: set of edges in $\tilde{G}$

$|E|$: number of edges in the network topology graph G_{nt}

$Efp[i,j]$: effective priority for the data request of $Rq[i]$ from its j-th requesting location

$E[i,j][k]$: k-th direct edge from $V[i]$ to $V[j]$ in G_{nt}

$E[S_h]$: effect of the scheduling scheme S_h

$\tilde{G}$: a weighted and directed graph

G_{nt}: network topology graph of the communication system that illustrates the connectivity of the machines

$Im[i,r]$: set of machines that $M[r]$ will forward its copy of $Rq[i]$ to according to the generated shortest paths

k_l: the argument k that minimizes $A_L[s,r][i][k]$

$L[i,j][k]$: k-th direct virtual communication link from $M[i]$ to $M[j]$

$Let[i,j][k]$: link ending time when $L[i,j][k]$'s availability terminates

$Lst[i,j][k]$: link starting time when $L[i,j][k]$ becomes available

m: number of machines in the communication system

$M[i]$: i-th machine in the communication system, $0 \le i < m$

n: number of the data items with distinctive values available in the communication system

$N\delta[i]$: number of different machines that the data item $\delta[i]$ is located at initially

$Nl[i,j]$: total number of direct virtual communication links from $M[i]$ to $M[j]$

$Nrq[j]$: number of different machines where a request for $Rq[j]$ is initiated

P: highest priority possible and implies to be most important for any data request

$\pi[v]$: predecessor field (or predecessor vertex) of vertex v

$Priority[j,k]$: priority for the data request of the data item $Rq[j]$ on its k-th requesting location

$Request[j,k]$: k-th location of the request for data item $Rq[j]$, $0 \le Request[j,k] < m$

$Rft[j,k]$: finishing time (or deadline) after which the data item $Rq[j]$ on its k-th requesting location is no longer useful

ρ: number of the requested data items with distinctive values in the corresponding communication system

$Rq[j]$: j-th requested data item in the communication system

$R_T[i,r]$: time for the last machine in $Im[i,r]$ to receive its copy of $Rq[i]$ from $M[r]$

$Sat[i,r](j)$: satisfiability function associated with a the j-th requesting location for data item $\delta[i]$ and a contingent vertex $V[r]$

S_h: a specific schedule for the communication steps of transmitting requested data items

σ: number of distinct schedules for the communication steps of transmitting requested data items

$Source[i,j]$: j-th initial source location of the data item $\delta[i]$

$Srq[S_h]$: set of two-tuples $\{(j,k) \mid k$-th request of the data item $Rq[j]$ is satisfiable$\}$

$Urgency[i,j]$: urgency for the data request of $Rq[i]$ from its j-th requesting location

V: set of m vertices for G_{nt} that corresponds to m machines

$|V|$: number of vertices in the network topology graph G_{nt}

$\tilde{V}$: set of vertices in $\tilde{G}$

v_d: a specific destination vertex

V_D: set of destination vertices

$V_D[i]$: set of destination vertices corresponding to $Rq[i]$

V_F: set of vertices whose final shortest paths from any $v_s \in V_S$ have been determined during the execution of Dijkstra's algorithm

$V[i]$: i-th vertex of G_{nt} that corresponds to machine $M[i]$

v_s: a specific source vertex

V_S: set of source vertices

$V_S[i]$: set of source vertices corresponding to $Rq[i]$

$W[i]$: relative weight of the i-th priority

W_E: relative weight for the effective priority factor in the scheduling

W_U: relative weight for the urgency factor in the scheduling

References

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

[2] 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, March-April 1997, pp. 18-27.

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

[4] R. Chandrasekaran and A. Dauchety, "Location on tree networks: P-centre and n-dispersion problems," *Mathematics of Operations Research*, Vol. 6, No. 1, February 1981, pp. 50-57.

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

[6] G. Cornuejols, G. L. Nemhauser, and L. A. Wolsey, "Worst-case and probabilistic analysis of algorithms for a location problem," *Operations Research*, Vol. 28, No. 4, July-August 1980, pp. 847-858.

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

[8] A. P. Hurter and J. S. Martinich, *Facility Location and The Theory of Production*, Kluwer Academic Publishers, Norwell, MA, 1989.

[9] 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-August 1995, pp. 661-669.

[10] M. J. Lemanski and J. C. Benton, *Simulation for SmartNet Scheduling of Asynchronous Transfer Mode Virtual Channels,* Master's Thesis, Dept. of Computer Science, Naval Postgraduate School, June 1997 (Advisor: D. Hensgen).

[11] I. D. Moon and S. S. Chaudhry, "An analysis of network location problems with distance constraints," *Management Science*, Vol. 30, No. 3, March 1984, pp. 290-307.

[12] A. J. Rockmore, "BADD functional description," Internal DARPA Memo, February 1996.

[13] SmartNet/Heterogeneous Computing Team, "BC2A/TACITUS/BADD integration plan," Internal Report, August 1996.

[14] D. R. Shier, "A min-max theorem for p-center problems on a tree," *Transportation Science*, Vol. 11, No. 3, August 1977, pp. 243-252.

[15] H. J. Siegel, H. G. Dietz, and J. K. Antonio, "Software support for heterogeneous computing," in *The Computer Science and Engineering Handbook,* edited by Allen B. Tucker, Jr., CRC Press, Boca Raton, FL, 1997, pp. 1886-1909.

[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*, Vol. 8, No. 8, August 1997, pp. 857-871.

Biographies

Min Tan received his PhD degree from the School of Electrical and Computer Engineering at Purdue University, West Lafayette, Indiana, USA in 1997. He currently is with the technical staff of Cisco Systems, Inc. He attended Shanghai Jiao Tong University, Shanghai, People's Republic of China, in 1988. In 1991, he went to Western Maryland College, Maryland, USA, and received a BA degree in Mathematics and Physics in 1993. In 1994, he received an MSEE degree from the School of Electrical Engineering at Purdue University. While at Purdue, he received the "Estus H. and Vashti L. Magoon Outstanding Teaching Assistant Award" in 1996. He also served as a program committee member for the 1998 Heterogeneous Computing Workshop. His research interests include data source management in heterogeneous computing, data staging issues for network communication, video compression and financial applications on parallel and distributed systems, and dynamic partitionability for reconfigurable parallel processing machines. He has authored or coauthored 15 technical papers in these and related areas. He is a member of IEEE, the IEEE Computer Society, and the Eta Kappa Nu honorary society.

Mitchell D. Theys is a PhD student and Research Assistant in the School of Electrical and Computer Engineering at Purdue University. He received a Bachelor of Science in Computer and Electrical Engneering from the school of Electrical Engineering in 1993 with Highest Distinction from Purdue University, and an MSEE from the school of Electrical Engineering at Purdue University in 1996. He has received a Benjamin Meisner Fellowship from Purdue University for the 1996-1997 academic year, and an Intel Graduate Fellowship for the 1997-1998 academic year. He is a member of the Eta Kappa Nu honorary society, IEEE, and IEEE Computer Society. He was elected President of the Beta Chapter of Eta Kappa Nu at Purdue University and has held several various offices during his stay at Purdue. He has held positions with Compaq Computer Corporation, S&C Electic Company, and Lawrence Livermore National Laboratory. His research interests include design of single chip parallel machines, heterogeneous computing, parallel processing, and software/hardware design.

H. J. Siegel is a Professor in the School of Electrical and Computer Engineering at Purdue University. He is a Fellow of the IEEE (1990) and a Fellow of the ACM (1998). He received BS degrees in both electrical engineering and management (1972) from MIT, and the MA (1974), MSE (1974), and PhD degrees (1977) from the Depart-

ment 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* (second edition 1990). He was a Coeditor-in-Chief of the *Journal of Parallel and Distributed Computing* (1989-1991), and was on the Editorial Boards of the *IEEE Transactions on Parallel and Distributed Systems* (1993-1996) and the *IEEE Transactions on Computers* (1993-1996). 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.

Prof. Siegel's heterogeneous computing research includes modeling, mapping heuristics, and minimization of inter-machine communication. He is the Principal Investigator of a joint ONR-DARPA/ISO grant to design efficient methodologies for communication in the heterogeneous environment of the Battlefield Awareness and Data Dissemination (BADD) program. He is an Investigator on the MSHN project, supported by the DARPA/ITO Quorum program to create a management system for a heterogeneous network of machines.

Prof. Siegel's other research interests include parallel algorithms, interconnection networks, and the PASM reconfigurable parallel machine. His algorithm work involves minimizing execution time by exploiting architectural features of parallel machines. Topological properties and fault tolerance are the focus of his research on interconnection networks for parallel machines. He is investigating the utility of the dynamic reconfigurability and mixed-mode parallelism supported by the PASM design ideas and the small-scale prototype.

Noah B. Beck is pursuing an MSEE degree from the School of Electrical and Computer Engineering at Purdue University, where he is currently a Research Assistant. His main research topic is data staging in heterogeneous networks. He has held many positions supporting Siemens Stromberg-Carlson's technical support staff, and has also worked as a Design Engineer in one of Intel Corporation's microprocessor design groups. Noah received his BS degree from Purdue University in 1997 in Computer Engineering. His research interests include parallel computing, computer architecture and organization, and heterogeneous computing. He is an active member of the Eta Kappa Nu honorary society.

Michael Jurczyk 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. Currently, he is an assistant professor at the Computer Engineering and Computer Science Department at the University of Missouri - Columbia. His research interests include parallel and distributed systems, interconnection networks for parallel and communication systems, ATM-networking, and networked multimedia.

An Efficient Group Communication Architecture over ATM Networks

Sung-Yong Park, Joohan Lee, and Salim Hariri
High Performance Distributed Computing (HPDC) Laboratory
Department of Electrical Engineering and Computer Science
Syracuse University
Syracuse, NY 13244
{sypark, jlee, hariri}@cat.syr.edu

Abstract

NYNET (ATM wide-area network testbed in New York state) Communication System (NCS) is a multithreaded message-passing tool developed at Syracuse University that provides low-latency and high-throughput communication services over Asynchronous Transfer Mode (ATM)-based high-performance distributed computing (HPDC) environments. NCS provides flexible and scalable group communication services based on dynamic grouping and tree-based multicasting. The NCS architecture, which separates the data and control functions, allows group operations to be implemented efficiently by utilizing the control connections when transferring status information (e.g., topology information, routing information). Furthermore, NCS provides several different algorithms for group communication and allows programmers to select an appropriate algorithm at runtime.

In this paper we overview the general architecture of NCS and present the multicasting services provided by NCS. We analyze and compare the performance of NCS with that of other message-passing tools such as p4, PVM, and MPI in terms of primitive performance and application performance. The benchmark results show that NCS outperforms other message-passing tools for both primitive performance and application performance.

1 Introduction

We are experiencing a rapid deployment of high-performance distributed systems (HPDS) that are typified by a heterogeneous collection of machines with widely differing performance characteristics and are connected by one or more high-speed networks. These systems combine workstations, shared-memory multiprocessors, and distributed-memory multicomputers. The high-speed network technologies used include Asynchronous Transfer Mode (ATM) [1], Myrinet [2], Gigabit Ethernet [3], High Performance Parallel Interface (HIPPI) [4], and wireless technologies. Consequently, the development of high-performance distributed computing (HPDC) applications is a non-trivial task that requires a thorough understanding of the application requirements and architecture, and the communication services provided.

HPDC applications require low-latency and high-throughput communication services comparable to that experienced in a bus-based parallel computer. HPDC applications have different Quality of Service (QoS) requirements and even one single application might have multiple QoS requirements during the course of its execution (e.g., interactive multimedia applications). Furthermore, a significant fraction of the traffic in HPDC applications is multi-point (e.g., video-conferencing, collaborative computing). In order to meet the requirements of a wide variety of HPDC applications, the parallel and distributed software systems should provide high performance and dynamic group communication services. The group communication services provided by traditional message-passing tools such as p4 [11], Parallel Virtual Machine (PVM) [12], Message-Passing Interface (MPI) [13], Express [15], and PARMACS [16] are fixed and thus can not be changed to meet the requirements of different HPDC applications. Furthermore, some message-passing tools such as PVM implement group communication operations by repeatedly calling send routines for each participant, which is computationally expensive and not scalable. There have been several distributed computing software tools specially designed to support group communication services such as Isis [18], Horus [19], Totem [20] and Transis [21]. However, most of them are designed to support spe-

0-8186-8365-1/98 $10.00

cial functionalities (e.g., fault tolerance, message ordering, virtual synchrony, group partition) rather than to achieve high throughput.

NYNET Communication System (NCS) [7, 8, 9] is a multithreaded message-passing tool for an ATM-based HPDC environment that provides low-latency and high-throughput communication services. NCS capitalizes on a thread-based programming model to overlap computation and communication, and develop a dynamic message-passing environment with separate data and control paths. This leads to a flexible, adaptive message-passing environment that can support multiple flow-control, error-control, and multicasting algorithms. This paper overviews the general architecture of NCS and presents the multicasting services provided by NCS. NCS multicasting services are based on dynamic grouping, where each process can dynamically create, join, or leave a group. NCS uses a binary tree to implement multicasting operations, which is more efficient and scalable than repetitive techniques especially when the number of groups is large. Furthermore, NCS group communication services can be implemented using different group communication algorithms. These algorithms can be selected by the application at runtime.

The rest of the paper is organized as follows. Section 2 outlines the general architecture of NCS. Section 3 discusses an approach to implement the NCS multicasting services. Section 4 analyzes and compares the multicasting performance of NCS with that of other message-passing tools such as p4, PVM, and MPI. Section 5 contains the summary and conclusion of the paper.

2 NCS Overview

In this section we present an overview of the NCS architecture. Additional details about NCS architecture can be found in [9].

Figure 1 shows the general architecture of NCS. An NCS application consists of multiple *Compute_Threads* that include programs to perform the computations of the application. NCS supports both the *host-node* programming model and the Single Program Multiple Data (SPMD) programming model. In both models processes are created at each node by using the *hostfile* that specifies the initial configurations of machines to run NCS applications. After each process is spawned, it creates multiple *Compute_Threads* according to the computation requirements of the application. The advantage of using a thread-based programming paradigm is that it reduces the cost of context switching, provides efficient support for fine-grained applications, and allows the overlapping of

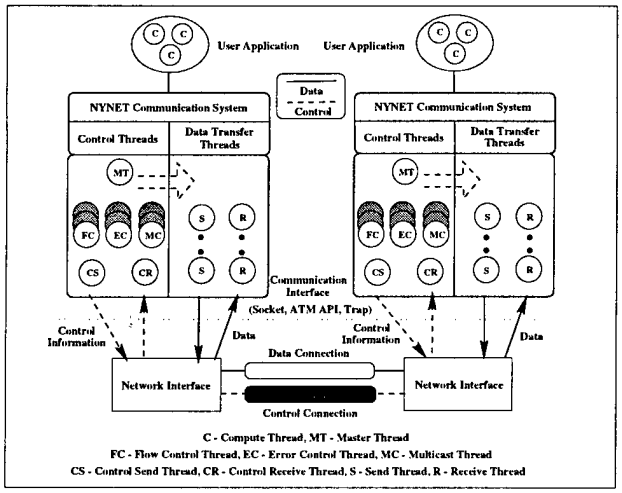


Figure 1: NCS General Architecture

computation and communication.

NCS separates control and data functions by providing two planes (see Figure 1): a *control plane* and a *data plane*. The control plane consists of several threads that implement important control functions (e.g., connection management, flow control, error control) in an independent manner. These threads include *Master_Thread*, *Flow_Control_Thread*, *Error_Control_Thread*, *Multicast_Thread*, *Control_Send_Thread*, and *Control_Receive_Thread* (we call them *control threads*). The *data transfer threads* (*Send_Thread* and *Receive_Thread*) in the data plane are spawned on a per-connection basis by the *Master_Thread* to perform only the data transfers associated with a specific connection. Furthermore, the control and data information from the two planes are transmitted on separate connections. All control information (e.g., flow control, error control, configuration information) is transferred over the control connections, while the data connections are used only for the data transfer functions. The separation of control and data functions eliminates the process of demultiplexing control and data packets within a single connection and allows the concurrent processing of control and data functions. This allows applications to utilize all available bandwidth for the data transfer functions and thus improves the performance.

NCS supports multiple flow-control (e.g., window-based, credit-based, or rate-based), error-control (e.g., go-back N or selective repeat), and multicasting algorithms (e.g., repetitive send/receive or a multicast spanning tree) within the control plane to meet the QoS requirements of a wide range of HPDC applications. Each algorithm is implemented as a thread and

programmers activate the appropriate thread when establishing a connection to meet the requirements of a given connection.

NCS provides three application communication interfaces such as *socket* communication interface (SCI), ATM communication interface (ACI), and high-performance interface (HPI) in order to support HPDC applications with different communication requirements. The SCI is provided mainly for applications that must be portable to many different computing platforms. The ACI provides the services that are compatible with ATM connection-oriented services where each connection can be configured to meet the QoS requirements of that connection. The HPI supports applications that demand low-latency and high-throughput communication services.

3 Multicasting Support in NCS

The implemented NCS multicasting algorithm is based on dynamic grouping, where each NCS process can dynamically create, join, or leave a group during the lifetime of the process. Within each group, there is a single group server that is responsible for intergroup communications and multicasting. The multicasting operation in NCS is implemented by using a binary tree. This approach is more efficient than repetitive techniques especially when the number of groups is large. In addition, the separation of control and data functions facilitates the development of efficient multicasting. For example, when the status of each process has been changed, it can be broadcast promptly to other processes without interfering with data traffic. This allows NCS to prepare most of the information needed to activate multicasting operations (e.g., tree information, group information) in advance before the actual multicasting operations are initiated. This reduces the set-up time (e.g., time to build a tree at runtime) of the multicasting operations and thus improves the performance of NCS group communication services. Other multicasting algorithms can be incorporated into NCS and activated at runtime by user applications without changing the NCS architecture and its supported group communication services.

In what follows we define the NCS group communication primitives and describe the NCS multicasting algorithm to implement these primitives.

3.1 NCS Group Communication Primitives

Figure 2 shows a set of NCS primitives that provide group communication services.

NCS multicasting primitive (*NCS_mcast()*) supports three classes of multicasting operations: (1) *global broadcast*, (2) *local broadcast*, and (3) *global multicast*. The global broadcast is used to transmit messages to all groups defined in the NCS applications. The local broadcast is used to transmit messages to all members within the same group. The global multicast is used to transmit messages to the specified groups. For all three operations, the destination endpoint is not the members, but the group servers. They can be invoked with either a reliable mode or an unreliable mode. The data-type of message (e.g., *char, int, float, double, etc*) and the message type can be specified by providing parameters to the *NCS_mcast()* primitive.

```
int NCS_mcast(int mode, char *gname[], NCS_Dtype type,
        int tag, char *msg, int len);
   - Multicasts a message to the groups specified by gname[].

int NCS_group_create(char *gname, int com_mode, int fc, int ec,
        int mc, struct QoS);
   - Creates a group named gname. Returns the group identifier.

int NCS_group_join(char *gname);
   - Joins the group specified by gname.

int NCS_group_destroy(char *gname);
   - Destroys the group specified by gname.

int NCS_group_leave(char *gname);
   - Leaves the group specified by gname.

int NCS_group_num_members(char *gname);
   - Returns the total number of members in the group.
```

Figure 2: NCS Group Communication Primitives

Users can create a new group by using the *NCS_group_create()* primitive. In this case a particular communication scheme (e.g., error-control algorithm, flow-control algorithm, multicasting algorithm), a particular communication interface (e.g., SCI, ACI, HPI), and ATM QoS parameters can be assigned to the group communication channel (e.g., binary tree). All the new processes that join this group by invoking *NCS_group_join()* primitive use the same communication scheme and communication interface when sending data over the group communication channel. The attributes assigned to this channel cannot be changed by the group members during program execution and they are released when the group is destroyed by using the *NCS_group_destroy()* primitive.

3.2 NCS Multicasting Algorithm

At program startup, a default NCS group called *NCS_GRP* is created, and each NCS process in the

hostfile joins this group automatically (see Figure 3). The *hostfile* is used to specify a list of machines to run NCS applications. The first process specified in the *hostfile* becomes a *master group server* (MGS). Each process that creates a new group becomes a *local group server* (LGS) of that group. The MGS represents all the LGSs and coordinates the group communication operations between these servers. The LGS is responsible for multicasting operations within the local group and maintains the membership information of the local group. A *global multicasting tree* (GMT) is built to connect all the LGSs rooted at the MGS. All the group members within the same group are connected by a *local multicasting tree* (LMT) rooted at the LGS of that group. The MGS and LGSs periodically exchange the status information of each group over the control connections.

Since three classes of multicasting operations (e.g., global broadcast, local broadcast, and global multicasting) are implemented using similar schemes, we will only describe the algorithm for global broadcast. The multicasting algorithm for global broadcast consists of six steps, as shown in Figure 4:

1. When the *Compute_Thread* of a process invokes the *NCS_mcast()* primitive, the *Multicast_Thread* of that process activates the corresponding *Send_Thread* to transmit an actual message to the MGS.
2. The MGS transmits the received message to the other LGSs using its GMT.
3. If the *NCS_mcast()* is invoked with reliable mode, each LGS that received the message sends an acknowledgment back to the MGS along the GMT.
4. An LGS maintains two buffers. The first buffer is used to assemble the messages, which are then transferred to the second buffer. The second buffer is used to retransmit the messages to the members that have not correctly received the messages.
5. Each LGS locally multicasts the message to its group members using its LMT.
6. If the *NCS_mcast()* is invoked with a reliable mode, each member that received the message sends an acknowledgment back to the LGS along the LMT. If there is any group member that has not received a message within the timeout period, the LGS of the group retransmits the message. This reduces the retransmission traffic from the source process.

The pseudo code for this algorithm is presented in Figure 5.

4 Benchmarking Results

In this section we analyze and compare the performance of NCS with that of other message-passing tools such as p4, PVM, and MPI using two levels of performance evaluation [10]: tool performance level (TPL) and application performance level (APL). In TPL we benchmark the performance of the broadcasting primitives provided by each message-passing tool, while in APL we compare the execution time of two applications (e.g., Back-Propagation Neural Network (BPNN) learning algorithm and static voting algorithm).

All experiments have been conducted over six SUN-4 workstations and four IBM RS/6000 workstations interconnected by an IBM 8260 ATM switch and a Cabletron MMAC-Plus ATM switch. In all measurements we used the NCS implementation over SCI. Consequently, the effect of error control and flow control is not considered in these experiments. The socket buffer size was set to 32 Kbytes and the TCP_NODELAY option was enabled. It is reported in [5] that setting those two options improves the socket throughput. For the PVM (Version 3.3.11) applications, we used the PVM Direct mode, where the direct TCP connection is made between two endpoints. The MPICH [14] (Version 1.0.13) was used to benchmark the MPI applications.

4.1 Tool Performance Level (TPL)

Figure 6 compares the performance of broadcasting primitives (e.g., *NCS_mcast()*, *pvm_mcast()*, *p4_broadcast()*, and *MPI_Bcast()*) of four message-passing tools over an ATM network when message sizes vary from 1 byte to 32 Kbytes. The group size varies from two to ten. Since ten heterogeneous workstations (six SUN-4 workstations and four IBM/RS6000 workstations) were used for measuring the timings, the results for the group size up to six represent the characteristics of broadcasting primitives over the SUN-4 platform, while the results for the group sizes of eight and ten represent the characteristics of broadcasting primitives over heterogeneous platforms.

As we can see from Figure 6, the execution time of each broadcasting primitive increases linearly for small message sizes up to 1 Kbytes, while it shows different patterns for large message sizes over 1 Kbytes.

NCS primitive (*NCS_mcast()*) achieved better performance (e.g., about five times faster than p4 and MPI) for various message sizes and group sizes. For

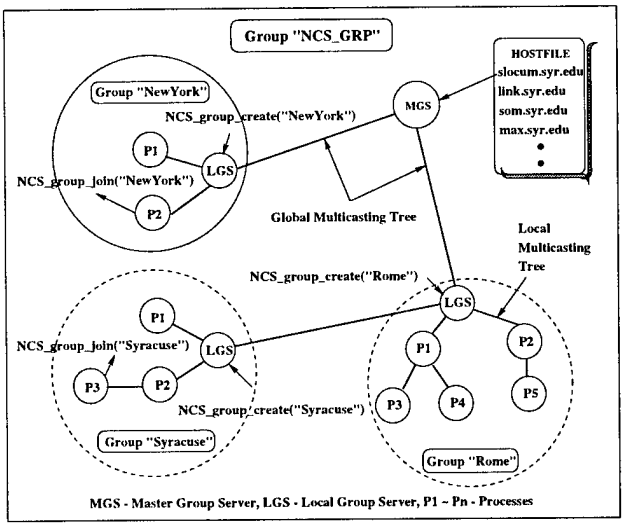


Figure 3: Group Structure in the NCS Environment

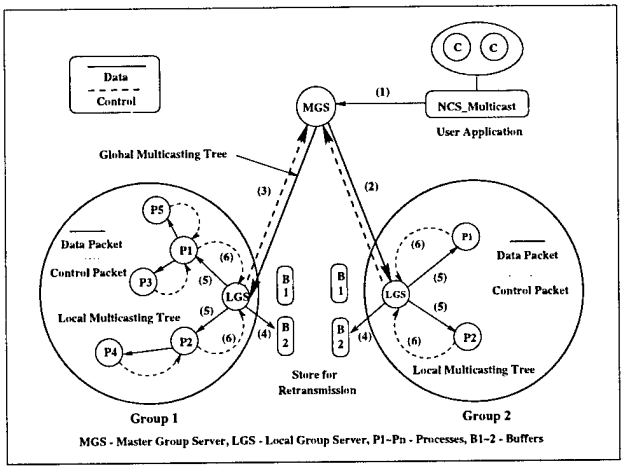


Figure 4: Multicasting in the NCS Environment

Thread *Master Group Server* (MGS)
  repeat Get the requests from other servers or members
    if group creation or destruction is requested **then**
      Update the GMT and send the information to the LGSs over the control path
    else if a *Global Broadcast* is requested
      Send the message to the LGSs along the GMT
      if a reliable multicast is requested **then**
        Check the ACKs from the LGSs and retransmit if necessary
      endif
    endif
  end

Thread *Local Group Server* (LGS)
  repeat Get the requests from other servers or members
    if group creation, destruction, join or leave are notified **then**
      Update the local database (or LMT) and send the information to all the members over the control path
    else if a message received for *Global Broadcast* or *Global Multicast* **then**
      Send the message to all the members along the LMT
      Route the message to other LGSs if necessary
      if a reliable multicast is requested **then**
        Merge the ACKs from the LGSs and send an ACK to its parent
        Check the ACKs from the members and retransmit if necessary
      endif
    else if a *Local Broadcast* is requested
      Send the message to all the members along the LMT
      if a reliable multicast is requested **then**
        Check the ACKs from the members and retransmit if necessary
      endif
    endif
  end

Thread *Multicasting Thread*
  if group creation, destruction, join or leave are notified **then**
    Update the local database for this information
  else if *Global Broadcasting* or *Local Broadcasting* are requested **then**
    Send the message to the MGS (*Global Broadcasting*) or LGS (*Local Broadcasting*)
  else if a *Global Multicasting* is requested **then**
    Setup a spanning tree at runtime and send the message to the LGSs along the new spanning tree
  endif

Figure 5: Pseudo code for the Multicasting Protocol

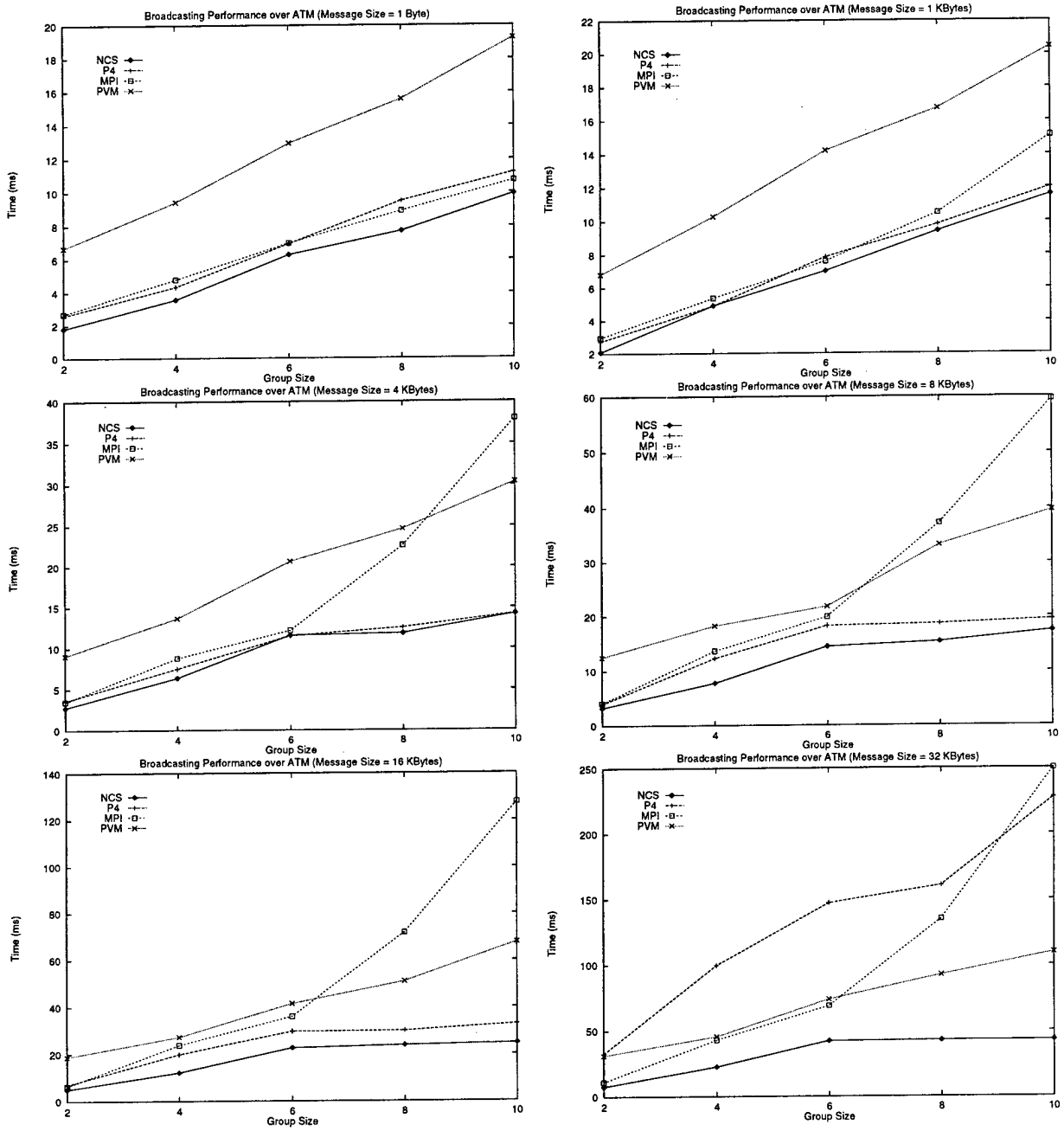


Figure 6: Comparison of Broadcasting Performance over ATM

example, given a message size of 32 Kbytes, the NCS broadcasting time is 42.966 *milli*seconds, while p4, PVM, and MPI took 227.568 *milli*seconds, 109.403 *milli*seconds, and 249.961 *milli*seconds, respectively. Furthermore, *NCS_mcast()* primitive shows almost similar performance for large group sizes as we increase the message size. For a message size of 16 Kbytes, the NCS broadcasting time using six members is 22.596 *milli*seconds and the broadcasting time using ten members is 24.623 *milli*seconds. In the *NCS_mcast()* primitive where most of the information for performing group communications (e.g., setup binary tree, setup routing information) is set up in advance by using the control connections, the start-up time for the broadcasting operations is very small. Also, the tree-based broadcasting scheme improves the performance as the group size gets larger. Consequently, the larger the message size and group size, the better is the performance of NCS when compared to that of other message-passing tools.

The performance of p4 primitive (*p4_broadcast()*) is comparably good except for large message sizes. For message size of 32 Kbytes, p4 performance gets worse rapidly as we increase the group size. One of the reasons for this is that p4 has also low performance for point-to-point communications with large message sizes, as shown in Figures 7 and 8.

The performance of PVM primitive (*pvm_mcast()*) is poor for small message sizes but as the message size and group size increase, its performance improves. In the *pvm_mcast()* where the broadcasting operation is implemented by repeatedly invoking a send primitive, the performance is expected to increase linearly as we increase the group size. Moreover, *pvm_mcast()* constructs a multicasting group internally for every invocation of the primitive, which results in a high start-up time when transmitting small messages as shown in Figure 6 (message size 1 byte and 1 Kbytes).

The MPI primitive (*MPI_Bcast()*) shows comparable performance to NCS and p4 for relatively small message sizes (e.g., up to 1 Kbyte) and small group sizes (e.g., up to 6 members) but its performance degrades drastically for message sizes larger than 4 Kbytes and large group sizes (e.g., over six members).

4.2 Application Performance Level (APL)

In this subsection we compare the performance of NCS with that of other message-passing tools by measuring the execution time of two applications (i.e., BPNN learning algorithm and static voting algorithm) that require intensive group communication services.

BPNN Learning Algorithm

Training BPNN for character recognition is one of the problems in Artificial Intelligence (AI) area that requires intensive group communications. We used a master/slave programming model to parallelize this application, as shown in Figure 9. In this algorithm the *master* process distributes the weight vectors between the input layer and the hidden layer to the *slave* processes. The *slave* processes receive weight vectors from the *master* process and compute the output values of the hidden nodes allocated to them, then transmit those output values back to the *master* process. After the *master* process receives the output values of the hidden nodes from the *slave* processes, it computes the output values of the output nodes, computes mean-squared error, and updates the weights vectors between input layer and hidden layer, and between hidden layer and output layer. These steps continue until the value of the mean-squared error falls under an appropriate value. This application intensively uses the broadcasting primitives when distributing the weight vectors to all the *slave* processes. The BPNN used in this experiment has 100 input nodes, 630 hidden nodes, and 4 output nodes to train 16 input vectors which represent the hexadecimal digits from 0x01 to 0x0F.

Static Voting Algorithm

Replicating data at different locations is a common approach to achieve fault tolerance in distributed computing systems. One well-known technique to manage replicated data is voting mechanisms. The algorithm used in this experiment is based on the static voting scheme proposed by Gifford [22]. In this algorithm (See Figure 10) we assume that there is a file server process in each node that handles *read* and *write* requests for a given file. Each file server process issues arbitrary *read* and *write* requests that were produced randomly using a random number generator. Whenever a server process issues a file access request, it sends a *Lock_Request* message for that file to the local lock manager and broadcasts a *Vote_Request* message to all other server processes. When the server process receives a *Vote_Request* message from other server processes, it sends a *Lock_Request* message for the requested file to the local lock manager. The server process then returns the version number of the replica and the number of votes assigned to the replica to the server process that initiated the *Vote_Request*. Based on the information returned from other server processes, the server process decides if the file access is granted and the file is the latest copy. If the local

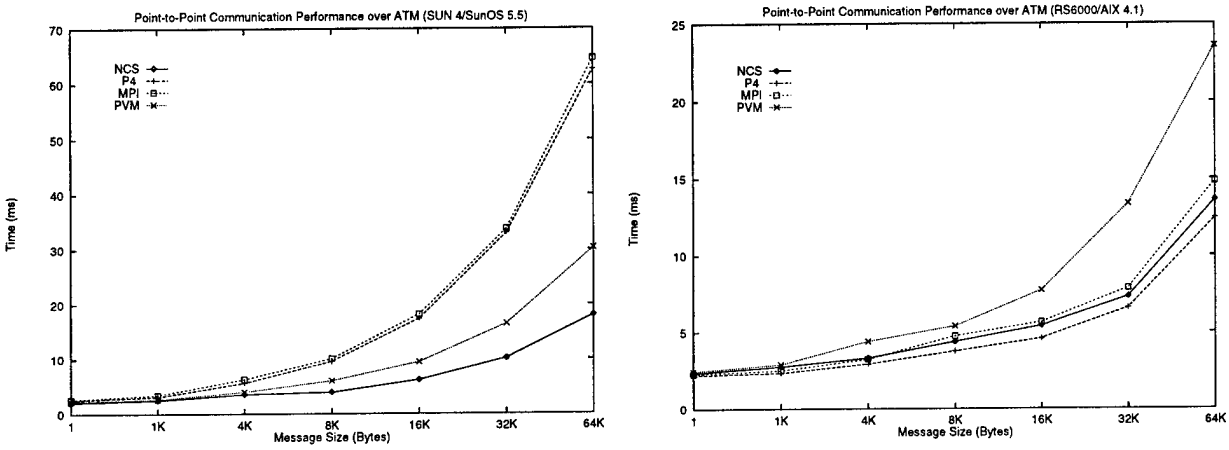


Figure 7: Point-to-Point Communication Performance over ATM Using Same Platform

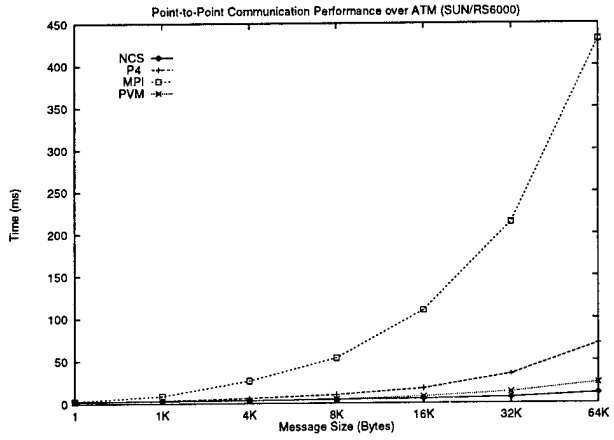


Figure 8: Point-to-Point Communication Performance over ATM Using Heterogeneous Platform

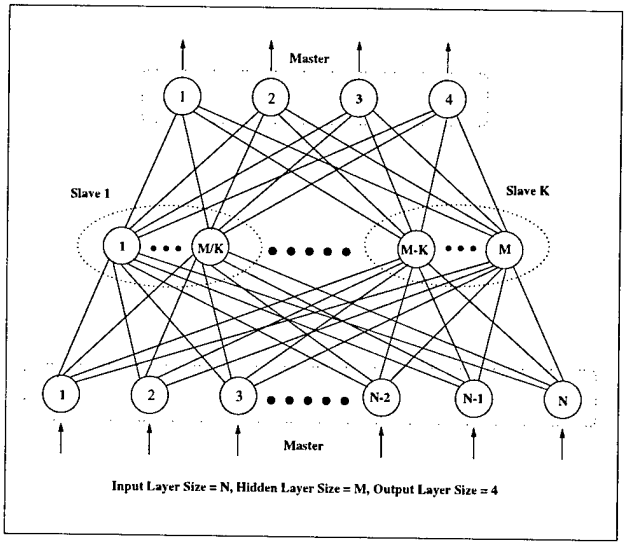


Figure 9: Back-Propagation Neural Network (BPNN) Learning Algorithm

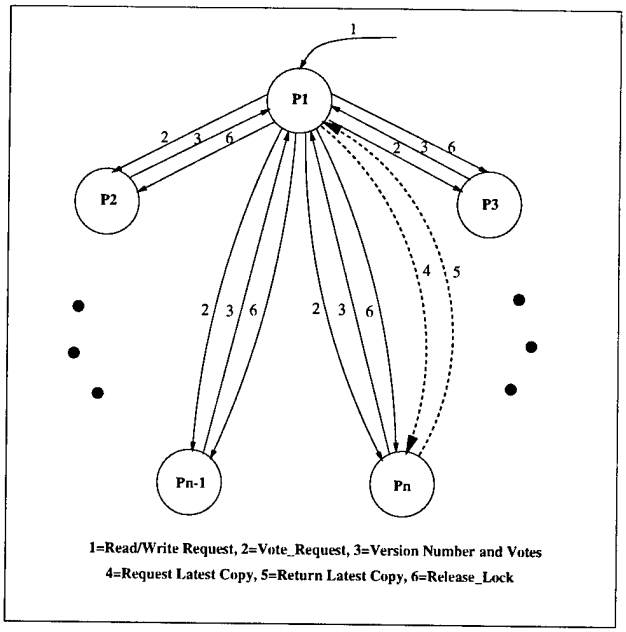


Figure 10: Static Voting Algorithm

copy is different from those replicated at other server processes, it gets the latest copy from other server processes. Finally, the file server process broadcasts a *Release_Lock* message to all other file servers if the file access is granted. In this experiment we assumed that there are 50 different files replicated at each node and each file server process generates 500 *read* or *write* requests for arbitrary files.

Performance Comparison

Figure 11 shows the performance of each message-passing tool to implement these two applications running over four homogeneous workstations (e.g., four SUN-4 workstations running SunOS 5.5 or four IBM/RS6000 workstations running AIX 4.1) and eight heterogeneous workstations (e.g., four SUN-4 workstations and four IBM/RS6000 workstations) interconnected by an ATM network. Due to the restrictions of the MPI broadcasting primitive (*MPI_Bcast()*), we couldn't implement the static voting algorithm using MPI. In MPI all messages broadcast using the *MPI_Bcast()* should be received by other processes using the *MPI_Bcast()* primitive instead of the receive primitive. Furthermore, one of the argument of this primitive represents the rank of the *root* process that initiated the broadcasting operation and this value should be identical on all processes that receive the message. Since the broadcasting operations in static voting algorithm are initiated randomly by different processes, it is difficult to obtain the *root* of the broadcasting operation. Consequently, implementing static voting algorithm using MPI is not straightforward.

As shown in Figure 11, the message-passing tool that has the best performance at TPL also has the best performance at APL. For example, NCS applications outperform other implementations regardless of the platform used. In the BPNN application using eight heterogeneous workstations, the execution time of NCS is 135 seconds, while p4, PVM, and MPI took 1088 seconds, 429 seconds, and 620 seconds, respectively. In the BPNN application where large messages are broadcast repeatedly, the performance improvement is noticeable and it improves further as we increase the group size. In the static voting application where the sizes of the broadcasting messages are small and the communications take place randomly, the performance of NCS is comparable to that of other message-passing tools for small size groups but the performance gap gets wider as we increase the group size. We believe that most of the improvements of NCS are due to overlapping of communications and computations and the use of tree-based broadcasting algorithm.

On the other hand, PVM implementations show better performance than MPI and p4 implementations in heterogeneous environment.

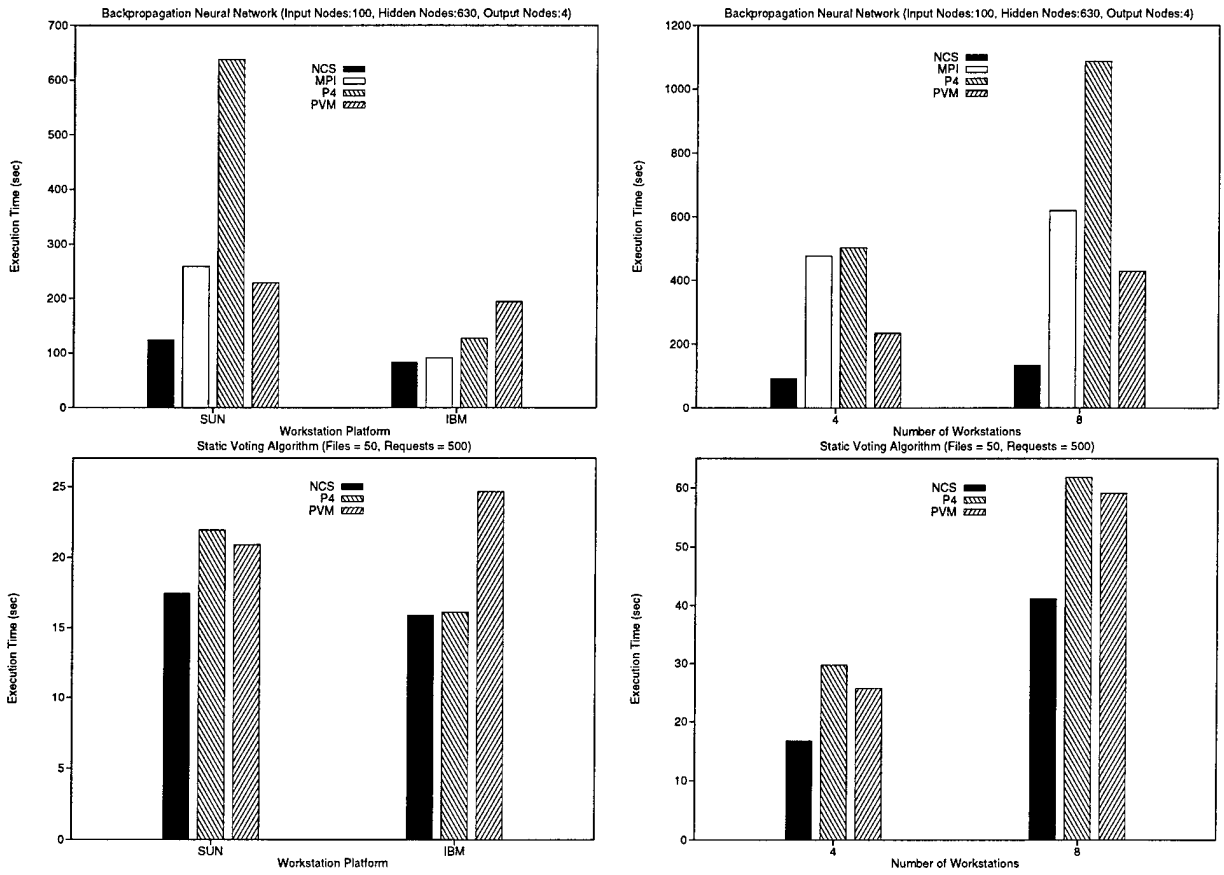


Figure 11: Comparison of Application Performance

5 Conclusion

In this paper we have presented NCS architecture that provides efficient and flexible group communication services over an ATM network. We have evaluated the performance of NCS group communication primitives and applications. The benchmark results showed that NCS outperforms other message-passing tools. It is clear that the NCS novel architecture, which separates the data and control functions and the use of tree-based multicasting scheme played an important role in improving the performance of the communication primitives and applications.

References

[1] J. Y. Le Boudec, "The Asynchronous Transfer Mode: a tutorial", *Computer Networks and ISDN Systems*, Vol. 24, No. 4, pp. 279–309, 1992.

[2] N. J. Moden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and W. Su, "Myrinet: A Gigabit-per-second Local Area Network", *IEEE Micro*, Vol. 15, No. 1, pp. 29–36, February 1995.

[3] Gigabit Ethernet Alliance, "Gigabit Ethernet Overview", White Paper, September 1997.

[4] D. Tolmie, and J. Renwick, "HIPPI: Simplicity Yields Success", *IEEE Network*, pp. 28–32, January 1993.

[5] M. Lin, J. Hsieh, D. Du, and J. Thomas, "Distributed Network Computing over Local ATM Networks", *IEEE Journal on Selected Areas in Communications*, Vol. 13, No. 4, pp. 733–747, May 1995.

[6] S. Hariri, S. Y. Park, R. Reddy, M. Subramanyan, R. Yadav, and M. Parashar, "Software Tool Evaluation Methodology", *Proc. of the 15th International Conference on Distributed Computing Systems*, pp. 3–10, May 1995.

[7] S. Y. Park, S. Hariri, Y. H. Kim, J. S. Harris, and R. Yadav, "NYNET Communication System (NCS): A Multithreaded Message Passing Tool over ATM Network", *Proc. of the 5th International Symposium on High Performance Distributed Computing*, pp. 460–469, August 1996.

[8] S. Y. Park and S. Hariri, "A High Performance Message Passing System for Network of Workstations", *The Journal of Supercomputing*, to appear.

[9] S. Y. Park, J. Lee, and S. Hariri, "A Multithreaded Communication System for ATM-Based High Performance Distributed Computing Environments", *Submitted to IEEE Transactions on Parallel and Distributed Systems*, 1997.

[10] S. Y. Park, J. Lee, and S. Hariri, "An Evaluation Methodology for Parallel/Distributed Software Tools", *Submitted to IEEE Transactions on Parallel and Distributed Systems*, 1997.

[11] R. Butler and E. Lusk, "Monitors, message, and clusters: The p4 parallel programming system", *Parallel Computing*, Vol. 20, pp. 547–564, April 1994.

[12] V. S. Sunderam, "PVM: A Framework for Parallel Distributed Computing", *Concurrency: Practice and Experience*, Vol. 2, No. 4, pp. 315–340, December 1990.

[13] MPI Forum, "MPI: A Message Passing Interface", *Proc. of Supercomputing '93*, pp. 878–883, November 1993.

[14] B. Gropp, R. Lusk, T. Skjellum, and N. Doss, "Portable MPI Model Implementation", Argonne National Laboratory, July 1994.

[15] J. Flower, and A. Kolawa, "Express is not just a message passing system. Current and future directions in Express", *Journal of Parallel Computing*, Vol. 20, No. 4, pp. 597–614, April 1994.

[16] S. Gillich, and B. Ries, "Flexible, portable performance analysis for PARMACS and MPI", *Proc. of High Performance Computing and Networking: International Conference and Exhibition*, May, 1995.

[17] L. Dorrmann, and M. Herdieckerhoff, "Parallel Processing Performance in a Linda System", *International Conference on Parallel Processing*, pp. 151–158, 1989.

[18] K. P. Birman, R. Cooper, T. A. Joseph, K. P. Kane, F. Schmuck, and M. Wood, "Isis - A Distributed Programming Environment", User's Guide and Reference Manual, Cornell University, June 1990.

[19] R. Renesse, T. Hickey, and K. Birman, "Design and performance of Horus: A lightweight group communications system", Technical Report TR94-1442, Cornell University, 1994.

[20] L. E. Moser, P. M. Melliar-Smith, D. A. Agarwal, R. K. Budhia and C. A. Lingley-Papadopoulos, "Totem: A Fault-Tolerant Multicast Group Communication System", *Communications of the ACM*, Vol. 39, No. 4, pp. 54-63, 1996.

[21] D. Dolev and D. Malki, "The Transis Approach to High Availability Cluster Communication", *Communications of the ACM*, Vol. 39, No. 4, pp. 64–70, 1996.

[22] D. K. Gifford, "Weighed Voting for Replicated Data", *Proc. of the 7th ACM Symposium on Operating System*, pp. 150–162, December, 1979.

Biographies

Sung-Yong Park received a BS degree in computer science from Sogang University, Korea, in 1987 and MS degree in computer science from Syracuse University, Syracuse, NY in 1994. He is currently working toward the PhD degree in computer science at Syracuse University. From 1987 to 1992, he has worked as a research engineer at LG Electronics (former Goldstar Telecommunication), Korea. From 1993 to 1996, he has worked at the Northeast Parallel Architectures Center (NPAC) at Syracuse University as a research assistant on ISDN and ATM networking. Since 1996, he has been with Computer Applications and Software Engineering Center (CASE) at Syracuse University. His research interests include high performance distributed systems, high speed networks, network computing, and multimedia.

Joohan Lee received BS and MS degrees in computer science from Sogang University, Korea, in 1993 and 1995 respectively, where he worked in artificial intelligence. He is currently pursuing a Ph.D. degree in computer science at Syracuse University. Since 1996, he has worked at High Performance Distributed Computing Laboratory at Computer Applications and Software Engineering Center (CASE) in Syracuse University. His research interests include high performance distributed computing and multimedia.

Salim Hariri is currently an Associate Professor in the Department of Electrical Engineering and Computer Science at Syracuse University. He is the director of the High Performance Distributed Computing Laboratory at Syracuse University (www.atm.syr.edu). He received his Ph.D. in computer engineering from University of Southern California in 1986and an MSc from the Ohio State university in 1982. His current research focuses on high performance distributed computing, high speed networks and protocols, network management, and performance.

Panel Session

Is Java the Answer for Programming Heterogenous Computing Systems?

Panel Chair

Gul A. Agha
University of Illinois, Urbana-Champaign, IL, USA

Panel Description:

One factor that complicates the programming of heterogenous computing systems is the absence of a portable, high-performance programming language. The widespread interest and use of Java for remote execution on the Web has demonstrated the ease and practicality of using high-level programming for heterogeneous computing. The focus of this panel is the use of Java for programming heterogenous computing systems with a higher degree of both inter- and intra-machine concurrency. Representative questions addressed by the panelists include:

Can Java deliver the potential of HC systems for high-performance, availability, etc.?

What additional constructs are needed for Java to effectively support concurrency and distribution?

Does Java provide true portability and mobility?

Will Java be accepted by application programmers of HC systems?

What are the challenges in using machine-dependent libraries with Java?

Modular Heterogeneous System Development: A Critical Analysis of Java

Gul A. Agha, Mark Astley, Jamil A. Sheikh, and Carlos Varela
Department of Computer Science
Univ. of Illinois at Urbana-Champaign
Urbana, IL 61801, USA
Phone: (217) 244-3087
Email: agha@cs.uiuc.edu

Abstract

*Java supports heterogeneous applications by transforming a heterogeneous network of machines into a homogeneous network of Java virtual machines. This approach abstracts over many of the complications that arise from heterogeneity, providing a uniform API to all components of an application. However, for many applications heterogeneity is an intentional feature where components and resources are co-located for optimal performance. We argue that Java's API does not provide an effective means for building applications in such an environment. Specifically, we suggest improvements to Java's existing mechanisms for maintaining consistency (*e.g. `synchronized`*), and controlling resources (*e.g. *thread scheduling*). We also consider the recent addition of a CORBA API in JDK 1.2. We argue that while such an approach provides greater flexibility for heterogeneous applications, many key problems still exist from an architectural standpoint. Finally, we consider the future of Java as a foundation for component-based software in heterogeneous environments and suggest architectural abstractions which will prove key to the successful development of such systems. We drive the discussion with examples and suggestions from our own work on the Actor model of computation.*

1 Classifying Heterogeneity

Heterogeneous computing environments arise in practice for a number of different reasons; heterogeneity, however, generates the same basic set of problems: code is not portable, shared data may need to be converted, the utilization of certain resources may be restricted to specific nodes, and so on. Nonetheless, the solution for these problems depends heavily on the types of applications that are deployed in the heterogeneous environment. As an example, consider the following two instances of heterogeneity:

- **System Evolution:** Corporate computing environments are continually evolving as outdated systems are gradually replaced with newer, more powerful systems. However, although the hardware is constantly replaced, corporations are often dependent on monolithic applications that must continue to run correctly in the presence of new hardware.
- **Specialized Hardware:** Certain computing environments are intentionally designed to be heterogeneous in order to utilize specialized hardware. Numeric simulations, for example, may be executed on massively parallel systems while monitoring and analysis is performed on graphics-intensive workstations. As another example, servers with high availability requirements are placed on hardware with large pools of available resources whereas clients execute on low-end workstations designed for single users.

The solution for an evolving corporate system depending on existing software might involve the development of a common execution environment atop each physical node. Thus, as long as existing applications are written in terms of this uniform environment, they will continue to be usable as future improvements are made. On the other hand, specialized hardware might be handled using an environment in which customized objects, targeted for specific hardware, coordinate with one another through a common interface for interactions. Still other environments, may utilize a hybrid of these two solutions.

Many languages and programming environments exist for managing heterogeneous computing environments. The Java programming language is an example

0-8186-8365-1/98 $10.00

which *directly* addresses the technical problems created by a heterogeneous environment. In the words of its designers [6]:

> Java is designed to meet the challenges of application development in the context of heterogeneous, network-wide distributed environments. Paramount among these challenges is secure delivery of applications that consume the minimum of system resources, can run on any hardware and software platform, and can be extended dynamically.

CORBA, COM and other Object Request Broker (ORB) based environments represent so called "middle-ware" solutions. That is, rather than address heterogeneity directly, these environments provide a mechanism for allowing interactions between applications executing in heterogeneous environments.

In general, we may characterize the Java approach as the transformation of a heterogeneous network of machines into a homogeneous network of Java virtual machines. Java makes no effort to abstract over network features or cater to highly-optimized (but non-portable) implementations. However, Java does greatly simplify network access and provides a native method interface as a loop-hole for incorporating non-Java code. On the other hand, ORB-based systems make little or no effort to transform heterogeneous systems into homogeneous ones. Instead, ORBs solve the problem of interactions between heterogeneous environments. While such a solution limits mobility, applications may directly access high-performance implementations executing on dedicated hardware.

Both the Java and ORB-based solution have their merits. However, we argue that in order for Java to become "the answer" for programming heterogeneous computing systems, it must incorporate many of the features already present in ORBs. In particular, to answer the challenge of high-performance systems, Java must make local, optimized servers more available to Java clients. Currently, there are joint efforts between Sun and OSF to link CORBA and Java for precisely this reason [11]. However, we believe that while Java should be more ORB-like, it should also overcome many of the weaknesses of existing ORBs such as the inability to customize interactions between ORB-served objects. Moreover, to effectively support concurrency and distribution, we claim that Java requires more powerful constructs for controlling synchronization and coordination between distributed entities. We find existing Java synchronization (e.g. the `synchronized` keyword) to be too low-level and unsuitable for distributed needs. The lack of control over resource management tasks such as thread scheduling is also undesirable.

We envision Java as evolving to support distributed *collections* of objects executing over heterogeneous computing environments. In such an environment, application developers may specify services consisting of (possibly) distributed collections of Java and native objects. Services would be composed with policies which manage both interactions as well as deployment. These policies would encapsulate many of the solutions currently employed for heterogeneous environments: protocols which marshal arguments, routing mechanisms which link client requests to optimized objects executing on custom hardware, and so on.

In the next section, we discuss some weaknesses of the current version of Java as well as potential solutions. In Section 3, we describe features of ORB-based models which we believe should be incorporated into Java. In addition, we propose solutions for a Java-ORB system which overcomes many of the current weaknesses of the ORB-based model. In Section 4, we present a future vision of Java as a tool for implementing large grain coordination and management for heterogeneous applications. We describe lessons learned from our research in Actor [2] systems and propose several abstractions to be incorporated in future Java developments. We present concluding remarks in Section 5.

2 Heterogeneity in Java

Software executing in a heterogeneous environment is naturally segmented into a collection of distributed, coordinating objects. As a result, desirable system features such as ease of management and high performance depend on the ability to specify error-free coordination mechanisms which exploit available concurrency. Java uses a passive object model in which threads and objects are separate entities. As a result, Java objects serve as surrogates for thread coordination and do not abstract over a unit of concurrency. We view this relationship between Java objects and threads to be a serious limiting factor in the utility of Java for heterogeneous systems. Specifically, while multiple threads may be active in a Java object, Java only provides the low-level `synchronized` keyword for controlling object state, and lacks higher-level linguistic mechanisms for more carefully characterizing the conditions under which object methods may be invoked. Java programmers often overuse `synchronized` and deadlock is a common bug in multi-threaded Java programs.

Java's passive object model also limits mechanisms for thread interaction. In particular, threads ex-

change data through objects using either polling or `wait\notify` pairs to coordinate the exchange. In decoupled environments, where asynchronous or event-based communication yield better performance, Java programmers must build their own libraries which implement asynchronous messaging in terms of these primitive thread interaction mechanisms. Active objects, on the other hand, greatly simplify such coordination and are a natural atomic unit for system building, but no such alternative is available in the current version of Java.

Finally, we find Java's position on thread scheduling to be inadequate. While it is reasonable to not *require* applications to use fairly scheduled threads, we believe that system builders should have the *option* of selecting fair scheduling if necessary. The lack of fair threads is a particularly devious source of race conditions which makes debugging multi-threaded applications all the more difficult.

In the remainder of this section, we elaborate on each of these criticisms and describe potential solutions.

2.1 Linguistic Support for Synchronization

Synchronization in Java is necessary to protect state properties associated with objects. For example, the standard class *java.util.Hashtable* defines a synchronized *put* method for adding key-value pairs, and a synchronized *get* method for hashing keys. Both methods are synchronized to avoid corrupting state when methods are simultaneously invoked by separate threads. This mechanism works well for classes like *Hashtable* because methods in these classes have relatively simple behavior and do not participate in complex interactions with other classes.

A side-effect of the convenience and simplicity of `synchronized`, however, is that it tends to be overused by application programmers: when software developers are not certain as to the context in which a method may be called, a *rule of thumb* is to make it `synchronized`. This approach guarantees safety in Java's passive object model, but does not guarantee liveness and is a common source of deadlock. Typically, such deadlocks result because of interactions between classes with synchronized methods. For example, consider the threads `t1` and `t2` in Figure 1. The thread `t1` executes the synchronized method `m` which attempts to invoke the synchronized method `n` in class `B`. Similarly, the thread `t2` executes the synchronized method `n` which attempts to invoke the synchronized method `m` in class `A`. In a trace in which both threads first acquire their local locks, this simple example results in a deadlock.

```
class A implements Runnable{
  B b;
  synchronized void m() {
    ...b.n();...
  }
  public void run() { m(); }
}

class B implements Runnable{
  A a;
  synchronized void n() {
    ...a.m();...
  }
  public void run() { n(); }
}

class Test {
  public static void main(String[] args){
    A a = new A();
    B b = new B();
    a.b = b;
    b.a = a;
    Thread t1 = new Thread(a).start();
    Thread t2 = new Thread(b).start();
  }
}
```

Figure 1: A simple example of thread interactions which may result in deadlock.

We view the `synchronized` keyword as too low-level for effective use by application developers. Specifically, requiring developers to implement sophisticated synchronization constraints in terms of low-level primitives is error prone and difficult to debug. Synchronizers [4, 3] are linguistic abstractions which describe synchronization constraints over collections of actors (see Figure 2). In particular, synchronizers allow the specification of *message patterns* which are associated with rules that enable or disable methods on actors. Synchronizers may also have state and predicates may be defined which use state in order to enable or disable methods.

Note that synchronizers are much more abstract than the low-level synchronization support provided in Java. Synchronizers may be placed on individual actors as well as overlapping collections of actors. Moreover, separating synchronization into a distinct linguistic abstraction, rather than embedding it in class

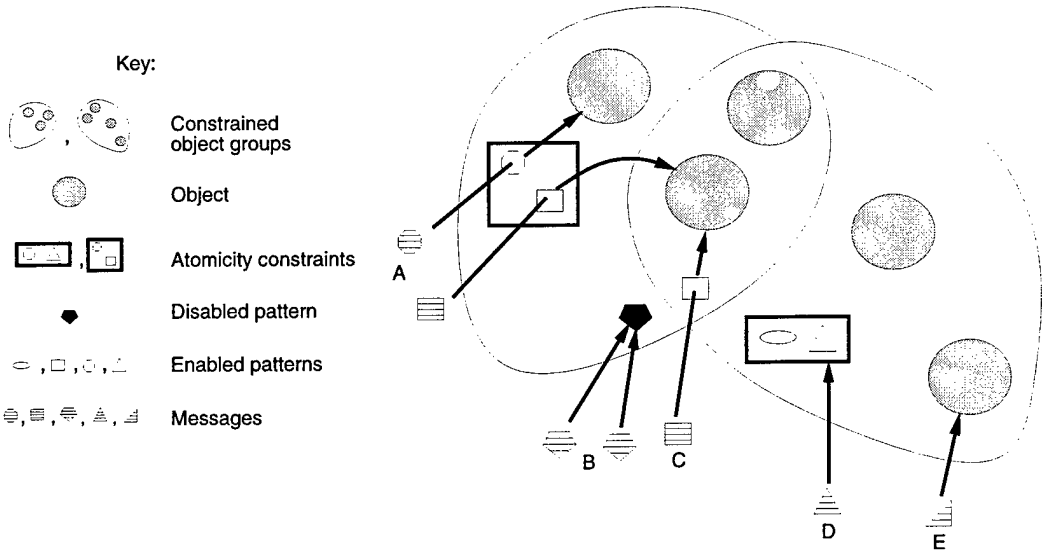


Figure 2: Synchronization constraints over a collection of actors.

definitions, allows constraints to be reused over different classes. As a simple example of how synchronizers may be specified linguistically, consider two resource managers, `adm1` and `adm2`, which distribute resources to clients. We wish to place a bound on the total number of resources allocated collectively by both managers. This can be achieved by defining the synchronizer given in Figure 3. The field `max` determines the total number of resources allocated by both managers.

We believe that heterogeneous environments, in which a wide variety of synchronization properties will be required, argue for an approach similar to synchronizers rather than the current Java solution of embedding low-level synchronization within classes.

2.2 Flexible Interactions

Distributed, heterogeneous systems require the ability to asynchronously participate in interactions in order to take advantage of available local concurrency. Because Java uses a passive object model, threads on a single virtual machine may interact either by polling on shared objects, or using `wait\notify`. Although these heavily synchronized methods of interaction are the most common in Java applications, asynchronous interactions may be implemented by spawning extra threads to handle interactions (see Figure 4).

As in the case of synchronization discussed in the last section, requiring the application developer to explicitly code such interaction mechanisms is prone to error. Asynchronous interactions are an important *basic* service that we believe should be standard in a heterogeneous programming environment. Thus, we argue for higher-level linguistic support in Java which provides such interaction mechanisms.

We believe that asynchronous interactions are best supported by an active object model such as that provided by actors. In such a model, method invocations are buffered in a *mailbox* and handled in a serialized fashion by a dedicated *master* thread. Active objects are thus a natural unit of concurrency and synchronization. Moreover, such objects need not be strictly serialized: intra-object concurrency may be added by allowing the master thread to spawn new threads which access specific internal methods. This form of intra-object concurrency differs from that in Java in that the master thread controls the conditions under which multiple methods may be active, rather than allowing arbitrary threads to execute in an object.

2.3 Resource Control

A final concern with using Java to develop heterogeneous systems is the lack of effective Java support for controlling system resources. A particular example is the ability of application programmers to control thread scheduling. While the Java language specification [5] encourages language implementors to write fair schedulers, this rule is not enforced. Hence, different environments may provide different schedulers emphasizing particular applications. A common solution is

```
AllocationPolicy(adm1,adm2,max)
    { init prev := 0

      prev >= max disables (adm1.request or adm2.request),
      (adm1.request or adm2.request) updates prev := prev + 1,
      (adm1.release or adm2.release) updates prev := prev - 1
    }
```

Figure 3: A Synchronizer that enforces collective bound on allocated resources.

```
class C {
  void m(){...}
  void am(){
    Runnable r = new Runnable {
      public void run(){
        m();
      }
    }
    new Thread(r).start();
    //  Code to continue executing
    //  after asynchronous method call
  }
}
```

Figure 4: A Java class which uses separate threads to handle interactions and execute local behavior.

to favor threads which are responsible for maintaining graphical user interfaces. However, while such an approach may be feasible for certain applications, other applications may fail as a result. Unfortunately, Java provides no mechanism for selecting features of the scheduler, leaving the application developer with the task of implementing custom scheduling if needed.

One possible solution is to include standardized thread scheduling libraries which may be invoked by applications desiring more control over scheduling. However, a user-level approach may not apply to certain critical threads in a system. For example, Java's RMI [12] package handles remote invocations using a separate, non-user controlled thread which invokes methods on user-defined objects. Because this thread is not under user control (and hence not subject to a user-level scheduling solution), unexpected pre-emption and deadlock may result[1]. As a specific solution, we favor the inclusion of lower-level policy selection which allows application developers to specify their scheduling needs. At a more general level, application developers should be able to specify abstract policies which govern more general classes of resources (see Section 4).

[1]It is possible to "hack" around this problem by modifying the RMI-created thread's properties once within a user-defined method. However, this may have unexpected side-effects since the thread was created for use by RMI.

3 Object Request Brokers

As of JDK 1.2, Java will incorporate an interface to the Common Object Request Broker Architecture (CORBA). The inclusion of ORB-based technology in Java indicates the widespread acceptance of Java as a platform for distributed computing, as well as the acceptance of CORBA as an appropriate technology for building component-based systems. In considering this recent combination of technologies, it is interesting to compare the Java Transaction Services (JTS) to the Object Transaction Services (OTS) used in CORBA. These two services are used to manage issues which arise in handling interactions between distributed objects. For example, marshaling data types, handling remote references, etc.

The design decisions evident in the JTS and OTS are a symptom of the relevant strengths and weaknesses of Java and CORBA, and attempt to combine the best of both worlds in a single package. Both Java and CORBA have their strong points and both have been used to develop successful applications. As discussed in the introduction, Java is a rich language with many features designed to simplify programming in heterogeneous environments. However, Java does not provide extensive support for matching clients to servers based on a service description. CORBA, on the other hand, facilitates service location and interaction in a heterogeneous environment. In particular, CORBA allows service description in terms of an Interface Definition Language (IDL), and provides mechanisms for locating services based on IDL descriptions. IDL specifications are an abstract specification of service which are independent of low-level system features such as resource requirements, procedural behavior, control-flow and so-on. Unfortunately, CORBA limits the types of data that can be commu-

nicated in interactions, and prohibits the passing of object references which is required to take advantage Java's more powerful features. The combination of Java and CORBA is intended to alleviate many (but not all) of these problems, while carrying over as much functionality as possible from existing remote interaction mechanisms in Java and CORBA.

In the remainder of this section we discuss some of the motivation behind combining ORB-based technology with Java. While we favor this marriage of technologies, we argue that such a combination still lacks many important features necessary for effective heterogeneous programming. Specifically, CORBA and its relatives still provide a closed model for interactions, and force application developers to embed interaction protocols within client and server code. Encryption protocols, for example, can not be defined as a property of the connection. Instead, both the client and server must embed appropriate endpoints for the protocol within the existing code for handling interactions. We propose an alternative approach in which these types of protocols may be factored out of application code and specified independently on a per-interaction basis.

3.1 Why Add ORB Technology?

Providing services among a collection of objects accessible via a shared network requires a common interaction layer which links clients, which request services, to servers, which implement those services. CORBA and related ORBs enable the construction and integration of distributed applications by providing such a layer. In particular, CORBA allows the dynamic placement and update of objects which implement services in a distributed, heterogeneous network. Moreover, these objects may be accessed using a common data exchange framework with many features critical to the development of heterogeneous systems. These features include:

- Multi-threading
- Debugging and Network Monitoring
- Connection Groups
- Synchronous and Asynchronous calls to servers
- Virtual Callbacks from the server
- Asynchronous operation
- Location Brokering for location transparency
- Naming Service †
- Event Service
- Life Cycle Service †
- Transaction Service †
- Concurrency Control Service
- Relationship Service †
- Query Service †
- Licensing Service
- Security Service †
- Object Trader Service †

Those items marked with a † indicate features that are present in the JTS as well as the OTS. A detailed description of each of these features is not within the scope of this paper. We refer the interested reader to [7] for more details.

In addition to the features described above, ORBs provide several other features which simplify system development. Among these are the ability to quickly design and implement larger object oriented systems, and a communication backplane with consistent semantics regardless of whether a system executes on a heterogeneous network or a single machine. However, as we discussed in the introduction, ORBs make no attempt to transform heterogeneous systems into homogeneous environments. As a result, although ORBs have been used for some time, it is only recently that issues such as load balancing, security, and transactions have received appreciable attention.

3.2 Other ORB-based Systems

CORBA is the most well-known ORB and is based on the Object Management Group's (OMG) Object Model. This model is backed by a large consortium of commercial system developers and hence has a significant role to play in the future of system development. However, although CORBA has achieved widespread success, several other systems have been developed which support a variety of object models (including CORBA).

The *Top-ORB* system from NCR will allow the connection of CORBA objects, Java Beans, DCOM objects and many other type of objects using the *Top End* framework as the underlying infrastructure. Top End is part of the Top End Service Interface Repository (TESIR) model designed by NCR for supporting access to legacy applications, and which defines a general object service mechanism [1]. NCR plans to launch the underlying infrastructure of Top-ORB in 1998.

The *Solaris NEO* system from Sun is similar to CORBA and designed around the same object model. JOE is another Sun product which provides for distributed client-server applications, and complies with the CORBA 2.0 standard. While supporting CORBA standards, both NEO and JOE also allow for the connectivity of Java applets to applications running on distributed servers. In particular, the object request broker used in JOE may be automatically downloaded

into web browsers, and used to connect Java applets to remote NEO objects. Another useful feature provided by JOE is an IDL compiler which generates Java classes from interface definitions of CORBA objects.

Finally, Java's *Remote Method Invocation* (RMI) provides for more primitive client-server functionality. In particular, RMI is not CORBA compliant, but does support interoperability among Java objects in distributed environments. However, RMI does not provide any explicit support for incorporating legacy (*i.e.* non-Java) objects. Such objects may only be included by adding a Java front-end which interacts with RMI.

3.3 Adding ORB Functionality to Java

The current release of Java supports RMI and JavaBeans and hence does not allow for integration with CORBA-like models of object systems. Despite the various other benefits of ORBs, however, ORB vendors including the OMG and Sun have placed technical emphasis on incorporating several object models within a single framework, rather than attempting to increase the functionality of ORB models as a whole. This trend is expected to continue as no single standard (*i.e.* object model) has been adopted for ORB-based systems.

Thus, while the next release of Java will provide greater flexibility in terms of incorporating existing object models, several key problems with ORBs are inherited with the new approach. Specifically, remote procedure call (RPC) remains as the primary mechanism for building distributed interactions. As with the `synchronized` keyword discussed in the previous section, RPC is often abused in the context of distributed interactions and leads to heavily synchronized, and therefore poorly performing applications. We have already argued for asynchronous modes of interaction in the previous section. More importantly, however, ORBs currently do not provide a mechanism for flexible specification of connection properties. Applications requiring specific policies must either use a custom coded ORB implementation, or embed policy code within clients and servers. Both approaches are error-prone and make systems less modular.

Our research in Actors has lead to a novel approach for separating communication policies from application code. Communicators [10] rely on a meta-architecture to abstract over the communication behavior of Actors. In particular, actor interactions are represented abstractly in terms of three operations (see Figure 5):

- A *transmit* operation is invoked when an actor attempts to send a message;
- A *deliver* operation is invoked when the system receives a message on behalf of an actor; and,
- A *dispatch* operation is invoked when an actor is ready to process the next message.

The communication behavior of actors are customized by installing meta-actors which redefine one or more of the basic actor operations. This technique may be used to implement a wide variety of protocols. For example, consider a simple protocol for implementing a FIFO channel between two actors. Figure 6 gives a Communicator specification which defines such a protocol.

Communicators effectively separate protocol code from application code allowing system designers to pick and choose the protocols necessary for interactions, without complicating code development by changing clients and servers. We believe that an ORB-Java combination must include similar abstractions in order to be an effective tool in distributed, heterogeneous environments.

4 Component-Based Systems

In the previous sections we have discussed the near-term limitations of Java as a tool for building heterogeneous systems. In this section, we present a future vision of software for heterogeneous systems and the features we expect to be incorporated into Java to make it a viable development environment.

The next logical step for component-based heterogeneous system development is higher-levels of granularity in which distributed *collections* of objects are managed as individual components and services. Currently, this is an active area of research in the software architecture community in which such systems are viewed as consisting of a collection of components, which encapsulate computation, and a collection of connectors, which describe how components are integrated into the architecture [9]. This separation of design concerns favors a compositional approach to system design; a methodology which is particularly important when specifying architectures for heterogeneous distributed systems. Heterogeneity, failure, and the potential for unpredictable interactions yield evolving systems which require complex management policies. Allowing architectural specifications in which these policies are separated into abstract connectors has clear advantages for system design, verification and reuse.

Note that policies for managing such systems (*e.g.* reliability protocols, load balance and placement, security constraints, coordination, etc.) not only assert

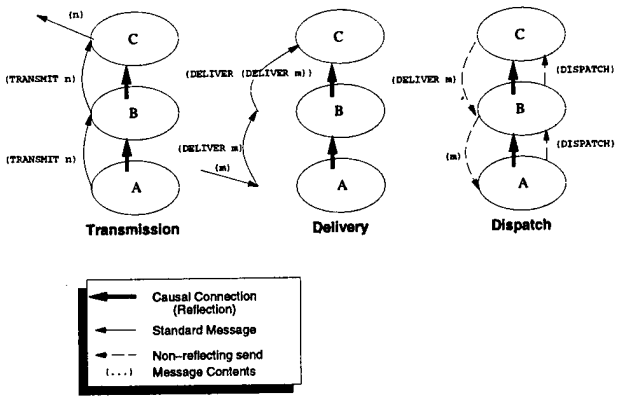


Figure 5: Customizing the communication operations of an actor. Actors B and C are meta-level customizations of actor A. Each operation of A results in an operation on B and/or C.

properties on the connections between component interfaces, but must also enforce constraints on how resources are allocated to components. For example, a reliable server may be developed by adding a backup to an existing server and installing an instance of the primary backup protocol. In addition to recording interactions at the backup, the primary backup protocol must also ensure that the backup and server use separate, failure-independent resources (*e.g.* they must execute on separate processors). The resulting collection of policies is quite different from those required to manage interactions in, for example, ORB-based models, and therefore requires new abstractions with the goal of fitting components to architectural *contexts*, rather than defining interconnections between component interfaces. Specifically, component interfaces abstract over functionality but not resource management. In the remainder of this section, we elaborate further on this point, and describe recent research using the Actor model which proposes a solution to these problems.

4.1 Extending Component Interfaces and Architectural Policies

Current notions of component interfaces are based on a functional representation of the services provided by a component. This abstraction is a natural extension of the object model. However, when placing an object in a heterogeneous architecture, this model fails to describe many important features such as:

- **Locality properties:** The distribution and communication behavior of internal computational elements.
- **Resource usage patterns:** Distinctions such as computation bound versus I/O bound elements, degree of concurrency, hardware dependencies, and the resources corresponding to critical and transient state.
- **Inter-level dependencies:** The relationships between management policies at various levels of granularity.

In general, components should provide a comprehensive model of *architectural context*: the relationships between component behavior and architectural features such as those described above. A natural solution would be to extend current interfaces with additional functional entry points for selecting, for instance, placement policies, reliability features (*e.g.* fault-tolerance protocols), and so on. However, such an approach complicates component code by embedding orthogonal, context-specific concerns. The more preferable approach would be to design generalized components which may be customized to particular architectural contexts. Connectors would encapsulate these customizations, preserving compositional system development. Note that such a solution solves both sides of the heterogeneity problem: general components may be adapted to new environments by composing them with appropriate policies, while hardware-sensitive components may be used in a general context by adding policies which guarantee appropriate resource allocation to this class of components.

A key challenge for specifying more general, resource-based policies is the problem of composing policies while respecting object-integrity. The connection-oriented customizations we described in

```
protocol FIFO_channel {

  Installation asymmetric;
  Isolated-Interaction;

  role local-client { }

  role client {
   int tag;

   method init() {
    tag = 0;
   }

   method out(msg m) {
    server.tagged_in(tag, m);
    tag = tag + 1;
  } }

  role server {
   MsgBag delays;
   int intag;

   method init() {
    intag = 0;
   }

   method tagged_in(int t,msg m) {
    msg next;

    if (t == intag) {
     next = m;
     while (next) {
       deliver next;
      intag = intag + 1;
      next = delays.get(intag);
     }
    } else delays.put(t,m);
} }
```

Figure 6: The Communicator specification for a FIFO channel between actors.

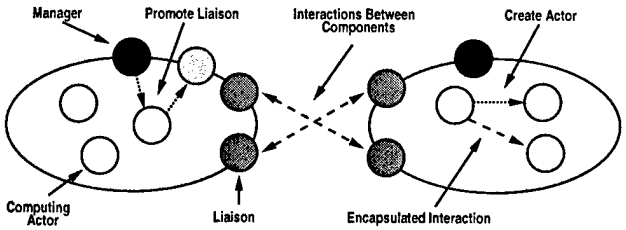


Figure 7: Components are an encapsulated collection of actors. *Liaisons* are a subset of the collection which may participate in external interactions. The *manager* negotiates new connections and promotes actors to liaisons.

Section 3 avoid this problem because they operate strictly on component interfaces. However, specifying policies which control the allocation of resources may require access to component internals. Thus, abstractions which support these policies must be carefully designed to avoid exposing object features which are not normally exported through an interface. We describe our model for such policy composition in the next section.

4.2 Specifying Policies for Connection and Context

In order to reason about architectural context, we require a model of component computation which represents component behavior in terms of interactions with a set of default system services. Relative to computational behavior, the semantics of these services will remain the same regardless of architectural context. However, the semantics of the *implementation* of these services will vary as components are placed in different architectures. This distinction allows compositional development, in which generalized components are fitted to particular architectures, not by changing their computational behavior (which would break encapsulation), but by customizing the interactions between components and the particular implementation of underlying services.

We build on the actor model extensions described in previous sections by modeling components as encapsulated collections of actors in which a distinguished subset, called *liaisons*, are used for interactions with other components (see Figure 7). Interactions between liaisons in different components define component connection properties. In particular, by customizing these interactions, specific protocols may be enforced. Moreover, the architectural context of a component is represented by the service invocation behavior of internal (*i.e.* non-liaison) actors. Thus, the

collective behavior of a component relative to architectural features is captured by the interactions through its liaisons and the resource access patterns of its internal actors. Both behaviors are represented uniformly in terms of invocations of the basic actor primitives, providing a clean representation for architectural customization.

Components are customized by designing policies which define how components access a collection of basic system services (see Figure 8). Liaisons are the only externally visible elements of a component. Thus, connectors which specify protocols between components are naturally represented in terms of customizations applied to individual liaisons. However, connectors which specify resource management policies are more challenging because they customize internal component elements. In particular, we would like to specify arbitrary customizations of internal actors while respecting the encapsulation properties of a component. To this end, policies are constructed from two types of meta-level behavior:

- **Roles:** A role is a specific customization applied to one or more liaisons. Roles are used to implement protocols on connections between components. For example, an encryption protocol may be implemented by customizing the "send" behavior of one liaison (*e.g.* to encrypt outgoing messages) and the "receive" behavior of another (*e.g.* to decrypt incoming messages). Roles are installed explicitly on a set of liaisons.

- **Context:** A context is a single meta-level behavior which customizes *all* actors within a component and is automatically installed on any dynamically created actors. Contexts are used to manage the allocation of resources. For example, a local load balancing strategy may be implemented by customizing the "create" behavior of all actors within a component.

Because roles are installed on liaisons, there is no danger of compromising object integrity as liaisons are already exported by components. Contexts, on the other hand, must be installed on internal component members. However, the structure of the meta-level architecture and the encapsulation properties of components prohibit contexts from destroying internal component elements or exporting non-liaison addresses. Specifically, a meta-level customization may only modify actor interactions with system services, and may not change the internal behavior of an actor. Similarly, regardless of meta-level customizations, managers control component namespaces and determine which actors may participate in external interactions.

A remaining open issue is the question of whether or not policies are composable (both with components or with other policies). In particular, as component compositions encompass larger systems, there is a greater potential for detrimental interactions between existing policies on sub-components. We are currently in the process of extending our abstractions to model and reason about such interference.

5 Conclusion

We have discussed the Java approach to solving the heterogeneity problem and identified several areas for improvement in the current release of Java. In particular, we claim that relative to the needs of heterogeneous computing, current synchronization mechanisms in Java are too low-level and hence prone to misuse. Similarly, we argue that Java does not provide enough control over resource usage, particularly threads, and that existing interaction mechanisms between Java tasks (*i.e.* threads) are too heavily synchronized and lack an alternative communication medium such as asynchronous messaging. We presented several examples from our own work on Actors which demonstrate the utility of more powerful synchronization constructs.

We have considered the recent marriage between Java-based computing and existing CORBA-like systems in the context of heterogeneous computing. While incorporating ORB-based technology into Java is a significant step, we argue that ORBs are still too closed with respect to interaction policies. We presented several examples of policies which may be factored out of object code and applied to the endpoints which implement the connection itself. Such an approach simplifies debugging and makes components more reusable. Moreover, system designers may select only those policies appropriate to their environment, rather than having to pay the price of layering policies atop an existing interaction mechanism.

Finally, we discussed the future of Java in the realm of component-based software development and described our preliminary work on policies for resource management in a distributed, heterogeneous setting. We model components as hierarchical collections of actors with interfaces defined as dynamic sets of actors called liaisons. Components are customized according to the needs of a particular environment by accessing an open implementation of the interface between actors and their underlying system services. We factor customizations into two categories: *roles* are ex-

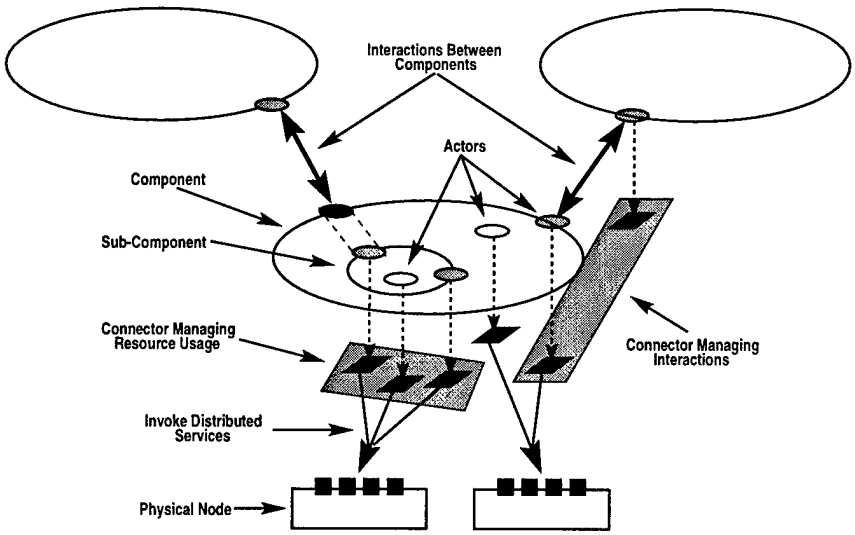


Figure 8: Components are customized by policies which redefine interactions between liaisons, and the invocation of basic system services.

plicit customizations of liaisons, while *contexts* are implicit customizations of all actors within a component. Roles allow the enforcement of interaction policies over connections between components. Contexts support component-wide resource management and coordination. Composition at the meta-level allows multiple customizations to be applied to a single component.

Despite our reservations, we believe that Java is an important step towards developing appropriate tools for building heterogeneous systems. In particular, we have used Java as the development environment for a prototype actor system which incorporates many of the abstractions described above [8].

Acknowledgments

We thank past and present members of the Open Systems Laboratory who aided in this research. The research described has been made possible in part by support from the National Science Foundation (NSF CCR-9619522) and the Air Force Office of Science Research (AF BASAR 2689 ANTIC).

References

[1] YOU'RE THE TOP: A research note from the standish group. Available at http://www.ncr.com/product/integrated /analyst_reports/standish-your/index.htm.

[2] G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT Press, 1986.

[3] S. Frølund. *Coordinating Distributed Objects: An Actor-Based Approach to Synchronization.* MIT Press, 1996.

[4] S. Frølund and G. Agha. *Object-Based Models and Languages for Concurrent Systems*, chapter Abstracting Interactions Based on Message Sets. Lecture Notes in Computer Science. Springer-Verlag, 1995.

[5] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification.* Addison Wesley, 1996.

[6] J. Gosling and H. McGilton. The java language environment: A white paper. Technical report, Sun Microsystems Inc., May 1996. Available at `http://www.javasoft.com/docs /white/index.html`.

[7] Object Management Group. CORBA services: Common object services specification version 2. Technical report, Object Management Group, June 1997. Available at http://www.omg.org/corba.

[8] Open Systems Lab. The actor foundry: A java-based actor programming environment. Available for download at http://www-osl.cs.uiuc.edu/~astley/foundry.html.

[9] M. Shaw, R. DeLine, D. V. Klein, T. L. Ross, D. M. Young, and G. Zelesnik. Abstractions for software architecture and tools to support

them. *IEEE Transactions on Software Engineering*, April 1995.

[10] D. C. Sturman. *Modular Specification of Interaction Policies in Distributed Computing.* PhD thesis, University of Illinois at Urbana-Champaign, May 1996.

[11] The Java Team. JDK 1.2 Beta specification. Available at http://developer.javasoft.com/developer/earlyAccess/jdk12.

[12] The Java Team. Rmi specification. Available at ftp://ftp.javasoft.com/docs/jdk1.1/rmi-spec.ps.

Biography

Gul Agha is director of the Open Systems Laboratory at the University of Illinois at Urbana-Champaign and an associate professor in the Department of Computer Science. He serves as editor-in-chief of IEEE Concurrency, associate editor of ACM Computing Surveys, and associate editor of Theory and Practice of Object Systems. His research interests include models, languages and tools for parallel computing and open distributed systems. Agha has received the Incentives for Excellence Award from Digital Equipment Corporation in 1989, and he was named a Naval Young Investigator by the US Office of Naval Research in 1990. He received an MS and PhD in computer and communication science, and an MA in psychology, all from the University of Michigan, Ann Arbor, and a BS in an interdisciplinary program from the California Institute of Technology.

Mark Astley is a doctoral candidate and research assistant in the Open Systems Laboratory at the University of Illinois at Urbana-Champaign. His research interests include visualization, and architecture description languages and environments for open distributed systems. He received an MS in 1996 in computer science from the University of Illinois at Urbana-Champaign, and a BS in computer science and BS in mathematics, magna cum laude with honors, in 1993 from the University of Alaska - Fairbanks.

Jamil A. Sheikh is a visiting scholar in the Open Systems Laboratory at the University of Illinois at Urbana-Champaign and doctoral candidate at Quaid-e-Azam University - Islamabad. Previously he served as a Computer Systems Engineer at the Department of Nuclear Power. His research interests include real-time distributed computing, concurrent object-oriented programming and high-speed computer networks. He received an MS in 1994 in Nuclear, Computer & Control Engineering from Quaid-e-Azam University, and a BS in Computer Systems Engineering in 1992 from NED University - Karachi.

Carlos Varela is a doctoral candidate and research assistant in the Open Systems Laboratory at the University of Illinois at Urbana-Champaign. His research interests include web-based applications and concurrent programming in Java. He received an MS in 1997 in computer science, and a BS in 1992 in computer science with honors, both from the University of Illinois at Urbana-Champaign.

Fault-Tolerance: Java's Missing Buzzword

Lorenzo Alvisi
Department of Computer Sciences
The University of Texas at Austin
Austin TX 78712

Abstract

Java has been described as a simple, object-oriented, distributed, interpreted, robust, secure, architectural neutral, portable, high-performance, multithreaded and dynamic language, prompting some to describe it as the first buzzword-compliant programming language. We submit that to deserve full certification—and in the process establish itself as the natural choice for developing large-scale distributed applications—Java misses a crucial buzzword: fault-tolerant. We outline some promising research directions for building reliable Java-based applications.

1 Introduction

Java may well be the most exciting technology of our time [2], but the excitement never appeared to leave its proponents speechless. Simple, object-oriented, distributed, interpreted, robust, secure, architectural neutral, portable, high-performance, multithreaded and dynamic [5]—these are some of the buzzwords that have been used to characterize Java since its first introduction. The vision of enterprise computing—a seamless integration of data and communication across thousands of machines to provide better services to citizens and greater opportunities to businesses—seemed to be at hand.

Remarkably, Java has substantially fulfilled many of the promises behind these buzzwords. Today, Java stands unchallenged in its ability to support true platform-independence and in its integration of security mechanisms. In addition, Java provides adequate support for distribution through Remote Method Invocation and run-time loading of classes. Performance is also becoming acceptable, thanks to the development of ever more sophisticated Just-in-Time compilers.

And yet, we submit that Java will fall short of what is required to realize the vision of enterprise computing until it explicitly addresses a conspicuous buzzword that it has so far overlooked: fault-tolerance.

2 Fault-Tolerance: the Ugly Duckling

Building distributed applications forces one to consider the possibility of partial failures. In fact, the kind of wide-area network applications that are likely to be programmed in Java are more vulnerable to partial failures than the relatively constrained applications that are common today. Furthermore, even as languages such as Java make it easier to develop distributed applications and to launch them on the Internet, they do not relieve programmers from the challenge of writing *correct* distributed algorithms. As distributed applications become common-place, we envision that fault-tolerance will become a pressing concern for many more application users and developers.

That Java makes no explicit provisions for fault-tolerance may be partly due to historical reasons: Java was originally conceived as a language for developing software for consumer electronics operating in an environment much more reliable than the Internet. Also, it probably does not help that fault-tolerance is hardly a source for sexy demos. Finally, in a marketplace that rewards the products that reach the market first, fault-tolerance is bound to be an afterthought at best.

There is no fundamental reason, however, that prevents Java from addressing fault-tolerance effectively. Indeed, Java's architecture provides an excellent opportunity to address the issue at the core of all fault-tolerance techniques: controlling nondeterminism.

2.1 Fault-Tolerance and Nondeterminism

If processes were deterministic, then tolerating failures would be trivial: a faulty process could be recovered simply by restarting it from its initial state and rolling it forward. In general, however, the state of a process depends on events that are non-deterministic. To recover a failed process, it is necessary to reproduce the non-deterministic choices that the process made before failing. For instance, in an asynchronous distributed system in which processes communicate by exchanging messages, the state of a process depends on the order in which a process delivers messages, which in turn depends on many factors, including pro-

0-8186-8365-1/98 $10.00

cess scheduling, routing, and flow control. Unless the order of message delivery is somehow recorded and reproduced during recovery, it will not in general be possible to restore a failed process to a state that is consistent both with the states of the other processes in the system and with the external environment.

Other examples of non-deterministic events that may affect a process state include preemptive scheduling of threads, readings of the processor clock, and responding to signals.

3 Fault-Tolerance in Java

There are two obvious levels at which nondeterministc events can be captured in Java:

1. At the virtual machine level
2. At the method invocation level

Capturing Nondeterminism in the JVM The idea of using a virtual machine to manage nondeterminism has been first explored by Bressoud and Schneider [1], who implemented a virtual machine for HP's PA-RISC architecture. In their scheme, fault-tolerance is achieved by replicating the computation on two independently failing processes, using a well-known technique called the state-machine approach [3]. To make this technique work, however, the two replicas must be deterministic. To ensure this, Bressoud and Schneider identified the non-deterministic commands processed by their virtual machine and designed protocols that guarantee that any non-deterministic choice is resolved identically at both replicas. For instance, their implementation guarantees that virtual-machine interrupts are delivered at the same point in the execution of both replicas, and that instructions that read the time-of-day clock return the same values for both replicas.

In Java, a virtual machine is already available for free. Typical implementations of the Java Virtual Machine (JVM), however, do not support deterministic replication of non-deterministic commands. Indeed, it is not obvious which non-deterministic commands are executed by the JVM, although it is reasonable to expect that many will occur at the Java Native Interface. One expects that once these commands have been identified, the techniques developed in [1] and [4] could be used to guarantee the reproducibility of nondeterministic choices during recovery.

Capturing Nondeterminism through Method Logging The logging of message ordering information described above can be generalized easily to distributed object computation systems. Instead of recording the messages delivered to a process, *method logging* protocols record the method invocations made upon an object. Not only do most of the message logging mechanisms apply to method logging, but method logging provides opportunities to improve upon these mechanisms.

Two of the main reasons that make capturing nondeterministic events a challenge are that these events are not easy to identify, and that they can be executed frequently, making it expensive to keep track of their effects.

The object-oriented context of method logging should help in capturing and limiting the effects of nondeterministic execution. For example, nondeterministic events could be encapsulated in method invocations that are tagged in their class definitions. The compiler could then use data flow analysis techniques to determine whether the value produced by a nondeterministic method invocation might affect the value of a parameter in a subsequent method invocation. If it does not, then the non-deterministic choice would not need to be recorded.

4 Conclusions

Java is in a uniquely positioned to emerge as the platform that will finally enable distributed computing to become, in the words of Ken Birman, a mass-market commodity. We believe that the key to Java's long-term success will depend on its ability to support the development of truly reliable applications, capable of tolerating both intentional security attacks and the less glamorous, but potentially as disruptive, spontaneous failure of subsets of their components.

References

[1] T. Bressoud and F.B. Schneider. Hypervisor-based fault-tolerance. *ACM Transactions on Computer Systems*, 14(1):41–79, February 1996.

[2] Sun Microsystems Computer Company. Java computing home page. http://www.sun.com/java/, January 1998.

[3] Fred B. Schneider. Implementing fault–tolerant services using the state machine approach: A tutorial. *Computing Surveys*, 22(3):299–319, September 1990.

[4] J.H. Slye and E.N. Elnozahy. Supporting nondeterministic execution in fault-tolerant systems. In *Proceedings of the 26th IEEE International Symposium on Fault-Tolerant Computing*, pages 250–259, June 1996.

[5] Sun Microsystems Computer Company. *Java Language Overview.* White paper available at ftp://ftp.javasoft.com/docs/papers/java-overview.ps.

Heterogeneous Parallel Computing With Java: Jabber Or Justified?

H. G. Dietz

Purdue University, School of Electrical and Computer Engineering
West Lafayette, IN 47907-1285
http://dynamo.ecn.purdue.edu/~hankd/

Is Java a good language for programming heterogeneous parallel computing systems? It is a well-designed modern language that, combined with the Java Virtual Machine (JVM), offers a myriad of modern programming features and excellent portability. However, in speedup-oriented heterogeneous computing, our primary concern is obtaining the best possible execution speed from the heterogeneous system. This paper briefly discusses what heterogeneous parallel computing is really about, lists some of the key features of Java, and finally summarizes how well Java matches the task of programming for heterogeneous parallel computing.

1. What Is Heterogeneous Computing?

Heterogeneous computing refers to the concept of using a collection of machines, in which each machine may have properties somewhat different from the others, to achieve speedup on a computation.

1.1. Architecture

Generally, a heterogeneous collection of machines will be arranged as either a cluster or a group of networked computers.

A **cluster** is a parallel system whose component computers are both physically and logically near each other. For example, a group of workstations and supercomputers within a single facility, all connected by SCI, HiPPI, Myrinet, PAPERS, or other high-performance networks, would be a typical cluster configuration.

The alternative is a more loosely connected group of **networked computers**, in which communication between machines is possible, but perhaps very indirect, with limited bandwidth and high latency. The largest example of this type of heterogeneous system is the Internet, although many groups of computers connected by LANs (local area networks) are also best viewed in this way.

Both arrangements of machines are possible and useful, but the focus is different. Machines in a heterogeneous cluster can truly cooperate, whereas the loose coupling of networked computers generally requires that machines be able to work independently for relatively long periods of time. Note that the time a processor can operate independently is related to how much memory space is available — having more memory frequently allows local buffering of data to be substituted for some communication operations.

Another significant difference is that clusters are generally assembled by design, whereas networked computers are often simply whatever machines happened to be available. Heterogeneous clusters tend to have heterogeneity because it is directly useful; for example, integrating a SIMD supercomputer with a shared-memory MIMD supercomputer so that different portions of a parallel program can each be executed using the parallel execution model that yields the best speedup. In contrast, heterogeneous networked computers often are workstations from several not-quite-compatible vendors, with essentially the same execution model and only minor performance variations.

1.2. Speedup

In a heterogeneous system, speedup can be achieved by two separate mechanisms. **Parallelism across machines** achieves speedup, ideally proportional to the number of machines, by simultaneously executing portions of the computation simultaneously on different machines. However, speedup also can be achieved by increasing the appropriateness/efficiency of the mapping of the computation onto the special abilities of each machine. In most cases, this centers on use of **parallelism within each machine**.

Clearly, the nature of the hardware heterogeneity places emphasis on one or the other of these two mechanisms. Performance of a cluster containing a few parallel supercomputers will critically hinge on the effective use of parallelism within each machine; performance of a network mixing comparable DEC, HP, Sun, SGI, etc., workstations will just as critically depend on parallelism across these machines. Of course, failing to use all appropriate machines or failing to use each machine efficiently lowers performance no matter what structure the heterogeneous system has.

1.3. Portability

Because all we care about is being able to execute the appropriate portions of a program on each machine, and there is nothing (excluding development and maintenance time and cost) to prevent us from writing code specifically tailored to each machine, portability simply is not a requirement. Indeed, if the machines are heterogeneous in the sense that some machines have access to specialized I/O devices that others cannot access, portability is meaningless: running code ported from a computer-controlled milling machine on a workstation is unlikely to get any parts milled. It can be just as difficult to achieve good results when porting a SIMD-optimized algorithm implementation to a MIMD machine.

Accepting that portability is not required, it is a highly desirable goal that all programs be expressed in portable notations. Further, if a program is primarily concerned with "pure" computation rather than I/O, the goal of portability can be achieved. There are two basic approaches.

0-8186-8365-1/98 $10.00

Portability by simulation: typically, this is done by compiling each program to an idealized, simplified, architecture which is in turn simulated by software on the target machine. This approach first became widely accepted in the form of P-Code implementations of Pascal. In fact, the Java Byte Code and Java Virtual Machine have many striking similarities to the UCSD Pascal P-System (which also incorporated graphics support and a protected operating environment).

Portability by transformability: instead of simulating another target, one can use a combination of compiler analysis and transformation technologies to literally re-write and optimize the program for the specific features of the target machine. Transformation is more difficult to implement than simulation, but the benefit is higher performance.

As a general rule, simulation works best when the idealized architecture has data types and other basic properties that are very similar to the actual target hardware characteristics, but "instructions" that are typically much higher-level than individual target machine instructions. The need for basic properties to be similar is obvious; the need for higher-level instructions is due to the fact that overhead to decode and prepare each simulated instruction for execution generally is typically about 20 target machine instructions. Simulating a "matrix multiply" instruction easily hides a 20-instruction overhead, while simulating "32-bit integer add" may result in a 20x slowdown relative to executing the operation transformed directly into native target code.

Many simulators now incorporate incremental compilers that can perform some of these transformations, however, the apparent simplicity of transforming an add instruction into machine code is misleading. The major complication is essentially the use of parallelism.

1.4. Transformability

If the simulated instructions are simple and not parallel, but the target machine is parallel, we are confronted with the age old problem of automatically parallelizing. Perhaps even more complex is the problem of transforming parallel instructions into a different target parallelism. There are at least three key aspects of a parallel execution model that must be matched when generating code: execution mode, communication model, and grain size/data layout.

1.4.1. Execution Mode

SIMD, MIMD, and VLIW/Superscalar execution modes each perform best under different circumstances, but it is very difficult, for example, to transform arbitrary MIMD code into efficient SIMD code.

SIMD (Single Instruction Stream Single Data Stream), which is now *the most common parallel execution mode* (what do you think all those multimedia enhancements are?), is very effective in implementing algorithms that require tight synchronization of similar activities across many processing elements. For example, communication operations do not need buffering, interrupts, etc., because it is trivial for all processing elements to know precisely when data exchanges will occur. However, SIMD serializes operations that are different on each processing element.

MIMD (Multiple Instruction Stream Multiple Data Stream) is the most general parallel execution mode, capable of simultaneously executing arbitrary operations on different processing elements. Because each processing element can have its own independent clock and program counter, processing elements do not have to wait for the slowest to complete each operation before advancing to the next. However, this independence comes at the expense of the ability to efficiently, globally, coordinate actions as we described for SIMD communications.

VLIW (Very Long Instruction Word) and **Superscalar** execution models are logically somewhere between SIMD and MIMD, offering more generality than SIMD while preserving the ability to globally coordinate parallel actions. The problem is that these techniques require structures that do not scale well, so parallelism width is generally limited as compared to SIMD or MIMD models.

1.4.2. Communication Model

There are at least three fundamentally different classes of communication models, and the best choice for each algorithm or target machine varies.

A **shared memory** model communicates using what appear to be simple memory load/store operations, but there are many flavors differing in how much of memory is shared by which processing elements (shared everything vs. shared something), access time as a function of address reference pattern (logical, physical, and cache structures), and even rules for atomicity and coherence (what references are atomic, how are access races resolved). These complications make shared memory models the most difficult to use efficiently and correctly, and also the most difficult to transform into other communication models — even into other shared memory models. Unfortunately, these are also the most efficient models for many machines.

A **message passing** model creates, sends, and receives messages, generally with one sender and one receiver for each message. The key advantage in message passing systems is the ability of a single message to contain a large data payload; this makes it more effective than shared memory models in utilizing reasonable-bandwidth, high-latency, interconnection mechanisms like UDP or TCP protocols over fast Ethernet. There are also various flavors of message passing, differing primarily in the types and lengths of messages allowed and the possible orderings of sends and receives.

An **aggregate function** model, unlike the other two models, allows any number of processing elements to directly participate in each communication operation. The simplest example is a barrier synchronization, in which no processor enters the next phase of a computation until all have signaled completion of the current phase. More complex aggregates include "collective communications" such as permutations, multi-broadcasts, personalized all-to-all, associative reductions, scans (parallel prefix operations), voting operations, etc. Because aggregates are N-ary operations, whereas shared memory and message passing operations tend to be point-to-point, aggregates offer much better performance on hardware that supports them... aggregates also are the key to efficiently

implementing SIMD and VLIW execution modes on what would generally be considered MIMD hardware.

Not only is it difficult to transform between these three different classes of models, but it also can be difficult to convert between different models within the same class. For example, it can be surprisingly complex to convert between two shared memory models that differ only in atomicity and coherence properties.

1.4.3. Grain Size And Data Layout

The amount of computation that each processing element will do between communication operations, and the layout of data structures across processors to make data accesses within a granule local, vary widely depending on the target machine, and transformations are difficult. For example, the HPF (High-Performance Fortran) effort was largely driven by the desire to have programmers specify the data layouts — compiler technology to automatically pick the best layout, or even to efficiently transform between specified parallel layouts, is still under development.

1.4.4. Semantics Enable Transformation

In summary, it is very difficult in the general case (perhaps impossible) to transform code written in one form into efficient code in a different form. Put another way, the above are all really **execution model** characteristics. A **programming model** should be designed to use only constructs that have known efficient implementations for a wide range of execution models.

For example, consider an abstract parallel `if` statement:

```
if (parallel_expr) {
    then_action;
} else {
    else_action;
}
```

SIMD semantics would specify that *then_action* would execution before *else_action* in the case that *parallel_expr* was true for some processing elements and false for others. This implies that side-effects, such as communications in each actions, are strictly ordered with respect to each other, and thus are inherently race-free.

In contrast, MIMD semantics would permit *then_action* and *else_action* to execute simultaneously, thus requiring other mechanisms to enforce ordering of communications within the actions.

The point is simply that, to be efficiently transformable, a parallel language (or programming model) must facilitate transformation into efficient code for any target by using **semantics that are consistent with all possible target models**. Thus, our parallel *if* statement should have semantics that *allow* both actions to occur simultaneously, but do not require this.

The result is that when compiling for a SIMD target, there may be redundant synchronizations in the source program (that were placed there to enforce communication ordering for MIMD execution of the actions), but these are easily mechanically removed by existing compiler technology. Thus, careful selection of transformable parallel semantics is the key to making a parallel programming model directly support heterogeneous supercomputing.

2. Java

Java is a well-designed modern language that, combined with the Java Virtual Machine (JVM, also known as the Java byte code interpreter), offers both flexibility and portability. There is a lot to like about Java, and about its byte code (which, for analysis and transformation purposes, is nearly equivalent to the Java source code).

2.1. Data Types

Java specifies the size of basic data types: `byte` is -128 to 127, `short` is -32768 to 32767, `int` is -2147483648 to 2147483647, `long` is -9223372036854775808 to 9223372036854775807, and `char` is 0 to 65535. This is a great benefit for heterogeneous computing in that it removes precision differences from our concerns in using a variety of machines. Although the lack of unsigned types is somewhat disturbing, they are effectively supported by unsigned operators: >>> is the unsigned shift right operator.

Another benefit in Java's type handling is the use of IEEE floating point features such as NaN (Not-A-Number), Infinity, and +/-0 instead of exception mechanisms. The need to create a valid state when an exception occurs is a subtle, yet serious, constraint on the compiler's ability to transform code for parallel execution; Java's definition removes this problem.

Java's high-level handling of arrays as single objects, and the deliberate omission of C-style pointers, make data alias analysis significantly easier for typical constructs than it is for C. The handling of object allocation/deallocation is also cleaner, although the garbage collection scheme significantly complicates the runtime environment and may seriously degrade performance using threads.

2.2. Object Oriented

One of the most praised features of Java is its support for object-oriented programming. In many ways, this is a useful abstraction, but object-oriented indirect function calls make static analysis for parallelization more difficult and less effective.

Java further complicates matters by allowing functions to be specified in a way that facilitates using independently-developed binary code modules together within a program. This very late binding of function calls to code makes it very difficult for static compiler analysis to perform global optimizations, such as those needed for some types of parallelism transformations (e.g., conversion from MIMD to SIMD). Just-in-time or other incremental compilation technologies can help in this respect, but these are not really good solutions, largely because the analysis would be repeated at runtime for each new run by each machine. It is also important to note that, for example, a SIMD supercomputer, which seriously needs the global analysis and transformation, may be a totally inappropriate machine on which to execute the analysis and transformation compiler code.

2.3. Threads

Java directly supports parallel execution using a built-in version of threads. Although Java threads essentially implement a shared memory model, unlike most thread libraries, Java precisely specifies how threads distinguish between actions taken on a thread's local working memory and effects on main memory. The result is a model somewhat more flexible with respect to ordering of accesses, thus hopefully more efficiently implementable on a variety of other shared memory mechanisms. As in C, `volatile` attribute is provided to enforce the programmed ordering of accesses.

In comparison to thread libraries, Java provides higher-level synchronization primitives that take advantage of the difference between a library call and a language construct. The standard lock management to ensure that only one thread is allowed to be within a particular segment of code at any given time (mutual exclusion) is remarkably clean in Java:

```
synchronized(t) {
    // exclusively executed code
}
```

This construct not only manages the locking at entry and unlocking at a normal exit of the segment, but also correctly handles unlocking for any other type of exit from that segment.

2.4. Java, The Standard Language

One of the best aspects of Java is the speed with which a precise standard definition of the language has emerged. There is currently an international standardization effort in progress.

The reason that the standard has moved so quickly is because Sun is essentially in full control. As a general rule, international standards are not created by a single for-profit company, but by non-profit groups; when the vote was taken in late 1997 on formation of an ISO standard for Java, the United States cast its vote against the standardization effort largely because Sun is placed in a controlling position. In fact, Sun even retains all rights to their trademark on the name Java. The vote passed, although the US was not alone in its concern about Sun's control of the standard.

Despite this tightly-controlled approach to popularizing Java, it is important to acknowledge that Sun has done a very good job thus far. Not only is Java a relatively clean language design, but Sun has employed some of their best people to ensure that the language is a success. Sun is also pushing a 100% Java certification effort to discourage people from using non-standard features.

I like the 100% effort; it is the only way to achieve the portability that Java seeks. However, it is important to note that a program that calls native routines is not 100% Java. In other words, Java does not provide any way to use parallelism other than threads; any other parallelism would violate the 100% certification rules. This is a fundamental problem with respect to using Java for high performance computing.

2.5. Support For Networking

Java supports network communication via sockets, and provides library routines for higher-level protocols such as HTTP and FTP. It is easy to imagine, or to create, pure Java code library functions for higher-level parallel-processing message passing. Unfortunately, the basic socket interface, although portable, is not particularly efficient, especially within a cluster connected by SAN (System Area Network) hardware.

2.6. Graphics Support

Java provides a very strong set of graphics facilities that are remarkably portable, independent of host machine OS, windowing system, etc. Then again, you already knew this from watching a certain little Java Aplet wave at you from within your WWW browser.... Java is an excellent way to graphically present and browse data, even within a heterogeneous parallel computation, provided that the graphics computations are not too intense.

3. Conclusions

Is Java appropriate for speedup-oriented heterogeneous parallel computing? The answer really depends on what type of heterogeneity your target system employs.

The thought of thousands of random workstations, PCs (Personal Computers), and NCs (Network Computers) scattered across the Internet being used as a parallel computer appalls me... but I've actually participated in such efforts. Virtually all of these machines are relatively minor variations on the same uniprocessor 32-bit architecture, heterogeneous primarily in the sense that some level of product differentiation is important in establishing a vendor's marketing strategy. The same Java code easily runs on all these machines, and it is relatively simple for the Java code to send an occasional message from one machine to another.

However, the Java byte code will be interpreted very slowly, and with variable and difficult to predict performance, on most machines. Of course, a factor of 20x slowdown on each machine can be repaired by simply using 20x more machines... if you have enough parallelism and machines available. Still, more parallelism means more communication unless you can buffer nearly all data in local memories, and Java communication is also slow. In my opinion, the class of applications for which one should be able to achieve both reasonable efficiency per unit of compute hardware used and good speedup is vanishingly small.

The more important definition of a heterogeneous system is the one in which the component machines truly do offer a variety of special characteristics, especially clusters in which each machine may offer a different type of internal parallelism. In these cases, Java does not solve any of the key problems and non-Java code would need to be invoked to efficiently use the hardware parallelism. Use of Java threads can be seen as creating serious problems by forcing a non-transformable shared-memory MIMD execution model.

In summary, Java has many features that should be a part of a programming model for heterogeneous parallel computing. Unfortunately, 100% Java is not an appropriate model.

On the Interaction between Mobile Processes and Objects

Suresh Jagannathan Richard Kelsey
NEC Research Institute
4 Independence Way
Princeton, NJ 08540
suresh|kelsey@research.nj.nec.com

Abstract

Java's remote method invocation mechanism provides a number of features that extend the functionality of traditional client/server-based distributed systems. However, there are a number of characteristics of the language that influence its utility as a vehicle in which to express lightweight mobile processes. Among these are its highly imperative sequential core, the close coupling of control and state as a consequence of its object model, and the fact remote method calls are not properly tail-recursive. These features impact the likelihood that Java can easily support process and object mobility for programs which exhibit complex communication and distribution patterns.

1 Introduction

Distributed systems have historically been concerned with issues concerning the partitioning and transmission of data among a collection of machines [15]. Typically, these systems allow code to be distributed and accessed in one of two ways. In some systems, each node holds code controlling the resources found on that node. In others, the same code image is found on all nodes. In either case, some form of message-passing [19] is used to invoke operations on remote sites. By and large, mobility has not been an issue of significant importance. In client-server based systems, process mobility is essentially irrelevant: tasks are heavyweight and control resources resident on a particular machine. In systems where all machines share the same code image, process mobility may be used to help performance by improving locality and load-balancing. However, tasks typically execute heavyweight procedures often closed over a large amount of local state, making task migration expensive. Indeed, devising an efficient task migration policy that has a simple well-understood semantics is still a subject of active research.

The past few years has seen an increasingly apparent shift to a new computational paradigm. Instead of regarding the locus of an executing program as a single address-space physically resident on a single processor, or as a collection of independent programs distributed among a set of processors, the advent of languages like Java [8] has offered a compelling alternative. By allowing concurrent threads of control to execute on top of a portable, distributed virtual machine, Java presents a view of computation in which a single program can be seamlessly distributed among a collection of heterogeneous processors. Unlike distributed systems that require the same code to be resident on all machines prior to execution, Java allows new code to be transmited and linked to an executing process. This feature allows Java to upload functionality dynamically in ways not possible in a traditional distributed system.

Currently, the Java core only supports migration of whole programs; threads of control are not transmitted among machines. However, extensions like Java/RMI [20] that enable client-server (RPC-style) semantics do allow data as well as code to be communicated among machines in a Java ensemble. Such extensions permit Java programmers to view a computation not merely as a single monolithic unit moving from machine to machine in the form of applets, but as a distributed entity, partitioned among a collection of machines. By using a architecture independent virtual machine, information from one active portion of a computation can be sent to another without deep knowledge of the underlying network infrastructure connecting these pieces together.

In this paper, we explore how distributed extensions to Java such as RMI handle issues pertinent to lightweight task migration. We are particularly interested in how Java's object model, which pervades all aspects of its design and implementation, affects lightweight task and object migration. Some of the questions we examine include:

1. What impact does the close coupling of data and

0-8186-8365-1/98 $10.00

code in Java have on the implementation of mobile processes?

2. How does Java's imperative sequential core influence influence the design of its distributed extensions?

3. How does a synchronous client-server communication model affect the construction of mobile objects?

4. How do we express where a task should execute, and how an object should migrate as part of an object's behavior? In other words, can we separate issues of concurrency and distribution from issues related to an object's sequential behavior?

Using Java/RMI as a base example, we conclude that features endemic to the Java design make it difficult to express lightweight process mobility. On the other hand, Java's object system significantly extends the functionality found in other distributed languages, most notably in its support for distributed dynamic linking of new code objects.

Our foucs will be on the interaction between mobile processes and objects. After providing some motivation, we present a simple view of a distributed system in the context of an Algol-like (imperative) setting. We then discuss interface description languages like CORBA [14], and their approach to handling distribution. Finally, we describe distributed computation from the Java perspective, and contrast these approaches. We conclude that while Java with RMI support offers significant advantages over its more traditional counterparts to programming heterogeneous networks, it lacks certain features that would enhance its expressivity.

2 Motivation

Like many distributed languages [3, 5], Java's sequential core is object-based. A Java class defines a datatype, and an object is an instance of that type. Operationally, an object defines a collection of data along with operations on that data. A distributed Java program can be now viewed as a collection of objects resident on different address spaces or machines, communicating with each other through visible methods declared by these objects. Under a static view of a distributed computation, processes executing on different machines provide a global service or protect some shared resource. Once allocated to a given machine, however, processes remain stationary. Objects are a natural abstraction for expressing the behavior of distributed processes since they encapsulate shared data through instance variables, and define the operations permitted on this data by other processes through methods and interfaces. For example, a client of a resource need only have access to the operations provided by that resource, and not the actual data manipulated by the resource in order to use the resource effectively. Indeed, for many distributed applications, a static partitioning of shared resources and services via distributed objects is ideal.

Nonetheless, not all applications exhibit such static behavior. As the size of distributed systems grows, or as the complexity of an application increases, defining an efficient static partition of a collection of logically distributed processes becomes problematic. In this case, finer mobility of code and data becomes important [10]. For example, a process executing on a machine in an overloaded network ensemble may need to migrate dynamically to a less-loaded one. In highly heterogeneous systems like the Internet, this functionality becomes even more pronounced: an application may be distributed over many different kinds of machines with widely different capabilities, and may need to be frequently reconfigured to take advantage of changing work-loads or conditions among the nodes on which it executes. To achieve mobility of this kind, code and data must be more loosely coupled: migrating a process from one machine to another should not necessarily entail copying all of the data it may potentially reference as well. Similarly, moving data closer to where a computation requires it, should not entail copying all other processes that also happen to share references to that data. A mobile process should be able to move among a collection of nodes without communicating its intention to the node from whence it came.

3 Mobility in an Imperative Context

We first consider distributed execution in the context of an imperative Algol-like language. These languages have two features that inhibit distribution in general and mobility in particular. The first is that programs generally make progress via side effects, either by updating variables or modifying data structures. The second is that they are first order. Procedures can neither be returned from other procedures, or passed as arguments except in the most trivial cases. The only data available to a procedure are arguments and global variables. It is difficult to only use procedures to simulate the behavior of objects in object-oriented languages, or closures in functional languages.

The result is that computation involves frequent modifications to shared global data, which is exactly

what a distributed program needs to avoid. There are two basic approaches to dealing with the problem: distributed shared memory (DSM) [2, 13] and remote procedure call (RPC) [4, 18]. With DSM, the distributed nature of the computation is largely invisible to the programmer, at a high cost in implementation complexity and communication overhead. All data is conceptually associated with a global address. Thus, the machine where a thread executes no longer influences the behavior of the program: dereferencing a global address may involve a remote communication to the node "owning" the contents of that address. While DSM provides a mechanism to implement parallel dialects of imperative languages in a distributed environment, programmers have little control in specifying how coherence and consistency are realized. In particular, issues of process mobility become largely irrelevant since the distribution of data and tasks is implicitly handled by the implementation, and not explicitly managed by the program. The high communication costs associated with DSM make it unattractive in a truly distributed environment. The alternative is to move the burden of handling communication from the implementation to the programmer.

Remote procedure call provides a way of breaking a program into discrete parts each of which runs in its own address space. Unlike DSM, communication is explicit in the program, so programmers have complete control over costs. The difficulty is that the semantics of RPC is substantially different from that of an ordinary procedure call. When procedure P makes an RPC call to Q, the arguments to Q are marshalled and shipped to the machine where the computation should be performed. Stub generators on procedures linked to the application program are responsible for handling representation conversion and messaging. Arguments passed to a remote procedure are passed by copying. Thus, side effects to shared structures can no longer be used for communication between caller and callee. As a result, imperative programs must be substantially modified to run in a distributed environment using RPC.

Process mobility is especially difficult [16, 6, 17]. The imperative nature of these languages means that a large percentage of data found in programs must be global. Communication among processes is enabled via side-effect, and not via allocation and copy. Thus, the advantages of having mobile processes is greatly mitigated. Conceptually, processes are highly mobile in these languages because they carry no state, but because they must frequently reference global (shared) data, process migration becomes useful only if the data they access moves along with the process requiring them. Given that global data is likely to be shared among several processes, the implicit coupling of data and code in imperative languages greatly weakens the utility of process mobility in these languages.

4 Distributed Glue Languages

CORBA [14] and ILU [9] are two well-known object-oriented glue languages that can be used to connect sequential components into a distributed program. The sequential components can be written in a variety of languages and components written in different languages can be freely intermixed within a single distributed program. These glue languages are based on object-oriented interface description languages. The programmer writes a description of each component in the interface language which is then compiled into stub programs. One stub program, in the language in which the component is written, is used by the component to communicate with clients. The rest are used by clients to communicate with the server and can be generated in any of the languages the glue language supports.

Unlike imperative languages, the use of objects alleviates the problems caused by limiting procedures to being first-order. Because each instance of an object has its own local state, the number of side effects on the program's global state is reduced. Unfortunately, commication between components is still done using RPC. The values that may be sent between components are immutable ones: numbers, characters, sequences of values, and so forth. The only references that can be sent over the wire are references to the glue language's global objects. Because components may be written in different languages, each component has not only its own address space but, potentially, its own data representations. A data format used in one component may unrepresentable in another. For example, ILU can be used to connect a component written in C with one written in Common Lisp; the representation of arrays in C, and the operations on them, are quite different from those in Common Lisp. There is no way to send either code or mutable data between components. An object resides permanently on the machine on which it is created. Thus, these distributed glue languages work for large-grain distributed programs with simple, static interfaces. Unlike imperative languages that provide little support for distributed communication, glue languages allow data (either in the form of base types or objects referenced via a global handle) to be accessed in a distributed setting. However, since a CORBA program may consist of modules written in many different languages, there is no sup-

port for code or process mobility. Processes run in an address space executing a language processor for the language in which they were written.

Programs written in distributed glue languages are likely to perform well if written in a client/server style. The client/server model partitions computation among nodes in a network: all server-related computation is exercised on the node where the particular server object resides, and all client-related computation is exercised on the client. Because the location of clients and servers is highly static, decisions on where computation is executed is static as well. Once a remote method is invoked, control-flow within the method body remains on the node of the corresponding remote object; the caller of the remote method blocks until the remote method returns.

As long as computation is uniformly distributed among clients and servers, having code remain resident on the remote object site is acceptable. However, for mobile computations, a client/server model is clearly inappropriate. For applications in which computation is non-uniform or which is not easily partitioned into client- and server-code, a more flexible distribution model is required.

Consider a thread of control T executing on some machine M. Suppose T makes a remote method call to a remote object O (possibly written in a different language) executing on some other machine M'. The execution of this method can occur in one of two ways. The method can execute on M as part of T's control-flow, or it can execute on M' as part of a newly instantiated thread. The first option, rejected by distributed glue languages, is useful if references to O within M are infrequent. In this case, the overhead of introducing a new thread control on M' may quite likely be greater than the overhead of retrieving necessary state information from O. The second approach transfers control-flow to M'; this is the approach taken in Java/RMI.

5 Threads and Distributed Objects

In contrast to imperative languages, class-based object-oriented languages like Java encapsulate data and code together. A computational unit in Java is an *object*. An object includes a collection of data called instances variables, and a set of operations called methods to operate on this data. Object state is accessed and manipulated from the outside through publically visible methods. Because it provides a natural form of encapsulation, an object-oriented paradigm seems well-suited for a distributed environment. Objects provide regulated access to shared resources and services. In contrast to distributed glue languages, distributed extensions of Java permit objects as well as base types to be communicated. Moreover, certain implementations such as Java/RMI also permit code to be dynamically linked into an address space on a remote site.

Since a primary goal of Java was to support code migration in a distributed environment, the language provides a socket mechanism through which processes on different machines in a distributed network may communicate. Sockets, however, are a flexible *low-level* communication abstraction. Applications using sockets must layer an application-level protocol on top of this network layer, responsible for encoding and decoding messages, performing type-checking and verification, etc. This is generally agreed to be error-prone and cumbersome.

5.1 Remote Method Invocation

As we discussed earlier, RPC provides one way of abstracting low-details necessary to use sockets. RPC is a poor fit, however, to an object system. In Java, for example, communication takes place among objects, not procedures *per se*. Requiring methods in different objects to communicate directly with one another would break object boundaries and thus would violate the basis of Java's object model. Java/RMI is a variant of remote procedure call tailored for the object semantics defined by Java's sequential core. Instead of using procedure call as the basis for separating local and remote computation, Java/RMI uses objects. A remote computation is initiated by invoking a method on a *remote object*. Clients access remote objects through surrogate objects found on their nodes. These objects are generated automatically by the compiler, and compile to code that handles marshalling of arguments, etc. Like any other Java object, remote objects are first-class, and may be passed as arguments to or returned as results from a method call. Remote objects are implicitly associated with global handles or uids, and thus are never copied across nodes. However, any argument which is not a remote object in a remote object method call is copied, in much the same way as in an RPC semantics. This means that remote calls have different semantics from local ones even though they appear identical syntactically. The fact that Java is highly imperative means that distributed programs must be carefully crafted to avoid unexpected behavior due to unwanted copying of shared data.

Nonetheless, Java/RMI does fit into Java's object model in a number of other ways. Communication takes place via proxies to remote objects, and the encapsulation benefits provided by objects is preserved. In addition, Java/RMI supports a number of features

not available in distributed extensions of imperative languages or distributed glue languages. Most important among them is the ability to transfer behavior to and from clients and servers. Consider a remote interface I that defines some abstraction. A server may implement this interface, providing a specific behavior. When a client first requests this object, it gets the code defining the implementation. In other words, as long as clients and servers agree on a policy, the particular mechanism used to implement this policy can be altered dynamically. Clients can send behavior to servers by packaging them as tasks which can then be directly executed on the server. Again, if the method to be executed is not already found on the server, it is fetched from the client. Remote interfaces thus provide a powerful device to dynamically ship executable content *with* state among a distributed collection of machines.

5.2 Tail Recursive Communication and Mobility

Although Java/RMI can be used to express non-client/server style applications, we expect this will not often be the case. By default, a remote method call in Java/RMI is synchronous: the caller waits for the callee to return before proceeding. Stated another way, remote method calls in Java/RMI are never properly tail-recursive. For example, suppose method A on machine M_1 wishes to make a remote method call to method C on M_3 supplying the result of calling remote method B on M_2. Ideally, we would like to have B invoke C directly. However, to do this requires modifying B's implementation. In certain agent-based systems [11], or distributed systems which support first-class continuations [1, 7], B may invoke C directly, wrapping A's continuation around the call. When C finishes, it returns immediately to A, avoiding an unnecessary communication with B. Although a form of asynchronous communication can be expressed in Java/RMI using an agent interface, it is cumbersome. (See Fig. 1.)

5.3 Thread and Object Migration

Despite its added functionality over distributed glue languages, Java's object model (like other object-based distributed systems [10]) encourages a close coupling of code and data. Because all non-local variable references in a method are either global, or refer to instance variables of the object (via self), the environment within which a method executes is explicitly expressed in the definition of the corresponding class. Unlike imperative languages, the location where a thread executes thus becomes very important. Since threads represent control-flow through methods, and methods are the only means of accessing internal object state, distributed object-oriented languages usually dicate that a thread executing a method evaluate on the same node as the method's object. Hence, thread migration is difficult to express: migrating a thread moves it away from the state accessed by the executed method via self. Indeed, Java/RMI provides no programmer-controlled mechanism to express thread migration: once a thread begins execution on a node, it remains resident on that node until the method it is executing completes. Furthermore, since objects contain mutable state, copying data to where the caller resides is likely not to be beneficial since the data must written back to the object when the method completes.

Part of the reason why thread migration appears to be a concept ill-suited in Java is because Java's thread model is so closely tied to its object model. Any object that inherits from the basic thread class, and provides a `run` method can be instantiated as a thread [12]. The code executed by the thread is the code found in the object's run method. Because threads are no different from any other object, thread and object migration are essentially the same. Moving a thread from one node to another is tantamount to moving an entire object, not just control. To achieve the benefits of lightweight thread migration, however, two features are required. First, we should be able to separate code from data, or at least not be required to explicitly package the two together. Second, we should be able to denote a piece of code as a thread without having to first define a class template. An arbitrary Java expression can be viewed as a thread only if it is encapsulated as a method within an object whose class implements or extends the basic `Thread` class. This leads to a significantly greater burden on the programmer to build lightweight threads.

5.4 Specifying a Locus of Control

Another ramification of code and data coupling is that decisions about whether a method is remote or not is hardwired as part of the class specification in which the method is defined. Any object instantiated from a class that implements a `Remote` interface is treated as remote. Thus, all calls to methods found in such objects are executed remotely on the site where the object is located. It is not possible to have some methods execute locally and others execute remotely without having them defined in separate classes. For example, consider a class that contains an array. Methods which operate on all the elements in the array are best implemented via remote method call since it may be potentially expensive to move the

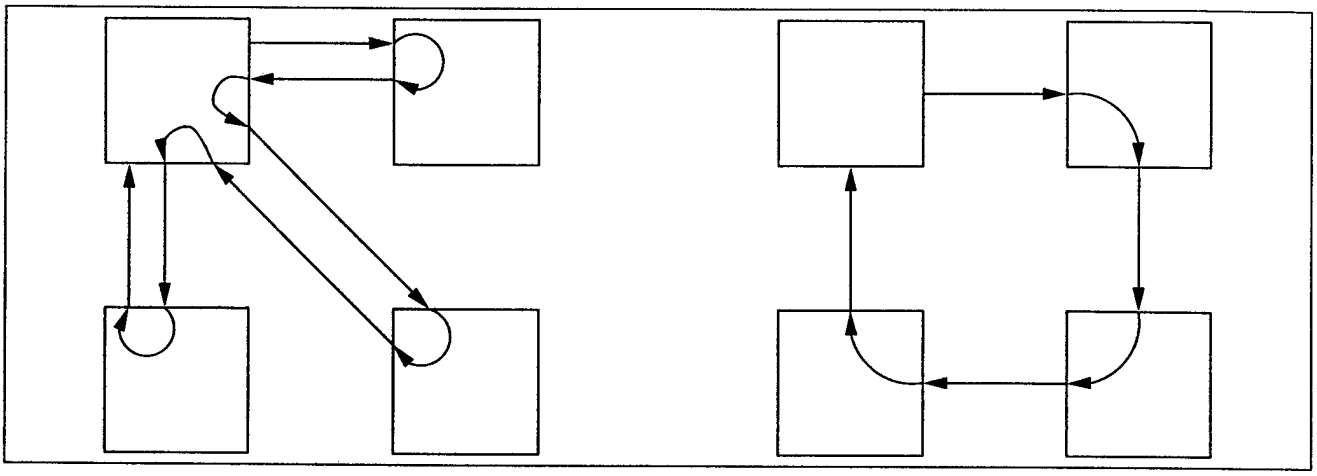


Figure 1: A client-server communication model requires control always to return back to the sender. In agent-based systems, control may move freely among nodes before ultimately returning.

array frequently to the object's clients. On the other hand, it may be more efficient to execute a method which extracts a particular field and operates on that field exclusively locally on the client since communication overhead is likely to be small in this case. In Java/RMI, these two methods cannot co-exist in the same object.

We believe that the choice of where methods execute should be under programmer control. By default, control should remain resident on the node where the thread making the call is currently executing. Field accesses thus involve copying data from the associated object's home. On the other hand, when a method M is annotated as an RPC method, calls to M translate to a shift in control to the location where M's object O is found. If O subsequently migrates while M is still executing, the corresponding thread created to evaluate M migrates as well. Neither of these protocols influence correctness, but their choice impacts efficiency. This finer granularity on control-flow would permit a given object to define methods that support both protocols.

Taken together, these features of Java (and similar class-based languages) make it unwieldy as a language in which to express mobile processes, and dynamic thread and object migration. Because remote calls by default are not tail-recursive. the creator of a thread on a remote node is closely coupled with the thread itself, limiting opportunities for the thread to migrate freely. Requiring that concurrency be expressed through the object system means that expression whose evaluation is to take place in a separate thread must be named as a method found in a `Runnable` object. Since Java provides no mechanism to reify scheduler state, programmers do not have the ability to capture a runnable thread, and move it to another node, or manipulate it in other ways. Any thread or object migration decisions are handled exclusively within the virtual machine. The close explicit coupling between state and code means that moving object state necessarily causes the locus of control for executing methods in that object to shift as well. This is because a class defines an explicit packaging of state used by the methods it defines. Because state referenced by a method is not implicitly constructed by the implementation, it is meaningless to consider mechanisms to distribute control (i.e., threads) as any different from mechanisms to distribute state (i.e., objects).

One way that Java addresses the latter point is through the use of inner classes. An inner class provides many of the features that closures in functional languages provide; in particular, an inner class, allows the same piece of code (an inner class definition) to be closed over many different environments. However, the Java specification requires that free variables in an inner class be final, i.e., immutable. In a functional language, such a requirement does not impose great burdens on expressivity, but functional programming in Java is hard to do because many of its most important features are defined in an imperative style. Thus, we suspect implementations are unlikely to view task migration as a critical issue because distributed programs written in Java will not be able to take advantage of inner classes to separate control from state.

6 Conclusions

Distributed extensions of Java such as Java/RMI combine features found in both agent-based languages and more traditional RPC-based distributed systems. While providing the encapsulation and protection benefits of traditional client/server RPC systems, Java's program model allows code to be dynamically linked and executed on remote nodes.

However, we have identified three fundamental characteristics of Java that we believe make it hard to express lightweight distributed mobile processes. First, the highly imperative nature of its sequential core complicates the semantics of distributed programming via message passing. The semantics of remote method invocation differs substantially from that of local method inocation. Second, a client/server communication model requires a remote call to return back to the sender once the call is complete. Since the continuation of the call cannot be explicitly supplied, mobility is hampered. Third, the close coupling of data and state make it difficult to express task migration as an issue orthogonal to object migration even though the circumstances under which the two would be exercised are very different.

We expect that there is much to be gained by exploring the interaction between an object semantics and distributed programming via mobile processes.

References

[1] AGHA, G. *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT Press, Cambridge, Mass., 1986.

[2] BENNETT, J. K., CARTER, J. B., AND ZWAENEPOEL, W. Munin: Distributed Shared Memory Based on Type–Specific Memory Coherence. In *Symposium on Principles and Practice of Parallel Programming* (March 1990).

[3] BIRRELL, A., NELSON, G., OWICKI, S., AND WOBBER, E. Network Objects. In *14th ACM Symposium on Operating Systems Principles* (1993 December).

[4] BIRRELL, A. D., AND NELSON, B. Implementing remote procedure call. *ACM Transactions on Computer Systems 2*, 1 (1984), 39–59.

[5] BLACK, A., HUTCHINSON, N., JUL, E., LEVY, H., AND CARTER, L. Distribution and abstract data types in emerald. *IEEE Transactions on Software Engineering 13*, 1 (1987), 65–76.

[6] CARRIERO, N., GELERNTER, D., JOURDENAIS, M., AND KAMINSKY, D. Piranha scheduling: Strategies and their implementation. *International Journal of Parallel Programming 23*, 1 (1995), 5–35.

[7] CEJTIN, H., JAGANNATHAN, S., AND KELSEY, R. Higher-Order Distributed Objects. *ACM Transactions on Programming Languages and Systems 17*, 5 (1995), 704–739.

[8] GOSLING, J., JOY, B., AND STEELE, G. *The Java Language Specification.* Sun Microsystems, Inc., 1995.

[9] JANSEN, B., SPREITZER, M., LARNER, D., AND JACOBI, C. *ILU 2.0alpha12 Reference Manual.* Xerox Corporation, 1997.

[10] JUL, E., LEVY, H., HUTCHINSON, N., AND BLACK, A. Fine-grained mobility in the Emerald system. *ACM Transactions on Computer Systems 6*, 1 (January 1988), 109–133.

[11] KOTZ, D., GRAY, R., NOG, S., RUS, S., CHAWLA, S., AND CYBENKO, G. AGENT TCL: Targeting the Needs of Mobile Computers. *IEEE Internet Computing 1*, 4 (1997), 58–67.

[12] LEA, D. *Concurrent Programming in Java: Design Principles and Patterns.* Sun Microsystems, Inc., 1996.

[13] LI, K., AND HUDAK, P. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems 7*, 4 (1989), 321–359.

[14] MOWBRAY, T., AND ZAHAVI, R. *The Essential CORBA: Systems Integration Using Distributed Objects.* Wiley, 1996.

[15] MULLENDER, S., Ed. *Distributed Systems.* Addison-Welsey, Reading, Mass., 1993.

[16] POWELL, M., AND MILLER, B. Process migration in demos/mp. In *Proceedings of the 9th ACM Symposium on Operating Systems Principles* (New York, 1983), ACM, pp. 110–119.

[17] ROGERS, A., CARLISLE, M., REPPY, J., AND HENDREN, L. Supporting Dynamic Data Structures on Distributed-Memory Machines. *ACM Transactions on Programming Languages and Systems 17*, 2 (1995), 233–263.

[18] SCHRODER, M., AND BURROWS, M. Performance of firefly rpc. *ACM Transations on Computer Systems 8*, 1 (1990), 1–17.

[19] SNIR, M. *MPI: The Complete Reference.* MIT Press, 1996.

[20] WOLLRATH, A., WALDO, J., AND RIGS, R. Java-Centric Distributed Computing. *IEEE Micro 2*, 72 (May 1997), 44–53.

Biographies

Suresh Jagannathan received his Ph.D from the Massachusetts Institute of Technology in 1989. He has spent the past eight years as a Research Scientist at the NEC Research Institute. Prior to joining NECI, he was a Research Faculty member at Yale University. His interests are in the areas of program analysis for mostly-functional languages, compiler design, and distributed programming languages.

Richard Kelsey received his Ph.D from Yale University in 1989. He joined the NEC Research Institute as a research scientist in 1993 after spending three years on the faculty of Northeastern University. His interests are in the areas of programming language design and implementation, usually concentrating on Scheme and similar languages and on building simple, general-purpose compilers.

Steps toward Understanding Performance in Java

Doug Lea
Computer Science Department
State University of New York at Oswego
Oswego NY 13126

Abstract

Java's design goals of portability, safety, and ubiquity make it a potentially ideal language for large-scale heterogeneous computing. One of the remaining challenges is to create performance models and associated specifications and programming constructs that can be used to reason about performance properties of systems implemented in Java.

1 Introduction

Java is the first mass-market concurrent, distributed, object-oriented language. To the extent that heterogeneous computing requires near-universal platform support for a given language ("Write once, run anywhere"), Java is currently the *only* answer for programming non-experimental heterogeneous systems. But is Java a *good enough* answer? Can it become good enough?

Java provides support for the common demands of system-wide heterogeneous computing: Concurrency via threads, locks and monitors; Distribution via Remote Method Invocation (RMI) and related frameworks; User interaction via the AWT; Persistence via serialization and database connections; Mobility via class loaders; and Security via principals, security managers, etc. Across all of these domains, as well as base language constructs, the primary design goals have surrounded portability, safety, historical precedent, and minimality. These goals have been traded off against performance, exploitation of machine-specific capabilities, and availability and real-time guarantees.

2 Performance Models

Reasoning about performance is an integral part of system-level development. Currently, system developers face more extreme versions of the kinds of problems that beset early developers of simple Java applets and applications. Developers could not be confident that a given Java Virtual Machine (JVM) would meet even the most minimal correctness and performance criteria needed for acceptable execution. The widespread deployment of Just-In-Time compilers, dynamic compilation, and more efficient run-time systems have alleviated some of these concerns. Others have been addressed by providing high-level, portable choices for mapping designs to implementations with different performance characteristics. For example, user interface programmers may now choose between "heavyweight" AWT components that are implemented directly by native windowing systems versus "lightweight" components that are implemented mainly in Java proper. Neither is always best with respect to performance and other design criteria.

However, these concerns become much more challenging at a systems level, and have yet to be addressed systematically by JVM implementors. Example issues include:

- How are threads mapped to different processors in SMPs?
- How is persistence mapped to high-performance random access devices (mainly disks), serial devices (mainly networks), transactional processing, etc?
- How is locality exploited in message-based remote communication?
- How is Java synchronization mapped to spinlocks versus JVM scheduling versus kernel scheduling?
- How are known regularities exploited for resource management?
- How can system-wide control and monitoring be extended, for example to include checkpointing and deadlock detection?
- How can soft-real-time requirements be used to influence scheduling?

Java is currently silent about most of these issues, leaving too much freedom in the hands of JVM and Java library and tool implementors, and hence too much uncertainty for developers to be able to reason

0-8186-8365-1/98 $10.00

about performance. Right now, the only way for developers to deal with this is to build their own custom JVMs, support libraries, and/or tools. While the laxity of Java specifications allows this, it is an unacceptable solution in the long run since it allows developers to reason about performance only on particular implementations.

An alternative is to construct a portable system-level performance model for Java, that is honored by JVM, library, and tool implementors. Aspects of such models have been implicit in most in-the-small performance-related efforts. However, they must be made explicit to scale to systems-level concerns. The heart of a performance model is an abstraction of a computer system providing just enough detail to express mappings and choices among mappings, yet noncommittal enough to apply to JVMs residing on smartcards, supercomputers, and everything in between. Such a model could then be used to provide various styles of rules:

- System-specified mappings: If a capability exists, it will be mapped in a certain fashion.
- Default mappings: Rules that apply unless overridden by programmers.
- Programmer-specified hints: Constructions that allow programmers to heuristically influence or tune parameters of a mapping. These need not take the form of tuning APIs, but may instead for example associate performance properties with different programming constructions.
- Programmer-specified mappings: APIs that allow programmers to plug in control modules and the like.
- Multiple mappings: Different APIs with different performance characteristics, that programmers may choose among.
- Intentional opacity: Reserving the right of implementors to make any mapping choice, unknowable by programmers.

The main challenge is to identify those components of a performance model that significantly impact the ability to reason about performance, yet can be used as the basis of usable, portable, and readily implementable programming constructions. Members of the heterogeneous computing community have much to contribute toward such efforts.

Perhaps in an ideal world, all rules would be of the first type, requiring "optimal" mappings to system capabilities. However, the world is rarely this ideal. For example, the benefit of placing threads on different processors of an SMP generally varies inversely with communication rates among threads. It is hard to imagine placement strategies that would not benefit from information that reveals expected communication rates. Such hints would of course be ignored or used in some other heuristic fashion (for example to help choose between user-level versus kernel-level threads) when programs are run on uniprocessors.

And even in an ideal world, some mappings must remain opaque; for example those that would otherwise reveal information that would compromise safety and security properties.

JVM-level performance models may in turn give rise to application-level models. For example, a common Java programming dilemna surrounds how to map object communication to any of many available forms, including direct method invocations, notifications among threads, JavaBean-style events, structured RMI-style messages, applet-style class transport, serialized mobile-code-style commands, database transactions, and so on. While performance concerns are typically only one factor in such decisions, the ability to approximately predict the performance characteristics of different choices can lead to development of more usable and more useful Java-based systems.

Heterogeneous Programming with Java: Gourmet Blend or just a Hill of Beans?

Charles C. Weems Jr.
Department of Computer Science
University of Massachusetts
Amherst, MA 01003-4610
weems@cs.umass.edu

Abstract

The heterogeneous parallel processing community has long been struggling to bring its approach to computation into the mainstream. One major impediment is that no popular programming language supports a sufficiently wide range of models of parallelism. The recent emergence of Java as a popular programming language may offer an opportunity to change this situation. This article begins with a review of the special linguistic and computational needs of heterogeneous parallel processing by considering the user communities that would benefit most from the approach. It then reviews the pros and cons of Java as a language for expressing and realizing heterogeneity, and concludes with some possible changes that would make Java more suitable for such use.

1. Heterogeneous Programming: Who Needs It?

Before we look at the relationship between Java and heterogeneous programming, we should first review what is involved in programming heterogeneous systems: where are they used and how? Once we identify the requirements for supporting software development for heterogeneous systems, we have a better basis for judging the applicability of a programming language. What follows is not meant to be an exhaustive survey of the field, but merely a discussion of some well-known examples to motivate the identification of a set of requirements.

There are three basic reasons for writing programs that involve heterogeneous parallelism: because we need to use heterogeneous hardware, because our problem is inherently heterogeneous in nature, or because we are faced with some combination of the two. In practice there are many gray areas between these distinctions. For example, to some applications, a distributed shared memory parallel processor may be completely homogeneous, whereas others may be sensitive to differences in memory access time and thus see such hardware as heterogeneous. Likewise, while one approach to solving a problem may be inherently heterogeneous, there may be other approaches that are more homogeneous in nature. In what follows it is implicit that programmers are always faced with a spectrum of choices and that the use of heterogeneity in any given instance is a matter of degree rather than absolute.

2. Heterogeneous Hardware Users

In some situations, the system architect is forced to turn to heterogeneous hardware. The necessity for heterogeneity can be due to space and power requirements as in embedded processing, or due to cost considerations as in clustered workstation farms, or simply a matter of physical limitations of technology as with large-scale shared memory multiprocessors. Heterogeneity can also result from systems that change their configuration dynamically, as in the case of adaptive computing hardware or network computing in which the availability of nodes is subject to change. In the sections that follow, we consider some of the special programming issues that are associated with each of these situations.

2.1 Embedded Systems

Most embedded systems are strongly constrained by limitations such as size, weight, power and cost. Many embedded systems are not high-performance in nature, and the goal is simply to minimize cost while achieving the necessary level of performance. However, when requirements for high performance are combined with embedded system limitations, there is often a considerable benefit to employing heterogeneous parallelism. For example, combining a digital signal processor (DSP) with a microprocessor and some custom logic can be more cost

0-8186-8365-1/98 $10.00

effective or achieve a higher level of performance than using multiple identical microprocessors.

Achieving high performance with DSP and custom logic, however, involves an especially high degree of optimization of certain algorithms for the hardware. There may be just a single way of optimally coding an algorithm for a DSP that was specifically envisioned by its designer. For example, some DSP architectures include address arithmetic instructions that are unique to a Fast Fourier Transform (FFT), and their use can speed up the inner loop of that algorithm by nearly an order of magnitude. Typically, these algorithms are hand-coded in assembly language and provided as external libraries.

While the library approach works in limited situations, it presents problems of portability and flexibility. One of the goals of heterogeneous programming is to reduce the dependence on hard-coded machine-specific libraries so that code can be ported to different heterogeneous platforms with minimal effort. A program that is written with such library calls can't be ported to another platform (or even run on a uniprocessor) until the library is rewritten for the new platform.

The alternative is that we write the library's algorithms in a high level programming language so they can be compiled for whatever system we choose. Of course, we then generate suboptimal object code for the DSP. While we could perhaps build a compiler to recognize and optimize certain key DSP algorithms carefully written in some canonical form, it would be difficult to handle the broader spectrum of DSP algorithms or even minor variations on the key subset.

A simple but effective solution is to provide the programmer with the ability to uniquely name an algorithm that is implemented in multiple ways (i.e., in high-level code and in libraries) and to indicate either a specific target or the conditions that determine the appropriate target for each implementation. For example, a program might include the code for a generic FFT, and the compiler might detect that there is a corresponding FFT library function for one of the target processors. Depending on how the code is partitioned among the processors, the compiler either generates new FFT code or a library call. Database researchers refer to this as ad-hoc polymorphism, and we have previously called it pseudomorphism [Weems1994] because it is analogous to the mineralogical form of the same name in which a crystal is chemically replaced by another compound in such a manner that the external appearance remains unchanged.

Implicit in the foregoing discussion is the notion that the compiler or some other tool is able to partition code among processors of multiple types. Partitioning implies that there is some means of estimating the performance and cost of mapping code segments to processors. While it is sometimes possible for partitioning tools to analyze code and identify first-order factors affecting its performance, it is also the case that the programmer may have specific information that can help to guide partitioning, and should be given a means to express it.

Partitioning tools also need hardware-specific cost estimators both for the individual target processors and for the communication mechanisms that connect them. This can either be in the form of dedicated software for each target or more general software that bases its estimates on hardware descriptions expressed in some language. This isn't necessarily the same language that the programmer uses, but it is difficult to decide whether it is best to create a whole new language or to extend an existing language with constructs that most programmers will never use.

2.2 Adaptive Computing

Processors that can change their configuration, such as field programmable gate arrays (FPGA) present challenges that are similar to heterogeneous computing systems. They are usually employed in embedded applications where separate processing phases require different custom computing hardware and thus it is possible to use a single component that reconfigures itself between phases. Adaptive hardware is often used as a coprocessor in a system that includes a DSP or traditional microprocessor.

Like DSP systems, adaptive systems often rely on libraries of manually optimized functions. An alternative approach for programming adaptive devices is to generate configurations automatically. Currently this is done only from hardware description languages (HDL) or from customized high-level languages that enable users to express computations in ways that are more suited to hardware layout (e.g., dataflow with datapath width information).

In terms of heterogeneous programming, the implications of the library approach are similar to those for DSP-based embedded systems. However, for automatic generation of configurations, the implications are that a language should provide some features similar to those of hardware description languages, including pipelining, clocking and synchronous communication, datapaths and functional units of varying widths.

The implications of adaptive computing for partitioning and mapping are that the cost model is more complex and performance estimates depend more on detailed analyses of the actual circuitry. Because there are many ways to lay out a particular circuit that affect different aspects of its performance, there is a larger mapping space to explore. The mapping space could be

considerably constrained by additional information from the programmer.

2.3 Clustered Workstations

Heterogeneous computing is often most closely associated with networked workstations in which multiple models are employed so that nodes differ significantly from each other in terms of performance and capacity. In many cases, the workstations are used in a manner similar to a homogeneous parallel processor and it is simply a matter of partitioning operations in parallel across the available resources. The reason for adopting this approach is typically to save cost by using existing hardware resources and free software to build an ad-hoc parallel processor, although some clusters are purpose-built.

Because clusters typically employ a standard computer network, communication between nodes has high latency and limited bandwidth. Thus it is common to partition jobs in a manner that minimizes communication, such as having a master processor distribute work to slaves that compute intensively for some period before returning a result. Partitionings of this nature are naturally expressed via message passing, and a significant amount of legacy code now exists that uses either the PVM or MPI library. Thus, in the near term a language must support interfaces to these libraries.

In the long term, a goal of heterogeneous computing research should be to support more automation of code partitioning, mapping, and distribution in these environments. However, their sheer diversity combined with a focus on low cost and modest software effort may make it difficult to provide a more sophisticated solution that is acceptable to this particular user community.

2.4 Nonuniform Memory Access

As MIMD parallel processors scale up in size, they encounter various physical limits that force their designers to sacrifice uniformity of memory access latency. One approach that has been adopted is to cluster processors in groups of two to eight within which they have uniform access latency, and access to shared memory outside of the cluster is slower (Figure 1). In some cases, the clusters are also grouped into a hierarchy. Another approach is to connect the clusters with a message-passing network with the result that programs can either employ a heterogeneous mixture of shared and distributed memory, or they can use software emulation of shared memory outside of clusters with a resultant increase in latency.

All of these architectures benefit from appropriate partitioning and mapping to enhance locality of reference. Traditional memory placement optimizations can be modified to some extent to deal with the nonuniform access latencies, and in doing so start to resemble partitioning strategies for heterogeneous systems. High Performance Fortran (HPF) is a recent attempt to extend a language with constructs that enable the programmer to provide additional information to aid the partitioning of data.

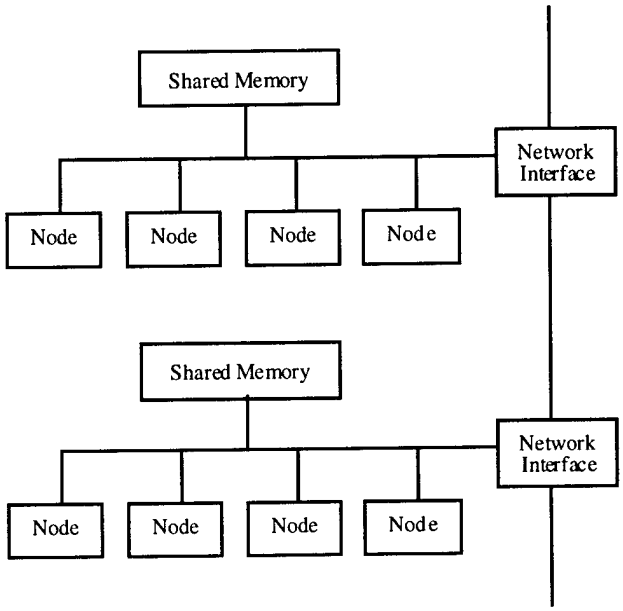


Figure 1. Clustered Parallel Architecture

While some sort of programmer-provided partitioning information is probably needed, the HPF extensions and especially their interactions with the Fortran-90 parallel extensions have proved to be particularly troublesome to compiler writers. Thus, an important consideration in evaluating a language for heterogeneous programming is to ensure that its features do not conflict with each other.

3. Solvers of Heterogeneous Problems

As we move beyond embarrassingly parallel applications to address problems with greater complexity, we find that they are often most naturally expressed with a combination of parallel processing modes. In the previous sections, we focused on the language features that are needed to enable users to inform partitioning tools so that they can distribute code onto heterogeneous hardware. In the sections that follow, we consider via some examples the language features needed to express the natural heterogeneity of parallelism in some applications.

3.1 Scientific Computing

Scientific codes are relying more on irregular computations, as in the case of simulating turbulent fluid flow, in which areas outside of turbulence zones are regular and sparse but within turbulence zones meshes are dense and irregular in their topology. Applications such as these require language features that support the definition of appropriate data structures, such as variable density triangular meshes, and new arithmetic operators that process them. One approach to implementing these is to use an index array that points into a data array and explicitly manage the adjacencies in the mesh. A set of functions can be written to carry out arithmetic operations on the mesh. However, all of this structure obscures the true relationships between the data elements and between operators, making it difficult to optimize them.

Expressing an irregular data structure at a higher level should both simplify programming and optimization. For example, making the property of adjacency explicit might enable programmers to define an adjacency matrix and automatically inherit certain operations that depend on adjacency. While this could be syntactically expressed with a class hierarchy, the compiler sees only the underlying implementation and cannot take advantage of higher level aspects of adjacency as a property in optimizing computations.

Linguistically, the ability to overload arithmetic operators provides a convenient syntactic sugarcoating that hides the functional implementation of operations on new data types. But overloading by itself does not enable the compiler to optimize expressions of these new operators in the same way that it optimizes expressions made up of built-in operators. The ability to add semantic property information to definitions to facilitate optimization is a necessity. Such a capability need not resort to the generality and complexity of mechanisms such as denotational semantics [Gordon1979]. Rather, simply having a list of properties that the compiler recognizes and which can be attached to definitions would be adequate.

Simulations of whole systems or processes are a natural source of heterogeneous parallelism. For example, simulating an entire jet engine involves various stages of compression, fuel-air mixture, combustion and exhaust, as well as mechanical stresses and thermodynamics. Simulating each of these aspects of the engine involves different computational techniques with different degrees of parallelism, connected in a dataflow structure that mimics the physical relationships between the engine components.

3.2 Computer Vision

The complexity of interpreting visual information necessitates many different processing techniques. Image processing and feature extraction provide opportunities for fine-grained data parallel processing. Image sequences can be processed in a pipelined manner and multiple features can be extracted simultaneously with MISD parallelism. Extracted features can be combined with SPMD parallelism into larger structures, and shared memory MIMD parallelism can be used to search the extracted features for matches to multiple objects at once. The overall processing may be coordinated with a data flow model.

The implications of our two example application domains for a programming language are that it must be able to support a wider range of modes of parallelism than merely data parallel and multiprocessing in order to facilitate programming of heterogeneous applications. In addition, it must support novel combinations of parallelism. Ideally, it would also enable the programmer to define new models of parallelism to suit a specific problem.

4. Summary of Requirements

From the foregoing discussion, we list the following requirements for a programming language suited to heterogeneous programming:

- The ability to uniquely identify algorithms.
- Express partitioning and mapping information.
- Interface to PVM, MPI, etc.
- Support message passing over networks.
- Avoidance of conflicting language constructs.
- Overloading of arithmetic operators.
- Ability to add semantic information.
- Ability to define irregular structures.
- Express structural relationships such as adjacency.
- Support a wide range of parallel models, including
    - SIMD,
    - SPMD,
    - MISD,
    - MIMD,
    - shared and distributed memory,
    - dataflow,
    - pipelining.
- Allow parallel models to be combined flexibly.
- Permit extension to new models of parallelism.

In our previous survey of programming languages for heterogeneous parallelism [Weems1994], we identified a

set of requirements that make up the abilities to flexibly combine models of parallelism into new models:

- Support for a range of data and control grain sizes.
- Ability to define a communication abstraction.
- Ability to define a synchronization abstraction.
- Ability to specify patterns of data distribution.
- Ability to specify patterns of process distribution.
- Able to define new first-class types.

5. Why Java?

5.1 It's Hot.

At one time, introducing a new programming language was just a matter of creating a compiler and making it available. In the world of computing today, where legacy code and compatibility tend to dominate the economics of software development, it takes a confluence of many factors to enable a new programming language to enter the mainstream. Java appears to be the right language to emerge at the right place and the right time.

Because Java is rising fast but is also still evolving, it presents an opportunity for many special interests to try to influence the design of what may become the dominant programming language of the next decade. The parallel and heterogeneous computing communities have been struggling for years to develop languages that would gain some measure of acceptance. If Java can be made suitable for their purposes with just a few minor additions, then perhaps those approaches will finally enter the mainstream and researchers can move on to new levels of research. Java was originally designed for embedded systems, so it would superficially appear that it should address the concerns of that segment of the heterogeneous processing community.

5.2 It's Simple

In comparison to other modern object-oriented languages, the essential syntax of Java is reasonably simple, and it's core is familiar to any C programmer [Gosling1996]. It avoids many of the pitfalls of larger languages like C++ and Ada95 but does not sacrifice convenience for the sake of minimalistic purity. It may certainly be argued that there are syntactically "better" languages, but there is general agreement that Java is an improvement over many of its predecessors.

Simplicity also implies that a language will be easier to learn, Java thus has the potential to win over converts from existing languages used in heterogeneous, parallel, embedded, and adaptive programming such as Fortran, C, C++, Ada, and VHDL. Of course, to do so, it must also provide a similar level of convenience of expression of essential constructs in each of those domains.

Being syntactically simple, Java offers a better base upon which to add constructs in support of heterogeneity, if necessary, because the resulting interactions between new and old constructs are easier to enumerate. Java's simplicity includes the avoidance of many features of earlier languages that were machine-dependent or that gave the user too much low-level control (e.g., common, equivalence, pointer address arithmetic, explicit memory management). It thus offers a stronger foundation for extensions.

5.3 It's Portable

Because Java targets a virtual machine (the JVM [Lindholm1997]), it can be executed on any machine with an implementation of the JVM. This approach to a run-time environment overcomes many of the difficulties that are faced in trying to distribute code across heterogeneous systems. Issues of word size, endianness, register file size, operating system, and display environment all disappear. In effect, Java homogenizes the heterogeneous world for us through its virtual machine.

The result of this homogenization is that we can focus on the bigger picture of high-level performance characterization in support of partitioning and mapping. All code is generated for a single instruction set and descriptions of target machines can be in simpler terms such as memory capacity and performance on some Java benchmark kernels.

5.4 It's Object-Oriented

Besides being a popular buzzword, object-orientation gives a language great syntactic power for extension. It is possible to develop classes that provide convenient abstractions for many forms of parallelism. For example, one could build an irregular array class that makes it simple to express the kinds of computations that are performed in turbulent fluid-flow simulations. That abstract interface can hide either a sequential or a parallel implementation.

Object orientation also offers greater potential for reuse and extension of user constructs by others. Given classes that implement different models of parallelism, we could combine them in a variety of ways to enable heterogeneity to be expressed.

5.5 It's Garbage Collected

By avoiding explicit memory management by the programmer, Java greatly reduces the opportunity for

errors resulting from memory leaks or bad address arithmetic. However, for parallel processing, garbage collection offers an opportunity for redistribution of processing load. As a garbage collector sweeps through memory compacting live and dead values, instead of merely gathering the live values into one place it could redistribute them to other (possibly heterogeneous) computational nodes.

5.6 It's Threaded

Java's threads provide a simple but powerful mechanism for expressing shared-memory parallelism. Many models of concurrent programming can be expressed with the Java's threads [Lea1997]. It's runtime system, the JVM, already supports multiprocessing so that a consistent abstract parallel environment can be used on all targets.

5.7 It's Network Aware

Java applets provide a means for distributing code across a local area network. In addition, its object-oriented approach can be used as a form of message passing through the remote method invocation facility. More directly, it has the ability to explicitly handle network connections as streams. When this is combined with Java's reflection capabilities, we find that there are many options for implementing parallelism in Java.

6. Java Weaknesses

6.1 It's Hot

The fact that Java is new and being influenced by many interest groups means that the language itself is continuing to evolve at a fast pace, making it difficult to identify specific extensions without risking conflicts that could arise from other proposed extensions. Java may still evolve into a language with features that make it unsuitable for heterogeneous parallel applications. Given the ongoing battle between Microsoft and Sun over Java standards, it is even still possible that Java will fail to reach a critical level of acceptance or be replaced by a different language.

Because of Java's perceived popularity, the heterogeneous programming community has very little influence over Java's evolution, and so must ride the coattails of others in any effort to extend Java for their benefit. A domain-specific language would not be subject to the same restrictions.

6.2 It's Simple

One of the key simplifications that the Java designers chose was to avoid operator overloading. Unfortunately, the lack of operator overloading precludes programmers from writing new arithmetic classes that are as convenient to use as the built-in arithmetic types. Without operator overloading, it will be difficult to convince scientific and engineering users of languages like Fortran 90, HPF, C++, and Ada to switch to Java. In effect, they must return to the levels of notational convenience that were available in Fortran 77, C and Pascal.

While Java provides access to the IEEE floating point standard's special values of plus and minus infinity, plus and minus zero and not-a-number, it does not provide the ability to control the various states of the IEEE standard, such as the rounding mode or the use of subnormal values. Java does not generate any exceptions for floating point operations, and for integers the only exception is division (or remainder) by zero. In particular, overflow and underflow are not detected. These limitations do not help to convince programmers of numerically-oriented applications to switch to Java.

6.3 It's Portable

Java's portability comes at the price of another layer of abstraction (the JVM), which reduces performance. If the abstraction layer is implemented by a bytecode interpreter, then the cost is quite high over native code execution, both in terms of time and space. If JIT compilation of the bytecodes is used, then the time penalty can be reduced as long as the compilation time can be amortized over enough execution time for each run. Off-line translation of bytecodes into native code avoids the JIT compilation cost on each run, but bytecodes are then effectively a low-level intermediate representation of the program that prevents the translator from applying higher-level optimizations before generating code. The result is object code that is less optimized than a native compiler could produce from source code.

Java's approach to portability thus results in a significant reduction in performance. This should be anticipated simply from the way in which Java attempts to homogenize the world of processor architectures. There is little point in using heterogeneous hardware if it will be programmed in a manner that ignores the special hardware features that were selected to improve performance. Java's bytecodes were designed in part to facilitate embedded processing such as appliance controllers, but not high-performance embedded processing such as DSP.

6.4 It's Object-Oriented

Object orientation provides only the illusion of language extensibility. However, there is still a rigid distinction between the first-class constructs and types of the language and user-defined objects. First-class objects carry additional semantic properties that are used for key optimizations. The programmer has no way of providing similar semantic information when defining new objects. The programmer cannot even indicate that existing semantic properties associated with first-class types apply equally to a new type or operator. For example, in defining a new numeric type, the programmer has no way to indicate that the addition operator is associative.

While object oriented programming can be used to define a triangular mesh data type that syntactically simplifies the coding of applications, the compiler has to optimize at a much lower level of abstraction and will miss opportunities for high-level optimizations. This is problematic for partitioning and mapping in heterogeneous systems, where knowing the organization and access patterns of larger structures is especially valuable.

6.5 It's Garbage Collected

The time-penalty of garbage collection is difficult to predict. Depending on the available system memory and on the data being processed, the frequency with which the garbage collector is invoked and the time that it takes can vary considerably. In many systems, long pauses for batch collection are unacceptable, but incremental garbage collection can be used to distribute the time so that the user does not notice delays but merely sees some variations in performance.

In hard real-time embedded systems, however, the performance of the processor must be highly predictable to avoid missing deadlines. The alternative is to assume a worst-case execution time that is so great that much of a processor's performance goes unused because tasks are scheduled under the assumption that they will all suffer a maximum penalty for garbage collection.

6.6 It's Threaded

While threading provides at least one native model of parallelism in the language, it is not sufficient by itself for efficiently defining all other models of parallelism. Combining threads with object-oriented techniques and other features of the language could make it possible to syntactically express other models at least in a round about way. For example, SPMD parallelism could be crudely implemented with an array of threads, onto which data arrays are partitioned, and locks could be manually programmed to ensure synchronization at the end of each basic block. However, that is far less convenient than a FORALL or PARDO construct.

The appearance of SIMD parallelism could be obtained with suitable whole-structure operators on structured classes. But the only actual means of implementation would be external library calls. Without operator overloading and semantic extensibility, however, there would be no opportunity for optimizing across calls. For example, an expression (written as method calls) that operates on a parallel array type could not have common subexpressions eliminated, nor could the registers of the parallel hardware be scheduled efficiently.

In essence, Java provides one basic approach to parallelism, and does not facilitate the use of other models. It thus neglects the needs of people who need to solve heterogeneous problems. It has also been noted that the primitive nature of Java's threading facility is analogous to the combination of pointer arithmetic and explicit memory management in C in that it provides sufficient rope to hang the programmer [Lewis1997]. Various user communities would be better served by a parallelism facility that is less prone to deadlock, livelock, and orphaned threads.

6.7 It's Network Aware

Java's network awareness is built on top of TCP/IP and HTML. Connections are established between Java programs using internet addresses and port numbers, and applets are referenced by internet addresses plus file names. While this greatly simplifies the network interface and makes it widely portable, the latency of the layered network protocol is significant. For a cluster of workstations, where the communication network is also built around this protocol, there is little choice but to accept the high latency. However, in purpose-built clusters or distributed-memory parallel processors, where communication is mainly between trusted peers, the latency is unnecessary.

While this is partly an operating system issue, Java does not inherently provide a mechanism to distinguish between secure communication and trusted low-latency communication. Adding a library built on special OS calls would allow programmers to work around this limitation, but then their code would not be portable.

There are no provisions in Java's model of network communication to support parallel operations such as broadcast and reduction. Again, these could be provided with manual workarounds, such as a native interface to MPI, but the bottom line is that Java was not designed for parallel processing network communication.

7. Making Gourmet Java

The foregoing discussion highlights various features of Java that make it both attractive and unsuitable for heterogeneous programming. In this section we consider some possible changes that would make Java more suitable for heterogeneous programming. It should be noted that these are not all syntax changes. Some of the changes involve the compiler and the virtual machine.

7.1 Syntactic Additions

Operator overloading is a key syntactic addition that is needed to enable programmers to write new numeric classes that match the expressiveness of other languages. Syntactically this is a minor change to the language if it is restricted to overloading the existing operators, although it significantly affects parsing, conversions, and promotions. If generalized to enable arbitrary monadic and dyadic operator forms for method invocations, then it would be a more significant syntactic change.

To support pseudomorphism, the syntax for the `implements` clause in a class declaration should be extended to support multiple implementations of the same `interface` by classes with the same name in a single package. The redundant classes must then be distinguished by a predicate. For example, we might write

`implements` *interface-list* `when` *Boolean-expression*

to indicate the conditions that determine when a particular class from the set of identical classes is selected as the appropriate implementation of an interface.

The conditions could be resolved at compile time, load time, or run time, according to the information that they depend on, which implies that the compiler, loader, and JVM must all be extended to perform these tests in a heterogeneous environment. For example, a condition might call `getProperties` to determine the type of target onto which it is being loaded so that only the implementations appropriate to the target would be loaded onto it. If multiple implementations are loaded at run time, then when the first member of the interface is invoked, the run-time system must test their conditions to identify the one that will be used. Once an implementation has been selected, it is used for as long as the class instance exists. If the conditions for multiple implementations are satisfied then the run-time system chooses the implementation. If none of the conditions are satisfied, then an exception is thrown.

If conditions change such that it would be desirable to switch to a different implementation (e.g., due to changing system load), then the class instance must be destroyed and a new instance created using the alternate implementation. It would be up to the programmer to decide how to convert the state of one implementation into another in such a transition.

7.2 Semantic Extensibility

A limited but sufficient mechanism for semantic extensibility could be achieved by enriching the set of attributes explicitly recognized by the Java compiler and virtual machine, and making them accessible at the source level. The existing attribute facility of the virtual machine is sufficient to tag any class, field, or method with additional information. For example, the `ConstantValue` attribute indicates to the JVM that a field is a constant.

If the list of attributes is extended to explicitly include all of the currently implicit semantic properties used to trigger or enable optimizations, and these are made available to the programmer, then it becomes possible to extend the language with new first class types. For example, we might write the following form (which also assumes an extension for overloading arithmetic operators):

```
attributes(associative, commutative)
static complex dyadic +
                (complex left, complex right)
```

This code would define a new + operator for a complex type that would carry the built-in attributes necessary to enable high-level optimizations of expressions containing it.

If the user specifies an attribute that does not exist in the system, a warning would be issued, but it would not result in a fatal error. The JVM is specified to ignore unrecognized attributes. This would allow implementations to carry attributes that are specific to certain JVMs but not others (e.g., a parallel JVM). In addition, because the attributes are carried through to the bytecode representation, it is possible for a bytecode to native code translator to employ some of its own high-level optimizations. For example, the translator could use the attributes to enable more aggressive register scheduling.

One attribute that would facilitate the creation of irregular types would be a means of indicating the adjacency relationships between elements in a data structure. Attributes could also be used to carry information to guide partitioning and mapping of structures.

7.3 Compiling

Compiling Java in a heterogeneous environment is likely to involve the initial bytecode generation, together with native code generation (either off-line or JIT) once the bytecodes are downloaded to a specific target. As mentioned in the previous section, a wider range of attributes must be carried in the `class` file containing the bytecodes to enable delayed optimization. It may, in fact, be necessary to carry a higher-level intermediate representation (IR) of the code in the `class` file to enable all of the desired optimizations. The IR might end up being comparable in sophistication to a program dependence graph augmented with source node-type information and links to the generated bytecode. In essence, this would enable the native code translator to start with a higher level view of the code and generate all new code for the particular target. Having such information available would especially facilitate target-specific partitioning of parallel operations.

7.4 Garbage Collection

Currently, Java supports the `gc` method to force garbage collection to occur. However, there is no way to ensure that collection does not occur during a time-critical section of code. The gc method should be extended to accept various parameters, such as `gc(FALSE)` to turn off collection until it is either explicitly enabled with `gc(TRUE)`, or the current method exits.

In addition, the user could be given more control over the collection process, such as indicating whether a more costly incremental collection should be run to minimize pauses or that a faster periodic sweep that results in noticeable pauses is acceptable.

7.5 Other Models of Parallelism

There are other extensions that would make it easier to explicitly write certain kinds of parallel code, such as FORALL or WHERE, but it can also be argued that parallelizing compiler technology can often identify these cases when normal loops are written to directly implement sequential versions of them.

Directly supporting multiple models of parallelism in Java depends not so much on changes to the language as to the JVM, which currently recognizes only threads as being concurrent. Given operator overloading, pseudomorphism, and semantic extensibility, we can express nearly all forms of parallelism. For example, the features of a data parallel language like ZPL [Snyder1994] can be realized with object-oriented techniques, although not with the same syntactic simplicity. Consider that ZPL regions can be implemented as arrays with the appropriate attributes and region specifiers can be written as methods that manage a global context variable. The result will not be as pretty, but if it carries the appropriate attributes through to a parallel runtime environment, the resulting code should have similar efficiency.

Of course, the ultimate in extensibility would be the capability to define new control structures. However, such an extension would involve the ability to pass expressions and code blocks to methods, where they could be executed with some level of intervention. When combined with support for pseudomorphism, however, the result would be the ability to directly express parallel operations whose implementation is determined by the available hardware; which is precisely what users of heterogeneous processing are seeking.

8. Conclusion

Java offers a combination of opportunities and features that make it an attractive language for heterogeneous parallel processing. However, a deeper study reveals some serious shortcomings that will make it difficult to attract users from the major communities that need heterogeneity. In particular, it lacks key support for scientific and high-performance embedded processing. In addition, the way in which it homogenizes the world to achieve portability directly conflicts with the fundamental reason for employing heterogeneity.

However, with some modest extensions to the language, and suitable restructuring of the compiler, loader, and run-time system (including native code generation from bytecodes and a higher level IR), Java could be made much more suitable for heterogeneous parallel processing. The major syntactic changes would be to enable operator overloading, extend the `implements` clause so that multiple classes with the same name can implement an interface in a manner that allows the system to choose between them, and make an enriched set of standard attributes accessible to the programmer. Support for user-defined control structures would be a significant change in syntax and semantics, but would allow expression of parallel operations with the syntactic directness of parallel languages such as Fortran90, HPF, and ZPL.

9. Acknowledgments

This work was supported in part by a grant from the Defense Advanced Research Projects Agency, monitored by the Naval Research Laboratory under contract number N00014-94-1-0742. The author wishes also to thank Eliot Moss, Kathryn McKinley, Steve Dropsho, Brendan

Cahoon, Glen weaver and John Cavazos for their helpful discussions and comments. Some of the ideas presented here are closely related to concepts behind the heterogeneous compiler, Scale, which we are presently building.

10. References

[Gordon1979] Michael Gordon, "The Denotational Description of Programming Languages", Springer-Verlag, New York, 1979.

[Gosling1996] James Gosling, Bill Joy, Guy Steele, "The Java Language Specification", Addison Wesley Pub. , Reading, MA, 1996.

[Lea1997] Doug Lea, "Concurrent Programming in Java", Addison Wesley Pub. , Reading, MA, 1997.

[Lewis1997] Ted Lewis, "If Java Is the Answer, What Was the Question?", IEEE Computer, Vol. 30, No. 3, March, 1997, pp. 133-136

[Lindholm1997] Tim Lindholm, Frank Yellin, "The Java Virtual Machine Specification", Addison Wesley Pub. , Reading, MA, 1997.

[Snyder1994] Lawrence Snyder, "A ZPL Programming Guide", Dept. of Computer Science and Engineering, Univ. of Washington, 1994.

[Weems1994] Charles Weems, Glen Weaver, Steve Dropsho, "Linguistic Support for Heterogeneous Parallel Processing: A Survey and an Approach", Proc. IEEE Heterogeneous Computing Workshop, April, 1994, Cancun, Mexico, IEEE Press, Los Alamitos, CA, pp. 81-88.

Biography

Charles C. Weems, B.S. 1977 (honors) and M.A. 1979, Oregon State University, Ph.D. 1984, University of Massachusetts at Amherst. Since 1984 he has directed the Specialized Parallel Architectures research group at the University of Massachusetts, where he is an Associate Professor. His research interests include parallel architectures for computer vision, benchmarks for vision, heterogeneous and adaptive parallel architectures, compilation for heterogeneous and adaptive systems, architectural issues for hard real-time systems, and parallel vision algorithms. He led the development of two generations of both the hardware and software for a heterogeneous parallel processor for machine vision, called the Image Understanding Architecture, under DARPA support. He is the author of numerous technical articles, has served on over a dozen program committees, is chairing the 1997 IEEE International Workshop on Computer Architecture for Machine Perception, the 1998 IEEE Symposium on the Frontiers of Massively Parallel Processing, and co-chairing the 1999 IEEE International Parallel Processing Symposium. He edited special issues of Machine Vision and Applications and IEEE Computer, serves on the board of ACSIOM Inc., is a member of the technical advisory committee to the board of Amerinex Applied Imaging, Inc., and is also the co-author of four widely used introductory computer science texts, and co-edited a book entitled Associative Processing and Processors. Dr. Weems is a member of ACM, a Senior Member of IEEE, a member of the Executive Committee of the IEEE TC on Parallel Processing, the IAPR TC on Special Purpose Architectures, and is an area editor for the Journal of Parallel and Distributed Computing and the SPIE/IEEE series on Imaging Science and Engineering.

Addendum

Scheduling Resources in Multi-User, Heterogeneous, Computing Environments with SmartNet

Richard F. Freund*
Michael Gherrity*
Stephen Ambrosius*
Mark Campbell†
Mike Halderman*
Debra Hensgen‡
Elaine Keith†
Taylor Kidd‡
Matt Kussow†
John D. Lima†
Francesca Mirabile*
Lantz Moore§
Brad Rust†
H. J. Siegel¶

Abstract

It is increasingly common for computer users to have access to several computers on a network, and hence to be able to execute many of their tasks on any of several computers. The choice of which computers execute which tasks is commonly determined by users based on a knowledge of computer speeds for each task and the current load on each computer. A number of task scheduling systems have been developed that balance the load of the computers on the network, but such systems tend to minimize the idle time of the computers rather than minimize the idle time of the users. This paper focuses on the benefits that can be achieved when the scheduling system considers both the computer availabilities and the performance of each task on each computer. The SmartNet resource scheduling system is described and compared to two different resource allocation strategies: load balancing and user directed assignment. Results are presented where the operation of hundreds of different networks of computers running thousands of different mixes of tasks are simulated in a batch environment. These results indicate that, for the computer environments simulated, SmartNet outperforms both load balancing and user directed assignments, based on the maximum time users must wait for their tasks to finish.

*NCCOSC RDT&E Division (NRaD)
†Science Applications International Corporation
‡Naval Postgraduate School
§University of Cincinnati
¶Purdue University

1 Introduction

1.1 Overview

The computational resources available to an individual user may range from a personal computer to workstations to a variety of high-performance computers, all connected through combinations of local and wide area networks. In this distributed computing environment, the jobs of a single user are often affected by the jobs of other users, computer unavailability, and network congestion. It is not unusual for a user's jobs to be significantly delayed because of other jobs saturating network routes or sharing computational resources. Although in some cases this may be unavoidable, in many cases a user's jobs can be executed on alternate computers on the network. A resource manager with a global perspective of the network resources might be able to get the user's jobs completed in less time by executing these jobs on such alternate computers.

SmartNet is a resource scheduling system for distributed computing environments. It allows users to execute jobs on complex networks of different computers as if they were a single machine, or **metacomputer**. A user need not be concerned with the activi-

0-8186-8365-1/98 $10.00 © 1998 IEEE

ties of other users, nor even whether various machines in the metacomputer are temporarily unavailable. To allow users to continue with interfaces with which they are familiar, SmartNet can also function as a scheduling advisor to existing resource management tools [24].

SmartNet schedules and can manage the execution of each user's jobs in coordination with the other jobs in the metacomputer in a attempt to maximize the performance of the metacomputer for all users. From this global perspective, the emphasis is not on maximizing the efficient use of the machines in the metacomputer, but rather on maximizing the efficiency of the users of the metacomputer. The SmartNet performance metrics are based on how well the users are served, rather than on how well each machine is used.

Section 2 describes the current capabilities of the SmartNet system. This includes the ability to obey data dependencies between jobs, to account for the effect of different inputs on job execution times, and to use a variety of scheduling algorithms. In Section 3, the performance of a metacomputer from a global perspective is measured by the maximum amount of time any user must wait for jobs to finish. Using this metric and some basic assumptions regarding the use of a metacomputer (specified in Section 3), it is shown that for a network of heterogeneous machines, a scheduling system that considers both the machine availabilities and the affinity of jobs to machines significantly outperforms a scheduling system that attempts to balance the load across machines in the network, and also outperforms a system where users select the fastest machine to execute each of their jobs. Section 4 describes the future direction of SmartNet development.

1.2 Implementing Superconcurrency

The term **superconcurrency** [14, 15, 36] has been used to describe a technique for selecting the optimal suite of machines in a metacomputer to execute a given set of jobs [17]. To fully exploit the capabilities of a metacomputer, a single job must be decomposed into tasks, such that each task has relatively homogeneous computational requirements. These computational requirements are defined by the different machine architectures available in the metacomputer. This decomposition allows full exploitation of the fact that different tasks execute at different speeds on the different machines in the metacomputer.

Job decomposition can occur at several levels. In many cases, large projects are decomposed by developers into programs that can be executed separately by a computer operating system. Often each program is written to take advantage of the special computational abilities of a given computer architecture. Such programs can share data by reading and writing files. In what follows, an executable program will be called a **task**, and a project consisting of one or more tasks with data dependencies will be called a **job**. This paper focuses on the benefits that can be achieved when the full heterogeneity among the tasks in the the jobs and among the machines in the metacomputer is considered when scheduling the execution of the jobs on the metacomputer.

For some tasks, a finer level of decomposition may be beneficial. In this case a task may be decomposed into subtasks, each with homogeneous computational requirements [33]. Considerable effort has been applied to perform this decomposition automatically, but many open problems remain [34]. Several language extensions for parallel computation have been developed that allow this decomposition to be done by programmers. Systems that use such language features include AHS [8], HeNCE/PVM [2, 35], Legion [20], Mentat [19], and P4 [5]. These systems allow programmers to decompose tasks into subtasks to fully utilize the heterogeneous processing capabilities of a metacomputer. Although this paper concentrates on scheduling tasks on a metacomputer, scheduling subtasks on a metacomputer involves many of the same issues.

1.3 Task Scheduling on a Metacomputer with Multiple Users

How tasks are assigned to the machines of a metacomputer is an important factor that affects the performance of the metacomputer. The assignment of a task to a machine on which it executes slowly can significantly reduce overall performance. Likewise, resource contention must be considered when scheduling tasks on a metacomputer with multiple users. For example, an optimal assignment of tasks to idle machines can easily become suboptimal if one of the machines is suddenly loaded with a task from another user. There are several important issues that a scheduler for a network of heterogeneous machines with multiple users should consider.

Heterogeneity: A number of systems have been developed for managing the execution of tasks on a network of machines. Examples include Condor [4], OSF DCE [29], and PBS [22]. Such systems schedule tasks in order to evenly balance the load on the machines in the metacomputer [27]. Many of these load balancing schemes are modified in an attempt to account for differences in the capabilities between machines. One common method is to adjust the load on a machine based on its speed on a single task relative to the other

machines. However, this does not account for the different affinities between tasks and machines that occur as a result of heterogeneity [15].

Figure 1 shows a simple example where the schedule that minimizes the completion time of all the tasks of a given set of jobs (the **schedule length** [6]) distributes the load unevenly among three machines, actually leaving one of the machines in the metacomputer idle. Specifically, executing any task on machine 2 will cause the schedule length to be at least 8 time units, whereas if machine 2 is left idle a schedule length of 6 is possible. This shows the importance of considering task and machine affinities in scheduling tasks on a metacomputer.

The importance of considering heterogeneity is further demonstrated in table (a) of Figure 1 where machine 1 is four times as fast as machine 2 on every task, but the speed of machine 3 relative to machine 1 varies depending on the task it is executing. This is an appropriate model if, for example, machine 3 is a vector machine and some of the tasks are not vectorizable. Tasks that are vectorizable would tend to execute more quickly on the vector machine.

Figure 1 also shows that the optimal schedule does not assign task A to its best machine (machine 3) but rather to its second best machine (machine 1). If the user of task A assigns this task to machine 3, its best machine, then the other tasks will not finish until at least time 8. This illustrates that simply allowing users to assign their tasks to the machines that execute them fastest may not provide all users with the best performance of the metacomputer. A scheduler that considers all the tasks of the metacomputer may improve its performance for most users.

Task Execution Times: To exploit the heterogeneity among the tasks in the jobs and among the machines in a metacomputer, it is beneficial to estimate the execution time of each task on each machine. Such estimates can usually be made empirically, although some deterministic models have been developed [1, 38]. Section 3 presents simulation results showing that variations in the estimate of task execution times has a secondary effect on scheduler performance compared to the appropriate consideration of heterogeneity.

Of course, different input data can dramatically affect the execution times of many tasks, and this effect can vary depending on the machine executing the task. It is assumed the effect of input data on the execution time of a task on a given machine is deterministic. For example, the execution time of a Monte Carlo simulation task on a single processor machine might tend to be linearly related to the number of iterations specified in its input data. A scheduler can benefit by having access to both the characteristics of the input data that affect a task's execution time and an equation for how the task execution time can be estimated from these characteristics when executed on a given machine.

Network Usage: To fully exploit the capabilities of a metacomputer, it is not only beneficial to estimate the execution times of the tasks, but also to estimate the network traffic that occurs as tasks communicate. If the tasks of one user are heavily loading a network route between two machines, it may be beneficial to schedule other tasks on machines where a different network route can be used.

Sections 2 and 4 describe SmartNet's current and future approaches, respectively, to address these issues.

1.4 Relationships with Other Work

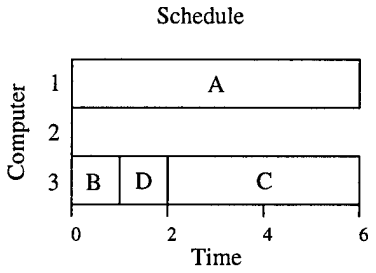
The importance of scheduling for the efficient use of distributed systems is well known, e.g., [10]. However, much of this research has been directed at scheduling the tasks of a single job on a network of processors, where there are no conflicting jobs or where the resource requirements of other jobs are unknown to the scheduler. In contrast, the SmartNet scheduler was designed to address multiple users competing for the resources of a metacomputer, and is most effective when the resource usage requirements of the jobs of each user can be estimated.

The importance of scheduling to achieve acceptable performance from a metacomputer with multiple users is also a major theme of the AppLeS [3] system. In this system, each task (called an application in AppLeS) would have its own AppLeS agent to select the metacomputer resources that will best meet that task's performance criteria. Because each agent schedules to maximize the performance of its associated task, the schedule that results would differ from the SmartNet schedule. As illustrated in Figure 1, SmartNet would attempt to minimize the longest execution time over all users. The emphasis in AppLeS is for each agent to optimize its own performance criteria rather than to cooperate with other agents to minimize a performance criteria shared by all the metacomputer users.

Task	Computer 1	2	3
A	6	24	5
B	2	8	1
C	8	32	4
D	2	8	1

Task Execution Times

(a)



(b)

Figure 1: An example where neither load balancing nor user directed assignments are the best scheduling strategies for a metacomputer. Table (a) gives the execution times of each task of a set of jobs on each machine. A Gantt chart of the schedule that minimizes the schedule length is shown in (b).

2 Current Implementation

2.1 Overview

The first version of SmartNet was operational in early 1994, and work has continued since then to increase its capabilities. The current version has proven to be useful in improving the performance of networks of heterogeneous machines, as will be shown in Section 3. Experience with this version has directed future efforts to enhance the system, and these will be described in Section 4.

Because batch systems such as Condor [4], OSF DCE [29], and PBS [22] perform many of the operations to manage the execution of scheduled tasks, this section will concentrate on the SmartNet scheduling capabilities not found in these systems. In fact, the SmartNet scheduling algorithms have been added to local versions of Condor [24] and Cray Research Inc.'s NQE.

2.2 How a Metacomputer and Tasks are Modeled in SmartNet

As described in Section 1.2, this paper will focus on scheduling executable tasks on a network of heterogeneous machines. The current version of Smartnet uses a **directed acyclic graph (DAG)** to specify data dependencies among the tasks of a job, and all the tasks of a single job communicate this data through files on a file server shared by all the tasks of the job. Different jobs may have different file servers. Communication among tasks can occur at the start or finish of a task. It is also assumed that tasks use a constant percentage of the processing and memory resources of the machine to which they are assigned, and that the executable program for each task is locally available to each machine which may execute the task.

It is assumed that the execution time of each task on each machine can be reasonably estimated. Task execution times are generally a function of a small number of input parameters, and better schedules result if both the values of these parameters and the appropriate time complexity function is available at the time the task is scheduled. For example, it may be possible to estimate the execution time of a task that contains a doubly nested loop by computing:

$$\text{execution time} = \alpha nm,$$

where α is a constant that may be found empirically, and n and m are the bounds on the loops. The user could provide this execution time formula to the SmartNet database for this task on the given machine, and the values of n and m could be given by the user when the task is submitted to the scheduler. SmartNet uses interpolation and extrapolation algorithms to extend rote learning of task execution times if either the parameters or the function are missing or only approximately correct.

The current version of SmartNet can account for latency and bandwidth between remote sites of the metacomputer, and also between the machines and a file server. Task priorities and data dependencies can be enforced by the scheduler. A background load on each machine and each network route can be considered when scheduling, however many complex network and processor contention issues are not considered in the current version. Although in many cases these network contention effects are negligible, there are problem domains where these effects are significant.

2.3 SmartNet Scheduling Algorithms

As shown in Figure 1, neither load balancing nor user directed assignment of resources may be as effective as a global scheduler for networks of heteroge-

neous machines. To most effectively schedule tasks on a metacomputer, the SmartNet scheduling heuristics all require estimates of the task execution times, and in some cases additional information about communication, memory, and processor usage. Estimates of task execution times are obtained through a combination of experiential information gathered automatically from trial runs, and, optionally, by time complexity equations (see Section 2.2) provided by the user.

In general, optimal multiprocessor scheduling is NP-complete [18], and hence SmartNet uses a variety of scheduling algorithms to attempt to obtain near-optimal schedules for different problems. The algorithms described below use a **deterministic execution simulation** [28] which performs a deterministic simulation of each task assuming the task execution times are equal to their estimated average values. A brief description of several of the SmartNet scheduling algorithms follows.

Maxmin and Minmin Algorithms:

If all the tasks to schedule are independent and compute intensive, then several of the algorithms described in [25] can be used. Specifically, algorithms D and E in [25] have been implemented and are called **maxmin** and **minmin** algorithms, respectively. Both have time complexities of $\mathcal{O}(mn^2)$, where m is the number of machines in the metacomputer and n is the number of tasks to schedule.

Both the maxmin and minmin algorithms consider a hypothetical assignment of tasks to machines, projecting when a machine will become idle based on the hypothetical assignment. Both algorithms determine, for each unassigned task, the earliest (minimum) time the task can be completed given the projected idle times of each machine and the estimated execution time of the task on each machine. The algorithms differ in their selection of which task to assign next given these minimum finish times. The maxmin algorithm selects the task that will take the maximum time to finish, whereas the minmin algorithm selects the task that could finish in the minimum time. Once selected, the task is assigned, the projected machine idle time is updated, and the task is removed from the set of unassigned tasks. The process is repeated until all tasks are assigned.

A schedule for 1000 tasks on a metacomputer of 100 machines can be determined in less than a minute using a typical workstation. Although [25] shows pathological problems where the resulting schedule length may be up to a factor of m worse than optimal, tests with environments such as those described in Section 3 have indicated that the schedule lengths are generally within 20% of optimal. Such tests were performed using simulations of small metacomputers and few tasks in order to perform an exhaustive search for an optimal schedule to be used for comparisons. In addition, for large metacomputers with many tasks, comparisons were possible against schedule lengths that were provably shorter than the optimal value [31].

Dependency Algorithms:

The following algorithms compute schedules when there are data dependencies between tasks. Although these tend to be the most frequently used SmartNet scheduling algorithms, a discussion of their effectiveness is outside the scope of this paper. A brief description is provided for completeness. For the experiments described in Section 3, there are no dependencies between tasks.

A Generational Algorithm:

This is a straightforward, cyclic method for mapping a set of dependent tasks onto available machines, that provides comparatively good schedules in a relatively short time [16]. During each cycle, a limited part of the scheduling problem is considered. Each task that has not satisfied all of its precedence constraints is considered ineligible for execution. All ineligible tasks are filtered out of the scheduling problem, forming a new smaller scheduling problem composed only of those tasks immediately eligible for execution. An auxiliary scheduling algorithm is then used to determine a schedule for the non-precedence-constrained problem. Upon detection of a rescheduling event, a new precedence-constrained scheduling problem is formed and the process repeats. One possible (indeed likely) rescheduling event is the completion of a previously scheduled task.

This generational algorithm is closely related to scheduling strategies such as Heaviest Node First scheduling [31] and Mapping Heuristic scheduling [9]. However this algorithm differs from these schedulers in that (1) all eligible tasks are rescheduled at each rescheduling event, and (2) the algorithm is designed to run dynamically as new task sets

are constantly added and completed.

A Clustering Algorithm:
A clustering algorithm [10] is provided for scheduling tasks with data dependencies that only use a portion of the available processing resources. For example, some tasks may consistently use only 50% of the CPU due to file I/O operations. In such cases, it is best to schedule several such tasks on a single machine (provided the machine has an operating system capable of multiprogramming [7], such as UNIX). In this case, the scheduler ensures that the memory requirements of the concurrent tasks are within the memory available on the machine, and that the concurrent tasks do not exceed the available network bandwidth.

Other Techniques:
A variety of experimental scheduling algorithms are also included in the current release of SmartNet. These include an algorithm using evolutionary programming [12], and another using a combination of genetic algorithms and simulated annealing [23, 32].

The added complexity of considering data dependencies in both the generational and clustering algorithms make these methods several times slower than the maxmin and minmin algorithms.

3 Simulation Results

3.1 Overview

It is difficult to precisely evaluate the performance of a scheduling algorithm for a metacomputer because this is dependent on many factors, such as the distribution of task execution times and characteristics of the metacomputer hardware. Tests using a specific metacomputer are not conclusive because the results are only valid for that metacomputer. Simulation studies provide the ability to demonstrate the effectiveness of a scheduling algorithm over a broad range of conditions, although there is no generally accepted set of benchmarks.

Although a large number of different metacomputers can be tested using simulation studies, there is an issue with the accuracy of the simulation. This section presents simulation studies where all tasks to be run are known initially (batch processing), tasks are independent, and network use is not significant, hence the need to accurately model specific features of a metacomputer for accurate simulation results are minimized. In addition, in this case, an assumption of single-programming (i.e., each machine executes a single task at a time) is appropriate to maximize the performance of the metacomputer.

Two simulation studies will be presented where a large number of randomly generated problems are scheduled. Section 3.2 describes simulations that were performed with results from the NAS parallel benchmarks [30] to model a metacomputer used in a typical production environment. Section 3.3 presents simulations that model a typical academic environment. Section 3.4 evaluates the effect of variations in the estimated task completion times for both the production and academic environments. As described in Section 2, there are SmartNet schedulers that consider communications between tasks, but the evaluation of these algorithms is outside the scope of this paper. The focus here is on demonstrating the benefits of considering heterogeneity in task scheduling. Additional information about SmartNet can be found in [13].

3.2 Simulating a Typical Production Environment

In the production environment modeled here, it is assumed that tasks tend to have long execution times. This would be typical of, say, batch jobs run overnight using a sizable metacomputer. To model this case, random sets of task execution times and machine speeds were generated using the NAS parallel benchmarks [30] as a template.

Specifically, ten machines were arbitrarily selected from the NAS database and the execution times of the eight NAS benchmarks on these machines (using the class A data size) were used. Table 1 shows the selected machines, and Table 2 shows the corresponding job execution times on each machine given in [30]. Notice the significant heterogeneity both in machine speeds for the same job (across rows) and job execution times for the same machine (across columns) shown in Table 2. No single machine is fastest on all the jobs, and the ratio of execution times on different machines is very much dependent on the job being executed.

Each test problem modeled a network of 20 machines with 100 tasks. The problems were randomly generated from Tables 1 and 2 as follows. Each of the 20 machines was selected by randomly picking one of the ten machines listed in Table 1 using a uniform distribution with replacement. Similarly, each of the 100 tasks was selected to correspond to one of the eight NAS jobs shown in Table 2. A complete 100×20 matrix of execution times was then created using the corresponding times in Table 2.

For these tests the maxmin algorithm was used by

index	Machine	Processors
1	HP/Convex Exemplar SPP2000	16
2	CRAY C90	16
3	CRAY T3E	256
4	CRAY Y-MP	8
5	DEC Alpha Server 8400 5/440 (437 MHz)	12
6	Fujitsu VPP700	32
7	IBM RS/6000 SP Thin-node2 (67 MHz)	128
8	Intel Paragon MP (SunMos turbo)	512
9	Kendell Square KSR2	64
10	SGI Origin2000 (195 MHz)	32

Table 1: The arbitrarily selected machines from the NAS database used to generate random test cases.

	Machine									
job	1	2	3	4	5	6	7	8	9	10
EP	10.5	2.36	0.4	15.87	8.46	1.03	1.31	0.87	13.0	3.55
MG	4.3	0.71	0.2	2.96	NA	0.25	0.63	1.36	5.7	2.12
CG	2.7	0.34	0.4	2.38	NA	0.67	1.48	NA	6.1	2.04
FT	NA	0.80	0.2	4.19	NA	0.33	1.30	1.92	6.5	3.16
IS	2.21	0.27	0.2	1.85	NA	0.98	0.61	2.29	3.9	1.21
LU	NA	10.17	4.2	49.5	67.97	10.06	15.9	NA	102.0	18.9
SP	56.6	12.82	6.0	64.6	91.43	4.53	20.6	NA	131.0	30.5
BT	55.3	20.3	6.2	114.0	103.5	4.93	20.8	113.0	130.0	30.0

Table 2: The execution times of each NAS job on the selected machines.

SmartNet, and both a load balancing algorithm and a user directed assignment algorithm were used for comparisons. The load balancing algorithm schedules each task on the machine that becomes idle first. Thus, load balancing considers machine availability, but ignores heterogeneity issues. The user directed assignment algorithm schedules each task on the machine that executes it fastest. Thus, user directed assignment considers heterogeneity, but ignores machine availability issues. The SmartNet maxmin algorithm, described in Section 2.3, considers both heterogeneity and machine availability.

Figure 2 shows the distribution of the ratio of the schedule length from the load balancing algorithm over the SmartNet schedule length for 1000 test problems. On average, the SmartNet schedule length is 36 times shorter than the load balance schedule length. Figure 3 shows the distribution of the ratio of the schedule length from the user directed assignment algorithm over the schedule length from SmartNet for 1000 test problems. On average, SmartNet performs two times better than the algorithm simulating user assignment.

3.3 Simulating a Typical Academic Environment

It has been shown [21] that the execution times of tasks are distributed exponentially in typical academic environments. This is quite different than the distribution of task execution times used in Section 3.2, which modeled a typical production environment using the NAS benchmarks. A similar series of tests as described in Section 3.2 were performed where this exponential distribution was used rather than the distribution based on the NAS benchmarks. Again, a batch submission of tasks is assumed.

Specifically, task execution times were randomly selected using an exponential distribution with a minimum of 10 and a mean of 1000. The results of 1000 samples from this distribution is shown in Figure 4. To provide some heterogeneity in the network, it was assumed that all machines had identical architectures but some were faster than others. Unlike the NAS case described in Section 3.2, the faster machines run all tasks faster than slower machines. The relative speeds of each of the 20 machines on the network were determined by randomly drawing from an exponential distribution with a minimum of 1 and a mean of 5.

It was found that the minmin algorithm was superior to the maxmin algorithm for this environment, and hence SmartNet was run using the minmin algorithm to schedule the tasks. Figure 5 shows the distribution of the ratio of the schedule length from the load balancing algorithm over the SmartNet schedule length for 1000 test problems. On average, the SmartNet schedule length is 2.2 times shorter than the load balance schedule length for this environment. Figure 6 shows the distribution of the ratio of the schedule length from the user directed assignment algorithm over the schedule length from SmartNet for 1000 test problems. On average, the SmartNet schedule length is 2.8 times shorter than the user assignment schedule length for this environment.

3.4 The Effects of Variations in Actual Task Execution Times on SmartNet Scheduling

The test results described in Sections 3.2 and 3.3 assume that the execution times of all the tasks on each machine were known prior to actually executing these tasks. The technique of load balancing is dynamic in that such information is not needed by the algorithm. When a task finally finishes, the machine on which it was running becomes idle and the next task in the queue can be started on that machine. In contrast, all the SmartNet schedulers assign all tasks to machines prior to the execution of any of these tasks.

To determine the sensitivity of the SmartNet schedulers to inaccurate estimates of task execution times, the tests described above were repeated with random noise added to the estimated task execution times. Specifically, after the SmartNet scheduler had determined the assignment of tasks to machines, the actual execution time of each task was determined by drawing a random time from a normal distribution with a mean of the estimated task execution time and a standard deviation a percent of the estimated task execution time. The load balancing algorithm used this actual task execution time rather than the estimated time used by the SmartNet scheduler.

Figure 7 shows a gradual increase in the load balance schedule length that results from adding random noise in this fashion to the task execution times. Although the average task execution time is not changed by adding this noise, the maximum execution time does increase. Because the load balancing scheduler occasionally assigns a task to the machine with one of these worst case times, and these bad assignments tend to dominate the schedule length, the schedule length can be seen in Figure 7 to increase as the noise level increases.

Figures 8 and 9 compare this dynamic load balancing schedule length with the SmartNet schedule length. Because the SmartNet schedulers consider heterogeneity, they are seen to be no more sensitive than a dynamic load balancing scheduler to this type of noise

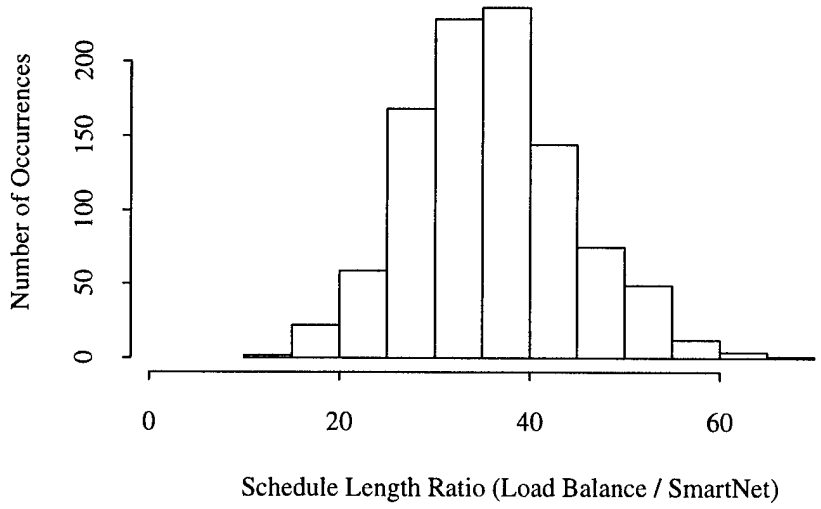


Figure 2: A comparison of the performance of a SmartNet scheduler to a load balancing scheduler using 1000 random problems modeling a typical production environment.

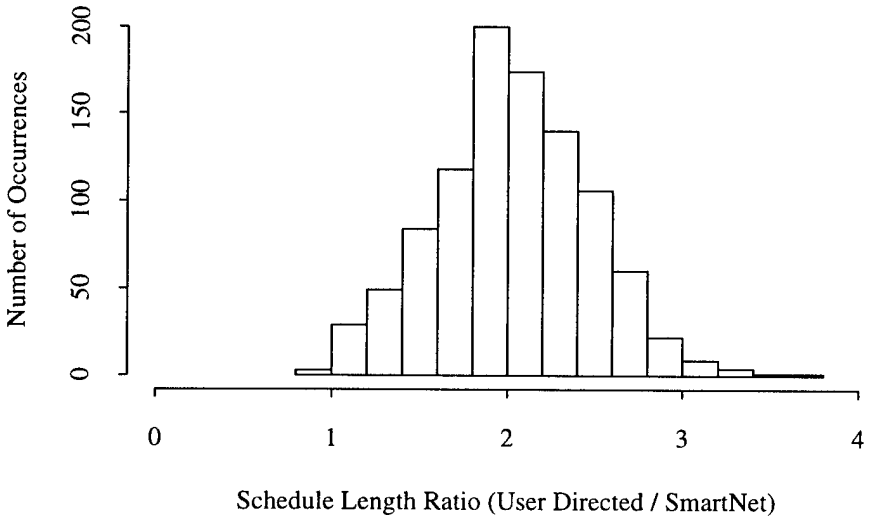


Figure 3: A comparison of the performance of a SmartNet scheduler to a scheduler simulating user directed assignment in a typical production environment.

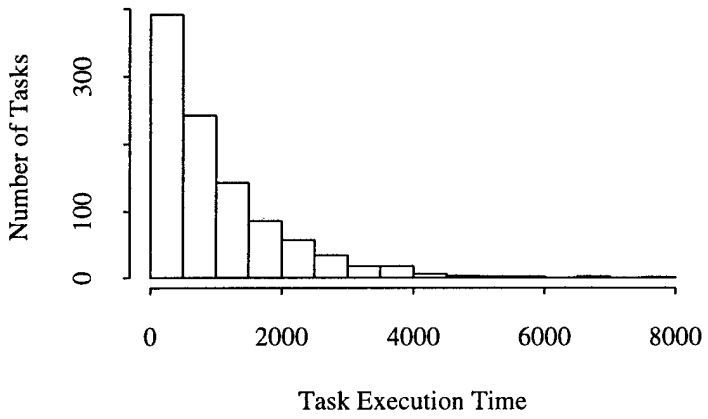


Figure 4: The distribution of task execution times used to simulate a typical academic environment. The results of 1000 random samples are shown.

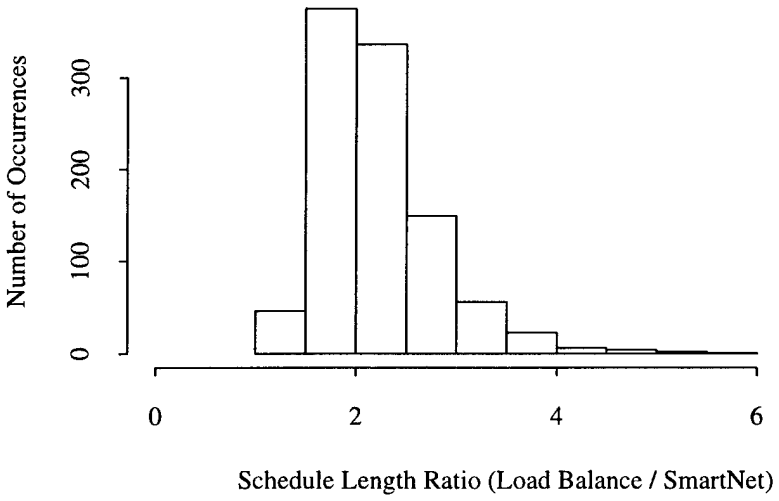


Figure 5: A comparison of the performance of a SmartNet scheduler to a load balancing scheduler using 1000 random problems modeling a typical academic environment with little heterogeneity.

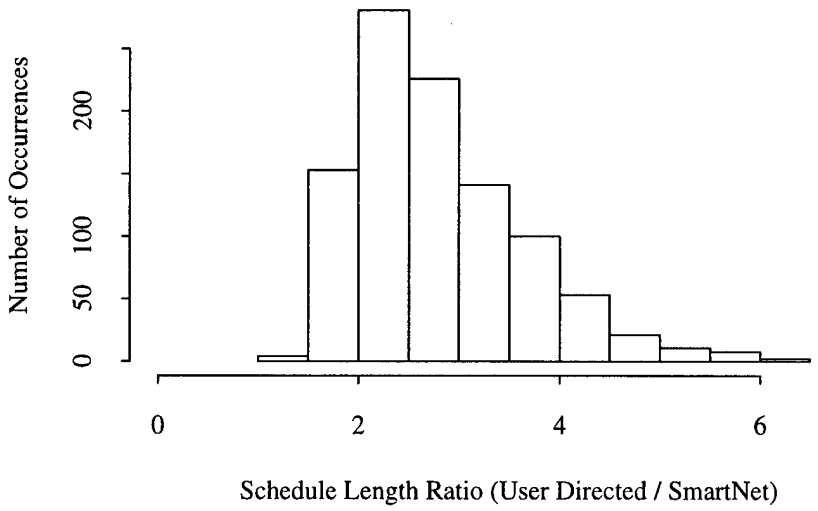


Figure 6: A comparison of the performance of a SmartNet scheduler to a scheduler simulating user directed assignment using 1000 random problems modeling a typical academic environment with little heterogeneity.

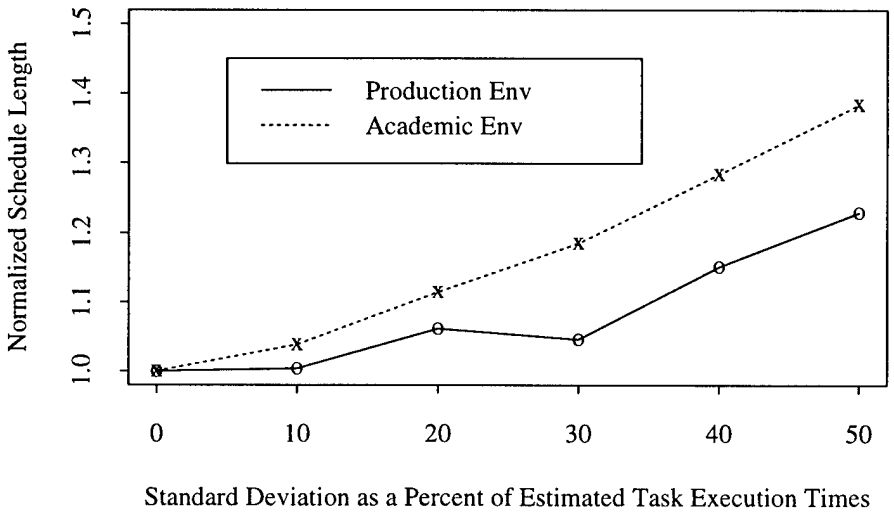


Figure 7: The performance of a dynamic load balancing scheduler when random noise is added to the task execution times. Each data point represents 100 random problems.

in the estimated task execution times. In fact, Figure 8 shows that the SmartNet scheduler is actually slightly better than the dynamic load balancing scheduler as the noise level is increased in this production environment.

4 The Future of Metacomputer Scheduling

Although an optimal solution to the task scheduling problem for a metacomputer is intractable, there are a number of polynomial time heuristic algorithms that provide solutions that are significantly better than merely balancing the load on all machines, or specifying a fixed assignment of tasks to machines. Section 3 presented several demonstrations of such an improvement in the simple case of tasks with no data dependencies or communication delays. Experiments using SmartNet to schedule problems where data dependencies and communication delays are included show that the improvements in the schedule length similar to those shown in Sections 3.2 and 3.4 is achieved, due to the benefit of scheduling tasks on the machines that run them best. Additional studies using SmartNet for scheduling tasks with data dependencies are documented in [26].

The SmartNet system has been operational since 1994 and is in its eighth release. Each release has significantly improved its ability to accurately model real-world computing environments. The development team has a Software Engineering Institute (SEI) level 3 rating, indicating the maturity and stability of the software. The system has been used for a variety of DARPA sponsored projects, as well as at the NIH, and is being tested for use at NASA. Section 2 has briefly outlined the current capabilities of the system. Future releases are planned to better account for networks and multiprogrammed machines. Specifically:

Network Routing:
A metacomputer can include numerous machines on several networks. Future versions of SmartNet are planned to better account for communication delays between tasks with data dependencies when tasks are scheduled on machines that require this communication to occur over several networks. Scheduling considering the contention effects of complex network routing has been a secondary concern due to increasing network speeds and the compute intensive nature of the applications that have been studied.

Resource Contention:
In many cases, it is appropriate to assume that a task controls a percentage of a resource throughout its entire execution time. Provided this portion of the resource is available, the task execution time is not affected by other tasks using other portions of the resource. This is the model used in the current SmartNet release. However it is also common that resources are shared among tasks, and such sharing causes the task execution times to be extended. This effect has been called **interference** in [37] and **slowdown** in [11]. Future releases of SmartNet will consider the effects of resource sharing on the execution time of tasks.

Optimization Criteria:
As described in Section 1, the SmartNet schedulers all attempt to minimize the schedule length. In some cases, notably for highly interactive environments, minimizing the average task finish time may be a more appropriate optimization criteria. It is planned that future releases of SmartNet will include schedulers that minimize this average task finish time (called **flow time** in [6]).

5 Conclusions

It has been shown in several cases that scheduling tasks considering both the machine availabilities and the heterogeneity of the machines in a metacomputer can increase the metacomputer performance. The objective of such scheduling is to minimize the total time users must wait for their tasks to finish.

Many scheduling approaches have been based on load balancing, which minimizes the idle time of the machines on the network rather than the idle time of the users of the network. In networks of homogeneous machines, the idle time of users can be minimized by minimizing the idle time of the machines. This is not the case in networks of heterogeneous machines. The schedulers used in the SmartNet system account for the heterogeneity of both the network machines and the user tasks. There are many factors that may create heterogeneity in what would appear to be a network of homogeneous machines, such as memory size differences between machines, network differences, or even differences in background loads.

The benefits of a global, centralized scheduler such as SmartNet diminish as the number of machines in the metacomputer grows large, due both to the time delays of the scheduling process itself, and to the associated increase in network traffic to and from the centralized scheduler. However, the simulation results described in Section 3 demonstrate conditions where a global scheduler like SmartNet is beneficial. For a metacomputer that covers the machines accessible to

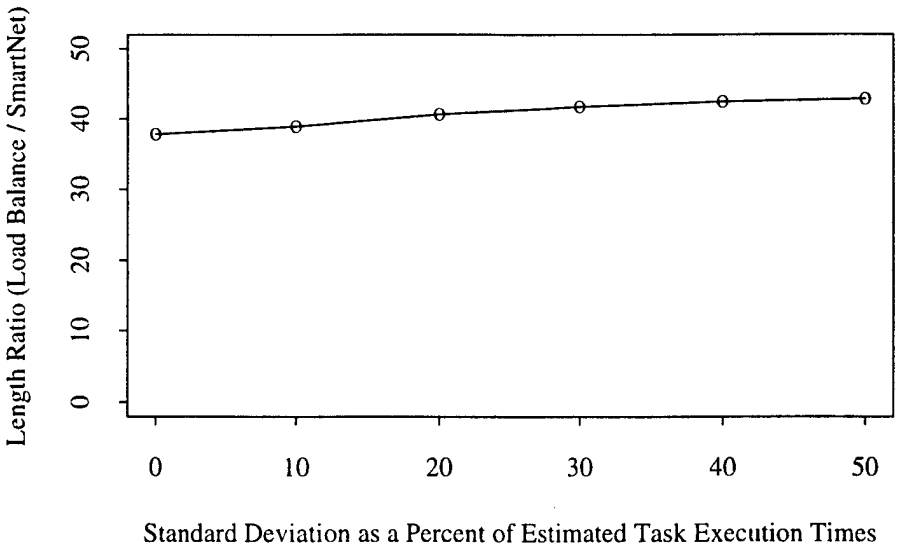


Figure 8: A comparison of the performance of a SmartNet scheduler to a load balancing scheduler when the estimated task execution times are not accurate. Each data point represents 100 random problems based on the NAS benchmarks.

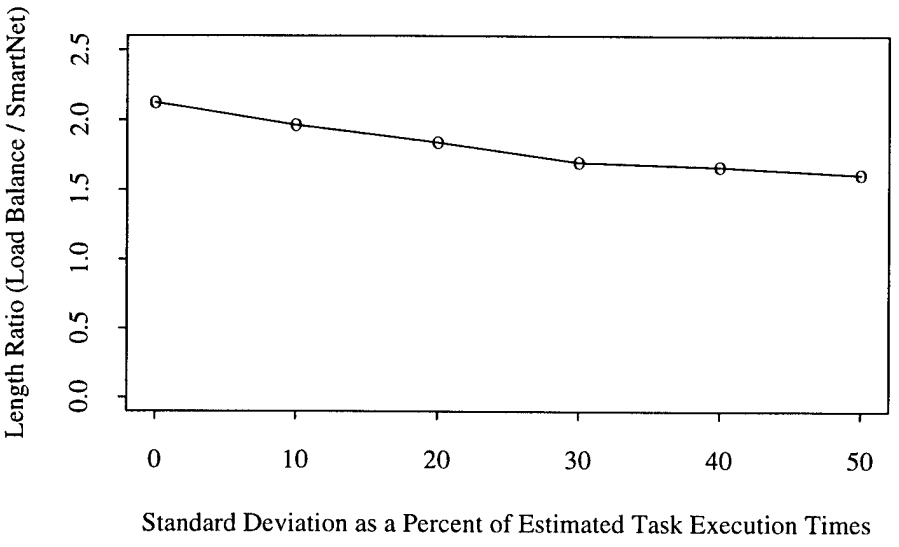


Figure 9: A comparison of the performance of a SmartNet scheduler to a load balancing scheduler when the estimated task execution times are not accurate. Each data point represents 100 random problems modeling a typical academic environment with little heterogeneity.

a typical department at a university, or a corporation, such global control is manageable. However, problems with scale clearly arise as the size of the metacomputer grows large [20]. Other tests, not described in this paper, indicate that the performance benefits of a global scheduler are still possible with metacomputers of several hundred machines when the task execution times are sufficiently long. For larger metacomputers, a hierarchy of SmartNet schedulers is currently being investigated.

Unlike many of the systems mentioned in Section 1, SmartNet does not constrain the user to a particular programming language, nor does it require the construction of special wrapper code for legacy programs. For best results, users need only provide a description of the time complexity of their tasks, and there are many tools that can help provide this information. By coordinating the execution time of user tasks, considering both machine availability and heterogeneity, the performance of a metacomputer may be substantially improved.

Acknowledgments

Portions of this work were supported by the NRaD Independent Research program, NASA, and DARPA. The authors thank the many people who have contributed to the SmartNet project, including Bill Adsit, Thomas Bayless, Michael Godfrey, Mitch Gregory, Joan Hammond, Roberta Hilton, Terry Koyama, Wanda Lam, Mary Lewis, Kathy Nolan, Sue Patterson, Bruce Rickard, Rod Roberts, Dave Schwarze, Dan Watson, Marc Weissman, and Bob Wellington.

References

[1] V. S. Adve. *Analyzing the Behavior and Performance of Parallel Systems.* PhD thesis, University of Wisconsin-Madison, December 1993.

[2] A. Beguelin, J. Dongarra, A. Geist, and R. Manchek. HeNCE: A heterogeneous network computing environment. *Scientific Programming*, 3(1):49–60, 1994.

[3] F. Berman and R. Wolski. Scheduling from the perspective of the application. In *The Fifth IEEE International Symposium on High Performance Distributed Computing*, August 1996.

[4] A. Bricker, M. Litzkow, and M. Livny. *Condor Technical Summary.* University of Wisconsin, Madison, version 4.1b edition, January 1992.

[5] R. M. Butler and E. L. Lusk. Monitors, messages, and clusters: the p4 parallel programming system. Technical report, Mathematics and Computer Science division, Argonne National Laboratory, Argonne, Illinois, 1992.

[6] E. G. Coffman, Jr. Introduction to deterministic scheduling theory. In E. G. Coffman, Jr., editor, *Computer and Job-Shop Scheduling Theory*, chapter 1, pages 1–50. John Wiley & Sons, New York, 1976.

[7] H. M. Deitel. *An Introduction to Operating Systems.* Addison-Wesley Publishing Co., Reading, Massachusetts, 1984.

[8] H. G. Dietz, W. E. Cohen, and B. K. Grant. Would you run it here... or there? (AHS: Automatic heterogeneous supercomputing). In *The 1993 International Conference on Parallel Processing*, volume II, pages 217–222, Saint Charles, Illinois, August 1993.

[9] H. El-Rewini and T. G. Lewis. Scheduling parallel program tasks onto arbitrary target machines. *Journal of Parallel and Distributed Computing*, 9(2):138–153, 1990.

[10] H. El-Rewini, T. G. Lewis, and H. H. Ali. *Task Scheduling in Parallel and Distributed Systems.* PTR Prentice Hall, Englewood Cliffs, New Jersey, 1994.

[11] S. M. Figueira and F. Berman. Modeling the effects of contention on the performance of heterogeneous applications. In *The High Performance Distributed Computing Conference*, 1996.

[12] D. B. Fogel. *Evolutionary Computation: Toward a New Philosophy of Machine Intelligence.* IEEE Press, New York, 1995.

[13] R. Freund, T. Kidd, D. Hensgen, and L. Moore. Smartnet: A scheduling framework for heterogeneous computing. In *The International Symposium on Parallel Architectures, Algorithms, and Networks*, Beijing, China, June 1996. IEEE Computer Society Press.

[14] R. F. Freund. Optimal selection theory for superconcurrency. In *Supercomputing '89*, pages 699–703, New York, NY, November 1989. ACM Press.

[15] R. F. Freund. SuperC or distributed heterogeneous HPC. *Computing Systems in Engineering*, 2(4):349–355, 1991.

[16] R. F. Freund, B. R. Carter, D. W. Watson, E. Keith, and F. Mirabile. Generational scheduling for heterogeneous computing systems. In *International Conference on Parallel and Distributed Processing Techniques and Applications*, pages 769–778, August 1996.

[17] R. F. Freund and H. J. Siegel. Heterogeneous processing. *Computer*, 26(6):13–17, June 1993.

[18] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman and Company, New York, 1979.

[19] A. S. Grimshaw, J. B. Weissman, and W. T. Strayer. Portable run-time support for dynamic object-oriented parallel processing. *ACM Transactions on Computer Systems*, 14(2):139–170, May 1996.

[20] A. S. Grimshaw, Wm. A. Wulf, and the Legion team. The Legion vision of a worldwide virtual computer. *Communications of the ACM*, 40(1), January 1997.

[21] M. Harchol-Bulter and A. B. Downey. Exploiting process lifetime distributions for dynamic load balancing. Technical Report UCB/CSD-95-887, University of California, Berkeley, November 1995.

[22] R. L. Henderson and D. Tweten. *PBS: Portable Batch System requirements specification.* NASA Ames Research Center, April 1995.

[23] R. A. Henry, N. S. Flann, and D. W. Watson. A massively parallel SIMD algorithm for combinatorial optimization. In *The 1996 International Conference on Parallel Processing, vol. II*, pages 46–49, August 1996.

[24] D. A. Hensgen, L. Moore, T. Kidd, R. Freund, E. Keith, M. Kussow, J. Lima, and M. Campbell. Adding rescheduling to and integrating Condor with Smartnet. In *The 4th Heterogeneous Computing Workshop*, pages 4–12, April 1995.

[25] O. H. Ibarra and C. E. Kim. Heuristic algorithms for scheduling independent tasks on nonidentical processors. *Journal of the Association for Computing Machinery*, 24(2):280–289, April 1977.

[26] M. Janakiraman. Simulation results for heuristic algorithms for scheduling precedence-related tasks in heterogeneous environments. Master's thesis, University of Cincinnati, 1996.

[27] J. A. Kaplan and M. L. Nelson. A comparison of queueing, cluster and distributed computing systems. Technical Report NASA TM 109025, NASA Langley Research Center, June 1994.

[28] D. A. Menascé, D. Saha, S. C. Da Silva Porto, V. A. F. Almeida, and S. K. Tripathi. Static and dynamic processor scheduling disciplines in heterogeneous parallel architectures. *Journal of Parallel and Distributed Computing*, 28(1):1–18, 1995.

[29] Open Software Foundation, 11 Cambridge Center, Cambridge, Massachusetts. *Distributed Computing Environment: An overview*, January 1992.

[30] S. Saini and D. H. Bailey. NAS parallel benchmark (version 1.0) results 11-96. Technical Report NAS-96-18, NASA Ames Research Center, November 1996.

[31] B. Shirazi, M. Wang, and G. Pathak. Analysis and evaluation of heuristic methods for static task scheduling. *Journal of Parallel and Distributed Computing*, 10(3):222–232, November 1990.

[32] P. Shroff, D. W. Watson, N. S. Flann, and R. F. Freund. Genetic simulated annealing for scheduling data-dependent tasks in heterogeneous environments. In *The 4th Heterogeneous Computing Workshop*, pages 98–104, April 1995.

[33] H. J. Siegel, J. K. Antonio, R. C. Metzger, M. Tan, and Y. A. Li. Heterogeneous computing. In A. Y. Zomaya, editor, *Parallel and Distributed Computing Handbook*, chapter 25, pages 725–761. McGraw-Hill, New York, NY, 1996.

[34] H. J. Siegel, H. G. Dietz, and J. K. Antonio. Software support for heterogeneous computing. In A. B. Tucker, Jr., editor, *The Computer Science and Engineering Handbook.* CRC Press, Boca Raton, FL, 1997.

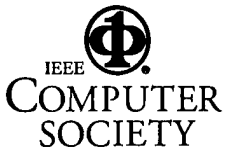
[35] V. S. Sunderam. PVM: A framework for parallel distributed computing. *Concurrency: Practice and Experience*, 2(4):315–339, December 1990.

[36] M-C. Wang, S-D. Kim, M. A. Nichols, R. F. Freund, H. J. Siegel, and W. G. Nation. Augmenting the optimal selection theory for superconcurrency. In *Workshop on Heterogeneous Processing*, pages 13–22, Los Alamitos, California, March 1992. IEEE Computer Society Press.

[37] J. Weissman. The interference paradigm for network job scheduling. In *The 5th Heterogeneous Computing Workshop*, pages 38–45. IEEE Computer Society Press, April 1996.

[38] Y. Yan, X. Zhang, and Y. Song. An effective and practical performance prediction model for parallel computing on non-dedicated heterogeneous NOW. *Journal of Parallel and Distributed Computing*, 38(1):63–80, October 1996.

Author Index



Press Activities Board

IEEE Computer Society Publications

The world-renowned Computer Society publishes, promotes, and distributes a wide variety of authoritative computer science and engineering texts. These books are available in two formats: 100 percent original material by authors preeminent in their field who focus on relevant topics and cutting-edge research, and reprint collections consisting of carefully selected groups of previously published papers with accompanying original introductory and explanatory text.

Submission of proposals: For guidelines and information on Computer Society books, send e-mail to cs.books@computer.org or write to the Acquisitions Editor, 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.

IEEE Computer Society Proceedings

The Computer Society also produces and actively promotes the proceedings of more than 130 acclaimed international conferences each year in multimedia formats that include hard and softcover books, CD-ROMs, videos, and on-line publications.

For information on 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

4/15/97