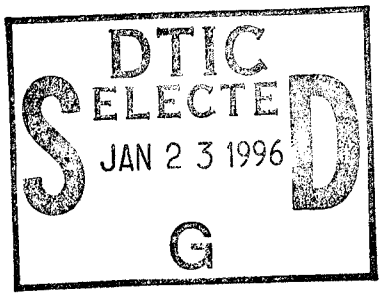
RL-TR-95-132
Final Technical Report
July 1995

METHODOLOGIES FOR MAPPING TASKS ONTO HETEROGENEOUS PROCESSING SYSTEMS

Purdue University

H.J. Siegel and John K. Antonio



APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

Rome Laboratory
Air Force Materiel Command
Griffiss Air Force Base, New York

This report has been reviewed by the Rome Laboratory Public Affairs Office (PA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

RL-TR-95-132 has been reviewed and is approved for publication.

APPROVED: Richard C. Metzger

RICHARD C. METZGER
Project Engineer

FOR THE COMMANDER: John A. Graniero

JOHN A. GRANIERO
Chief Scientist
Command, Control & Communications Directorate

If your address has changed or if you wish to be removed from the Rome Laboratory mailing list, or if the addressee is no longer employed by your organization, please notify RL (C3CB) Griffiss AFB NY 13441. This will assist us in maintaining a current mailing list.

Do not return copies of this report unless contractual obligations or notices on a specific document require that it be returned.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave Blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	July 1995	Final Jan 94 - Jan 95

4. TITLE AND SUBTITLE
METHODOLOGIES FOR MAPPING TASKS ONTO HETEROGENEOUS PROCESSING SYSTEMS

5. FUNDING NUMBERS
C - F30602-94-C-0022
PE - 62702F
PR - 5581
TA - 18
WU - PF

6. AUTHOR(S)
H.J. Siegel and John K. Antonio

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Purdue University
School of Electrical Engineering
1285 Electrical Engineering Building
West Lafayette IN 49707-1285

8. PERFORMING ORGANIZATION REPORT NUMBER
N/A

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Rome Laboratory (C3CB)
525 Brooks Rd
Griffiss AFB NY 13441-4505

10. SPONSORING/MONITORING AGENCY REPORT NUMBER
RL-TR-95-132

11. SUPPLEMENTARY NOTES
Rome Laboratory Project Engineer: Richard C. Metzger/C3CB/(315) 330-7650

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

Complete application tasks, of the type that would be of interest to Rome Laboratory, are large and complex. One approach to dealing with them is heterogeneous computing. Two types of heterogeneous computing systems are: (1) mixed-mode, wherein multiple types of parallelism are available on a single machine; and (2) mixed-machine, wherein a suite of different high-performance computers is connected by high-speed links. In this effort, we studied ways to decompose an application into subtasks and then match each subtask to the mode or machine, which results in the smallest total task execution time. Our accomplishments include: (1) conducting a mixed-mode case study; (2) developing an approach for automatically decomposing a task for mixed-mode execution, and assigning modes to subtasks; (3) extending this approach for use as an heuristic for a particular class of mixed-machine heterogeneous computing systems; (4) surveying the state-of-the-art of heterogeneous computing, and constructing a conceptual framework for automatic mixed-machine heterogeneous computing; (5) examining how to estimate non-deterministic execution of subtasks and complete tasks; and (6) devising an optimal scheme for inter-machine data transfers for a given matching of subtasks to machines.

14. SUBJECT TERMS	15. NUMBER OF PAGES
Heterogeneous computing systems, Tasks, Subtasks, Static allocation, Mixed-mode, Mixed-machine	198
	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	UL

Table of Contents

List of Figures vi

1. Introduction 1

2. Parallel Algorithms for Singular Value Decomposition 4

2.1. Introduction 4

2.2. Background Information 5

2.3. Data Mapping 7

2.3.1. Overview 7

2.3.2. Mappings Being Analyzed 8

2.4. Performance Analysis 10

2.4.1. Analysis Overview 10

2.4.2. Operation Counts 11

2.4.3. Relation of Number of PEs to Operation Count 12

2.4.4. Performance Prediction 13

2.4.5. Implementation Comparison 14

2.5. Performance Evaluation 15

2.5.1. Experimental Algorithm Performance 15

2.5.2. Modes of Parallelism 17

2.6. Conclusions 20

3. A Block-Based Mode Selection Model for SIMD/SPMD Parallel Environments 22

3.1. Introduction 22

3.2. Machine and Language Model 26

3.3. Parallel Performance Issues 27

3.4. Single-Mode Selection 30

3.4.1. Overview of Single-Mode Selection 30

3.4.2. Assumptions and Definitions 30
3.4.3. Description of the Single-Mode Selection Technique 32
3.4.4. A Probabilistic Model for Data-Dependent Operation Execution Times 38
3.4.5. Effects of Block Juxtaposition 41
3.4.6. Summary of Single-Mode Model 42
3.5. Mixed-Mode and Mixed-Machine Analysis 42
3.5.1. Assumptions and Definitions 42
3.5.2. Optimal Selection of Modes for Mixed Algorithms 44
3.5.3. Computational Aspects of Optimal Selection of Modes for Mixed Algorithms 48
3.5.4. An Example of Optimal Mixed-Algorithm Selection 52
3.6. Summary and Future Work 54
4. Static Program Decomposition Among Machines in an SIMD/SPMD Heterogeneous Environment with Non-Constant Mode Switching Costs 56
4.1. Introduction 56
4.2. Machine and Language Model 57
4.3. The BBMS Framework 58
4.3.1. Framework Overview 58
4.3.2. With Constant Switching Costs 60
4.3.3. With Non-Constant Switching Costs 63
4.4. Extension of BBMS 64
4.4.1. A Simplified Parallel Program Behavior Model 64
4.4.2. Heuristic Based on BBMS 66
4.4.3. Simulation Study 67
4.5. Summary 71
5. Heterogeneous Computing 72

5.1. Overview 72
5.2. Mixed-Mode Machines 75
5.2.1. Introduction 75
5.2.2. PASM 80
5.2.3. TRAC 80
5.2.4. OPSILA 81
5.2.5. Triton 82
5.2.6. EXECUBE Chip 83
5.2.7. Conclusions 83
5.3. Examples of Uses of Existing HC Systems 84
5.3.1. Simulation of Mixing in Turbulent Convection at the Minnesota Supercomputer Center 84
5.3.2. Interactive Rendering of Multiple Earth Science Data Sets on the CASA Testbed 85
5.3.3. Using VISTAnet to Compute Radiation Treatment Planning for Cancer Patients 87
5.4. Examples of Existing Software Tools and Environments 88
5.4.1. Overview 88
5.4.2. Linda 89
5.4.3. p4 89
5.4.4. Mentat 91
5.4.5. PVM, Xab, and HeNCE 93
5.5. A Conceptual Model for Heterogeneous Computing 96
5.6. Task Profiling and Analytical Benchmarking 98
5.6.1. Overview 98
5.6.2. Definitions of Task Profiling and Analytical Benchmarking 100

5.6.3. Methodologies for Performing Task Profiling and Analytical Benchmarking 100
5.6.4. A Mathematical Formulation for Task Profiling and Analytical Benchmarking 107
5.6.5. Summary 108
5.7. Matching and Scheduling for HC Systems 109
5.7.1. Overview 109
5.7.2. Characterizing Matching and Scheduling for HC Systems 109
5.7.3. Examples of Techniques and Formulations for Matching and Scheduling for HC Systems 111
5.7.4. Summary 120
5.8. Conclusions and Future Directions 121
6. Estimating the Distribution of Execution Times for SIMD/SPMD Mixed-Mode Programs 123
6.1. Introduction 123
6.2. Overview of the Approach 126
6.3. Basic Probability Theory 129
6.4. Execution Time Distribution of a Code Block 131
6.5. Single-Mode Execution of a Program 133
6.5.1. Overview of Single-Mode Execution 133
6.5.2. A Series of Blocks 133
6.5.3. Pure Loop 134
6.5.4. Pure Data Conditional Construct 136
6.5.5. Arbitrary Program Construct 137
6.6. Mixed-Mode Execution of a Program 137
6.6.1. Overview of Mixed-Mode Execution 137

6.6.2. A Series of Mixed-Mode Blocks 138
6.6.3. Pure Loop 139
6.6.4. Arbitrary Program Construct 140
6.7. Numerical Example 142
6.8. Summary and Future Work 144
7. Scheduling and Data Relocation for Sequentially Executed Subtasks in a Heterogeneous Computing System 145
7.1. Introduction 145
7.2. A Mathematical Model for Matching, Scheduling, and Data Relocation in HC 148
7.3. Impact of Data-Reuse Without Considering Multiple Data-Copies 151
7.4. Impact of Multiple Data-Copies and Temporally Interleaved Execution of Atomic Input Operations for Different Subtasks (TIE) 153
7.5. A Minimum Spanning Tree Based Algorithm for Finding the Optimal Set of Data-Source Functions and the Corresponding Set of Ordering Functions 161
7.5.1. Description of the Algorithm 161
7.5.2. Proof of Correctness of the Algorithm 166
7.6. Two-Stage Approach for Matching, Scheduling, and Data Relocation in HC 166
7.7. Summary 167
8. Publication List 169
9. References 171

List of Figures

Figure 2.1: High-level algorithm for finding SVD using Givens rotations.
Figure 2.2: Column transfer model for 2CPP mapping.
Figure 2.3: Column transfer model for 1CPP mapping.
Figure 2.4: Comparison of PP_{2CPP} and PP_{1CPP}; CR = 0.119.
Figure 2.5: Experimental performance of 2CPP vs. 1CPP.
Figure 2.6: 2CPP algorithm execution time comparison" "for different modes of parallelism; 4×4 matrix.
Figure 3.1: Execution of variable-time instructions in SIMD and MIMD mode.
Figure 3.2: Example preliminary flow-analysis tree.
Figure 3.3: Final flow-analysis tree after modification to include overhead for `for` and `if` constructs.
Figure 3.4: SIMD/SPMD trade-off tree for the flow-analysis tree in Figure 3.3.
Figure 3.5: Overlapped execution of CU and PE instructions.
Figure 3.6: Modification of `for` loop to avoid unnecessary mode switches for the first iteration.
Figure 3.7: Example of a general multistage optimization problem and the aggregate structure of the intermediate and final solution graphs.
Figure 3.8: Transformation from flow-analysis tree to multistage graph.
Figure 3.9: Mixed-algorithm analysis example for a parallel code segment.
Figure 3.10: Selection of parallel modes of the example of Figure 3.9.
Figure 4.1: Example of transformation from program to flow-analysis tree to trade-off tree..**IE**
Figure 4.2: Example of a general multistage optimization problem and its solution.
Figure 4.3: Transformation from flow-analysis tree to multistage graph.
Figure 4.4: Simplified model of parallel program behavior.
Figure 4.5: Heuristic building on the multistage technique.
Figure 4.6: Effectiveness of machine selection policies for data/execution ratio of 0.1.
Figure 4.7: Effectiveness of machine selection policies for data/execution ratio of 10.
Figure 4.8: Effectiveness of machine selection policies for data/execution ratio of 1000.
Figure 5.1: A hypothetical example of the advantage of using heterogeneous computing [Fre91], where the execution time for the heterogeneous suite includes inter-machine communications. Percentages are based on 100% being the total

execution time on the baseline serial system, but are not drawn to scale.

Figure 5.2: (a) Distributed memory SIMD machine model. (b) Distributed memory MIMD machine model.

Figure 5.3: "Sum of max's" versus "max of sums" effects.

Figure 5.4: Bitonic sequence-sorting algorithm [FiC91].

Figure 5.5: A task tree (instruction tree and data tree) of TRAC 1.1 [AlG89].

Figure 5.6: Conceptual model of the assignment of subtasks to machines in an HC environment

Figure 5.7: Example program segment and its associated flow-analysis tree [WaS94].

Figure 5.8: Transformation from flow-analysis tree to multistage optimization graph [WaS94].

Figure 5.9: Simplified model of parallel program behavior with an arbitrary choice of machine for each code block [WaA94].

Figure 5.10: Heuristic building on the multistage technique [WaA94].

Figure 6.1: Execution of a loop with variable execution time subtasks

Figure 6.2: Example program and its associated flow analysis tree

Figure 6.3: An example of mixed-mode tree reduction

Figure 6.4: Mixed-mode execution for the numerical example

Figure 7.1: Data-distribution situations in HC.

Figure 7.2: Network of four machines with the initial data element d_0 on $M[0]$.

Figure 7.3: Network of four machines with the initial data on $M[0]$ and $M[3]$.

Figure 7.4: The generation of the vertices for the atomic operations of **IS**

Figure 7.5: Subtask flow graph for the example application program.

Figure 7.6: Generating a spanning tree with respect to the set of data-source functions associated with the subtask flow graph.

Figure 7.7: Generating a minimum spanning tree for the example application program and its corresponding valid data-source functions.

1. Introduction

This is the final report for the project "Methodologies for Mapping Tasks onto Heterogeneous Processing Systems," supported by Rome Laboratory under contract number F30602-94-C-0022. The Co-Principal Investigators for this project were H. J. Siegel and John K. Antonio. The project period was from January 27, 1994 to January 26, 1995. Richard C. Metzger was the Rome Laboratory project manager.

The work done on this project focused on heterogeneous computing, both mixed-mode and mixed-machine. Section 2 describes a mixed-mode case study involving singular value decomposition. Based on information learned from this and other case studies, a method for selecting the best mode of parallelism to use for each block of a mixed-mode program was devised, and is presented in Section 3. Section 4 builds on the results of Section 3 to construct a heuristic for assigning subtasks to machines in a mixed-machine environment. A complete model of automatic heterogeneous computing is developed in Section 5 (the research in Section 4 represents one approach to one piece of the model). An evaluation and survey of the state-of-the-art of heterogeneous computing is also given in Section 5. Section 6 examines the estimation of non-deterministic execution times for use in enhancing the technique in Section 3. The heuristic of Section 4 is extended in Section 7 by considering optimal data relocations for a given matching of subtasks to machines. The publications that were supported by this effort are listed in Section 8. The rest of this section provides more details about Sections 2 through 7 of this report.

In motion rate control applications, it is faster and easier to solve the equations involved if the singular value decomposition (SVD) of the Jacobian matrix is first determined. A parallel SVD algorithm with minimum execution time is desired. One approach using Givens rotations lends itself to parallelization, reduces the iterative nature of the algorithm, and efficiently handles rectangular matrices. The research in Section 2 focuses on the minimization of the SVD execution time when using this approach. Specific issues addressed in this section include considerations of data mapping, effects of the number of processors used on execution time, impacts of the interconnection network on performance, and trade-offs among the SIMD, MIMD, and mixed-mode implementations. Results are verified by experimental data collected on the PASM mixed-mode parallel machine prototype.

One of the challenges for mixed-mode computing is, given a mode-independent parallel language, to generate executable code for a variety of computational models, and to identify those specific parallel modes for which portions of a program are well-suited. One part of this problem, developing a method for estimating the relative execution time of a data-parallel

algorithm in an environment capable of the SIMD and SPMD (MIMD) modes of parallelism, is presented in Section 3. Given a data-parallel program in a language whose syntax is mode-independent and empirical information about instruction execution time characteristics, the goal is to use static source-code analysis to determine an implementation that results in an optimal execution time for a mixed-mode machine capable of SIMD and SPMD parallelism. Statistical information about individual operation execution times and paths of execution through a parallel program is assumed. A secondary goal of this study is to indicate language, algorithm, and machine characteristics that must be researched to learn how to provide the information needed to obtain an optimal assignment of parallel modes to program segments.

The problem of minimizing the execution time of programs within a mixed-machine heterogeneous environment is considered in Section 4. Different computational characteristics within a parallel algorithm may make switching execution from one machine to another beneficial; however, the cost of switching between machines during the execution of a program must be considered. This cost is not constant, but depends on data transfers needed as a result of the move. Therefore, determining a minimum-cost assignment of machines to program segments is not straightforward. The block-based mode selection (BBMS) approach, presented in Section 3 for use in a single mixed-mode machine, is used as a basis to develop a polynomial time heuristic for assigning machines in a mixed-machine suite to program segments of data-parallel algorithms. Simulation results of parallel program behaviors using the heuristic indicate that good assignments are possible without resorting to exhaustive search techniques.

Section 5 is an overview of heterogeneous computing. A heterogeneous computing system provides a variety of architectural capabilities, orchestrated to perform an application whose subtasks have diverse execution requirements. To exploit such systems, a task must be decomposed into subtasks, where each subtask is computationally homogeneous. The subtasks are then assigned to and executed with the machines (or modes) that will result in a minimal overall execution time for the task. Currently, users typically specify this decomposition and assignment. One long-term pursuit in the field of heterogeneous computing is to do this automatically. This section presents a model for automatic heterogeneous computing. It also surveys and examines the state-of-the art in heterogeneous computing.

In Section 6, a methodology is introduced for estimating the distribution of execution times for a given data parallel program that is to be executed in an SIMD/SPMD mixed-mode heterogeneous computing environment. The program is assumed to contain operations and constructs whose execution times depend on input-data values. The methodology uses the block-based approach of Section 3 to transform the program into a flow analysis tree. It then computes the distribution of execution times for the program, given the execution modes for each node in the

flow analysis tree, an estimated execution time distribution for each operation in both modes, and an appropriate probabilistic model for each control and data conditional construct. A numerical example is given to illustrate the utility of the proposed methodology.

A mathematical model is presented in Section 7 for three of the factors that affect the execution time of an application program in a heterogeneous computing system: matching, scheduling, and data relocation schemes. As stated earlier, in a heterogeneous computing environment, an application program is decomposed into subtasks, then each computationally homogeneous subtask is assigned to the machine where it is best suited for execution. It is assumed in this section that, at any instant in time during the execution of a specific application program, only one machine is being used for program execution and only one subtask is being executed. Two data relocation situations are identified, namely data-reuse and multiple data-copies. It is proved that without considering multiple data-copies, but allowing data-reuse, the execution time of given application program depends only on the matching scheme. A polynomial algorithm, which involves finding a minimum spanning tree, is introduced to determine the optimal scheduling scheme and the optimal data relocation scheme with respect to an arbitrary matching scheme when data-reuse and multiple data-copies are considered. Finally, a two-stage approach for matching, scheduling, and data relocation in heterogeneous computing is presented. The work extends and enhances the matching heuristic developed in Section 4.

2. Parallel Algorithms for Singular Value Decomposition

2.1. Introduction

Decreasing the execution time of computerized tasks is the focus of a tremendous amount of study. The use of parallel computer systems is one method to help decrease these times. The performance of a parallel system, however, is dependent on the algorithm implementation and the parallel machine characteristics. Performance optimization is therefore complicated, due to the wide variety of algorithm characteristics [Jam87] and the rapidly growing variety of parallel machines that have been built or proposed. Thus, the study of mapping algorithms onto parallel machines is an important research area.

The singular value decomposition (SVD) of matrices has been extensively used in control applications, e.g., during the computational analysis of robotic manipulators [KlH83, Yos85]. The decomposition aids the computational solution of system equations such as the motion rate control formula $\dot{x} = J\dot{\theta}$, where $\underline{\dot{x}} \in \mathbf{R}^M$ specifies the end effector velocity, $\underline{\dot{\theta}} \in \mathbf{R}^N$ specifies joint velocities, and $\underline{J} \in \mathbf{R}^{M \times N}$ is the Jacobian matrix [Whi69]. For systems with many cooperating manipulators, the value of N can reach into the hundreds, resulting in a severe computational burden for achieving real-time control.

In general, computation of the SVD of an arbitrary matrix is an iterative procedure, so the number of operations required to calculate it to within acceptable error limits is not known beforehand. The control of many systems, however, is based on equations involving the current Jacobian matrix, which can be regarded as a perturbation of the previous matrix, i.e., $J(t+\Delta t) = J(t) + \Delta J(t)$. It has been demonstrated that for these cases knowledge of the previous state can be used during the computation of the current SVD to decrease execution time [MaK89]. This section describes and analyzes two SVD algorithm implementations for these cases. Experimental data obtained on the PASM prototype parallel computer [ArW93, SiS87] is provided that supports the conclusions of the algorithm analyses.

Subsection 2.2 provides background information about SVD, Givens rotations, and PASM. Descriptions of the two parallel SVD implementations being analyzed are presented in Subsection 2.3. Subsection 2.4 demonstrates an analysis approach to determine which

---

The co-authors of this section were Renard Ulrey, Anthony A. Maciejewski, and Howard Jay Siegel.

This research was supported by Sandia National Laboratories under contract 18-4379B, and by Rome Laboratory under contract F30602-94-C-0022. This research used equipment supported by the National Science Foundation under grant CDA-9015696.

This material appeared in the *Proceedings of the 8th International Parallel Processing Symposium*, sponsored by the IEEE Computer Society, April 1994, pp. 524-533.

implementation has the shorter execution time. The performances of SVD implementations on PASM are evaluated in Subsection 2.5.

2.2. Background Information

The SVD of a matrix $J \in R^{M \times N}$ is defined as the matrix factorization $J = UDV^T$, where $\underline{U} \in R^{M \times M}$ and $\underline{V} \in R^{N \times N}$ are orthogonal matrices of the singular vectors, and $\underline{D} \in R^{M \times N}$ is a nonnegative diagonal matrix. The singular values of J, $\underline{\sigma_i}$, are ordered from largest to smallest along the diagonal of D. It is assumed here that $M \leq N$.

The Golub-Reinsch algorithm [GoV83] is the standard technique for determining the SVD of a matrix. This method, however, has two unattractive aspects. The first is that the algorithm, as it is defined, cannot use knowledge of a previous matrix decomposition. The second is that the technique is relatively serial in nature, making more parallelizable algorithms desirable.

Several parallel SVD algorithms have been implemented for various machine architectures, including those proposed in [BrL85, ChC89, Luk80, Luk86, ScL86]. These implementations also do not allow their iterative natures to be reduced. Algorithms being studied in this section are based on a methodology presented in [MaK89], which exclusively uses Givens rotations [GoV83] to orthogonalize matrix columns.

Successive Givens rotations are used to generate the orthogonal matrix V that will result in $JV = B$, where the columns of $\underline{B} \in R^{M \times N}$ are orthogonal. A matrix with orthogonal columns can be written as the product of an orthogonal matrix U and a diagonal matrix D (i.e., $B = UD$) by letting the columns of U, u_i, equal normalized columns of B, b_i,

$$u_i = b_i / \|b_i\| \quad (\text{where } \|b_i\| = \sqrt{b_i^T b_i}), \tag{1}$$

and defining the diagonal elements of D to be equal to the norm of the columns of B

$$\sigma_i = \|b_i\|. \tag{2}$$

This results in the SVD of J.

The orthogonal matrix V that will orthogonalize the columns of J is formed as a product of Givens rotations, each of which orthogonalizes two columns. Considering the i-th and k-th columns of an arbitrary matrix A, a single Givens rotation results in new columns, a_i' and a_k', given by

$$a_i' = a_i \cos(\phi) + a_k \sin(\phi) \tag{3}$$

$$a_k' = a_k \cos(\phi) - a_i \sin(\phi). \tag{4}$$

The $\cos(\phi)$ and $\sin(\phi)$ terms necessary to achieve orthogonality are computed using the formulas in [Nas79], which are based on the quantities

$$p = a_i^T a_k,\ q = a_i^T a_i - a_k^T a_k,\ \text{and}\ c = \sqrt{4p^2 + q^2}\ . \tag{5}$$

Using these quantities, when $q \geq 0$

$$\cos(\phi) = \sqrt{(c+q)/(2c)}\ \text{ and }\ \sin(\phi) = p/(c \cdot \cos(\phi))\,. \tag{6}$$

When $q < 0$,

$$\sin(\phi) = \text{sgn}(p) \cdot \sqrt{(c-q)/(2c)}\ \text{ and} \tag{7}$$
$$\cos(\phi) = p/(c \cdot \sin(\phi))\,,$$

where sgn(p) equals 1 if $p \geq 0$ and -1 if $p < 0$. Two sets of formulas are given so that ill-conditioned equations resulting from the subtraction of nearly equal numbers can always be avoided.

To orthogonalize each possible pair of columns requires $N(N-1)/2$ rotations, referred to as a sweep [GoL83]. The matrix V can be computed by iteratively forming the product of a set of sweeps and testing for convergence. While the number of sweeps required to orthogonalize the columns of J is not generally known beforehand, it was shown in [MaK89] that by using the V matrix from the SVD of the previous J to find an initial estimate for B,

$$B(t+\Delta t) \simeq J(t+\Delta t) \times V(t)\,, \tag{8}$$

one can obtain a good approximation to the new SVD using a single sweep if $\Delta J(t)$ is small. Therefore, in this work the current V matrix is calculated using $V(t+\Delta t) = V(t) \times \prod_{i=1}^{N-1} \prod_{k=i+1}^{N} G_{ik}$, where G_{ik} denotes the Givens rotation to orthogonalize columns i and k. Only a single sweep is performed to update the matrix V.

The PASM (partitionable SIMD/MIMD) parallel processing system [ArW93, SiS87] was used to implement these algorithms. PASM, designed at Purdue University, supports mixed-mode parallelism, i.e., it can operate in either SIMD or MIMD mode of parallelism, and can switch modes at instruction level granularity with generally negligible overhead. A small-scale 30-processor PASM prototype has been built with 16 PEs (processor/memory pairs) in the computational engine. For inter-PE communications, PASM uses a partitionable circuit-switched multistage cube interconnection network [Sie90], also called an Omega [Law75]. The network can be used in both SIMD and MIMD modes.

PASM is capable of employing barrier synchronization [DiS89] in MIMD mode, called Barrier MIMD (BMIMD). Each PE executes its code independently until it arrives at a synchronization point called a barrier. Then, each PE waits at the barrier until all PEs indicate they have reached it. One use for this is to synchronize inter-PE transfers performed in MIMD mode.

2.3. Data Mapping

2.3.1. Overview

Based on the equations in Subsection 2.2, Figure 2.1 gives an algorithm to calculate V, D, and U using Givens rotations. This algorithm assumes that the SVD of the Jacobian matrix from the previous control sample period has been computed. Thus, for step 1, the previous V matrix is available on the system. It is assumed that the algorithm then converges with a single sweep of rotations in step 2.

step 1:
calculate initial estimate for B from J and previous V, using (8)
step 2:
for all column pairs (i,k) do /* one sweep */
  calculate p, q, and c using (5)
  calculate cos(ϕ) and sin(ϕ) using (6) or (7)
  perform rotation on columns i and k of B, similar to (3) and (4)
  perform rotation on columns i and k of V, similar to (3) and (4)
end for
step 3:
calculate D from B, using (2)
calculate U from B and D, using (1)

Figure 2.1: High-level algorithm for finding SVD using Givens rotations.

Referring to the parallel execution of a Givens rotation by all PEs as a rotation step, N–1 rotation steps must be performed on N/2 column pairs to form all N(N–1)/2 column pairs. With unique column pairs distributed among N/2 PEs, inter-PE communication is avoided within each rotation step. After the initial rotation step, however, an inter-PE communication is required before each remaining rotation step. This rotate-transfer-rotate sequence is required both to form all column pairs and to converge the B and V matrices to their single-sweep values. Newly updated columns are being transferred in each communication step.

As presented in Subsection 2.2, the calculations involved in this algorithm are straightforward. Of greater interest are ways to effectively map matrix elements to particular parallel machines, and the types of inter-PE communication these mappings dictate. Various implementations of column transfer operations have been devised, including those in [BrL85, ChC89, ScL86]. Each of these methods map a unique column pair to each of N/2 PEs. The availability of a multistage cube network on PASM allows matrix data to be distributed across more PEs than allowed by implementations in [BrL85, ChC89, ScL86], and thus increases the number of PEs that can perform useful work while still performing all necessary inter-PE

communications in single transfer steps.

Two different methods for mapping matrices to PEs are presented. These implementations assume that $M = 2^m \leq N = 2^n$ for the Jacobian matrix $J \in R^{M \times N}$. If the matrix does not have these dimensions, it can always be padded with zeroes. This subsection explains the data layout and communication patterns for each method. The algorithmic details for each are in Subsection 2.4.

2.3.2. Mappings Being Analyzed

A two columns per PE (2CPP) data mapping is the first to be considered. Assume that N/2 PEs are used, numbered from 0 to (N/2)−1. Let S be the number of PEs in a communicating subgroup (S a power of two), and let i be the address of a PE that is transferring a column through the network. The 2CPP method uses the interconnection function $\underline{Shift}_j(i,S) = S\lfloor i/S \rfloor + ((i+j) \bmod S)$, where j is the Shift length, to determine the address of the destination PE. This function allows the destination PE address to remain within a current communicating subgroup of size S. The resulting communication patterns are conflict-free transfers on a multistage cube network [Law75, Sie90].

Let each PE contain columns x and y. All possible column pairs are formed by iteratively performing a two-step process. To begin, $S = N/2$ so that all PEs being used are in a communicating subgroup. In the first step, all y columns are shifted to all other PEs in the subgroup by applying the function $Shift_1(i,S)$ a total of S−1 times. In the second step, the subgroup is split in two by exchanging x and y columns between subgroup halves using $Shift_{S/2}(i,S)$ and reducing S by a factor of two for the next iteration.

A model of the inter-rotation transfers for this algorithm is shown for N = 8 in Figure 2.2. Each row of blocks in the figure represents a rotation step where calculations are being performed in each of 8/2 PEs. Number pairs in the blocks denote the columns being updated/rotated in each PE. Arrows illustrate the inter-rotation column transfer steps. Beside each transfer step, the communication function used to achieve the interconnection pattern is specified, as well as the columns being exchanged (x is the left column number in each box, y is the right).

A second implementation was developed that allows all three algorithm steps to be implemented with a one column per PE (1CPP) distribution. Using the capabilities of the multistage cube network, a communication pattern was derived that forms all possible column pairs using N−1 column transfers. Assume N PEs are used, numbered 0 to N−1. PE i always contains the most recent version of column i. In the k-th rotation step, $1 \leq k < N$, PE i exchanges data with PE $i \oplus k$, where $\oplus$ is the bit-wise exclusive-or. These transfers are conflict-free on the

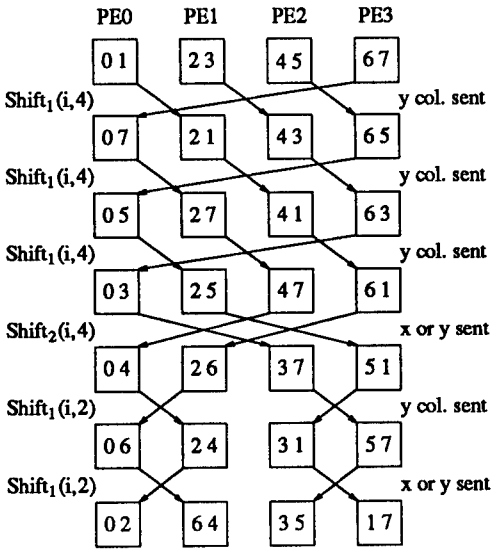


Figure 2.2: Column transfer model for 2CPP mapping.

multistage cube [Bat77].

Operations in step 2 require data from both columns of the pair being rotated. Therefore, the 1CPP mapping requires column transfers within each rotation step (intra-rotation transfers) rather than between rotation steps (inter-rotation transfers). A performance trade-off is immediately apparent with respect to the 2CPP method. Steps 1 and 3 can execute with half as many operations using the 1CPP mapping, but step 2 requires one additional column transfer to complete a full sweep. Later subsections compare both the expected performance and observed performance between the two methods.

A model of the intra-rotation transfers for this implementation is shown in Figure 2.3 for $N = 8$. Having the columns sequentially ordered after the decomposition may be an advantage for post-SVD operations.

With the 2CPP and 1CPP column distribution models now formed, it is a goal of this study to further utilize parallelism in the SVD algorithm to possibly decrease execution times. The approach taken divides each column of the B and V matrices into $\underline{R} = 2^r$ segments. The total number of PEs that are used increases by a factor of R. For this study, $R \leq M \leq N$.

In the 2CPP mapping, because RN/2 total PEs are being used, r+n−1 PE address bits can be used to fully define the column segment distribution. To map column segments onto PASM, PEs whose addresses agree in the n−1 most significant bits contain different segments of the same column, and PEs with the same r least significant bits have corresponding segments of different columns. Similarly, for the 1CPP mapping, r+n address bits define the column segment distribution among the RN PEs. PEs with the same n most significant bits contain segments of

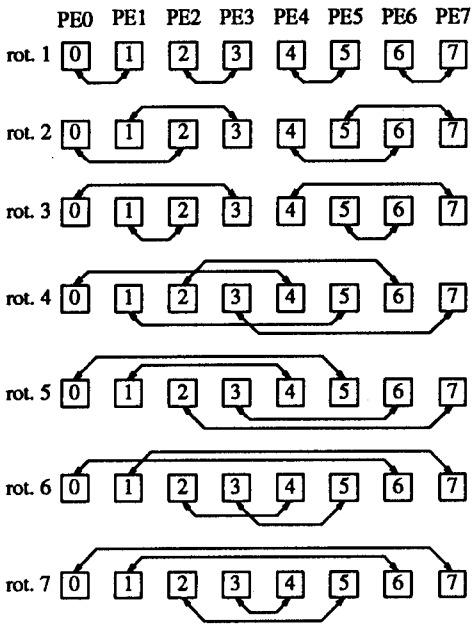


Figure 2.3: Column transfer model for 1CPP mapping.

the same column, and PEs with the same r least significant bits have the same segment number.

These segment mappings allow the system network to still perform the column transfer communications as explained for both the 1CPP and 2CPP methods. These communications will occur between PEs that have the same segment number, i.e., agree in a given set of address bits. All PEs can also perform simultaneous communications between PEs containing different segments of the same column as a conflict-free transfer. The addresses of these PEs will agree in a different set of bits. This is due to the partitioning properties of the multistage cube [Sie90].

Communication patterns between PEs that have different segments of the same column have not been discussed. The patterns that provide the fastest algorithm execution were found to be dependent on both the column mapping (1CPP or 2CPP) and the current operation being performed. The communication patterns providing the best performance are detailed in [UlM95].

2.4. Performance Analysis

2.4.1. Analysis Overview

There are three goals of this analysis. The first is to demonstrate some considerations when examining algorithm performance. The second is to see whether a speedup of the SVD

algorithm can always be expected when more PEs are used (this is not always the case, e.g., [SaS93]). The third is to determine the conditions when one of the 1CPP and 2CPP implementations performs better.

An operation count analysis for the SVD implementations is the first step toward predicting the better algorithm mapping. The two main components of the SVD algorithm are considered to be computation and inter-PE communication. The number of floating-point operations (FLOPs) performed by each PE will be used as the measure of the amount of computation for this analysis.

In general, the time to perform FLOPs on a machine will depend on the operation to be performed, and possibly on the operands. For this analysis, it is assumed that all FLOPs and their associated address calculations take the same constant amount of time. It was shown in [SaS93] that using an experimentally-derived average time as the execution time of each FLOP can provide good results.

The time it takes to set up a valid network configuration in SIMD mode on the PASM prototype is close to that to perform a floating-point (FP) data transfer. For this reason, and because the inter-PE transfers performed throughout the SVD algorithm involve different numbers of data items, the time spent performing communications in this analysis is represented by the total number of single data transfers (DTs) performed by each PE. Experimental results presented in Subsection 2.5 will show that this is a good approximation.

2.4.2. Operation Counts

Various methods were derived to perform each of the three steps of the SVD algorithm with both the 2CPP and 1CPP approaches. A comparison of the operation counts for the different methods is detailed in [UlM95]. The methods with the smallest complexities were implemented.

Because of the distribution of columns and column segments among PEs, many operations require the combining of the partial sums of calculations performed by single PEs. In most cases, some variation of recursive doubling (described in [SiA92]) allowed execution with the fewest FP and DT operations. Other methods were also found to reduce the number of operations. The similarity of (6) and (7) is exploited so that both $\cos(\phi)$ and $\sin(\phi)$ are calculated using only 6 non-data-dependent FLOPs, regardless of the mode of parallelism being used. For the 1CPP approach, a method was developed for both SIMD and MIMD modes to perform column rotations on all PEs simultaneously, where half of the PEs rotate their own columns according to (3) and the other half according to (4). Again, the similarity of the equations is

exploited. These methods are detailed in [UlM95].

The complete complexity equations for both approaches are shown in Table 2.1. Because of the method chosen to perform the 2CPP approach, its total operation count has a special case when $R = 1$. For comparison purposes, the total number of FLOPs needed to perform the entire SVD algorithm using a single processor is also shown in the table.

2.4.3. Relation of Number of PEs to Operation Count

To find the number of PEs to use so that the fewest number of operations are performed, the derivative with respect to R of the equations shown in Table 2.1 are found and set to zero. Doing this for the FLOP count of the 2CPP implementation and rearranging the resulting equation reveals that the minimum number of FLOPs are performed when $R = (2N + (16NM - 8M - 2N)/(3N - 2))$. Similarly, for the 1CPP implementation, $R = (1.5N + (9NM - 5M - 1.5N)/(2N - 1))$. Recall that as R increases the number of PEs increases, and that for this study it is assumed $1 \le R \le M \le N$. Because $R > M$ in these two equations, the number of FLOPs used in each implementation continues to decrease as R (and the number of PEs) increases, up to the maximum allowed when $R = M$.

Floating-point Operation Count		
implementation	total	condition
2CPP (RN/2 PEs)	$(1/R)(6N^2-6N+16NM-8M) + r(3N-2)+(9N-10)$	$1<R\le M$
	$6N^2+3N+16NM-8M-9$	$R=1$
1CPP (RN PEs)	$(1/R)(3N^2-3N+9NM-5M) + r(2N-1)+(10N-10)$	$1\le R\le M$
one PE	$3N^3+(3/2)N^2+8N^2M -5NM-(9/2)N+M^2$	
Network Data Transfer Count		
implementation	total	condition
2CPP (RN/2 PEs)	$(1/R)(N^2-2N+NM-4M) + r(3N-2)+(N+2M-1)$	$1<R\le M$
	$N^2-N+NM-2M-2$	$R=1$
1CPP (RN PEs)	$(1/R)(N^2-N+NM-2M) + r(2N-1)+(N+M-1)$	$1\le R\le M$

Table 2.1: SVD algorithm operation count totals.

Setting the derivative of the DT count of the 2CPP approach to zero results in the mathematically optimal value of $R = ((N^2 - 2N + NM - 4M)/(3N - 2))$. In this equation, R may be less than M, depending on the values of N and M. Setting the derivative of the DT count of the 1CPP approach to zero results in the mathematically optimal value of $R = ((N^2 - N + NM - 2M)/(2N - 1))$. Again, R may be less than M, depending on the values of N

and M. An examination of this equation, however, provides interesting results. Letting $M = N$, the equation reduces to $R = N-(2N/(2N-1))$, so the optimal value of R will be between N–2 and N–1. Therefore, when using the 1CPP algorithm with $M = N$, the number of DTs will decrease as R increases from 1 to M–2. Also, if M in the original equation is reduced to less than $N \geq 4$ by some power of two, the mathematically optimal value of R is larger than its assumed maximum value of $M \leq N/2$, and the minimum number of DTs is always reached when the maximum number of PEs are used.

The possibility that the number of DTs performed by an algorithm may increase as the number of PEs increases means that there could be a case when the total algorithm execution time increases when more PEs are used. A method is presented in the next subsection for determining whether this is true for a given system and problem size.

2.4.4. Performance Prediction

A method is adapted from [SaS93] to predict the number of PEs to use that will minimize the execution time for the SVD algorithm. This method gives relative weights to the FP and DT operations by the determination of a communication ratio (CR). This ratio is used with the complexity equations in Table 2.1 to predict only whether performance improves as more PEs are used. Because the numbers of FP and DT operations do not account for the total execution time, machine-dependent data was collected to use for the prediction.

The CR is calculated in terms of average expected time to perform a DT over the average expected time to perform a FLOP (including memory access and array address calculation times). The units of measure for the CR are ((secs./DT)/(secs./FLOP)) = FLOPs/DT. Various methods can be used to determine the CR. The one chosen executes one implementation of the SVD algorithm on a small matrix, using the minimum number of PEs that the implementation allows. The 1CPP algorithm was arbitrarily selected to measure the CR, with four PEs being used to decompose a random 4×4 matrix. Although the PASM prototype can operate in different modes of parallelism, SIMD mode is used throughout this analysis for consistency. Hardware timers are used to measure the execution times of the operations being considered. Because the PASM prototype currently performs all FP calculations in software and has a relatively fast inter-PE communication network, its CR measured 0.119. It is assumed for this analysis that the CR does not vary with the number of PEs used.

Using the CR, the predicted performance (PP) of a machine running an SVD implementation is approximated by PP = (no. of FLOPs) + CR·(no. of DTs), and is a function of both matrix size and the number of PEs. With this definition, PP will have units of number of

FLOPs. Because the PP equations for the 2CPP and 1CPP approaches ($\underline{PP_{2CPP}}$ and $\underline{PP_{1CPP}}$) do not consider many overhead operations, they do not provide absolute execution times, but they are reasonable estimates of relative execution times as R, N, and M are varied. Therefore, they can be analyzed to determine the number of PEs that will provide minimum execution time on a particular machine.

2.4.5. Implementation Comparison

The operation counts of the 2CPP and 1CPP approaches are now compared. One comparison covers when the number of PEs equals the minimum common number that the two implementations can use (N PEs). A second comparison is for when the maximum common number of PEs are used (NM/2 PEs). These two cases are focused on because various numbers of PEs can be used, depending on the values N, M, and R. The third case directly compares PP_{2CPP} and PP_{1CPP}.

To compare the two implementations with N PEs, replacements are made for R and r in the equations of Table 2.1 which correspond to using N PEs with either approach. The 2CPP approach requires both fewer FLOPs and fewer DTs under the constraints that $M \geq 2$ and $N \geq 4$ (details in [UlM95]). Because these constraints are met for all values of N and M of interest, the 2CPP implementation is expected to be the fastest (neglecting differences in overhead between the two approaches) when the minimum common number of PEs are used.

To compare the two approaches using NM/2 PEs, the same method is followed, with different values replacing R and r in the equations of Table 2.1. Analysis in [UlM95] shows that the 1CPP implementation uses fewer FLOPs when NM/2 PEs are used, under the constraint that $M > 2$, which is true for all values of M of interest. It is also shown that the 1CPP implementation uses fewer DTs under the constraint $(M(N-1)\cdot(m+1)+M^2) > N^2$. This inequality is not true for all values of N and M, but it can easily be shown to be true when $M \leq N \leq M(m+1)$, which covers many cases. Thus, for this range of N, the 1CPP implementation is expected to be the fastest when the maximum common number of PEs are used. For N outside this range, the PP of the two implementations can be compared, taking into account the CR of the system.

PP_{2CPP} and PP_{1CPP} are directly compared for three matrix sizes of interest and a CR = 0.119 in Figure 2.4. The figure shows that as the number of PEs used increases from N to NM/2, the execution times of both methods are expected to decrease, and the fastest implementation is predicted to change from the 2CPP approach to the 1CPP approach. It also shows that the 1CPP implementation with NM PEs is expected to provide the overall minimum

execution time.

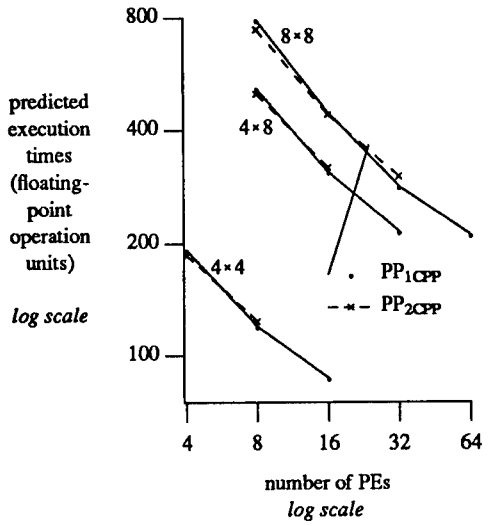


Figure 2.4: Comparison of PP_{2CPP} and PP_{1CPP}; CR=0.119.

2.5. Performance Evaluation

2.5.1. Experimental Algorithm Performance

Matrices of size 4×4, 4×8, and 8×8 allow timing data to be recorded while using different numbers of PEs on the 16-PE prototype. Both the 2CPP and 1CPP implementations were executed on the PASM computer with matrices of these sizes. Jacobian matrix data consisted of randomly generated FP values within the range (–5, +5). Algorithms were coded using a combination of a C language compiler, AWK scripts (for pre- and post-processing), and library routines for data-conditionals, network transfers, and data transfers from the control unit (CU) to the PEs. Matrix and column elements were stored in arrays. Values for M, N, and R were left as variables that could be updated before each execution so that several data points could be obtained easily. But, because M, N, and R were variables, all column and column segment operations involved loops that could not be unrolled.

The execution times were recorded for both the 2CPP and 1CPP implementations executed in SIMD mode on the PASM prototype. Matrices of each of the three dimensions specified were decomposed using both implementations, with all allowable numbers of PEs between one and 16, inclusive. The recorded data is plotted in [UlM95].

A comparison of the 2CPP and 1CPP implementation execution times is illustrated in Figure 2.5. The experimental timing data represents the average execution times of an algorithm run on 256 different Jacobian matrices of the given size. The experimental data is normalized to the average execution time of the SVD algorithm when decomposing a 4×4 matrix with a single PE. For the 4×4 matrix case, it is apparent that the fastest implementation switches from the 2CPP approach to the 1CPP approach when going from four to eight PEs. Also, the 1CPP approach can use 16 PEs when working with a 4×4 matrix, whereas the 2CPP approach cannot. The data obtained for 4×4 matrices is as expected. Similarly, for the 4×8 matrix case, the fastest implementation switches from the 2CPP to the 1CPP approach when going from eight to 16 PEs. When working with an 8×8 matrix, the 1CPP implementation execution time approaches that of the 2CPP implementation when going from eight PEs to 16 PEs. All of these observations of Figure 2.5 match exactly the comparison of the PPs shown in Figure 2.4.

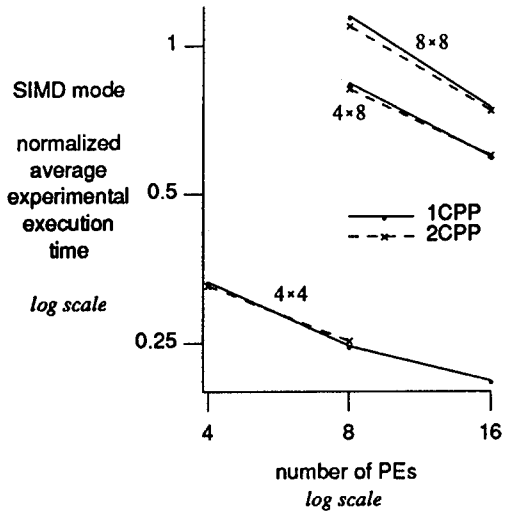


Figure 2.5: Experimental performance of 2CPP vs. 1CPP.

It was desired to determine how algorithm execution times are affected by the number of PEs when the CR is much higher than 0.119, as would be expected on a commercially available machine with FP coprocessors or digital signal processors. For this purpose, the 2CPP and 1CPP programs were changed to operate on integer data (the square-root FLOP, which is comparable to a FP division with an MC68881 FP coprocessor [Mot87], was replaced by a single integer division). This was done for experimental timing studies only; FP operations are needed to get the desired accuracy for this application. The CR for this new code was determined to be 1.205. The PP for the two algorithms with this CR are analyzed in [UlM95]. It is shown that for 4×4, 4×8, and 8×8 matrices, execution times are still expected to decrease as the number of PEs increases. These predictions were verified by experimental execution times on the PASM

machine [UlM95].

2.5.2. Modes of Parallelism

The 2CPP and 1CPP algorithms were performed on the PASM prototype using SIMD, MIMD, BMIMD, and mixed-mode parallelism. To determine the most effective mode mappings, both algorithms were divided into several code fragments. The fastest execution mode for each code fragment was then determined.

The SIMD and MIMD modes of parallelism each have several advantages and disadvantages [SiA92]. An advantage of SIMD is the ability to utilize CU/PE overlap. For the SVD implementations, this overlap occurs when the CU performs the overhead associated with loops while the PEs execute the loop bodies. Another advantage of SIMD is that the implicit synchronization after every instruction broadcast to the PEs implies that explicit synchronization is not required during communication. A SIMD disadvantage is that data conditional "then" and "else" clauses must both be broadcast to the PEs.

An advantage of MIMD is the ability to execute the clauses of data conditional statements without underutilizing PEs. A block of instructions whose execution times are data-dependent will complete faster [SiA92]. A MIMD disadvantage is that explicit sender/receiver synchronization is required before inter-PE communication can take place. On PASM, sending and receiving PEs must be synchronized for every value sent through the network in MIMD mode. In the BMIMD implementations, all operations are executed in MIMD with the exception that a barrier is executed once for every network setting. After the barrier, all required data transfers can be made as if the PEs were in SIMD mode, with less overhead than MIMD network transfers.

Mixed-mode implementations incorporate advantages of both the SIMD and MIMD mode implementations while trying to avoid the disadvantages of each. Various mode combinations were considered for the different program fragments of both the 2CPP and 1CPP approaches. The following is an analysis of the implementations that resulted in the shortest execution times of each method.

Table 2.2 shows how the 2CPP algorithm was divided into code fragments. Fragment 1 is a nested loop calculation of partial sums of two columns of the B matrix, where each of M column elements is determined from N/R matrix elements of J and V. This fragment is implemented in SIMD mode to maximize the advantage of CU/PE overlap. Fragment 2 is a set of transfers in a loop that combines the partial sums of segments of b_i and b_j. Fragment 2 is also implemented in SIMD to utilize both CU/PE overlap and implicit network transfer synchronization.

alg. step	code frag.	code fragment description
1	1	derive partial sums of columns b_i and b_j
	2	combine segments of b_i and b_j
2	3	determine columns to transfer in $Shift_{S/2}(i,S)$ op.
	4	transfer column segments via $Shift_{S/2}(i,S)$ op.
	5	transfer column segments via $Shift_1(i,S)$ op.
	6	reorder left/right columns by their number
	7	derive partial sums of $b_i^T b_i$, $b_j^T b_j$, and p
	8	combine $b_i^T b_i$, $b_j^T b_j$, and p partial sums
	9	calculate q, c, $\sin(\phi)$, and $\cos(\phi)$
	10	rotate b_i, b_j, v_i, and v_j
3	11	derive partial sums of $b_i^T b_i$ and $b_j^T b_j$
	12	calculate σ_i and σ_j, R=1
	13	determine $b^T b$ value to combine, R≠1
	14	combine $b^T b$ term, and partners exchange σ, R≠1
	15	conditional calculation of u_i and u_j

Table 2.2: Code fragmentation of 2CPP implementation for mixed-mode parallelism.

Inter-rotation column segment transfers are handled by code fragments 3 through 6. Fragments 3 and 4 are performed n−1 times during the execution of 2CPP, once for each $Shift_{S/2}(i,S)$ operation. Fragment 3 performs a short if-then-else conditional in MIMD mode to determine the column segments to be transferred. In fragment 4, a column segment of B and V is transferred by each PE. The matrix element transfers are performed in two loops. SIMD mode is used to take advantage of both CU/PE overlap and transfer efficiency. Code fragment 5 performs the $Shift_1(i,S)$ operation N−n−1 times throughout execution of 2CPP. These transfers are again executed in SIMD mode for the advantages of CU/PE overlap and transfer synchronization. Fragment 6 is performed after each inter-rotation Shift (i.e., N−2 times). It is an if-then-else operation executed in MIMD mode that reassigns i/j (left/right) column order.

Code fragments 7 through 10 are performed N−1 times during the execution of the 2CPP algorithm; once for each rotation. Fragment 7 calculates three partial sum values within a loop executed in SIMD mode. Fragment 8 combines these partial sums via recursive doubling transfers that get performed r times in a loop. Again, this loop is executed in SIMD to take advantage of both CU/PE overlap and implicit transfer synchronization. Code fragment 9 calculates the values q, c, $\sin(\phi)$, and $\cos(\phi)$. This is an in-line block of code requiring no loops, so MIMD mode is used because the instructions' execution times are data-dependent. Fragment 10 performs the rotation on two column segments of B and of V. Rotating column segments of B requires M/R iterations of a tight loop, and rotating column segments of V requires N/R loop iterations. These loops are again performed in SIMD mode to take advantage of CU/PE overlap.

Calculation of partial sums of the values $b_i^T b_i$ and $b_j^T b_j$ occurs in fragment 11 as another tight SIMD loop. Fragment 12 is performed only in the special case when R = 1. In this case, the values $b_i^T b_i$ and $b_j^T b_j$ found in fragment 11 are final values, not partial sums, and two square-root FLOPs will yield σ_i and σ_j. Fragment 12 is performed in MIMD mode because instruction execution times are data-dependent. When R≠1, fragments 13 and 14 are performed to find the final σ_i and σ_j values. Fragment 13 is a short if-then-else operation performed in MIMD that determines the single $b^T b$ value a given PE will be calculating (i.e., for column i or for column j). Fragment 14 performs transfers in a loop to combine the single $b^T b$ term in each PE, then finds the square-root of this final value to obtain σ, and exchanges σ with its partner PE (so both have the final σ_i and σ_j values). These transfers are performed in SIMD mode to take advantage of both CU/PE overlap and implicit transfer synchronization.

The final code fragment, 15, calculates column segments of U by dividing elements of B by the corresponding σ value. If σ is nonzero, the division is executed. If σ is zero, the corresponding column of U is replaced with zeroes. Fragment 15 is therefore performed in MIMD mode to take advantage of parallel "then" and "else" clause execution.

Figure 2.6 shows the execution times of the 2CPP implementation when run in different modes of parallelism on PASM. The data shown in the figure are the results of 4×4 matrix SVD. The minimum execution time was obtained using mixed-mode parallelism.

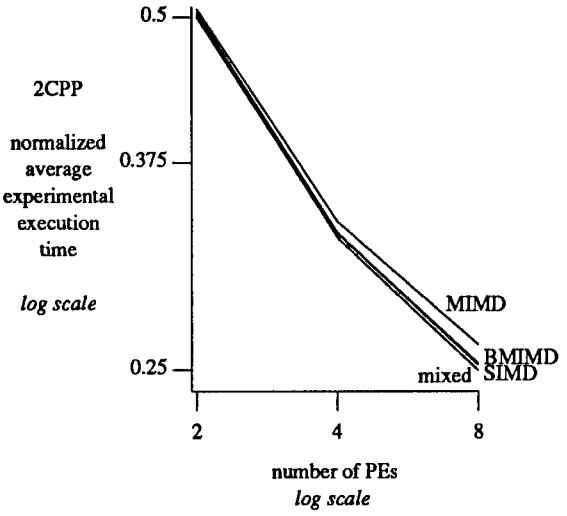


Figure 2.6: 2CPP algorithm execution time comparison" "for different modes of parallelism; 4×4 matrix.

From the figure, it is obvious that the advantage of strictly SIMD mode over MIMD increases as the number of PEs increases. It is also obvious that SIMD and BMIMD execution

provide similar execution times, meaning that the greatest advantage that SIMD has over MIMD for the SVD algorithm is implicit network transfer synchronization. Using the 2CPP DT count equation in Table 2.1, it can be determined that for a 4×4 matrix, the number of DTs increases as the number of PEs used increases. This means that network transfers become a larger portion of the operations performed, and that the SIMD advantage in transfer times becomes a greater asset.

The execution times displayed in Figure 2.6 also show that the advantage mixed-mode parallelism has over strictly SIMD increases as the number of PEs increases. It is apparent from the code fragment analysis that the fragments performed in MIMD generally do not operate on column segments, and therefore their performance is generally independent of the number of PEs, i.e., the value of R. Thus, the MIMD code fragment execution times become a larger fraction of the overall execution times as more PEs are used. As the overall execution times decrease, the MIMD advantage of those code fragments becomes more prominent.

The 1CPP algorithm was also fragmented to determine the best combination of modes of parallelism for fastest mixed-mode execution (details in [UlM95]). The 1CPP algorithm has fewer code fragments than the 2CPP approach, and each is analogous to one already presented in the 2CPP mixed-mode analysis. The observations made between the different modes of parallelism with the 2CPP approach held with the 1CPP approach.

2.6. Conclusions

Several methods for performing SVDs using column transformations have been previously developed. Many of these use rotation operations in an iterative construct to perform the decomposition. Those methods map a unique pair from N columns to each of N/2 PEs, and implement inter-PE communication patterns designed to accommodate their systems' interconnection network. This study presents a similar method, 2CPP, which utilizes the capabilities of a multistage cube network. Another method, 1CPP, was also developed, which maps one matrix column to each of N PEs. The method introduced here for dividing each matrix column into R segments provides the greatest impact on performance. It allows the use of R times the number of PEs previously used, by utilizing more of the inherent parallelism of the SVD algorithms. The approach works effectively with both the 2CPP and 1CPP mappings due to both the methods of data distribution and the capabilities of the multistage cube network.

The methods derived to implement one sweep of rotations (step 2 of the SVD algorithm) can also be applied to other SVD algorithms that iteratively perform multiple sweeps of column rotations. These algorithms would be useful when greater accuracy is needed in the

decomposition, or when successive matrices being decomposed cannot be considered as small perturbations of previous matrices. Using more PEs by distributing column segments among PEs may decrease the execution times of these algorithms as well. The performance prediction method presented in Subsection 2.4 can be used for this determination.

The analysis presented in Subsection 2.4 and supported by experimental data in Section 2.5 provides the following results. First, the PP analysis presented in Subsection 2.4 can be used to determine the number of PEs to use in a system to achieve the minimum execution time of either the 2CPP or 1CPP implementation. Second, the execution times of both implementations depends on the size of the matrix being decomposed, the number of PEs being used, and the CR of the system executing the algorithm. Third, when increasing the number of PEs being used from N to NM/2, the fastest implementation generally changes from the 2CPP approach to the 1CPP approach.

Experimental data presented in Subsection 2.5 demonstrates that the mode of parallelism used can have an affect on the execution time of an algorithm. The results obtained show that an SIMD implementation of either the 2CPP or 1CPP SVD approach performs better than an MIMD implementation regardless of the number of PEs used. By using barriers to reduce the synchronization overhead involved in MIMD mode network transfers, the BMIMD implementations outperformed the MIMD implementations. Finally, a mixed-mode implementation can outperform SIMD, MIMD, and BMIMD implementations by using the advantages of each mode on different program fragments.

3. A Block-Based Mode Selection Model for SIMD/SPMD Parallel Environments

3.1. Introduction

In general, writing effective programs for parallel computers remains a complex and difficult endeavor. In many cases, algorithms that work well under a serial execution model require reformulation for implementation on a parallel system. The problem is compounded by the rich diversity of parallel architectures available, each with its own attributes. Subsequently, parallel languages for these systems vary substantially, as vendors are (justifiably) concerned with maximizing performance for their products. One of the challenges for compilers and compiler-related tools is, given a machine-independent parallel language, to generate executable code for a variety of parallel computational modes, and to identify those specific modes for which the program is well-suited. This problem is even more difficult on a heterogeneous system [Fre91, FrS93], where different parallel modes and/or machines can be used to perform the segments of a single task.

One type of a heterogeneous system is a mixed-mode machine, in which the processors are capable of operating in either the SIMD or MIMD modes of parallelism and can dynamically switch between modes at instruction-level granularity with generally negligible overhead, e.g., PASM [SiS94], Triton [PhW93], and OPSILA [AuB86]. The focus here is the use of mixed-mode machines to perform data-parallel algorithms, where the data is distributed among the processor/memory pairs of a parallel machine and all processors use the same code to operate on their local data [HiS86]. In MIMD mode, data-parallel algorithms are associated with SPMD (single program - multiple data) operation, where all processors are executing the same program, but are doing it asynchronously with respect to one another, i.e., each processor follows its own control path through its copy of the program [DaG88]. The relative execution times of the segments of an algorithm using the SIMD or SPMD modes of computation must be estimated to map a data-parallel algorithm onto a mixed-mode machine in an effective way. Given a data-parallel program written in a language whose syntax is mode-independent and empirical information about instruction characteristics and execution paths, the goal addressed here is to

---

The co-authors of this section were Daniel W. Watson, Howard Jay Siegel, John K. Antonio, Mark A. Nichols, and Mikhail J. Atallah.
This research was supported by the Office of Naval Research under grant number N00014-90-J-1937, by Rome Laboratory under contract numbers F30602-92-C-0150 and F30602-94-C-0022, by NRaD under subcontract number 20-950001-70, and by the National Science Foundation under grant number CCR-9202807.
This material appeared in the *Journal of Parallel and Distributed Computing*, Special Issue on Heterogeneous Processing, Vol. 21, No. 3, June 1994, pp. 271-288.

make a static source-code based determination of the implementation that results in the optimal execution time on a mixed-mode machine.

One aspect of current research in parallel optimization and analysis techniques is concerned with extending traditional compiler optimization techniques, including loop concurrentization, code hoisting, and common subexpression elimination, for parallel programs. Many conventional optimization techniques employed by serial compilers are applicable in parallel compilers; however, in some cases, traditional optimization approaches without proper analysis can produce erroneous results [MiP90]. There are a number of research efforts directed specifically towards improving execution time in a parallel environment by performing compile-time analysis of programs. In [GuB92], communication costs in a parallel program are analyzed as a function of array dimension and the number of available processors. A time-cost approach based on analysis and simulation is developed in [QiS91] to determine time-cost behavior of parallel computations based on parameters such as input, algorithm, data structure, execution overhead, and execution environment. In [DiZ92] a timing analysis is employed to allow compilers for barrier MIMD machines to eliminate a significant percentage of run-time synchronization. The goal of code-type profiling is to identify partitions of a program with similar computational requirements [Fre89]. An "optimal selection framework" is presented in [WaK92] as a model for determining the optimal configuration of a heterogeneous suite of supercomputers to perform each task in a given set of applications.

One area where static source-code analysis may prove beneficial is in the execution of parallel algorithms in an environment capable of both SIMD and MIMD modes of parallelism. Heterogeneous systems exploit the diverse computational requirements of traditional supercomputing problems by the selection and use of different types of machines for the computation required [Fre91, FrS93, KhP93]. Typical examples of implementation studies using heterogeneous suites of machines are summarized in [Sun92]. Several studies [e.g., AuB87, BeK91, FiC91, GiW92] have examined the implementation of parallel programs in a mixed-mode machine, i.e., a machine that can operate in either the SIMD or MIMD mode of parallelism and can dynamically switch between modes at instruction-level granularity with generally negligible overhead. A common result from these studies is that cases exist where the execution time of a parallel application can be reduced by exploiting the mode of parallelism that best matches each portion of the program.

Recent academic and commercial interest in parallel computing systems has increased activity in the development of unifying parallel programming models. New programming languages and refined models of programming for parallel machines form an area of research that benefits from this increased interest. Some examples are "Refined C," CODE, and

SUPERB. The "Refined C" parallel language is developed in [DiK85] to provide programmers with a means of expressing algorithms for different parallel computers without imposing a specific parallel programming style. A computation-oriented display environment (CODE) is introduced in [BrA89] that encompasses most existing or proposed MIMD architectures and the programming systems related to them. The SUprenum ParallelizER Bonn (SUPERB) [ZiB88] is an interactive system for the semi-automatic transformation of FORTRAN 77 programs into MIMD and SIMD programs by using a catalog of parallelization transformations.

Another approach in the development of unifying programming models is to provide languages that support multiple modes of parallelism. Examples of languages designed to support either the SIMD or SPMD modes of data-parallel programming include CM-Fortran [ChC92], HPF [HPF92], and Fortran-D [HiK91]. Other languages, such as CSL [BrT82], Hellena [AuB87], and ELP [NiS93], have been developed for machines that are capable of mixed-mode parallelism, and include language elements for both SIMD and SPMD (or MIMD) computation. These "mixed-mode" languages provide support for executing different portions of the same program in different modes (e.g., SIMD versus SPMD). CSL CSL (Computation Structures Language) is a PASCAL-like explicitly parallel job control language used on TRAC that supports the high-level specification and scheduling of SIMD and MIMD tasks. Hellena is an explicitly parallel preprocessed version of PASCAL for OPSILA that supports dynamic mode switching at the instruction level. An Explicit Language for Parallelism (ELP) is an example of a language whose goal is for all of the statements and constructs to have functionally equivalent SIMD and SPMD interpretations, allowing segments of the same program to be compiled for different parallel models.

The development of mode-independent languages makes possible the incorporation of the decision-making process for selecting the appropriate parallel mode (heretofore performed by knowledgeable programmers) into the realm of the parallel code compiler. Toward that goal, this study proposes a framework for the static source-code based analysis of execution time for data-parallel algorithms on a mixed-mode machine. These algorithms are assumed to be written in a mode-independent language, and to be implemented in an SIMD/SPMD environment. By establishing this framework, the three elements described above, compile-time analysis, use of different parallel modes, and unifying programming models, are brought together to provide a basis for writing efficient parallel algorithms. The technique involves transforming the program into an SIMD/SPMD trade-off tree, whose structure represents the scope levels within the program. Information at the leaf nodes of the tree, representing blocks of code in the program, is then combined using rules to arrive at decisions for the best parallel mode in which to implement each portion of the program.

To illustrate underlying concepts for the methods presented, the techniques are first developed for the case where a parallel program is to be implemented in either pure SIMD mode or pure SPMD mode (distributed memory machines are assumed). The more general case of a single program that employs both modes during the course of execution on a mixed-mode machine is then considered. For ease of presentation, the programming model presented here is restricted to basic processor operations and elementary control-flow constructs. Statistical information about individual operation execution times and paths of execution through a parallel program is assumed. This basic model can be enhanced to include other language constructs.

The techniques presented here are directly applicable to the analysis of programs for implementation in a mixed-mode system. Furthermore, they provide a basis for studying the more general problem of optimizing resources in a multiple-machine heterogeneous computing environment. A secondary goal of this study is to indicate language, algorithm, and machine characteristics that must be researched to learn how to provide the information needed to obtain an optimal assignment of parallel modes to program segments.

The system model and representative mode-independent language assumed in later sections are described in Subsection 3.2. Many of the reasons why different parallel implementations exhibit disparate execution time performances are the result of inherent differences in parallel modes. These differences are examined in Subsection 3.3. In Subsection 3.4, techniques are presented for choosing the single-mode implementation of a parallel program that minimizes execution time. Basic definitions used throughout the rest of the section are included in Subsection 3.4.2, and the single-mode framework itself is developed in Subsection 3.4.3 using a simplified execution-time model for machine-level operations. In Subsection 3.4.4, a more refined model for operation execution times using probabilistic analysis is considered, and in Subsection 3.4.5 the effect of interaction between program segments is explored. The single-mode analysis of Subsection 3.4 introduces many concepts that are useful in Subsection 3.5. In Subsection 3.5.1, some of the challenges associated with mode-independent languages are considered by examining how programs written in a mode-independent manner can be executed on a mixed-mode machine. The single-mode analysis of Subsection 3.4 is expanded to find an efficient mixed-mode implementation in Subsection 3.5.2. Subsections 3.5.3 and 3.5.4 introduce a method to determine the minimum execution time in an SIMD/SPMD mixed-mode machine implementation through the use of multistage optimization techniques. Concluding remarks and areas for further research are summarized in Subsection 3.6.

3.2. Machine and Language Model

The mixed-mode machine model assumes a physically distributed memory. Thus, each processor is paired with a memory, forming a processing element (PE). It is assumed for mixed-mode machines that all PEs are in the same mode at any point in time (i.e., SIMD versus MIMD).

In the SIMD model used here, the control unit (CU) enqueues instructions to be performed on the PEs into an instruction queue. The PEs then fetch instructions from the instruction queue independently of CU operation. Thus, in SIMD mode, the CU can overlap its operations with those of the PEs.

One way of programming machines that are capable of mixed-mode parallelism is by using a mode-independent language, i.e., a language whose syntactic elements have interpretations under more than one mode of parallelism. An example of a mode-independent language is the Explicit Language for Parallelism (ELP), under development at Purdue University [NiS93]. ELP provides constructs for SIMD, MIMD, and mixed-mode parallelism. The ELP syntax is based on C, and allows the programmer to specify the SIMD, MIMD, and SPMD modes of parallelism within a program. Although ELP contains specifiers for full MIMD mode processing, this section focuses on SIMD and SPMD modes only. A goal of ELP is to provide uniformity with respect to the SIMD and SPMD modes of parallelism by having interpretations for the syntax within both of these modes that are identical in semantics. This is an important characteristic because it allows a data-parallel algorithm to be coded in a mode-independent manner, producing a data-parallel program for which: (1) only SIMD code is generated, (2) only SPMD code is generated, or (3) execution mode specifiers can be added to facilitate mixed-mode experimentation.

ELP associates with each variable defined in a program a variable class. A variable defined to be of class mono always has a single value with respect to all PEs, independent of execution mode; whereas a variable defined to be of class poly can have different values in each PE, independent of execution mode. Each mono variable has storage allocated for it on the CU and on all PEs. If a mono variable is referenced while in SIMD mode, its CU storage is active. If a mono variable is referenced while in SPMD mode, its PE storage is active and all PE copies of the mono variable will have the same value. For variables defined to be poly, each PE has its own copy with its own value, independent of execution mode.

In SIMD mode, operations on mono variables indicate work to be done on the CU, and they permit CU/PE overlap to be explicitly specified. This, in turn, allows the user to experiment with load balancing between the CU and the PEs. In SPMD mode, mono variables can be used

to force `if`, `while`, `do`, and `for` statements on different PEs to execute in the same fashion on all PEs; for example, mono variables could be used as the index variable and as the common upper bound for a `for` loop with all PEs. All PEs must execute the same instructions, but not necessarily at the same time (as in SIMD mode). Thus, mono variables in ELP are guaranteed to have the same value spatially, but not temporally. This spatial congruency is enforced syntactically by the ELP compiler. Mono variables also permit other SPMD operations to be performed in an identical fashion across all PEs, such as having each PE access the same element of an array.

In the assumed model, when changing between SIMD mode and SPMD mode, only the source of the instructions (control) is changed between the CU and each PE's local memory, respectively. In both the SIMD and SPMD modes of execution, the same local memory and registers are used at each PE to store poly variables. Thus, mode changes in themselves do not cause a need for data transfers except for mono variables, the time for which is assumed to be relatively negligible. A more detailed description of the ELP language and compiler can be found in [NiS93].

3.3. Parallel Performance Issues

There are trade-offs that exist between the SIMD and MIMD modes of parallelism that explain why some sequences of instructions are better performed in one mode than in the other [BeS91, SiA92]. Some of the advantages and disadvantages of each mode are summarized here.

Conditional statements in the synchronous execution of an SIMD program can introduce serialization. Consider an if-A-then-B-else-C statement. Let the conditional test A depend on PE data. In some PEs, A is true and in others false. Those PEs where A is false are disabled (masked off) during the execution of clause B. Once B has executed, the PEs where A is true are disabled and the PEs where A is false enabled. C is then executed. This serializes the execution of B and C. Conversely, in MIMD mode those PEs where A is true can execute B while the other PEs execute C. In MIMD mode, the maximum time to execute the if-then-else statement in a PE is approximately $T_A + \max(T_B, T_C)$, while in SIMD mode the time would be approximately $T_A + T_B + T_C$ (where a PE is idle for either T_B or T_C). Thus, in general, MIMD mode is more effective for executing conditional statements.

Another distinction between SIMD mode and MIMD mode pertains to synchronization overhead. In SIMD mode, the synchronization of program execution is implicit, because there is a single thread of control. However, when synchronization of program execution is required among PEs in MIMD mode, explicit synchronization mechanisms, such as semaphores or barriers, must be employed in the parallel program. Thus, synchronization costs are greater for

MIMD mode. One benefit of implicit PE synchronization becomes apparent when inter-PE data transfers are needed. In SIMD mode, when one PE sends data to another PE, all enabled PEs send data. Therefore, the "send" and "receive" commands are implicitly synchronized. Because all enabled PEs follow the same single instruction stream, each PE knows from which PE the message has been received and for what use the message is intended. Conversely, MIMD mode programs are executed asynchronously among all PEs. As a result, the PEs must execute explicit synchronization and identification protocols for each inter-PE transfer. While the details of the inter-PE transfer protocol in both SIMD and MIMD mode are implementation dependent, there is generally more overhead associated with MIMD mode inter-PE transfers. Like the synchronization overhead described above, this protocol overhead is a cost of the flexibility of programming in MIMD mode.

In SIMD mode, the CU can overlap its operations with those of the PEs. For example, the CU can perform the increment and compare operations on scalar-valued loop control variables, while the PEs execute the loop body. Furthermore, any operations common to all PEs, such as local array address calculations, can be performed in the CU while the PEs are performing other computations. In MIMD mode, CU/PE overlap does not occur, and the PEs must perform all of the instructions.

It is possible that the execution time of an instruction is data dependent, taking a variable length of time to perform on each PE. Such variable-time instructions execute at least as efficiently in MIMD mode as in SIMD mode. This is illustrated in Figure 3.1. In SIMD mode, PEs can execute the next instruction only after all PEs have completed the current instruction. Therefore, each instruction takes as long as it takes the PE that executes it most slowly. In MIMD mode, the PEs are not synchronized and each PE executes the next instruction independently. Let T_{ij} represent the time it takes instruction i to execute in PE j. Assume that T_{ij} in SIMD mode is equal to T_{ij} in MIMD mode. The execution time in SIMD mode of a sequence of variable-time instructions can be expressed as $\sum_i \max_j(T_{ij})$, for all i in the sequence. The time to perform the same sequence of instructions in MIMD mode can be expressed in terms of T_{ij} as $\max_j(\sum_i T_{ij})$ (Figure 3.1). Because $\max_j(\sum_i T_{ij}) \leq \sum_i \max_j(T_{ij})$, the time to execute the sequence of variable-time instructions in MIMD mode is less than or equal to the time to execute the same sequence of instructions in SIMD mode. Thus, because of this "sum of maxs" versus "max of sums" effect, MIMD mode is more appropriate for executing sequences of variable-time instructions.

The "sum of maxs" versus "max of sums" property is not limited to single instructions whose execution time is data dependent. In mixed-mode, an entire block of instructions whose

execution time varies on different PEs due to data-conditional statements and/or variable-time

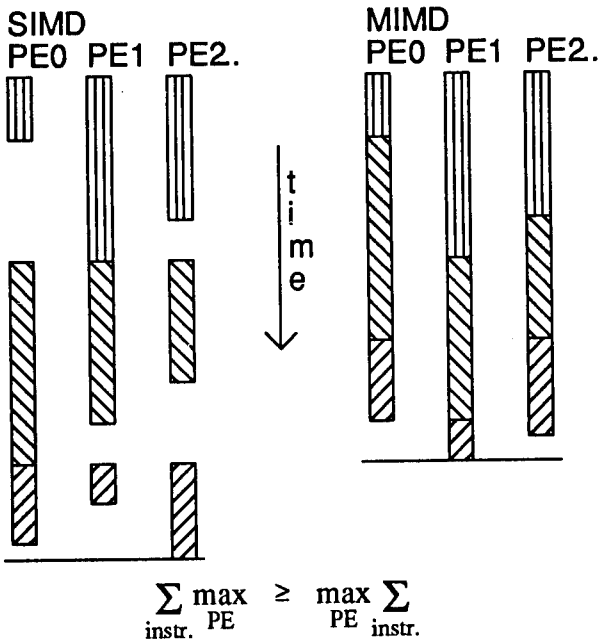


Figure 3.1: Execution of variable-time instructions in SIMD and MIMD mode.

instructions may exhibit the same performance characteristics on a "macro" level if synchronization is required after the block. Consider a loop body with two blocks A and B, such that the first block is best executed in MIMD and the second block is best implemented in SIMD. Because all PEs must synchronize after block A, before any PE can go on to block B, an added synchronization cost is encountered during each iteration of the loop. Consequently, the time penalty for synchronizing with the other PEs may outweigh the benefit of implementing block A in SIMD; i.e., the overall execution time may possibly be reduced by implementing the loop entirely in MIMD mode [BeK91]. The former case corresponds to the "sum of maxs" and the latter to the "max of sums."

From the discussion above it is evident that there are trade-offs between operating in SIMD mode and operating in MIMD mode. Although it is often clear in which mode a sequence of instructions should be implemented, this is not the case when counteracting trade-offs are involved. For example, a data-conditional clause may contain instructions that perform network transfers. Choosing the best mode of operation is not straightforward; i.e., conditional statements should be performed in MIMD mode while network transfers should be performed in SIMD mode. Studies examining the implementation of algorithms on mixed-mode machines have been performed [e.g., AuB87, BeK91, BrC90, FiC91, GiW92]. One long-term goal of

these efforts has been to increase the understanding needed to develop automatic, static source-code based determination of parallel modes for algorithm segments. In Subsections 3.4 and 3.5, a methodology for analyzing data-parallel programs to do this is examined.

By employing the methodology developed in Subsections 3.4 and 3.5, quantifications of all of the trade-offs discussed in this subsection can be incorporated in a straightforward manner, except for the "sum of maxs" versus "max of sums" trade-off, as well as the "macro" effect of this trade-off. Accurately modeling this trade-off is complex and replete with subtle nuances. In Subsection 3.4.4, a probabilistic basis for studying this aspect of SIMD/SPMD mode comparison is outlined.

3.4. Single-Mode Selection

3.4.1. Overview of Single-Mode Selection

A methodology is presented here to estimate the execution time of a data-parallel algorithm, written in a mode-independent language, that can be implemented in either the purely SIMD or purely SPMD mode of parallelism. This framework can be used to select the optimal single-mode for a mixed-mode parallel computer. Many of the concepts developed in this section are needed for the analysis in Subsection 3.5, where the use of both SIMD and SPMD within the same program is considered.

3.4.2. Assumptions and Definitions

Applications for this study are assumed to be data parallel [HiS86], i.e., the data for a program is distributed among the PEs, and the algorithm exhibits a high degree of uniformity across the data [Jam87]. This is in contrast to function (or control) parallelism [TuR88], where each PE executes a unique program.

Syntactic elements of the mode-independent language in which the algorithm is coded will be referred to as operations. Operations in a language represent the most explicit level at which program representation is identical for each mode of parallelism. Consider an operand fetch for a mono variable for use in a calculation that must be performed on the PEs. In SPMD mode, the operation is a simple fetch, because mono variables are stored on the PEs; however, in SIMD mode the mono variables are stored on the CU, and must be transferred to the PEs through the instruction queue or via a separate data bus. An operand fetch of a mono variable would therefore be considered a single operation for which execution-time information is known for both the SIMD and SPMD modes, even though different multiple machine level commands are required to perform the operation under each mode.

In general, the execution time of an operation may be fixed (and known) or data dependent. For the case of data-dependent operations, a probabilistic model is introduced in Subsection 3.4.4 to estimate the expected execution time for a block of operations. It is assumed that the necessary statistical information can be estimated empirically and is known a priori.

Even if the target architecture for both SIMD and SPMD implementations is virtually identical, as is typically the case for a mixed-mode machine, execution times of the same operation under SIMD and SPMD modes may differ significantly. For example, the expected time required to perform the mono operand fetch described in a previous paragraph would generally be greater under SIMD mode than under SPMD mode.

To estimate the overall execution time of a parallel program, code is examined in terms of constructs and blocks. Constructs include control constructs, such as looping structures and function calls, and data-conditional constructs, such as `if-then-else` and `case` statements. For the analysis here, the set of permissible control constructs is restricted to `for` loops for which the number of iterations, Q, is assumed to be known at compile time. Furthermore, because programs under consideration exhibit characteristics that make them candidates for implementation in SIMD mode (in which all PEs iterate the same number of times) it is also assumed that all PEs executing a loop in SPMD mode iterate the same number of times. In a practical sense, the expected value for Q may be obtained directly from the program source, either directly as a constant in the program, or as information provided to the compiler by the programmer in the form of a compiler directive. Alternatively, a sufficiently accurate value for Q might be obtained empirically by executing a prototype version of the parallel program on sample data sets. While time-consuming, this empirical approach may be reasonable for production codes.

The set of data-conditional constructs is limited here to `if-then-else` constructs, where all PEs may or may not perform the same blocks of code, depending on the outcome of a condition test of local PE data. It is assumed that the necessary statistical information regarding the probability that a branch will be taken can be estimated empirically or is obtainable by compiler directives provided by the programmer.

Future work will examine relaxing the restrictions to include other constructs, e.g., `while` and `case` statements. It will also investigate methods for obtaining the statistical information assumed to be known here, and for eliminating the assumption that all PEs iterate the same number of times.

Code blocks are the parallel equivalent of the serial basic blocks described in [AhS86]. Blocks of code are identified by their leading statements, called leaders. The first statement in a program is a leader, any statement that becomes the target of a conditional or unconditional

branch at the machine code level is a leader, and any statement that follows a conditional branch at the machine code level is a leader. In the approach described here, an additional constraint is used in the determination of blocks: any statement requiring synchronization and any statement that follows a statement requiring a synchronization is a leader, and any statement requiring an inter-PE data transfer and any statement that follows an inter-PE data transfer is a leader. This is an important distinction from the serial definition of a basic block in [AhS86], because synchronization points within an algorithm indicate when PEs are idle, waiting for one or more other PEs to arrive at the synchronization point.

Blocks consist of either pure scalar code or pure parallel code. Scalar blocks consist of instructions that, if performed in SIMD mode, would be executed on the CU (i.e., code that references only mono-valued variables). Parallel blocks consist of instructions to be performed on the PEs in SIMD mode (i.e., code that includes a reference to poly-valued variables). As an example, consider the execution of the loop `for i=1 to n` in SIMD mode, where both `i` and `n` are mono-valued variables, and where the loop body contains no references to mono-valued variables. One way to implement the loop is for the CU to perform an end-of-loop test, enqueue the appropriate SIMD instruction blocks to be broadcast to the PEs, perform an induction step, and branch back to the test. In this case, the PE instructions forming the body of the loop would be considered parallel blocks, while the increment and test code would each form a scalar block. Even if that portion of the program were to be executed in SPMD mode, the block definitions are the same as for the SIMD case.

3.4.3. Description of the Single-Mode Selection Technique

To determine the best single mode for a given program, the program is first transformed into a flow-analysis tree, which is a tree whose structure represents the scope levels within the algorithm. The flow-analysis tree is then used to create an SIMD/SPMD trade-off tree. For both trees, the leaf nodes of the tree represent parallel and scalar code blocks, and the non-leaf nodes correspond to control constructs and data-conditional constructs. The root node represents the scope of the entire program. Initially, only information about the execution times of the leaf nodes is assumed to be known. The proposed technique involves combining this known information to arrive at SIMD and SPMD execution times for the intermediate nodes. The optimal mode determined for the root node represents the "best" mode for the entire program. In this section, it is assumed that both the SIMD and SPMD execution times associated with each leaf node are known constants. In Subsection 3.4.4, methods are given for estimating SIMD and SPMD execution times for leaf nodes under the assumption that the execution times of the operations are random quantities with known probability distributions.

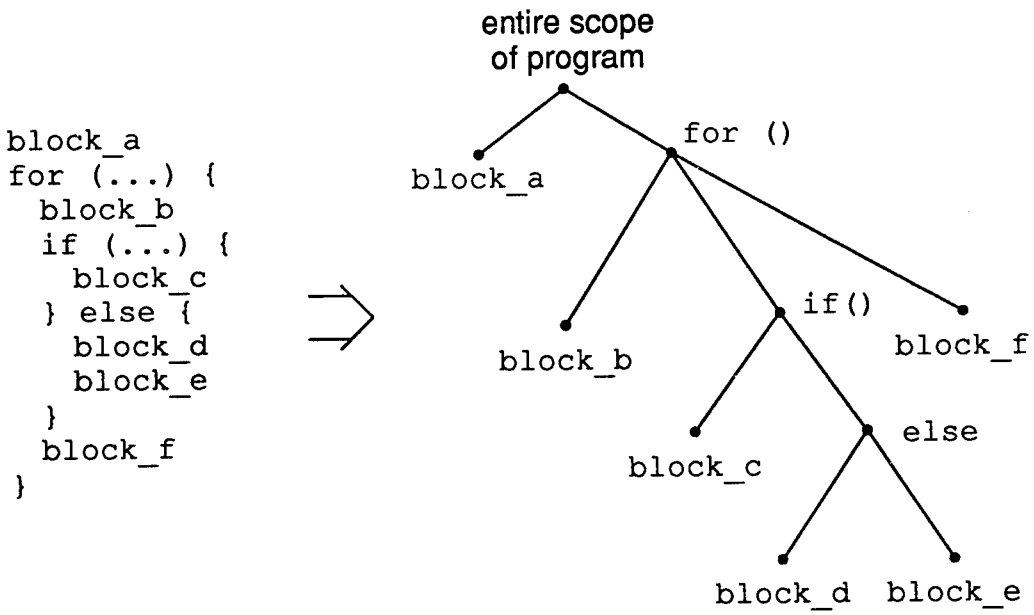


Figure 3.2: Example preliminary flow-analysis tree.

Figure 3.2 illustrates how a program is transformed into a preliminary flow-analysis tree. The program on the left side of the figure is composed of two elements, `block_a` and a `for` loop. Thus, the root node has two children, one for each of the elements. The `for` loop is composed of `block_b`, an `if` construct, and `block_f`, each of which is a child of the `for` node. For the `if` construct, the `then` clause is a single block (`block_c`), and is placed in the preliminary flow-analysis tree as a child of the `if` node. However, because the `else` clause is composed of `block_d` and `block_e`, an interior node representing the `else` clause is formed, whose children are the leaves `block_d` and `block_e` in Figure 3.2. At each level in the tree, the sibling blocks are represented in the same order (left to right) as they appear in the program. This order dependence
becomes important for the analysis of juxtaposed nodes and the use of SIMD/SPMD operations together, presented in Subsection 3.4.5 and Subsection 3.5, respectively.

To include the overhead required for `for` and `if` constructs, it is necessary to add nodes to the preliminary flow-analysis tree. This is illustrated for the previous example in Figure 3.3.

Associated with each `for` loop is a block required for initializing induction variables, and another block required to execute the induction step for the loop, perform an end-of-loop test, and conditionally branch back to the top of the loop. Because the initialization block is executed only once, it is represented as a separate child of the root node in Figure 3.3, labeled `for_init`. Because the increment, test, and conditional branch are executed during each iteration of the loop, they are represented by a single block as the last child of the `for` node in

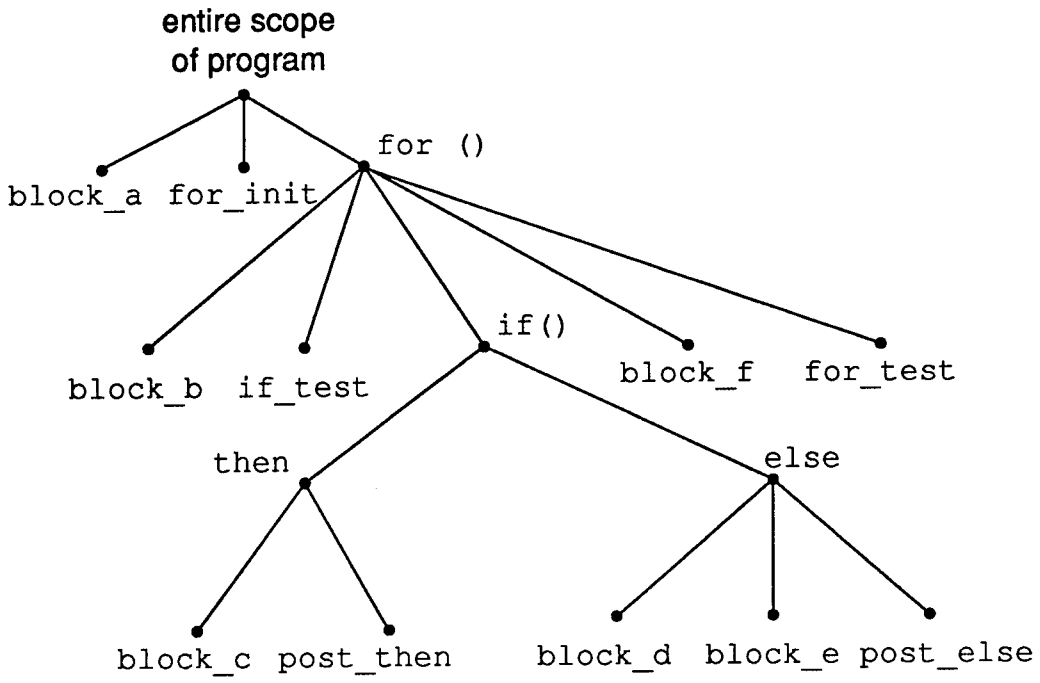


Figure 3.3: Final flow-analysis tree after modification to include overhead for `for` and `if` constructs.

Figure 3.3, labeled `for_test`.

Associated with each `if` construct is a conditional test (`if_test` in Figure 3.3), performed before either the `then` clause or the `else` clause is executed. There is also special processing required at the end of the `then` clause (`post_then` in Figure 3.3). In SPMD mode, this is an unconditional branch performed on each of the PEs for which the condition is true. In SIMD mode, this corresponds to disabling those PEs for which the condition is true and enabling those PEs for which the condition is false (with nested conditionals handled appropriately). Similarly, at the end of the `else` clause in SIMD mode, those PEs for which the condition is true must be re-enabled (`post_else` in Figure 3.3). Because there is no corresponding operation required in SPMD, the cost for executing the `post_else` in SPMD mode is 0. Because the `then` clause of the `if` construct is now composed of two blocks, an interior node is added representing the `then` clause, with leaves `block_c` and `post_then` as children.

After the final flow-analysis tree is formed, the following steps are performed to convert it to an SIMD/SPMD trade-off tree, as illustrated in Figure 3.4:

1) For each leaf l of the tree, assign an ordered pair (T_l^{SIMD}, T_l^{SPMD}), which represents the times for executing the block associated with leaf l in SIMD mode and SPMD mode, respectively.

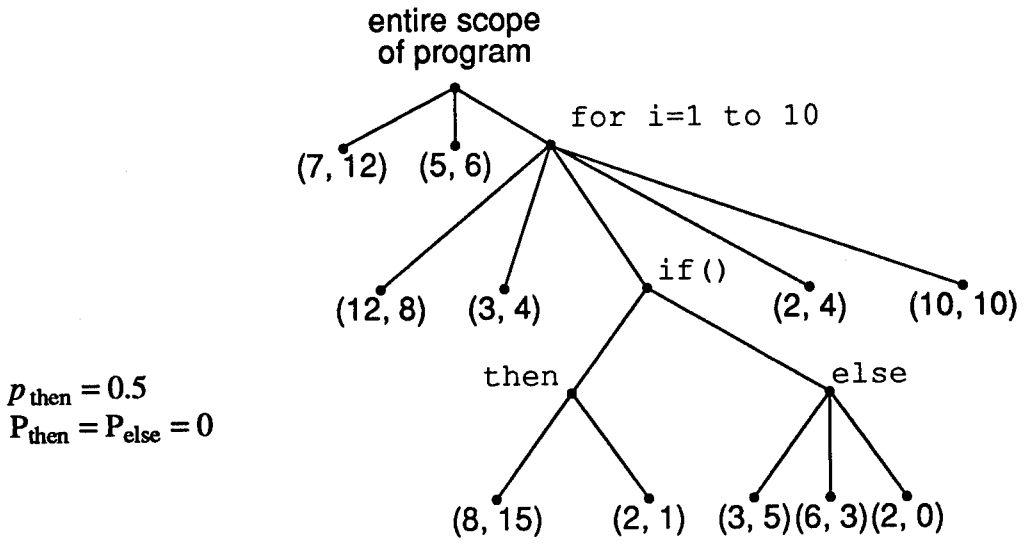


Figure 3.4: SIMD/SPMD trade-off tree for the flow-analysis tree in Figure 3.3.

Steps 2 and 3 are then performed for each non-leaf node in the order of a depth-first traversal of the tree.

2) For each non-leaf node *d* corresponding to a *d*ata-conditional construct, assign the ordered pair (T_d^{SIMD}, T_d^{SPMD}), which represents the times for executing the data-conditional construct at node *d*, including the time to execute all the children of *d* in SIMD mode and SPMD mode, respectively. To estimate the values of the ordered pair (T_d^{SIMD}, T_d^{SPMD}), consider how data conditionals are performed in SIMD and SPMD modes. In SIMD mode, if the conditional test for an `if-then-else` is true for *all* PEs, then only the PE instructions that belong to the `then` clause need to be broadcast to the PEs. Similarly, if the conditional test is false for *all* PEs, only the `else` clause needs to be broadcast. However, if the condition is true for some PEs and false for others, then the PE instructions for both the `then` and `else` clauses must be broadcast, effectively serializing the two clauses, as discussed in Subsection 3.3. Let p_{then} denote the probability that a PE executes the `then` clause, let P_{then} denote the probability that *all* PEs execute the `then` clause, and let P_{else} denote the probability that *all* PEs execute the `else` clause. For the special case when the outcomes of the conditional tests are mutually independent among N PEs, $P_{then} = (p_{then})^N$. In general however, the assumption of mutual independence cannot be made, and therefore P_{then} and p_{then} must be determined separately. The values of P_{then}, P_{else}, and p_{then} can be estimated in a similar manner to the value of Q as discussed in Subsection 3.4.1 (e.g., empirical data and/or compiler directives). Let T_u^{SIMD} be the execution time in SIMD associated with node *u*, and let

$$T_{\text{then}}^{\text{SIMD}} = \sum_{\substack{\text{all children } u \\ \text{of then clause}}} T_u^{\text{SIMD}}$$

and

$$T_{\text{else}}^{\text{SIMD}} = \sum_{\substack{\text{all children } u \\ \text{of else clause}}} T_u^{\text{SIMD}} .$$

Then the expected time required to perform the `if-then-else` in SIMD mode can be estimated by

$$T_d^{\text{SIMD}} = P_{\text{then}} T_{\text{then}}^{\text{SIMD}} + P_{\text{else}} T_{\text{else}}^{\text{SIMD}} + (1 - P_{\text{then}} - P_{\text{else}})(T_{\text{then}}^{\text{SIMD}} + T_{\text{else}}^{\text{SIMD}})$$

$$= (1 - P_{\text{else}})T_{\text{then}}^{\text{SIMD}} + (1 - P_{\text{then}})T_{\text{else}}^{\text{SIMD}} .$$

In contrast to SIMD mode, in SPMD mode each PE independently follows its own control path through the program. The `then` clause is executed on those PEs where the condition is true, and the `else` clause is executed on those PEs where the condition is false. Let T_u^{SPMD} be the execution time in SPMD associated with node u, and let

$$T_{\text{then}}^{\text{SPMD}} = \sum_{\substack{\text{all children } u \\ \text{of then clause}}} T_u^{\text{SPMD}}$$

and

$$T_{\text{else}}^{\text{SPMD}} = \sum_{\substack{\text{all children } u \\ \text{of else clause}}} T_u^{\text{SPMD}} .$$

Then the expected time to perform the conditional construct in SPMD mode is

$$T_d^{\text{SPMD}} = p_{\text{then}} T_{\text{then}}^{\text{SPMD}} + (1 - p_{\text{then}})T_{\text{else}}^{\text{SPMD}} .$$

The estimates for T_d^{SIMD} and T_d^{SPMD} derived above are *expected* execution times and will not necessarily be the actual (observed) times for any particular execution. To clarify the meaning and use of the formulas for T_d^{SIMD} and T_d^{SPMD}, consider a simple example. Assume there is a sequence of m data conditionals with expected execution times $(T_{d_1}^{\text{SIMD}}, T_{d_1}^{\text{SPMD}})$, $(T_{d_2}^{\text{SIMD}}, T_{d_2}^{\text{SPMD}})$, $\cdots$ $(T_{d_m}^{\text{SIMD}}, T_{d_m}^{\text{SPMD}})$. To simplify the illustration, assume that $p_{\text{then}} = 0.5$ for all m conditionals, that $T_{\text{then}}^{\text{SIMD}} = T_{\text{else}}^{\text{SIMD}} = T_{\text{then}}^{\text{SPMD}} = T_{\text{else}}^{\text{SPMD}} = T$, and that the conditional tests are mutually independent, which implies that $P_{\text{then}} = (p_{\text{then}})^N$ and $P_{\text{else}} = (1 - p_{\text{then}})^N$. Assuming N is moderately large, the formula for the expected execution time in SIMD mode yields $T_{d_i}^{\text{SIMD}} \cong 2T$, $i \in \{1 \cdots m\}$. The formula for the expected execution time in SPMD mode yields $T_{d_i}^{\text{SPMD}} = T$, $i \in \{1 \cdots m\}$. The expected execution time for executing all m conditionals in SIMD mode is therefore

approximately $2m$T, while the expected execution time for executing all m conditionals in SPMD mode is mT.

3) For each non-leaf node c corresponding to a control construct, assign the ordered pair (T_c^{SIMD}, T_c^{SPMD}), which represents the times for executing the control construct at node c, including the time to execute all the children of c in SIMD mode and SPMD mode, respectively. In SPMD mode, the control construct is performed entirely on the PEs. In SIMD mode, the control steps are performed on the CU, and then the PE instructions that form the loop body are placed in the instruction queue to be broadcast to the PEs. For SIMD, significant improvement in execution time can be obtained by overlapping the execution of operations on the CU and the PEs, and the need for adding this to the existing model in the future is discussed in Subsection 3.4.5. Disregarding for the moment the effects of CU/PE overlap, the values of the ordered pair (T_c^{SIMD}, T_c^{SPMD}) can be estimated by

$$T_c^{SIMD} = Q\left[\sum_{\substack{\text{all children}\\ u \text{ of } c}} T_u^{SIMD}\right]$$

$$T_c^{SPMD} = Q\left[\sum_{\substack{\text{all children}\\ u \text{ of } c}} T_u^{SPMD}\right],$$

where Q is the number of iterations in the construct and is assumed to be known, as discussed in Subsection 3.4.2.

4) For the node representing the root, assign the ordered pair $(T_{root}^{SIMD}, T_{root}^{SPMD})$, which represents the times for executing the entire program in SIMD mode and SPMD mode, respectively, where

$$T_{root}^{SIMD} = \sum_{\substack{\text{all children}\\ u \text{ of root}}} T_u^{SIMD}$$

and

$$T_{root}^{SPMD} = \sum_{\substack{\text{all children}\\ u \text{ of root}}} T_u^{SPMD}.$$

If $T_{root}^{SPMD} \le T_{root}^{SIMD}$, then implement the algorithm in SPMD mode, else implement it in SIMD mode.

As an example, consider the SIMD/SPMD trade-off tree in Figure 3.4, which corresponds to the the flow-analysis tree in Figure 3.3. In the tree, the ordered pairs (T_l^{SIMD}, T_l^{SPMD}) are assigned to each of the leaf nodes. For the example, let Q = 10, $p_{then} = 0.5$, and let $P_{then} = P_{else}$

= 0. The analysis algorithm performs a depth-first traversal beginning at the root node. When the algorithm considers the `then` clause, it assigns the ordered pair $(T_{then}^{SIMD}, T_{then}^{SPMD}) = (8+2, 15+1) = (10, 16)$. Similarly, for the `else` clause, $(T_{else}^{SIMD}, T_{else}^{SPMD}) = (3+6+2, 5+3+0) = (11, 8)$. Then the algorithm considers the `if` node, and assigns the ordered pair $(T_d^{SIMD}, T_d^{SPMD}) = (10+11, 0.5\times 16+0.5\times 8) = (21, 12)$. At the `for` node, the algorithm assigns the ordered pair $(T_c^{SIMD}, T_c^{SPMD}) = (10\times(12+3+21+2+10), 10\times(8+4+12+4+10)) = (480,380)$. For the root node, the ordered pair $(7+5+480, 12+6+380) = (492,398)$ is assigned, indicating that SPMD mode is best suited for this program.

3.4.4. A Probabilistic Model for Data-Dependent Operation Execution Times

The technique of Subsection 3.4.3 for estimating the execution time of a program operating in either SIMD mode or SPMD mode assumes that the execution times T_l^{SIMD} and T_l^{SPMD} are known constants for each leaf node l in the flow-analysis tree. While the leaf blocks do not contain conditional or control statements, times for these blocks may be data dependent if the block includes instructions whose execution times are data dependent (e.g., floating point addition). In this subsection, a probabilistic model to account for the uncertainty in the execution times of the blocks of code associated with the leaf nodes of the flow-analysis tree is considered.

The model presented here can be used as a basis for describing the general behavior of code segments whose execution time is data dependent. The model does not attempt to completely describe program behavior. Instead, it seeks to estimate expected execution times for individual code segments. Of particular interest is the difference between the expected execution time of a parallel code block across all PEs and the expected execution time of the same block for the PE which takes the most time to complete execution of the block. This difference represents a cost associated with performing a synchronization across PEs.

Let $\underline{k_l}$ denote the number of operations in the block of code associated with the leaf node l. Label the operations in the block of code associated with the leaf node l as $0,1,...,k_l-1$. For each leaf node l, define an array of continuous random variables denoted by X_{ij}^l, $i \in \{0,1,...,k_l-1\}$, $j \in \{0,1,...,N-1\}$, where N is the number of PEs executing that block. The value of the random variable $\underline{X_{ij}^l}$ corresponds to the execution time of operation i executing on PE j. In a mixed-mode machine, where all of the PEs are of the same architecture, it may be reasonable to assume that the probability distribution of the random variable X_{ij}^l is the same, regardless of the mode of execution of operation i. Examples of operations whose times will differ include statements that require the accessing of mono variables, for reasons discussed earlier, and inter-PE transfers,

where the software overhead for SPMD is much greater than for SIMD. To ease the notational burden, this possible distinction in the distribution of X_{ij}^l for SIMD and SPMD is not explicitly indicated, but implied.

The expected value of the random variable X_{ij}^l, i.e., $E\left[X_{ij}^l\right]$, is assumed to exist and is denoted by μ_{ij}^l. The variance of the random variable X_{ij}^l, i.e., $E\left[(X_{ij}^l - \mu_{ij}^l)^2\right]$, is assumed to exist and is denoted by $(\sigma_{ij}^l)^2$. Recall that the expected value of a function of a continuous random variable X, say $g(X)$, is defined by $E\left[g(X)\right] = \int_{-\infty}^{\infty} g(x) f_X(x) dx$, where $f_X(x)$ is the probability density function for the random variable X [Pap84].

For the purpose of this section, it is assumed that the random variables X_{ij}^l are mutually independent for all $i \in \{0,1,...,k_l-1\}$, $j \in \{0,1,...,N-1\}$. Furthermore, it is assumed that for each $i \in \{0,1,...,k_l-1\}$, the random variables X_{ij}^l are independent and identically distributed for all $j \in \{0,1,...,N-1\}$. Thus, for each value of i, μ_{ij}^l will be denoted as μ_i^l and $(\sigma_{ij}^l)^2$ will be denoted as $(\sigma_i^l)^2$.

The random variable for the execution time associated with implementing a block of k_l operations in SIMD mode is defined by the following transformation (the "sum of the maxs" referred to in Subsection 3.3):

$$X_l^{SIMD} = \sum_{i=0}^{k_l-1} \left[\max_{j \in \{0,1,...,N-1\}} \{X_{ij}^l\} \right].$$

The random variable for the execution time associated with implementing a block of k_l operations in SPMD mode is defined by the following transformation (the "max of the sums" referred to in Subsection 3.3):

$$X_l^{SPMD} = \max_{j \in \{0,1,...,N-1\}} \left\{ \sum_{i=0}^{k_l-1} X_{ij}^l \right\}.$$

The above formula assumes that the PEs are synchronized both when they enter and exit the block associated with leaf node l. If the block is preceded and/or followed by a SPMD block that does not require synchronization, then this formula represents a worst case estimate.

If the probability distribution for each X_{ij}^l is assumed to be known, then it is possible (although tedious) to determine the exact probability distribution for both X_l^{SIMD} and X_l^{SPMD}. For the purposes of the present section, only bounds for the expected values of these random variables will be determined. Upper bounds for $E\left[X_l^{SIMD}\right]$ and $E\left[X_l^{SPMD}\right]$ are derived next.

The expected value of X_l^{SIMD} is given by:

$$E\left[X_l^{SIMD}\right] = E\left[\sum_{i=0}^{k_l-1} \max_{j \in \{0,1,...,N-1\}} \{X_{ij}^l\}\right].$$

By the linearity of the E[·] operation, it follows that

$$E\left[X_l^{SIMD}\right] = \sum_{i=0}^{k_l-1} E\left[\max_{j \in \{0,1,...,N-1\}} \{X_{ij}^l\}\right]. \quad (1)$$

Because for each $i \in \{0,1,...,k_l-1\}$ the random variables X_{ij}^l are independent and identically distributed with mean μ_i^l and variance $(\sigma_i^l)^2$ for all $j \in \{0,1,...,N-1\}$, a standard result from order statistics [Dav81] can be applied to bound each term in the summation of Equation 1. In particular,

$$E\left[\max_{j \in \{0,1,...,N-1\}} \{X_{ij}^l\}\right] \le \mu_i^l + \sigma_i^l \frac{(N-1)}{(2N-1)^{1/2}}.$$

Thus, it follows that

$$E\left[X_l^{SIMD}\right] \le \sum_{i=0}^{k_l-1} \mu_i^l + \frac{(N-1)}{(2N-1)^{1/2}} \left[\sum_{i=0}^{k_l-1} \sigma_i^l\right]. \quad (2)$$

Next, an upper bound for $E\left[X_l^{SPMD}\right]$ is derived. Define $S_j^l = \sum_{i=0}^{k_l-1} X_{ij}^l$, for each $j \in \{0,1,...,N-1\}$. Thus, the random variable X_l^{SPMD} can be expressed as

$$X_l^{SPMD} = \max_{j \in \{0,1,...,N-1\}} \{S_j^l\}.$$

Because of the independence assumption, the expected value and variance of the random variables S_j^l are given by $\sum_{i=0}^{k_l-1} \mu_i^l$ and $\sum_{i=0}^{k_l-1} (\sigma_i^l)^2$, respectively. The upper bound result from [Dav81] can be applied to bound $E\left[X_l^{SPMD}\right]$, as follows:

$$E\left[X_l^{SPMD}\right] \le \sum_{i=0}^{k_l-1} \mu_i^l + \frac{(N-1)}{(2N-1)^{1/2}} \left[\sum_{i=0}^{k_l-1} (\sigma_i^l)^2\right]^{1/2}. \quad (3)$$

The upper bounds for $E\left[X_l^{SIMD}\right]$ and $E\left[X_l^{SPMD}\right]$ could be used to approximate the execution times T_l^{SIMD} and T_l^{SPMD} for large N, assumed to be given constants in the previous subsection. Determining the actual distributions for the random variables X_l^{SIMD} and X_l^{SPMD} could give a better indication of what the fundamental trade-offs are between the SIMD and SPMD modes. For instance, the exact value for the mean and variance of X_l^{SIMD} and X_l^{SPMD} could be computed. In certain applications, a moderate mean execution time with an associated small variance may be preferred over a smaller mean execution time with a relatively large

variance.

The above difference between SIMD and SPMD involves just one aspect of the relationship between these two modes of parallelism. There are other trade-offs, as discussed in Subsection 3.3, that must also be included in a determination of the best mode to use.

3.4.5. Effects of Block Juxtaposition

The total execution time for all children of a non-leaf node is estimated in Subsection 3.4.3 as the sum of the associated execution times. This estimate can be sharpened by considering the effects of juxtaposing blocks of code in either the SIMD or SPMD mode.

Consider first the case of estimating the total SIMD execution time for several sibling blocks. One factor that can have a significant effect is the overlapped execution of operations on the CU and PEs while in SIMD mode. A detailed analysis and explanation of CU/PE overlap, such as that presented in [KiN91], is beyond the scope of this section; however, the effect of CU/PE overlap can be accounted for in the present framework by using a simplified model. Consider a sequence of m blocks of code, say B_j, B_{j+1}, ... B_{j+m},where each block is a pure parallel code block or pure scalar code block. Let $Ov(B_j,B_{j+1}, \cdots B_{j+m})$, denote the amount of execution time overlap among the blocks. Thus, the total execution time for a sequence of m such blocks in SIMD mode is given by

$$T_{j,\ldots,j+m} = \sum_{i=j}^{j+m} T_i - Ov(B_j, \ldots, B_{j+m})$$

which represents the difference between the sum of the expected execution times of each of the blocks and the CU/PE overlap as a result of the juxtaposition of those blocks in SIMD mode.

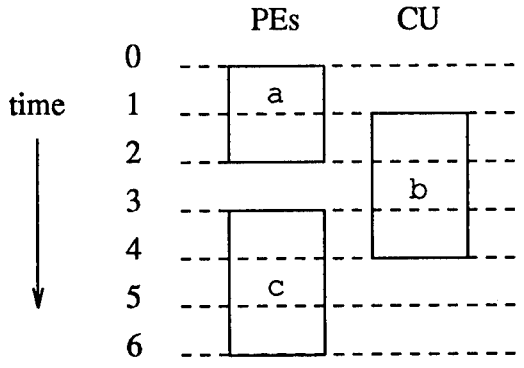


Figure 3.5: Overlapped execution of CU and PE instructions.

As an example, consider Figure 3.5, where two parallel blocks, `a` and `c`, are to be performed on the PEs, and block `b` is to be performed on the CU in SIMD mode. Blocks `a` and

b require a total of 5 time units; however, the CU and the PEs overlap execution for one time unit. Similarly, although blocks b and c require a combined total of 6 time units, they also overlap execution for one time unit. For this example, Ov(a,b,c) = 2, $T_{a,b,c} = T_a + T_b + T_c - Ov(a,b,c) = (2 + 3 + 3 - 2) = 6$.

In SPMD mode, PEs may need to perform some type of synchronization at the end of a block. The cost of synchronizing is related to the amount of processing performed since the previous synchronization.

For this section, $T^{SIMD}_{j,\ldots,j+m}$ and $T^{SPMD}_{j,\ldots,j+m}$ are approximated by the sum of the (known or estimated) individual block execution times, i.e., $T^{SIMD}_{j,\cdots,j+m} = \sum_{i=j}^{j+m} T_i^{SIMD}$ and $T^{SPMD}_{j,\cdots,j+m} = \sum_{i=j}^{j+m} T_i^{SPMD}$. Extending the probabilistic model of Subsection 3.4.4 to describe the probability distributions of the execution times of all types of leaf and non-leaf nodes is possible but not straightforward.

For example, consider the case of a sequence of two or more nodes to be implemented in SPMD mode. If no synchronization is required between nodes, then a smaller than expected execution time may result, because of the effect of the macro-level ''sum of maxs'' versus ''max of sums'' property described in Subsection 3.3. For a practical implementation, an extension of the concepts presented in Subsection 3.4.4 is needed.

3.4.6. Summary of Single-Mode Model

In this section, a model for determining the single mode of parallelism for which execution time is minimized has been described. Beginning with a program written in a mode-independent language, and given basic information about the execution time of the operations that are included in the language, useful execution time information for blocks within the program can be estimated. Employing a set of rules, block information is used to determine how fast a program will execute under either the SIMD or SPMD mode of parallel processing. This comparison technique can be used at compile time to determine which single-mode implementation on a mixed-mode machine would result in the minimum execution time.

3.5. Mixed-Mode Analysis

3.5.1. Assumptions and Definitions

In mixed-mode systems, it is possible for some portions of a single parallel algorithm to be implemented in SIMD mode, and other portions of the same program to be executed in SPMD mode. The programmer can select the mode of parallelism best-suited for each segment of the

program. The analysis of single-mode parallel programs developed in the previous subsection is expanded here for the implementation of data-parallel programs in SIMD/SPMD mixed-mode environments. In this section, an algorithm that may use both SIMD and SPMD modes will be referred as a mixed-algorithm.

The execution time estimation technique for mixed-algorithm programs is a generalization of the single-mode case. For the analysis in this section, several simplifying assumptions are made on the selection of modes within a program.

First, leaf blocks are assumed to be implemented completely in either SIMD mode or SPMD mode. Given that block boundaries are defined by control-flow and data-conditional constructs, synchronizations, inter-PE data transfers, and CU operations that can be overlapped in SIMD mode, there is no benefit in changing modes within a leaf block. Mode changes are therefore allowed only at inter-block boundaries.

Another assumption is that, if a block is to be executed more than once, i.e., as part of a looping construct, the mode of parallelism for that block is the same for all iterations. Because there are no branches or targets of branches within a block, and no inter-PE transfers or synchronization points within a block, there is no perceived advantage for the same block to execute in different modes from iteration to iteration; i.e., if a given mode is better for one instance of a block, it should be better for all instances.

Additionally, all blocks that comprise each data-conditional construct are to be implemented in the same mode of parallelism, i.e., for each `if` construct, all the blocks within the construct are either implemented in SIMD mode, or they are all implemented in SPMD mode. This assumption is necessary because mode changes within data-conditional constructs can translate into operations that cannot be implemented without excessive execution-time overhead. For example, consider an SPMD data-conditional construct with an embedded SIMD block. Because each PE independently performs a test to determine whether to perform the `then` clause or the `else` clause of a data-conditional construct, the CU cannot know which PEs are performing each clause. The CU therefore cannot know when all the appropriate PEs have arrived at the SIMD block, because some PEs may never execute the clause that contains the SIMD block. The situation can become worse when conditionals are nested. This type of problem led to the operational constraint of the mixed-mode model, given in Subsection 3.2, that all PEs must be in the same mode at a given point in time.

Finally, it is assumed that each iteration of a loop must begin and end execution in the same mode of parallelism. Consider, for example, a sequence of blocks that comprise the body of a loop, such that the first block is implemented in SIMD mode and the last block is implemented in SPMD mode. Between each successive iteration of the loop, a mode switch from SPMD to

SIMD is required to allow the PEs to perform the first block of the next iteration. Because the last block does not end in the same mode as the first block, a mode switch is added at the beginning of the loop body.

3.5.2. Optimal Selection of Modes for Mixed-Algorithms

To perform the execution time analysis and optimization for a mixed-algorithm, an SIMD/SPMD trade-off tree is constructed, where, as before, each block in the program is a leaf node and the data-conditional and control constructs form the interior nodes, with the root node representing the scope of the entire program. For mixed-algorithms, the mode of parallelism may be changed between adjacent sibling nodes, with an appropriate time cost. These mode changes are not nodes and are not represented in the SIMD/SPMD trade-off tree.

For the following discussion, let the ordered pair $(T^{SIMD}_{mixed\text{-}n}, T^{SPMD}_{mixed\text{-}n})$ represent the minimum mixed-algorithm execution time estimate for a non-leaf node n, where n is not the root node for the entire program, i.e., for a subtree with root n that begins and terminates execution in one of SIMD or SPMD, respectively, but may switch modes zero or more times during execution. The values for $T^{SIMD}_{mixed\text{-}n}$ and $T^{SPMD}_{mixed\text{-}n}$ include the time required for any mode switches that are performed. In the SIMD/SPMD trade-off tree, interior nodes can represent `if` constructs, `then` and `else` clauses, and `for` constructs. Recall that for non-leaf nodes corresponding to descendents of data-conditional constructs, mode changes are disallowed. Therefore, for nodes corresponding to `if`, `then`, and `else` nodes, $T^{SIMD}_{mixed\text{-}n}$ and $T^{SPMD}_{mixed\text{-}n}$ represent the estimated execution time for that node purely in SIMD or purely in SPMD, which can be determined using the techniques given in Subsection 3.4. Recall that for a sequence of nodes that are children of a `for` construct, the modes of the nodes can differ. However, the first and last nodes must be implemented in the same mode of parallelism or a mode switch must be added before the first node. Therefore, the mode of the `for` subtree is defined to be that of the last node that is a child of that `for` node. Thus, the only subtree that does not necessarily begin and terminate in the same mode is the node corresponding to the root of the entire program.

There is a cost $\underline{C^{SIMD}}$ associated with switching to SIMD mode, and a cost $\underline{C^{SPMD}}$ associated with switching to SPMD mode. The values of C^{SIMD} and C^{SPMD} are assumed to be known constants. The value of C^{SIMD} is determined by the execution time of the mode switching hardware and software, and does not include the time it takes between the first PE reaching the mode change synchronization point and the last PE reaching that point. Although the time to synchronize PEs is generally not a fixed constant because it depends on the amount of processing performed since the last synchronization, for the purposes of this framework a "typical" synchronization time is assumed to be included in the cost C^{SIMD} associated with

switching to SIMD mode. Explicitly including these times into the analysis relates to the discussion in Subsection 3.4.5 and is beyond the scope of this section.

For the case of a heterogeneous suite of parallel machines, C^{SIMD} and C^{SPMD} will typically not be constants and will generally be greater than for a single mixed-mode machine, because data may have to be moved between machines. The application of this technique under the relaxed assumption of non-constant mode-switching costs for a heterogeneous suite of parallel machines is currently under study [WaA94].

The methodology of Subsection 3.4.3 is adapted below for analyzing mixed-algorithms. The following steps are performed to determine the execution times for all nodes:

1) For each leaf l of the tree, assign an ordered pair (T_l^{SIMD}, T_l^{SPMD}), where T_l^{SIMD} is the SIMD execution time estimate for block l, and T_l^{SPMD} is the SPMD execution time estimate for block l, determined as in the single-mode analysis presented in Subsection 3.4.

Steps 2 and 3 are performed for each non-leaf node in the order of a depth-first traversal of the tree:

2) For each non-leaf node d corresponding to a data-conditional construct, assign an ordered pair $(T_{mixed\text{-}d}^{SIMD}, T_{mixed\text{-}d}^{SPMD})$, which represents the times for executing the data-conditional construct at node d, including the time to execute all the children of d. Analogous to the single-mode case presented in Subsection 3.4.3, an estimate for $(T_{mixed\text{-}d}^{SIMD}, T_{mixed\text{-}d}^{SPMD})$, is given by

$$T_{mixed\text{-}d}^{SIMD} = (1 - P_{else})T_{mixed\text{-}then}^{SIMD} + (1 - P_{then})T_{mixed\text{-}else}^{SIMD}$$

$$T_{mixed\text{-}d}^{SPMD} = p_{then}T_{mixed\text{-}then}^{SPMD} + (1 - p_{then})T_{mixed\text{-}else}^{SPMD} .$$

Because of the single execution mode constraint assumed for data-conditional constructs, $T_{mixed\text{-}then}^{SIMD} = T_{then}^{SIMD}$, $T_{mixed\text{-}then}^{SPMD} = T_{then}^{SPMD}$, $T_{mixed\text{-}else}^{SIMD} = T_{else}^{SIMD}$, and $T_{mixed\text{-}else}^{SPMD} = T_{else}^{SPMD}$, and the above equations reduce to the single-mode case. The formula used for $T_{mixed\text{-}d}^{SPMD}$ is the expected time for each PE to execute the data-conditional construct at node d, and not necessarily the maximum time taken over all PEs for a given execution. Thus, the value of $T_{mixed\text{-}d}^{SPMD}$ does not account for the time to synchronize the PEs for the case where the node following node d is executed in SIMD mode.

3) Case (a):

For each non-leaf node c corresponding to a control construct that is not in a subtree with a data-conditional construct as its root, assign an ordered pair $(T_{mixed\text{-}c}^{SIMD}, T_{mixed\text{-}c}^{SPMD})$, which represents the times for executing the control construct at node c, beginning and ending in SIMD and SPMD modes, respectively, including the time to

execute all the children of c. Let the ordered pair $(T^{SIMD}_{mixed\text{-}iteration}, T^{SPMD}_{mixed\text{-}iteration})$ represent the minimum mixed-algorithm execution time estimate for a single iteration of the loop corresponding to node c. Because the last block in the loop is the `for_test`, and because the first and the last block must be in the same mode of parallelism, $T^{SIMD}_{mixed\text{-}iteration}$ corresponds to performing the `for_test` in SIMD on the CU, and $T^{SPMD}_{mixed\text{-}iteration}$ corresponds to performing the `for_test` in SPMD on the PEs. Recall that Q is the number of iterations of the loop to be executed. Then $(T^{SIMD}_{mixed\text{-}c}, T^{SPMD}_{mixed\text{-}c})$ is given by

$$T^{SIMD}_{mixed\text{-}c} = Q \times T^{SIMD}_{mixed\text{-}iteration}$$

$$T^{SPMD}_{mixed\text{-}c} = Q \times T^{SPMD}_{mixed\text{-}iteration} \cdot$$

Recall that if the first node of a loop body and the `for_test` are in different modes, a mode switch is inserted before the first node in the loop body. This may lead to an unneeded mode switch for the first iteration. This situation is described in detail later.

Case (b):

If node c is the descendent of a node corresponding to an `if` construct, then only pure SIMD and pure SPMD implementations are considered, and the equations for the single-mode analysis methodology are used instead.

4) For the node representing the root, assign the ordered quadruple $(T^{SIMD/SIMD}_{root}, T^{SIMD/SPMD}_{root}, T^{SPMD/SIMD}_{root}, T^{SPMD/SPMD}_{root})$, where $T^{X/Y}_{root}$ corresponds to the minimum mixed-algorithm execution time required to perform all the children of the root node, beginning in mode X and ending in mode Y (where zero or more mode switches can occur between X and Y). Thus, it is possible for the root node to avoid a final mode change. The minimum quadruple value then represents the best mixed-mode implementation.

As the SIMD/SPMD trade-off tree is traversed, the deepest levels of the tree are combined by employing the above steps. As the analysis works its way up the tree, higher levels are combined, until only the root is represented. Then the parallel mode for each segment of the program can be assigned.

One effect that must be considered when traversing the tree is the possibility of adding unnecessary mode switches when going from the `for_init` block to the first child in the loop body of a `for` node. For example, consider a `for` construct such that the `for_init` block is implemented in SPMD, the first child in the loop body of the `for` node is implemented in SPMD, and the `for_test` block (the last child of the `for` node) is implemented in SIMD, as illustrated in Figure 3.6a. Because the last and first nodes are performed one after the other when iterating over the loop, a mode switch from SIMD to SPMD is required before executing the first node. Additionally, by the definition of a `for` loop node, the mode of the `for` node is

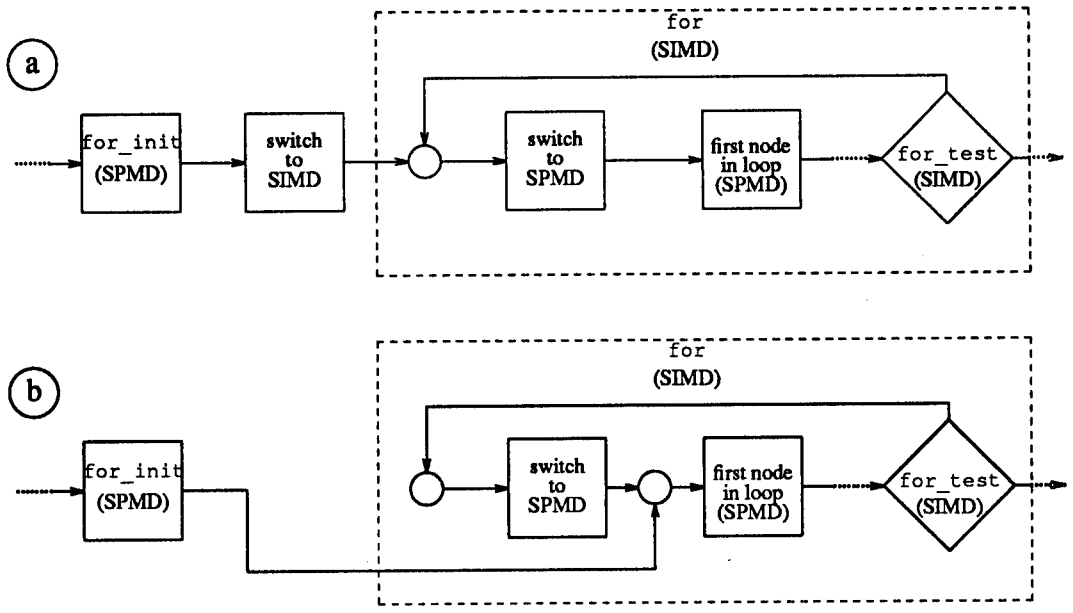


Figure 3.6: Modification of `for` loop to avoid unnecessary mode switches for the first iteration. (a) With unneeded mode switch. (b) Without unneeded mode switch.

the same as that of the `for_test` in the loop (as stated in Subsection 3.5.1). Thus, a mode switch from the SPMD `for_init` node to the SIMD `for` node is inserted before the first iteration of the loop body. However, immediately after the execution of the `for_init`, and before the first iteration, the parallel system is already in SPMD mode, which is the mode of parallelism required for the first node in the loop body. By detecting this case, the mode switch from SPMD to SIMD and from SIMD back to SPMD can be avoided for the first iteration, as illustrated in Figure 3.6b. Thus, it is beneficial for the analysis to recognize this case, and to remove mode switches that are not needed for the first iteration of the loop from the time-cost estimate.

One element of the analysis not addressed, which should be included in a practical implementation of the techniques presented here, is the effect of implementing a sequence of two or more nodes in SPMD mode. Recall from Subsection 3.4.5, if an SPMD node is directly followed by another SPMD node with no intervening synchronization, then a reduced execution time may result, due to the macro-level "sum of maxs" versus "max of sums" effect described in Subsection 3.3. This effect is particularly applicable to the estimated execution time of `for` constructs. Thus, for a practical implementation, an extension of the concepts presented in Subsection 3.4.4 is needed, where such adjacent SPMD nodes are treated as a single node for the purposes of the type of analysis discussed in that subsection.

3.5.3. Computational Aspects of Optimal Selection of Modes for Mixed-Algorithms

Exhaustively testing all possible implementations to find the minimum mixed-algorithm execution time for a node with m children would require testing 2^m combinations. For nodes with many children, this approach is impractical; therefore, an efficient method of finding the minimum execution time is needed when m becomes large.

An efficient way of computing ($T^{SIMD}_{mixed\text{-}n}$, $T^{SPMD}_{mixed\text{-}n}$) is to transform the subtree rooted at n into a multistage optimization problem. In multistage optimization problems, proceeding to stage $j+1$ is possible only by passing through stage j. There is a cost associated with proceeding from stage j to stage $j+1$, depending on the initial state (at stage j) and the final state (at stage $j+1$). In a multistage optimization graph, each state in each stage is represented by a vertex, and edges connecting vertices in stage j to vertices in stage $j+1$ indicate valid transitions, each with an associated cost. The goal of a multistage optimization problem is to find the minimum cost path between the initial and final stages, e.g., see [AnT91, BrH75].

Multistage optimization problems can be solved using Moore's algorithm [Moo57] (a variant of Dijkstra's algorithm [Dij59]) in $s-2$ iterations for an s-stage problem. In each iteration, two successive stages of the multistage optimization graph are reduced to a single stage, so that at the end of $s-2$ iterations, only the initial and final stages remain, with connecting edges indicating the minimum cost from each of the states in the initial stage to each of the states in the final stage.

An example of a general multistage optimization problem and its solution is illustrated in Figure 3.7. Initially, there are four stages, numbered from 0 to 3. In the first iteration, for each vertex g in stage 0, a minimum path is found from g to each vertex h in stage 2, passing through stage 1. Stage 1 is then removed from the graph, and new edges are drawn from vertices in stage 0 to vertices in stage 2, indicating the minimum cost to proceed from stage 0 to stage 2 for each possible (g,h) pair. The multistage optimization algorithm is applied again, and the problem is reduced to two stages, indicating the minimum cost to proceed from each initial state to each final state. By recording the minimum paths selected during each iteration of the algorithm, the path through the entire multistage problem resulting in the minimum cost for each initial/final pair is obtained. The correctness of the optimization algorithm is based on the principle that all sub-paths along an optimal (i.e., shortest) path must themselves be optimal.

To find the minimum execution time for all the children of a node in the SIMD/SPMD trade-off tree, let a stage and the edges exiting that stage in a multistage optimization graph correspond to a single child node. Let each mode of parallelism represent a separate state within each stage. Let the cost associated with each edge correspond to the cost of performing that node in SIMD mode or SPMD mode. Then, before the representation of each node in the multistage

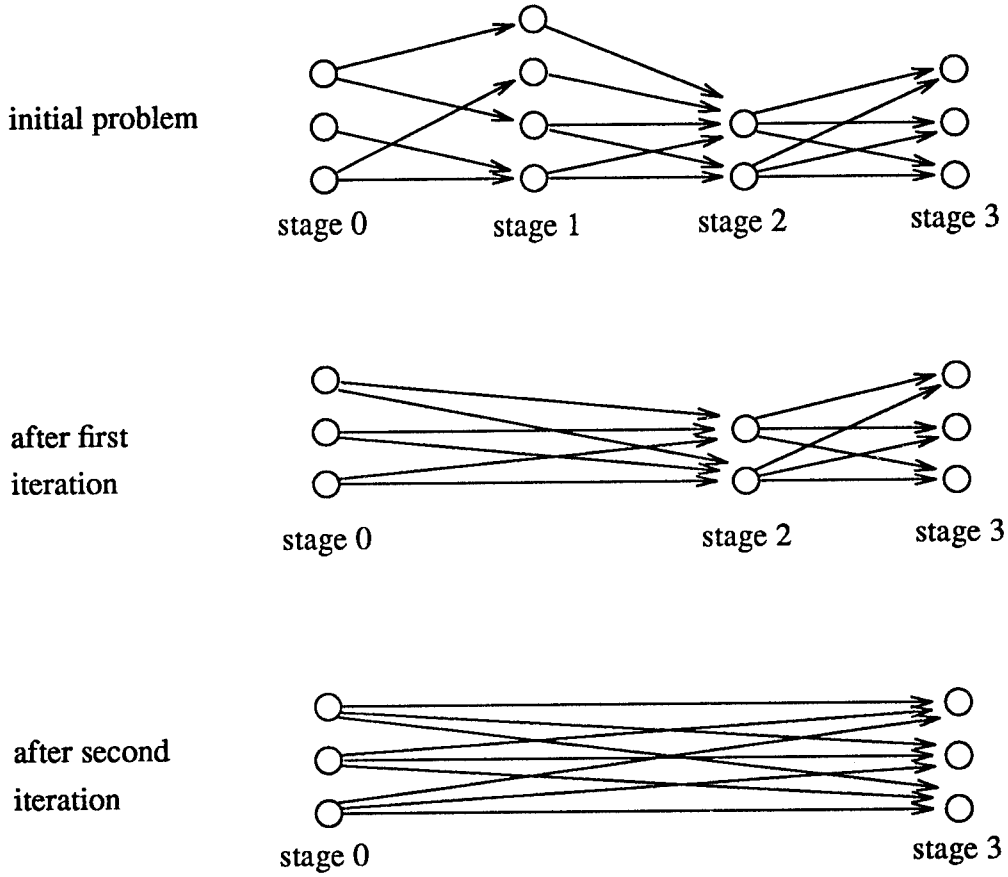


Figure 3.7: Example of a general multistage optimization problem and the aggregate structure of the intermediate and final solution graphs.

graph, include a stage and associated edges representing the cost of a possible mode switch between nodes.

As an example, consider the transformation illustrated in Figure 3.8a. To find the minimum mixed-algorithm execution time for a sequence of sibling nodes, each node is modeled as three stages of a multistage graph (numbered 0 to 2 in Figure 3.8a) and then joined together to form a multistage optimization graph for the entire sequence of nodes. In each stage, the upper vertex represents SIMD mode, while the lower vertex represents SPMD mode. For stage 0 and its associated edges, a possible mode change is represented. The edge from the upper vertex in the stage 0 to the lower vertex in stage 1 is labeled with the cost of switching from SIMD to SPMD mode. Similarly, the cost of switching from SPMD to SIMD is indicated as the label for the edge from the lower vertex in stage 0 to the upper vertex in stage 1. There is zero cost for remaining in the same mode of parallelism. The edges from stage 1 to stage 2 in Figure 3.8a represent the cost of executing the node in each mode. Figure 3.8b illustrates transforming an

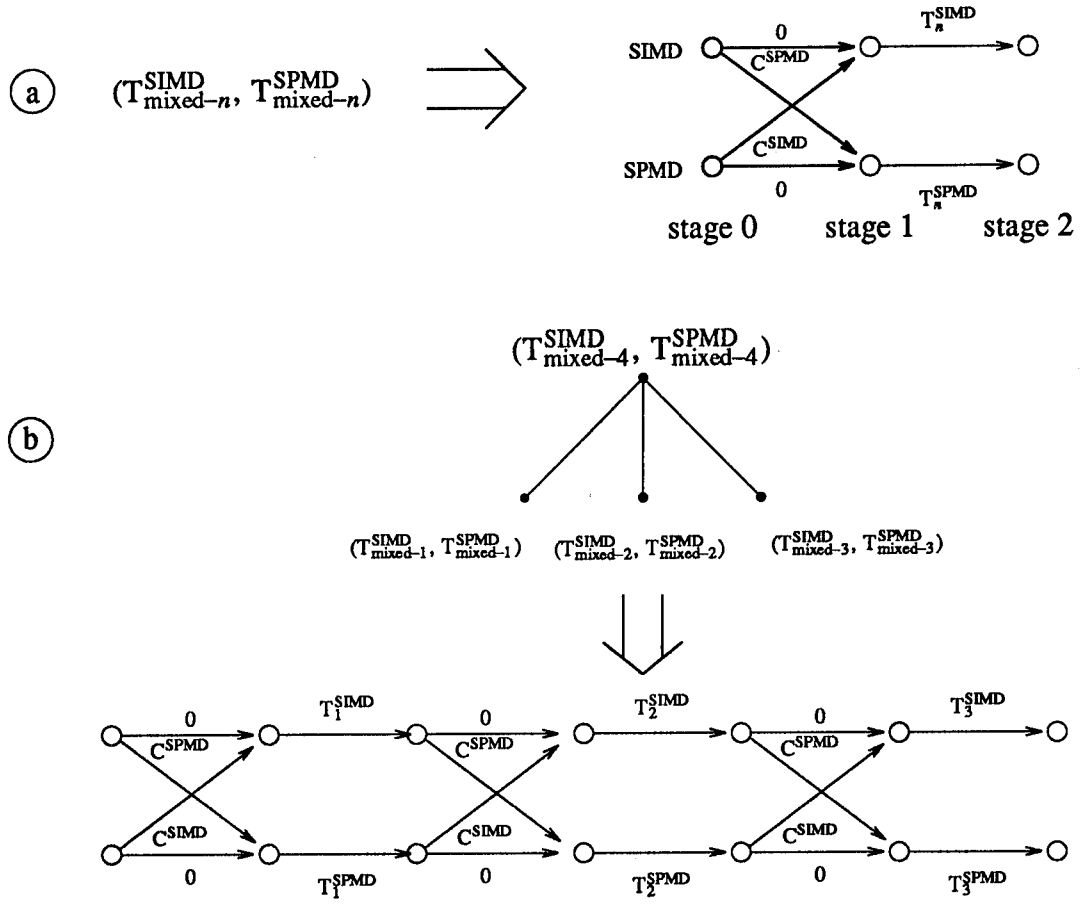


Figure 3.8: Transformation from flow-analysis tree to multistage graph. (a) Transforming a node into two stages. (b) Transforming a node with three children into a multistage graph.

SIMD/SPMD trade-off tree with three children into a multistage optimization graph. The ordered pair $(T^{SIMD}_{mixed-4}, T^{SPMD}_{mixed-4})$ is obtained from the solution of the multistage optimization problem shown in Figure 3.8b.

A node with m children is represented by a multistage optimization problem with $2m + 1$ stages: m stages to represent the nodes, m stages to represent possible mode switches, and one stage to represent the final states at the end of the multistage optimization graph. Thus, $2m + 1 - 2 \equiv 2m$ iterations of Moore's algorithm are required to find a solution. For each iteration corresponding to a reduction of the solved portion of the graph with a possible mode switch, $2^2 = 4$ comparisons and $2^2 = 4$ assignments are needed, while for each iteration corresponding to a reduction of the solved portion of the graph with a node execution, no

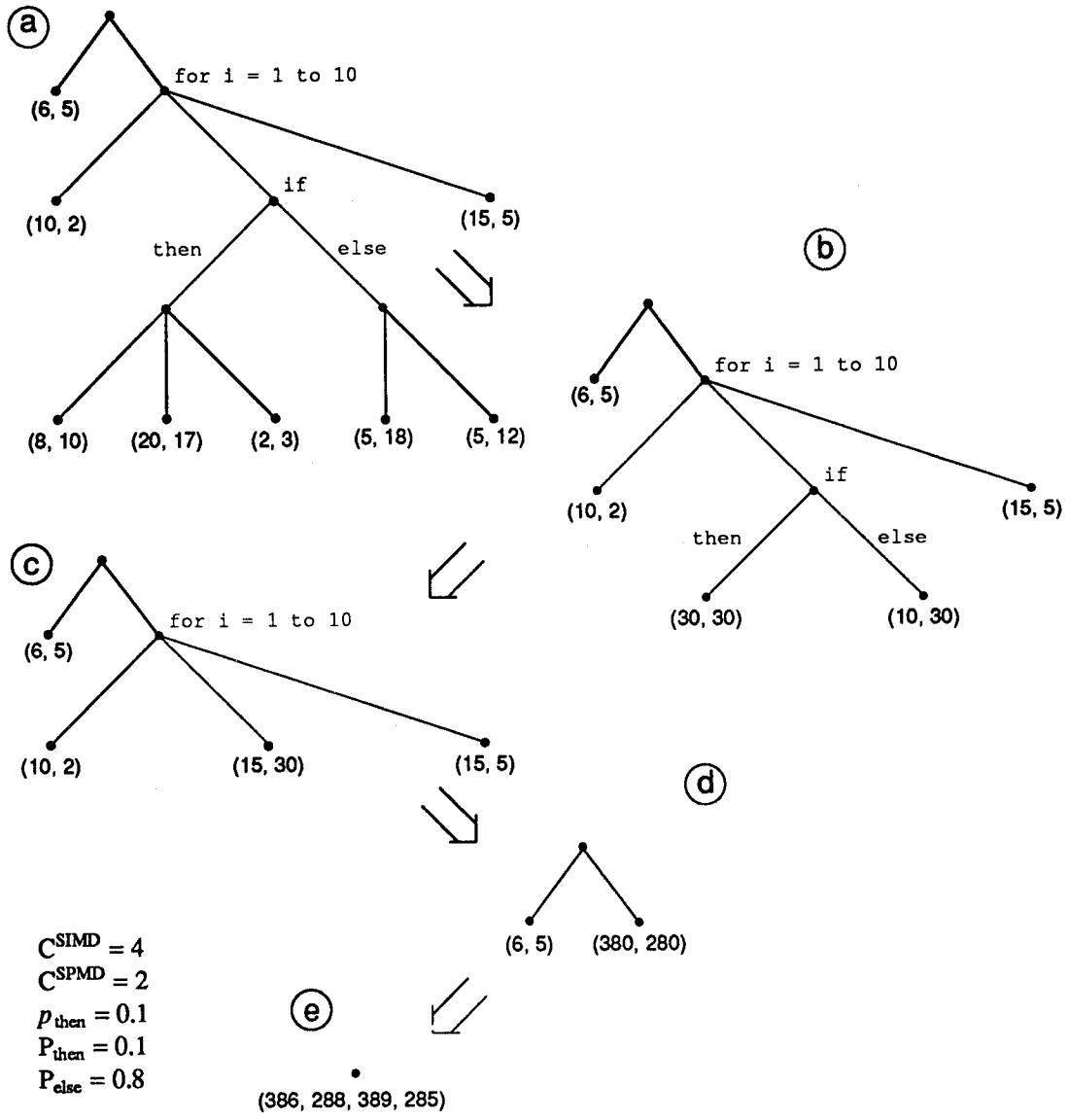


Figure 3.9: Mixed-algorithm analysis example for a parallel code segment. (a) Time assignments made for leaf blocks. (b) Time assignments for `then` and `else` nodes calculated. (c) Time assignments for `if` node calculated. (d) Time assignments for `for` loop calculated (except for initialization). (e) Time assignments for root node calculated.

comparisons and $2^2 = 4$ assignments are needed. Thus, finding the minimum mixed-algorithm execution time by this approach has a sequential time complexity of $8m + 4m = 12m = O(m)$ time.

By recording the minimum paths selected at each iteration, the mode of parallelism for each stage is selected. Each edge in the minimum-cost path selected corresponds to performing a node in SIMD, performing a node in SPMD, performing a mode switch, or staying in the same mode. When the multistage optimization is completed for all the children of a node in the SIMD/SPMD trade-off tree, the execution time information is used to generate the execution time for the parent node. The multistage approach is then applied to the parent node and its siblings. In this way, the analysis works up the trade-off tree until execution costs are obtained for the root node.

3.5.4. An Example of Optimal Mixed-Algorithm Selection

As an example of an optimal assignment of modes to the segments of a data-parallel program, consider the SIMD/SPMD trade-off tree in Figure 3.9, composed of a `for` loop, where one of the nodes of the loop is an `if` construct (the first child of the root node is the `for_init` for the `for` loop). For this example, let $Q = 10$, $C^{SIMD} = 4$, and $C^{SPMD} = 2$. For the `if` construct, assuming that the outcome of the conditional tests are not mutually independent, let $p_{then} = 0.1$, $P_{then} = 0.1$, and $P_{else} = 0.8$.

In Figure 3.9a, the SIMD/SPMD trade-off tree for the program is shown, where the ordered pairs for the leaf nodes have been determined. After the first transformation, the leaf nodes for the `then` clause have been combined to a single node, illustrated in Figure 3.9b. Because the `then` node is part of a data-conditional construct, the ordered pair $(T^{SIMD}_{mixed\text{-}then}, T^{SPMD}_{mixed\text{-}then})$ is simply the ordered sum of its children, $(8 + 20 + 2, 10 + 17 + 3) = (30, 30)$. Similarly, for the `else` clause, the ordered pair $(T^{SIMD}_{mixed\text{-}else}, T^{SPMD}_{mixed\text{-}else})$ is $(5 + 5, 18 + 12) = (10, 30)$.

In Figure 3.9c, an ordered pair for the `if` construct can then be found, as indicated in step 2. Specifically,

$$T^{SIMD}_{mixed\text{-}if} = (1 - P_{else})T^{SIMD}_{mixed\text{-}then} + (1 - P_{then})T^{SIMD}_{mixed\text{-}else} = (0.2 \times 30) + (0.9 \times 10) = 15$$

$$T^{SPMD}_{mixed\text{-}if} = p_{then}T^{SPMD}_{mixed\text{-}then} + (1 - p_{then})T^{SPMD}_{mixed\text{-}else} = (0.1 \times 30) + (0.9 \times 30) = 30$$

The ordered pair $(T^{SIMD}_{mixed\text{-}for}, T^{SPMD}_{mixed\text{-}for})$ can then be found using the multistage optimization approach and step 3. Using the multistage optimization approach for the children of the `for` construct, it is determined that $T^{SIMD}_{mixed\text{-}iteration}$, the minimum execution time for a single iteration beginning and ending in SIMD, is obtained by implementing the first block in SPMD mode, the

`if` construct in SIMD mode, and the last block in SIMD mode. There will need to be a mode switch before the `if` node (C^{SIMD}), and before the first node of the loop body (C^{SPMD}). Thus,

$$T^{SIMD}_{mixed\text{-}iteration} = C^{SPMD}\ (=2) + (10, \underline{2}) + C^{SIMD}\ (=4) + (\underline{15}, 30) + (\underline{15}, 5)$$

$$= 2 + 2 + 4 + 15 + 15 = 38\,.$$

The multistage optimization approach also determines that for $T^{SPMD}_{mixed\text{-}iteration}$, the first block is implemented in SPMD mode, the `if` construct in SIMD mode, and the last block in SPMD mode:

$$T^{SPMD}_{mixed\text{-}iteration} = (10, \underline{2}) + C^{SIMD}\ (=4) + (\underline{15}, 30) + C^{SPMD}\ (=2) + (15, \underline{5})$$

$$= 2 + 4 + 15 + 2 + 5 = 28\,.$$

Applying step 3 yields:

$$T^{SIMD}_{mixed\text{-}for} = Q \times T^{SIMD}_{mixed\text{-}iteration} = 10 \times 38 = 380$$

$$T^{SPMD}_{mixed\text{-}for} = Q \times T^{SPMD}_{mixed\text{-}iteration} = 10 \times 28 = 280\,.$$

This is shown in Figure 3.9d.

By applying step 4, using the multistage optimization approach, and including mode switch costs the values for the root node are obtained:

$$(T^{SIMD/SIMD}_{root}, T^{SIMD/SPMD}_{root}, T^{SPMD/SIMD}_{root}, T^{SPMD/SPMD}_{root}) = (386, 288, 383, 285)\,.$$

This is shown in Figure 3.9e.

To determine the value of $T^{SIMD/SPMD}_{root}$, the first node in Figure 3.9d, representing the `for_init`, requires a time of 5 in SIMD. The second node, representing a `for` loop to be executed in SPMD, requires 380. However, the calculation of the value of 380 for the body of the `for` loop includes the cost of switching from SIMD to SPMD, needed at the beginning of the loop body. This mode switch can be bypassed, as discussed in Subsection 3.5.2 and illustrated in Figure 3.6, saving 2 time units. Thus, $T^{SIMD/SPMD}_{root} = 5 - 2 + 380 = 383$.

For comparison, the estimated execution time for a pure SIMD or pure SPMD implementation for this example is 406 and 375, respectively.

The mode of parallelism for each portion of the tree can then be chosen, as illustrated in Figure 3.10. For this example, the first child and the last child of the `for` node are implemented in SPMD mode. Because the last child of the `for` node is implemented in SPMD, by definition the `for` construct is implemented in SPMD. The `if` node and all its descendents are implemented in SIMD mode.

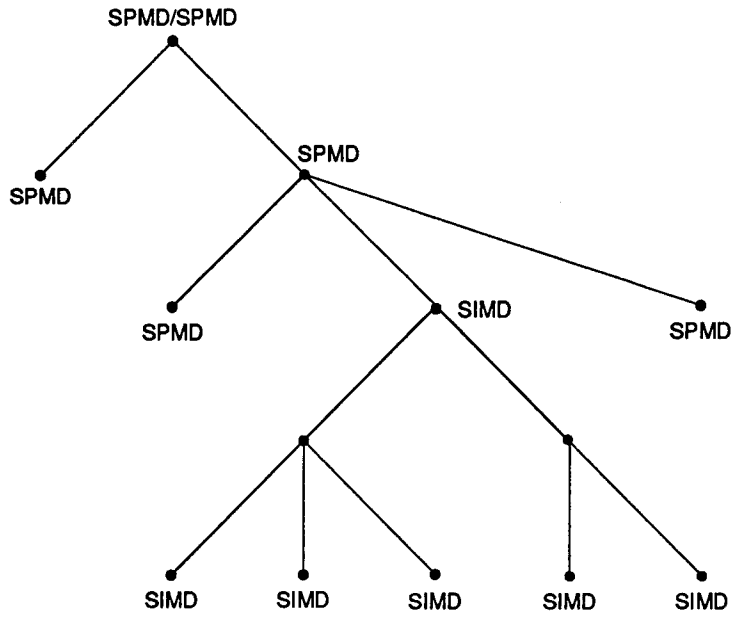


Figure 3.10: Selection of parallel modes for the example of Figure 9.

3.6. Summary and Future Work

The block-based mode selection model proposed in this section establishes a framework on which to build static source-code analysis techniques for the selection of parallel modes in a mixed-mode context. The model consists of an algorithm for determining information in an SIMD/SPMD trade-off tree, which is then combined using a set of rules and by employing a multistage optimization technique to determine the minimum mixed-algorithm execution time for a sequence of nodes. With this approach, parallel programs written in a mode-independent language can be executed on a mixed-mode machine with each program segment using the most appropriate mode of parallelism for minimum total program execution time.

There are various extensions to the model that form the basis for future work. The set of control and data-conditional constructs can be expanded to include other useful constructs, e.g. `while` statements, `case` statements, and function calls. The probabilistic model introduced in this section can be enhanced to consider more fully the effect of juxtaposed blocks on overall execution time. The framework can also include a more complete model of CU/PE overlap for SIMD operation. Other research may involve more practical aspects of the analysis, for example, the details of incorporating the techniques presented here into a parallel compiler. Methods for estimating the parameters used in the model, when they are not deterministic, must

be developed.

For a practical implementation in a heterogeneous environment composed of a suite of parallel machines, the time to move data among machines will not be a constant for all blocks, and decisions at one point will impact the quantity future data movements (and thus the amount of time required for inter-machine data transfers). Therefore, an important extension to this study is to examine the case where the cost of switching machines/modes is not constant, but depends on the size, location, and usage of data within the program [WaA94].

Another research area that is the subject of future work is the impact on the analysis of including computation models other than SIMD and SPMD, e.g., MIMD and vector processing. The incorporation of other parallel/vector models would form the basis for future efforts to provide programming tools for heterogeneous systems.

4. Static Program Decomposition Among Machines in an SIMD/SPMD Heterogeneous Environment with Non-Constant Mode Switching Costs

4.1. Introduction

One of the benefits of heterogeneous parallel processing is that programs can be executed on those machines that best exploit the parallelism in each part of the program. One of the associated challenges of implementing heterogeneous systems is finding a mapping of program segments to parallel machines. In [WaS93], a Block-Based Mode Selection (BBMS) framework for estimating the relative execution time of a data-parallel algorithm in an environment capable of the SIMD and SPMD (Single Program - Multiple Data) modes of computation was presented, where SPMD is MIMD with the restriction that all PEs are executing copies of the same program. The BBMS framework provides a methodology whereby static source-code based decisions of computational mode of execution can be made for each portion of an algorithm. One of the assumptions of the framework in [WaS93] is that the cost of performing mode changes in a heterogeneous system is constant.

In the model of heterogeneous computing assumed here, each segment in a program has a set of zero or more data elements that must be available to it (i.e., on the same machine where the program segment is to be executed). Because each data element may be used and/or generated by other program segments, and because program segments may execute on different machines within the system, it may be necessary to transfer data elements needed by a segment before execution can begin. The transfer cost is assumed to be proportional to the size and number of the data elements transferred. Thus, the cost of switching between machines during the execution of a program is not a constant cost, but depends on data transfers needed as a result of the switch.

Because the mode-switching costs are non-constant, determining a minimum-cost assignment of machines to program segments is not straightforward. The cost of executing a given program segment depends on previous machine selections and associated data transfers made in the past. Exhaustively testing each possible mapping of program segments to machines provides a minimum-cost schedule, but because the number of possible mappings grows

---

The co-authors of this section were Daniel W. Watson, John K. Antonio, Howard Jay Siegel, and Mikhail J. Atallah.
This research was supported by the Office of Naval Research under grant number N00014-90-J-1937, Rome Laboratory under contract numbers F30602-92-C-0150 and F30602-94-C-0022, and the National Science Foundation under grant number CCR-9202807.
This material appeared in the *Proceedings of the Third Heterogeneous Computing Workshop (HCW '94)*, sponsored by the IEEE Computer Society, April 1994, pp. 58-65.

exponentially with the number of segments, this approach is not computationally feasible.

The BBMS approach, which provides the optimal sequence of modes under the constant switching cost assumption, is used here as a basis for a heuristic method for associating parallel machines with data-parallel program segments in an environment with non-constant switching costs. The heuristic provides an efficient approach for selecting near-optimal mappings of program segments to machines. Simulation results based on randomly generated parallel program behaviors indicate that good assignments are generally provided by the heuristic.

The rest of this section is organized as follows. Underlying assumptions and basic terms are defined in Subsection 4.2. Subsection 4.3 overviews the BBMS framework and isolates the specific area of investigation. Also included in Subsection 4.3 is an explanation of why the BBMS must be extended for the case of data-location (i.e., non-constant) machine-switching costs. In Subsection 4.4, a heterogeneous program execution model is introduced, a static heuristic developed, and a series of simulations is presented to validate the approach.

4.2. Machine and Language Model

In a heterogeneous system [Fre91], different types of parallel machines can be used to perform a single task. One example of a heterogeneous environment is a mixed- mode machine [SiA92b], a single machine which is capable of operating in either the SIMD or MIMD mode of parallelism and can dynamically switch between modes at instruction-level granularity with relatively small overhead, e.g. PASM [ArW93], Triton [PhW93], and OPSILA [AuB86] (limited to SIMD/SPMD). Another type of heterogeneous system is a suite of independent machines of different types interconnected by a high-speed network, referred to as a mixed-machine system. Unlike mixed-mode systems, switching execution among machines in a mixed-machine system requires measurable overhead because data may need to be transferred among machines.

Mixed-machine systems considered here are assumed to have high-speed connections among machines that make decomposition at the sub-program level feasible. The emphasis for this study is on machines interconnected using a current or future high-speed network. Whenever the term "SPMD machine" is used, recall that any MIMD machine can operate as an SPMD machine.

Each system in a mixed-machine suite is assumed to have a physically distributed memory. Thus, each processor in the system is paired with a memory, forming a processing element (PE).

Applications for this study are assumed to be data parallel [HiS86], i.e., the data for a program is distributed among the PEs, and the algorithm exhibits a high degree of uniformity across the data [Jam87]. It is assumed for mixed-mode machines that all PEs are in the same

mode at any point in time (i.e., SIMD versus SPMD). Similarly for mixed-machine systems, a job is actively executed on only one machine at a time (i.e., only one SIMD or SPMD machine is being used for the program at any one time). It is assumed for this study that all machines in the system are available for the execution of any program segment. All PEs in a given machine must be synchronized at program segment boundaries before an inter-machine transfer can occur. Although the focus here is on SIMD versus SPMD machines, the approach can also be used to select among two or more machines of the same class (e.g., among several different SIMD machines).

One way of programming machines that are capable of mixed-mode and mixed-machine parallelism is by using a mode-independent language, i.e., a language whose syntactic elements have interpretations under more than one mode of parallelism. An example of a mode-independent language is the Explicit Language for Parallelism (ELP), currently under development at Purdue University [NiS93]. The ELP syntax is based on C, and allows the programmer to specify SIMD, MIMD, SPMD, and mixed-mode operation within a program. A goal of ELP is to provide uniformity with respect to the SIMD and SPMD modes of parallelism by having interpretations for all the elements of the syntax within both of these modes that are identical in semantics. This is an important characteristic because it allows a data-parallel algorithm to be coded in a mode-independent manner, producing a data-parallel program for which: (1) only SIMD code is generated, (2) only SPMD code is generated, or (3) execution mode specifiers can be added to facilitate mixed-mode experimentation. Other related unifying parallel language studies include [BrA89], [PhW93], and [ZiB88].

4.3. The BBMS Framework

4.3.1. Framework Overview

To determine the best machine for each portion of a parallel algorithm, the program is first transformed into a flow–analysis tree, which is a tree whose structure represents the scope levels within the algorithm. The flow-analysis tree is then used to create a trade-off tree that has a structure identical to the flow-analysis tree, but indicates the machine in a mixed-machine system on which to execute each portion of the program. For both trees, the root node represents the scope of the entire program, the non-leaf nodes correspond to control constructs and data-conditional constructs, and the leaf nodes of the tree represent parallel or scalar code blocks.

Code blocks are the parallel equivalent of the serial basic blocks described in [AhS86]. Blocks of code are identified by their leading statements, called leaders. The first statement in a program is a leader, any statement that becomes the target of a conditional or unconditional

branch at the machine code level is a leader, and any statement that follows a conditional branch at the machine code level is a leader. In the approach described here, in addition any statement requiring synchronization and any statement that follows a statement requiring a synchronization is a leader, and any statement requiring an inter-PE data transfer and any statement that follows an inter-PE data transfer is a leader. This is an important distinction from the serial definition of a basic block, because synchronization points within an algorithm indicate when PEs are idle, waiting for one or more other PEs to arrive at the synchronization point.

Initially, only information about the execution times of the leaf nodes is assumed to be known. The technique proposed in [WaS93] combines this known information to arrive at comparative mixed-mode execution times for the intermediate nodes, using a limited set of language constructs. Intermediate nodes, which correspond to looping and data-conditional constructs, are implemented such that they begin and terminate execution in either SIMD or SPMD, but may switch modes zero or more times during execution. For non-leaf nodes corresponding to descendents of data-conditional constructs, mode changes are disallowed, because these changes can translate into operations that cannot be implemented without excessive execution time overhead [WaS93]. For a sequence of nodes that are children of a looping construct, the modes of the nodes can differ. However, the first and last nodes must be implemented in the same mode of parallelism or a mode switch is added before the first node. Thus, the only subtree that does not necessarily begin and terminate in the same mode is the node corresponding to the root of the entire program.

It is assumed that both the SIMD and SPMD execution times associated with each leaf node are known constants. The case where execution times of the operations are assumed to be random quantities with known probability distributions (which impacts the distribution of execution time of combined adjacent blocks) is a consideration that will be examined in future studies. In [WaS93] and in this paper, iterative loop bounds and information regarding the probability of data-conditional outcomes is assumed known or estimated statistically.

Figure 4.1 illustrates how a program is transformed into a preliminary flow-analysis tree, and then used to generate a trade-off tree (details are in [WaS93]). At each level in the tree, the sibling blocks are represented in the same order (left to right) as they appear in the program. In the final trade-off tree, only machine selection information is retained.

In mixed-mode and mixed-machine systems, it is possible for some portions of a single parallel program to be implemented in one mode or machine, and other portions of the same program to be executed in a different mode or machine. This is referred to as a mixed-algorithm. Leaf blocks are assumed to be implemented completely in one mode/machine. Given the definition of a block, it is assumed that there is no benefit in changing modes except at

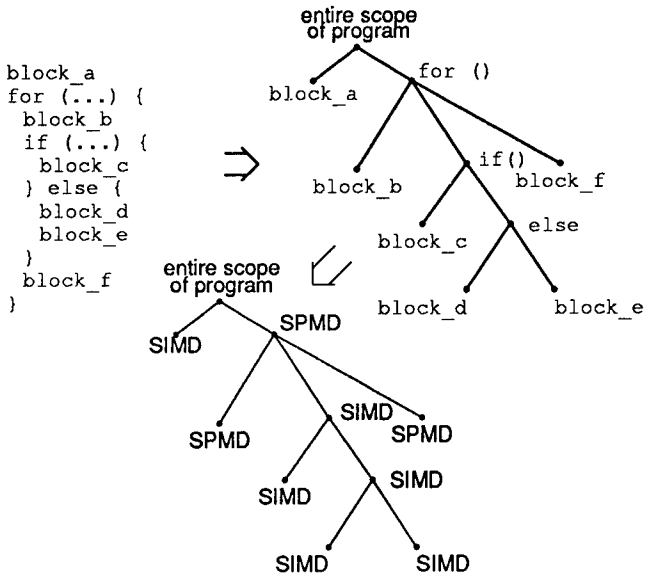


Figure 4.1: Example of transformation from program to flow-analysis tree to trade-off tree.

block boundaries.

4.3.2. With Constant Switching Costs

Consider the mixed-mode case, where a node may be executed in either SIMD or SPMD mode, and there is a cost (assumed for the moment to be constant) associated with switching from one mode of parallelism to the other. One portion of the block-based framework developed for this case is the approach taken to reduce a set of children in the flow-analysis tree under a common parent so that it is represented by a single node at the location of the parent node in the original tree. An ordered pair (T_i^{SIMD}, T_i^{SPMD}) is assigned to each leaf block, representing the execution time required to perform block i in SIMD or SPMD mode, respectively. Leaf nodes with a common parent are executed in order from leftmost node to rightmost node. For each non-leaf node n within the flow-analysis tree, (T_n^{SIMD}, T_n^{SPMD}) corresponds to the time required to perform the entire subtree rooted at n beginning and ending in either SIMD or SPMD mode, with no restriction on the number of mode changes that occur within the subtree.

Because there are two choices of mode for each of m children of a parent node, there are 2^m possible ways to execute the sequence of children nodes. Exhaustively testing all possible implementations to find the minimum mixed-algorithm execution time for a node with m children would therefore require testing 2^m combinations. For nodes with many children, this approach is impractical. A non-exponential method of finding the minimum execution time is

needed when m is large.

One efficient way of computing (T_n^{SIMD}, T_n^{SPMD}) is to transform the m children of node n into a multistage optimization problem. In multistage optimization problems, proceeding to stage $j+1$ is possible only by passing through stage j. There is a cost associated with proceeding from stage j to stage $j+1$ dependent on the initial state (at stage j) and the final state (at stage $j+1$). In a multistage optimization graph, each state in each stage is represented by a vertex, and edges connecting vertices in stage j to vertices in stage $j+1$ indicate valid state transitions, each with an associated cost. The goal of a multistage optimization problem is to find the minimum cost path between the initial and final stages, e.g., see [AnT91, BrH75].

Multistage optimization problems can be solved using Moore's algorithm [Moo57] (a variant of Dijkstra's algorithm [Dij59]) in $s-2$ iterations for an s-stage problem. In each iteration, two successive stages of the problem are reduced to a single stage, so that at the end of $s-2$ iterations, only the initial and final stages remain, with connecting edges indicating the minimum cost from each of the states in the initial stage to each of the states in the final stage. By redrawing the set of m children nodes of a parent as a multistage optimization problem, the minimum cost mapping of parallel modes to m leaves can be found for the corresponding ordered pairs and mode switching costs in $O(m)$ time.

An example of a general multistage optimization problem and its solution is illustrated in Figure 4.2. Initially, there are four stages, numbered from 0 to 3. In the first iteration, for each vertex in stage 0, a minimum path is found to each vertex in stage 2 (passing through stage 1). Stage 1 is then removed from the graph, and new edges are drawn from vertices in stage 0 to vertices in stage 2, indicating the minimum cost to proceed from stage 0 to stage 2 for each possible source/destination pair. This reduction procedure is applied again, and the problem is reduced to two stages, indicating the minimum cost to proceed from each initial state to each final state.

By recording the minimum paths selected during each iteration of the algorithm, the path through the entire multistage graph resulting in the minimum cost for each initial/final pair is obtained. To find the minimum execution time for all the children of a node in the SIMD/SPMD trade-off tree, let a stage and the edges exiting that stage in a multistage optimization graph correspond a single child node. Let each mode of parallelism represent a separate state within each stage. The cost associated with each edge corresponds to the cost of performing that node in SIMD mode or SPMD mode. Then, before the representation of each child node in the multistage graph, include a stage and associated edges representing the cost of a possible mode switch between nodes. If the mode switching cost is assumed to be constant ($\underline{C^{SIMD}}$ and $\underline{C^{SPMD}}$), then all the arc weights in the multistage graph are independent of past decisions, and

the minimum cost path can be determined.

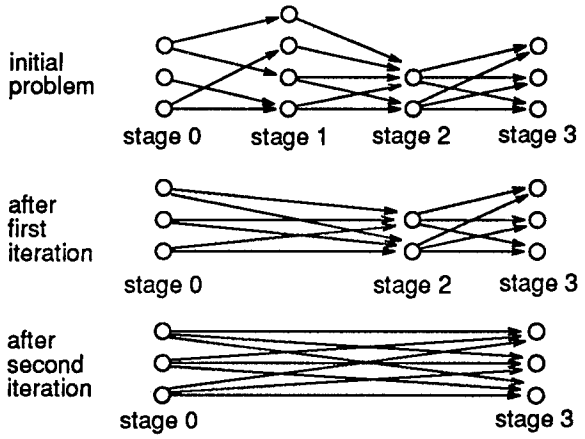


Figure 4.2: Example of a general multistage optimization problem and its solution.

As an example of the transformation of a flow-analysis subtree into into a multistage optimization problem, consider the transformation illustrated in Figure 4.3(a). To find the minimum mixed-algorithm execution time for a sequence of sibling nodes, each node is modeled as three stages of a multistage graph, numbered 0 to 2 in Figure 4.3(a), and then joined together to form a multistage optimization graph for the entire sequence of nodes. In each stage, the upper vertex represents SIMD mode, and the lower vertex represents SPMD mode. For stage 0 and its associated edges, a possible mode change is represented. The edge from the upper vertex in the stage 0 to the lower vertex in stage 1 is labeled with the cost of switching from SIMD to SPMD mode. Similarly, the cost of switching from SPMD to SIMD is indicated as the label for the edge from the lower vertex in stage 0 to the upper vertex in stage 1. There is zero cost for remaining in the same mode of parallelism. The edges from stage 1 to stage 2 in Figure 4.3(a) represent the cost of executing the node in each mode (or for a loop construct starting and stopping in that mode). Figure 4.3(b) illustrates the graph for a SIMD/SPMD trade-off tree with three children. The ordered pair $(T^{SIMD}_{mixed\text{-}4}, T^{SPMD}_{mixed\text{-}4})$, which represents the optimum mixed-mode implementation of the subtree, is obtained from the solution of the multistage optimization problem.

A node with m children is represented by a multistage optimization problem with $2m + 1$ stages: m stages to represent the nodes, m stages to represent possible mode switches, and one stage to represent the final states at the end of the multistage optimization graph. Thus, $2m + 1 - 2 \equiv 2m$ iterations of Moore's algorithm are required to determine a solution. For each iteration corresponding to a possible mode switch, $2^2 = 4$ comparisons and $2^2 = 4$ assignments

are needed, while for each iteration corresponding to a node execution, no comparisons and $2^2 = 4$ assignments are needed. Thus, finding the minimum mixed-algorithm execution time by this approach has a sequential time complexity of $8m + 4m = 12m = O(m)$ time.

By recording the minimum paths selected at each iteration, the mode of parallelism for each stage can be determined. Each edge in the minimum-cost path selected corresponds to performing a node in SIMD, performing a node in SPMD, or performing a mode switch. When the multistage optimization is completed for all the children of a node in the SIMD/SPMD trade-off tree (under the constant mode-switching cost assumption), the execution time information is used to generate the execution time for the parent node. The multistage approach is then applied to the parent node and its siblings. In this way, the analysis works upwards through the trade-off tree until execution costs are obtained for the root node.

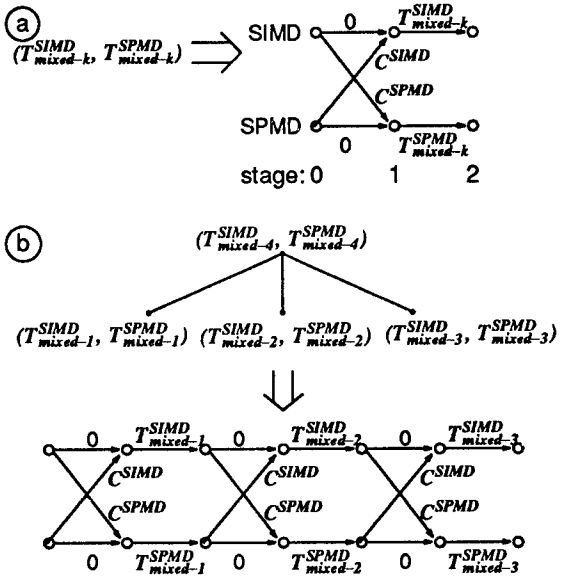


Figure 4.3: Transformation from flow-analysis tree to multistage graph.

4.3.3. With Non-Constant Switching Costs

In mixed-machine heterogeneous systems, the cost to switch from one machine to another is assumed to depend primarily on the cost of moving the required data between machines. A given machine may contain a data set because it was initially loaded there, it was received from another machine, or it was generated by that machine. (It is assumed that the assignment of program code will be determined, and code distributed, prior to execution time.) Thus, unlike the assumption of the previous subsection, the cost of switching execution from one machine to

another is dependent on which machine(s) contain the data structures that are required to execute the next block, which in turn is dependent on the machine choices made for previous blocks. Alternatively, the effectiveness of a current machine selection and associated data movement is impacted by the data movement required for the optimal machine assignment for a later program segment. Under these conditions, it can be shown that the multistage optimization approach described in the previous section does not directly apply because the cost of switching machines is not a known constant. However, as to be described, it forms the basis of a useful heuristic in this situation.

4.4. Extension of BBMS

4.4.1. A Simplified Parallel Program Behavior Model

Parallel programs are divided into a sequence of individual blocks (B_0, B_1, B_2, ...). Each block is considered exactly once, in a sequence beginning with the first block and ending with the last. In this way, the execution of a sequence of sibling nodes under a common parent node is modeled. A block sequence itself may, as the flow-analysis tree is reduced, form an interior node as part of another construct, but in this section the focus is on determining a low-cost path only for the immediate block sequence under a single parent node in a mixed-mode heterogeneous system.

Each block B_i in the parallel program can be executed on one of several machines. For each possible machine, there is an associated execution time assumed to be known a priori. Mixed-machine systems consisting of two parallel machines are considered here, but the results can be generalized to include systems with more than two machines.

Associated with a parallel program is a set of data structures that are used by the program during execution. Each data structure j is assigned a weight C^j, indicating the cost of moving that data structure to another machine in the system. Although the time cost of moving a data structure can be related to the source and destination machines, to clarify the concepts presented, the time cost for each data structure is assumed to be independent of the source and destination of the transfer. The analysis can be extended to include these source/destination dependent costs.

Also associated with each data structure is a location attribute. The location of a data structure can be enumerated as being on no machine, on a single machine, or on a set of machines. A data structure having no location corresponds to a structure that may be available at an I/O device for the system, but not available on any machine. According to the machine selection policy, the data structure can be assigned to the portion of the heterogeneous

environment where it is first used. A structure that has multiple locations is defined to have the same state at each location. This may correspond to a data set that is only read by a block.

A data location (DL) table is defined as a table with entries for each data structure used within a parallel program. The DL table initially corresponds to the starting location for each of the data structures used. As the static source-code based analysis of the program proceeds, the DL table is updated to reflect the movement of data structures that would occur as a result of machine choices made thus far in the program.

To differentiate the state of the DL table for different portions of the program, a subscript is added to indicate the block that corresponds to the state of the table before execution of that block. For example, DL_0 corresponds to the initial location of data structures in the heterogeneous system before B_0 is executed, and DL_1 reflects the changes in the location of data structures required as a result of B_0.

Associated with each block in the program is a data use (DU) table listing each data structure used in that block, as well as a usage type, which designates the way in which that structure is modified by the block associated with the DU table. Possible usage types include the read type, where a structure is used but not altered by a block; the create type, where the values of a structure are first generated; and the modify type, where one or more elements of the structure are modified, based on on the current state of the data structure. Other usage types are also possible.

As an example, consider the representation of a parallel program given in Figure 4.4. The program consists of three blocks, labeled B_0, B_1, and B_2, illustrated vertically in the center of the figure. The blocks labeled "begin" and "end" in the figure do not correspond to executable code, but instead to initial and final states. B_0 requires a time cost of 20 to execute on machine X, and 2 on machine Y. B_1 and B_2 each require 10 to execute on machine X, and 10 on machine Y. Each DU table is shown to the left of its corresponding block. In B_0 the data structure P is modified, in B_1 the data structure Q is created, and in B_2 the data structures P and R are modified.

The DL tables are shown for each step during the planned parallel execution. Before execution there are three data structures, P, Q, and R, initially placed on machine X, as indicated by DL_0. Assume the first block is be executed on machine Y. It modifies data structure P and this change is reflected in DL_1. Assume B_1 is also to be executed on machine Y. It creates data structure Q. Therefore, DL_2 indicates that Q is now also on machine Y. It is assumed B_2 is executed on machine X. Because it modifies P, P must be moved from Y to X, as indicated in DL_2 and DL_{final}. Block B_2 also modifies R, but R is already in X (see DL_2) so no data transfer is needed.

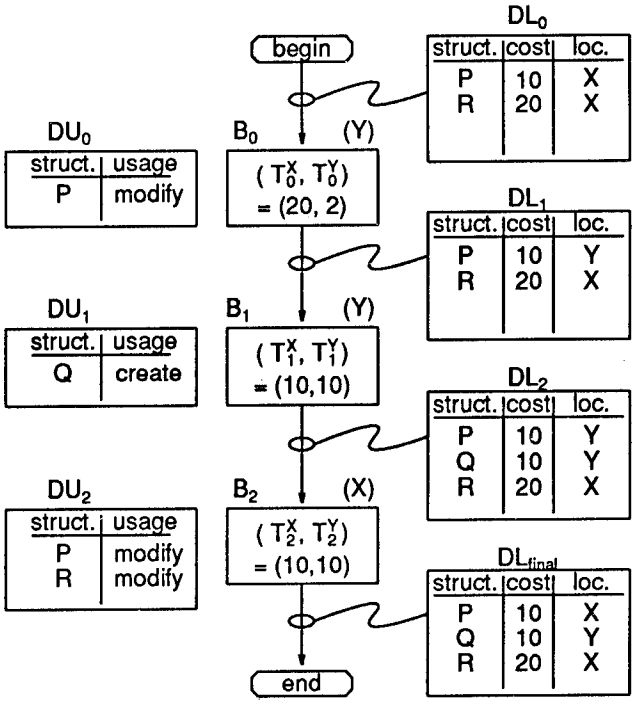


Figure 4.4: Simplified model of parallel program behavior.

4.4.2. Heuristic Based on BBMS

Given a sequence of blocks (B_0, B_1, B_2, ...), a set of data usage tables and associated costs to transfer the data structures between machines in a heterogeneous system, the goal to to find an assignment of machines to blocks that results in the minimum execution time. As discussed in Subsection 4.3, the multistage optimization technique presented in [WaS93] finds the optimum mapping of modes to program segments when the cost to switch modes is constant, but the technique cannot be applied directly to the mixed-machine case where non-constant data transfer costs are considered.

To reduce the complexity of finding an assignment of machines to program segments, a heuristic approach is used. In this approach, four independent sets of DL tables are used to track possible routes through the multistage optimization graph. To find the shortest path through a completely-weighted multistage graph with s stages, $s - 2$ iterations are required. During each iteration, the shortest path from each state in the initial stage to each state in the next stage is determined. Because there are only two initial states (representing machine X and Y) and two states in the next stage, only four paths are considered at each iteration. In the heuristic extension for non-constant switching costs, a DL table is associated with each path in the solved

portion of the multistage graph (Figure 4.5). Each DL table represents the location of all data structures needed by the parallel program that results from choosing the path designated by the corresponding arc in the solved portion of the multistage graph. Additionally, because programs with large data sets often are found to execute in minimum time without switching machines, the single-machine implementation is always considered for each machine in the heterogeneous system.

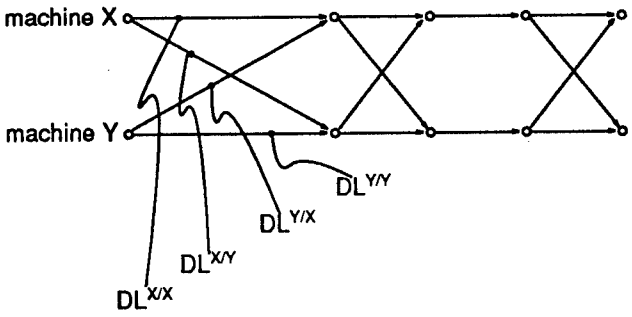


Figure 4.5: Heuristic building on the multistage technique.

4.4.3. Simulation Study

Examples exist for which the heuristic selects machine mappings that are not optimal. This is because machine choices, and hence data transfer decisions, do not consider the data location needs of blocks that follow. To determine if the heuristic generally provides good results, a simulation study has been performed. In the study, randomly-generated parallel program behavior representations were used to test the proposed heuristic. Then, the resulting assignment of machines was compared to the optimum assignment (found by exhaustive search) to determine the validity of the machine-selection decision. Parameters of the parallel program behavior being modeled are then modified to determine the effect of the parameters on the quality of the assignment decision.

One of the goals of the simulation study is to explore how the heuristic performs in a general-purpose environment. To examine the behavior of the approach for a variety of program structures, and because it is unclear what is meant by a "typical" parallel program, randomly-generated parallel program representations are used instead of specific program traces.

In the simulation, parallel program behaviors are generated using the simplified model in Subsection 4.1. No parallel code is generated; only parameters needed by the model. For each leaf block B_i in the simulated program, an ordered pair (T_i^X, T_i^Y) is randomly assigned with uniform distribution over a specified range. No correlation between T_i^X and T_i^Y is assumed.

The number of data structures used by the program is then assigned, also with uniform distribution over a specified range. Each data structure is given an initial location and weight (cost to transfer), again uniformly over a specified range without correlation. A DU table is then assigned to each block in the program, with data structure requirements and usage types also assigned. Again, all the values are uncorrelated. This is important because of interest is how the heuristic performs in the absence of predictable events, and because the characterization of "typical" parallel programs remains unclear in the general purpose computing field.

After the parallel program behavior is generated, four separate analyses are performed with the following policies: best, worst, heuristic, and random, each providing an assignment of machines to each block in the program. The best policy determines the optimum (minimum time cost) assignment of machines to program blocks by exhaustively testing every possible mapping. Because this approach is so expensive, simulations can be performed for only a limited number of stages (twelve in this study). The worst policy finds the most costly mapping, again by exhaustive search, and is used as a lower bound. The heuristic policy implements the policy described in Subsection 4.2. The random policy makes machine assignments by using the bit-wise representation of a randomly-generated integer to serve as a map for making machine selections.

Three sets of simulations are included here, differing in the ratio of transfer time to computation time. In each set, the number of stages in the generated parallel program behaviors is controlled, and is varied from three to twelve. At each setting of the number of stages to be simulated, 2^8 simulated parallel program behaviors were generated. For each generated program, each of the four policies were used to determine an assignment of machines to blocks. Then, the time cost for each selected mapping was determined.

Because the actual time cost for each parallel program varies significantly, the information for each heuristically and randomly selected mapping was normalized against the best and worst mapping for that algorithm. Specifically, the heuristic norm for a parallel program *k*, termed h-norm, is determined by the following definition:

$$\text{h-norm}_k = 1 - \left[\frac{(\text{heuristic}_k - \text{best}_k)}{(\text{worst}_k - \text{best}_k)} \right]$$

The random norm (r-norm) is defined similarly.

Using this definition, an h-norm of 1 indicates that the optimum mapping was selected, while an h-norm of 0 indicates that the worst possible mapping was selected. The results for all generated programs for each parameter setting are then averaged.

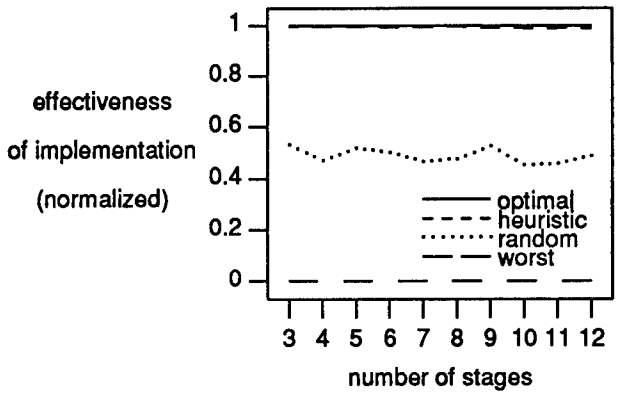


Figure 4.6: Effectiveness of machine selection policies for data/execution ratio of 0.1.

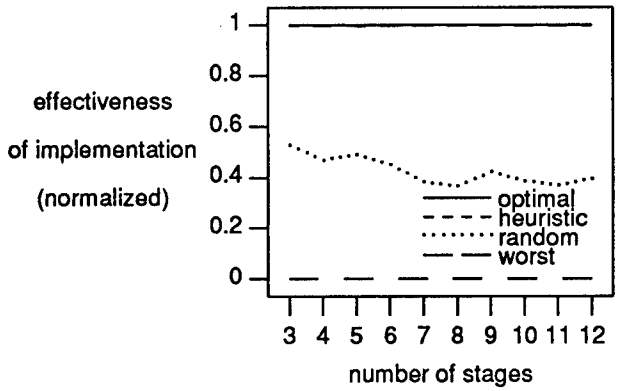


Figure 4.7: Effectiveness of machine selection policies for data/execution ratio of 10.

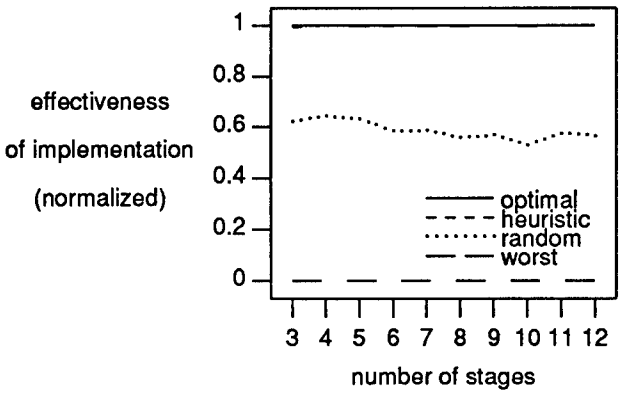


Figure 4.8: Effectiveness of machine selection policies for data/execution ratio of 1000.

In each of the three simulation sets, the relative size of data structures (i.e., data transfer cost) is controlled in relation to the computational cost of each stage. This is accomplished by restricting the range over which data structure sizes and execution times are chosen. In the first set of simulations, the data/execution cost ratio is set at 1/10, indicating that data size is generally small in relation to the execution requirements for the algorithm. For this set of simulations the heuristic is expected to perform well, because the effect of transferring data between machines is not significant, and therefore results should be similar to the constant-cost case.

Results for this simulation are shown in Figure 4.6. The horizontal axis in the graph indicates the number of stages in the simulated parallel program behavior. The vertical axis indicates the relative worth of the machine-selection decisions for the simulated program behaviors. The solid line in the graph represent the "best" policy, which is a constant value of 1. Similarly, the "worst" policy (large-dashed line) is a constant value of 0. The dotted line indicates the relative worth of the "random" policy. The heuristic policy is represented by the small-dashed line and performs close to the optimal. For many of the simulated cases, the heuristic policy chooses the same mapping as the "best" policy. Each data point in the graph represents the average of the r-norms and h-norms of 2^8 simulations.

In the second set (Figure 4.7), the data/execution ratio is set to 10. Again, the heuristic performs very well.

In the third set of simulations, shown in Figure 4.8, the data/execution ratio is 1000. Parallel programs for these parameter settings tend to become polarized in that they are normally implemented entirely in one machine, because switching from one machine to another normally incurs a very high data transfer penalty. Even under these extreme conditions, the machine selection heuristic still selects good mappings.

In Table 4.1, the percentage of optimal execution time required for both the random assignment policy and the heuristic assignment policy for the 12-stage simulation is given. As the data/execution cost ratio increases, the random assignment policy results in significantly higher cost assignments, while the heuristic policy finds mappings that are close to optimal (100%).

data/execution cost ratio	random assignment	heuristic assignment
0.1	140.7%	100.6%
10	449.1%	100.1%
1000	515.6%	100.5%

Table 4.1: Percentage of optimal execution time for random and heuristic assignments (# stages = 12).

4.5. Summary

The problem of minimizing the execution time of programs within a mixed-machine heterogeneous environment has been considered. A heuristic for determining the machine to execute each portion of a parallel program is examined, with an emphasis on efficiently determining the best assignment of machines to program segments in the presence of data-location dependent machine switching costs. Results of simulated parallel program behaviors indicate that good assignments are possible without resorting to exhaustive search techniques.

5. Heterogeneous Computing

5.1 Overview

A single application task often requires a variety of different types of computation (e.g., operations on arrays versus operations on scalars). Numerous application tasks that have more than one type of computational characteristic are now being mapped onto high-performance computing systems. Existing supercomputers generally achieve only a fraction of their peak performance on certain portions of such application programs. This is because different subtasks of an application can have very different computational requirements that result in different needs for machine capabilities. In general, it is currently impossible for a single machine architecture with its associated compiler, operating system, and programming tools to satisfy all the computational requirements of various subtasks in certain applications equally well [FrS93]. Thus, a more appropriate approach for high-performance computing is to construct a heterogeneous computing environment.

A heterogeneous computing (HC) system provides a variety of architectural capabilities, orchestrated to perform an application whose subtasks have diverse execution requirements. One type of heterogeneous computing system is a mixed-mode machine, where a single machine can operate in different modes of parallelism. Another is a mixed-machine system, where a suite of different kinds of high-performance machines are interconnected by high-speed links. To exploit such systems, a task must be decomposed into subtasks, where each subtask is computationally homogeneous. The subtasks are then assigned to and executed with the machines (or modes) that will result in a minimal overall execution time for the task. Typically, users must specify this decomposition and assignment. One long-term pursuit in the field of heterogeneous computing is to do this automatically. The field of HC is quite new, having been made possible

---

The co-authors of this section were Howard Jay Siegel, John K. Antonio, Richard C. Metzger, Min Tan, and Yan A. Li.

This research was supported by Rome Laboratory under contract number F30602-94-C-0022, NRaD under subcontract number 20-950001-70, and AFOSR under contract number RL JON 2304F2TK. Some of the research discussed used equipment supported by the National Science Foundation under grant number CDA-9015696.

This material appeared in Purdue University Technical Report TR-EE 94-37 December 1994. A version will appear in the *Handbook of Parallel and Distributed Computing*, A. Zomaya, ed., McGraw-Hill, 1995. Parts also appeared in: (1) the *Proceedings of the 23rd AIPR Workshop on Image and Information Systems: Applications and Oppourtunities*, sponsored by SPIE, October 1994; (2) the *Proceedings of the International Symposium on Parallel Architectures, Algorithms, and Networks*, sponsored by the JAIST, December 1994; and (3) the *Proceedings of the Petaflops Frontier Workshop at the 5th Symposium on the Frontiers of Massively Parallel Computation*, sponsored by the IEEE Computer Society, February 1995, proceedings to appear.

by recent advances in high-speed inter-machine communication. This section is a brief introduction to HC.

In the most general case, an HC system includes a heterogeneous suite of machines, high-speed interconnections, interfaces, operating systems, communication protocols, and programming environments [KhP93]. HC is the effective use of these diverse hardware and software components to meet the distinct and varied computational requirements of a given application. Implicit in this concept of HC is the idea that subtasks with different machine architectural requirements are embedded in the applications executed by the HC system. The goal of HC is to decompose a task into computationally homogeneous subtasks, and then assign each subtask to the machine (or mode of parallelism) where it is best suited for execution.

Figure 5.1 shows a hypothetical example of an application program whose various subtasks are best suited for execution on different machine architectures, i.e., vector, SIMD, MIMD, data-flow, and special purpose [Fre91]. Executing the whole program on a vector supercomputer only gives twice the performance achieved by a baseline serial machine. The vector portion of the program can be executed significantly faster. However, the non-vector portions of the program may only have a slight improvement in execution time due to the mismatch between each subtask's unique computational requirement and the machine architecture being used. Alternatively, the use of five different machines, each matched with the computational requirements of the subtasks for which it is used, can result in an execution 20 times as fast as the baseline serial machine.

Two types of HC systems are mixed-mode machines and mixed-machine systems [WaA94]. A *mixed-mode* machine is a single parallel processing machine that is capable of operating in either the synchronous SIMD or asynchronous MIMD mode of parallelism and can dynamically switch between modes at instruction-level granularity with generally negligible overhead [FiC91]. A *mixed-machine* system is a heterogeneous suite of independent machines of different types interconnected by a high-speed network. Unlike mixed-mode machines, switching execution among machines in a mixed-machine system requires measurable overhead because data may need to be transferred among machines. Thus, the mixed-machine systems considered in this section are assumed to have high-speed connections among machines that make decomposition at the subtask level feasible. Another difference is that in mixed-machine systems, the set of subtasks may be executed as an ordered sequence and/or concurrently. Mixed-machine HC has also been referred to as metacomputing [KhP93].

A programming language used in an HC environment must be portable. To allow full flexibility of execution targets, the language must be compilable into efficient code for any machine in the mixed-machine suite or any mode available in a mixed-mode machine. Thus, ideally, this

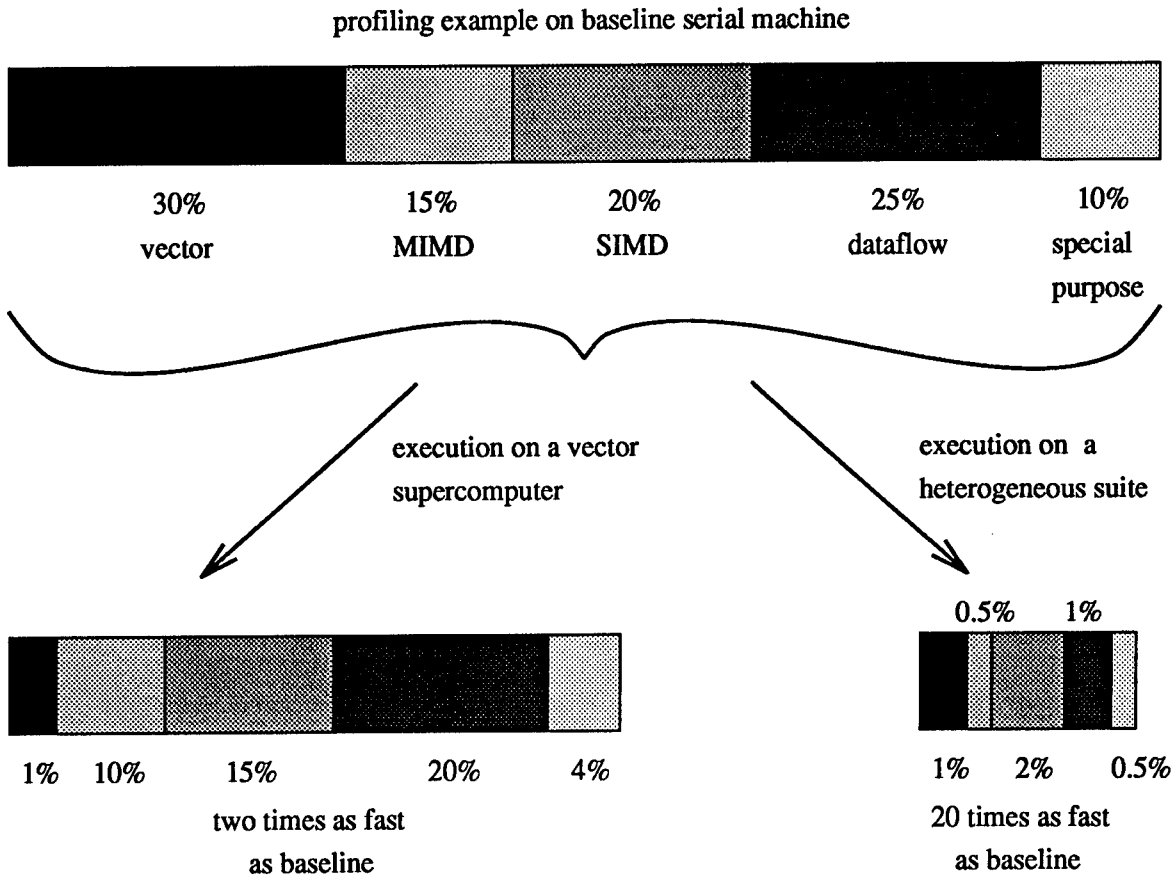


Figure 5.1: A hypothetical example of the advantage of using heterogeneous computing [Fre91], where the execution time for the heterogeneous suite includes inter-machine communications. Percentages are based on 100% being the total execution time on the baseline serial system, but are not drawn to scale.

portable programming language must be machine/mode-independent, and supply the compiler with the information it needs to produce efficient code for different target architectures and/or modes of parallelism. In this section, the future existence of such a language is assumed. More about this topic is in [WeW94], where a collection of parallel programming languages are surveyed and various aspects of programming parallel systems from the perspective of supporting HC are addressed.

In Subsection 5.2, examples of mixed-mode machines are given and the mechanism of switching modes for each example is discussed. After Subsection 5.2, "HC system" will imply "mixed-machine system," as it is most commonly used in that way. Descriptions of and applications for example existing mixed-machine systems are presented in Subsection 5.3. Subsection 5.4 provides examples of existing software tools and environments for HC systems. A

conceptual model for HC is introduced in Subsection 5.5. In this conceptual model, task profiling and analytical benchmarking are two steps necessary for characterizing an application program to automatically decompose it for processing on an HC system. Existing literature that presents explicit frameworks for performing task profiling and analytical benchmarking in the context of HC is overviewed in Subsection 5.6. Matching and scheduling are techniques for selecting machines for each subtask based on certain cost metrics. In Subsection 5.7, some basic characteristics of matching and scheduling techniques are described and some existing formulations are reviewed. Finally, open problems in the field of HC are discussed in Subsection 5.8.

5.2. Mixed-Mode Machines

5.2.1. Introduction

Two types of parallel processing systems are the *SIMD* (single instruction stream - multiple data stream) machine and the *MIMD* (multiple instruction stream - multiple data stream) machine. An SIMD machine typically consists of N processors, N memory modules, an interconnection network, and a control unit [Fly66]. Figure 5.2(a) shows a distributed memory SIMD architecture in which each processor is paired with a memory module to form N processing elements (PEs). In the SIMD mode of parallelism, there is a single program and the control unit broadcasts instructions of this program in sequence to the N PEs. All enabled PEs execute the same instruction (broadcast by the control unit) at the same time, but each on its own distinct data. The operand data for these instructions are fetched from the memory associated with each PE. The interconnection network provides inter-PE communication.

In an MIMD machine, each PE stores its own instructions and data. Distributed memory MIMD systems are typically structured like SIMD systems without the control unit, i.e., N PEs, an interconnection network, and multiple data streams [Fly66] (see Figure 5.2(b)). Each PE executes its own program asynchronously with respect to the other PEs. Thus, in contrast to the SIMD model, there are multiple threads of control (i.e., multiple programs). In both of the models in Figure 5.2, a PE processes data stored locally or received from another PE through the interconnection network. The use of SIMD and MIMD machines is discussed further in [SiW95].

SIMD machines and MIMD machines each have their own advantages when they are used to execute application programs. The advantages of SIMD mode include:

a) The single instruction stream and implicit synchronization of SIMD make programs easier to create, understand, and debug. Also, as opposed to MIMD architectures where common

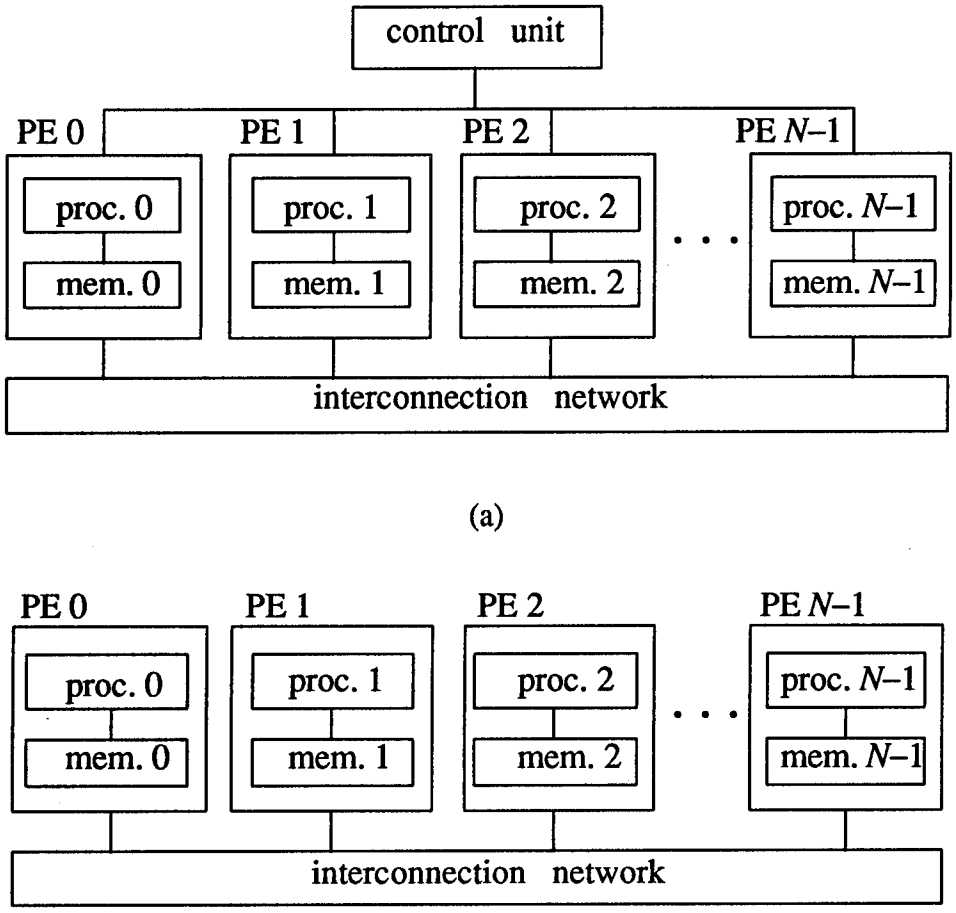


Figure 5.2: (a) Distributed memory SIMD machine model. (b) Distributed memory MIMD machine model.

programs can be executed asynchronously, the user does not need to be concerned with the relative timings among the PEs.

b) In SIMD mode, the PEs are implicitly synchronized at the instruction level. Explicit synchronization primitives, such as semaphores, may be required in MIMD mode, and generally incur overhead.

c) The implicit synchronization of SIMD mode also allows more efficient inter-PE communication. If the PEs communicate through messages, during a given transfer all enabled PEs send a message to distinct PEs, thereby implicitly synchronizing the "send" and "receive"

commands. The receiving PEs implicitly know when to read the message, who sent it, and why it was sent. MIMD architectures require the overhead of identification protocols and a scheme to signal when a message has been sent and received.

d) Control flow instructions and scalar operations that are common to all PEs (e.g., computing common local subimage data point addresses) can be overlapped (i.e., executed concurrently) on the CU while the processors are executing instructions (this is implementation dependent); this is referred to as CU/PE overlap [ArN91, KiN91].

e) Only a single copy of the instructions needs to be stored in the system memory, thus possibly reducing memory cost and size, allowing for more data storage, and/or reducing communication between primary and secondary memory.

f) Cost is reduced by the need for only a single instruction decoder in the CU (versus one in each PE for MIMD mode).

The advantages of MIMD mode include:

a) MIMD is very flexible in that different operations may be performed on the different PEs simultaneously (i.e., there are multiple threads of control). Thus, MIMD is effective for a much wider range of algorithms, including tasks that can be parallelized based on functionality (i.e., MIMD can exploit data parallelism and functional parallelism, while SIMD is limited to the former [Jam87]).

b) The multiple instruction streams of MIMD allow for more efficient execution of conditional statements (e.g., "if-then-else") because each PE can independently follow either decision path. In SIMD mode, when conditionals depend on data local to PEs, all of the instructions for the "then" block must be broadcast, followed by all of the "else" block. Only the appropriate PEs are enabled for each block.

c) MIMD's asynchronous nature results in a higher effective execution rate for a sequence of instructions each of whose execution time is data dependent (e.g., floating point operations on some processor architectures). In SIMD mode, a PE must wait until all the other PEs have completed an instruction before continuing to the next instruction, resulting in a "sum of max's" effect: $T_{SIMD} = \sum_{instr's} \max_{PE}$ (instr. time). MIMD mode allows each PE to execute the block of instructions independently, resulting in a "max of sum's" effect: $T_{MIMD} = \max_{PEs} \sum_{instr's}$ (instr. time) $\leq T_{SIMD}$ (see Figure 5.3).

d) MIMD machines do not have the added cost of a SIMD CU and the hardware for broadcasting instructions.

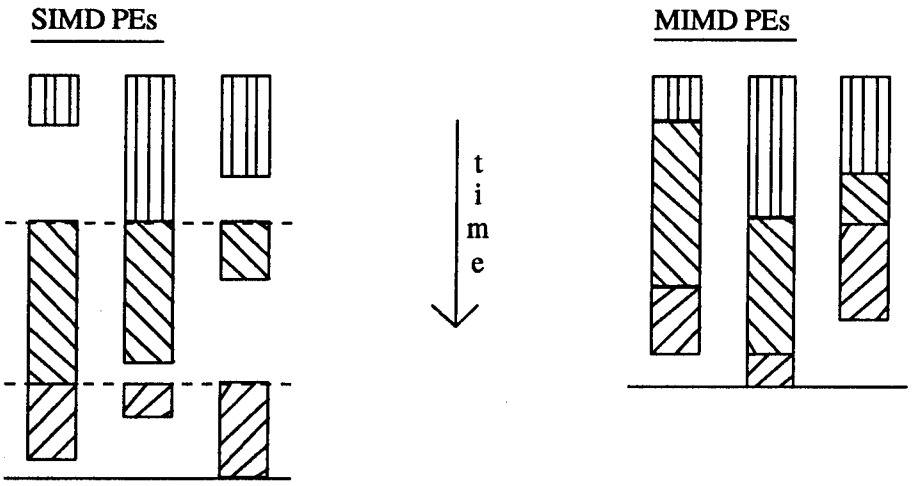


Figure 5.3: "Sum of max's" versus "max of sums" effects.

The trade-offs above are summarized from [BeS91]. The reader is referred to that paper and to [Jam87, SiA92b] for more details and examples. Because both SIMD and MIMD modes have advantages, various mixed-mode machines have been proposed.

A mixed-mode machine, which can dynamically switch between the SIMD and MIMD modes of parallelism at instruction-level granularity, allows different modes of parallelism to be applied to execute various subtasks of an application program. Various studies have shown that the mode of parallelism has an impact on the performance of a parallel processing system, and a mixed-mode machine may outperform a single-mode machine with the same number of processors for a given algorithm (e.g., [GiW92, SaS93, UlM94]).

As an example of the use of a mixed-mode machine, consider the bitonic sorting [Bat68] of sequences on the mixed-mode PASM prototype [FiC91]. Assume there are L numbers and $N = 2^n$ PEs, where L is an integer multiple of N, that L/N numbers are stored in each PE, and that the L/N numbers within a PE are in sorted order. The goal is to have each PE contain a sorted list of L/N elements, where each of the elements in PE i is less than or equal to all of the elements in PE k, for $i < k$. The regular bitonic sorting algorithm for $L = N$ is modified to accommodate the L/N sequence in each PE. As shown in Figure 5.4, an ordered merge is done between the local PE sequence X and the transferred sequence Y using local data conditional statements in merge(X, Y). The lesser half of the merged sequence is assigned the pointer X and the greater half is assigned the pointer Y. The pointers to the two lists may be swapped by swap(X, Y), based on a precomputed data-independent mask.

```
for k = 1 to log2N do
    for i = 1 to k do
        { for q = 1 to L/N do
            { load X[q] into network
              send to PE whose number differs in bit (k − i)
              Y[q] ← network output }
          merge(X, Y)
          swap(X, Y) }
```

Figure 5.4: Bitonic sequence-sorting algorithm [FiC91].

When choosing the mode of parallelism, the programmer must consider various characteristics of the algorithm. The ordered merge involves many comparisons, all of which can be more efficiently computed in MIMD mode. The innermost loop of the algorithm requires many network transfers, which are better performed in SIMD mode. In a mixed-mode implementation, the ordered merge and swap routines can be executed in MIMD mode, while the rest of the operations, including network transfers, are performed in SIMD mode. This approach has an advantage over pure SIMD or pure MIMD mode implementations because all comparisons are done in MIMD mode and all network transfers are done in SIMD mode. Additionally, there is potential in SIMD mode for overlapping operations done by the control unit (i.e., loop index variable increment and compare) with operations done by the PEs (i.e., the loop body). It is shown in [FiC91] that there is a noticeable improvement in execution time for the mixed-mode implementation. The mixed-mode results are shown to be the product of properties inherent to the modes of parallelism.

Most of the advantages of SIMD and MIMD modes can be realized with a mixed-mode architecture that allows the most appropriate mode to be selected at each step in the execution of a program. Disadvantages of mixed-mode parallelism include higher hardware cost (because mixed-mode machines must have the hardware needed for both modes), more complicated use (because the mode switching ability adds another dimension of complexity for the programmer), and, when switching from MIMD to SIMD mode, some PEs may remain idle while they wait for the other PEs to reach the switch point (which they may not need to do if only MIMD mode was used) [BeK91].

Very brief descriptions of four existing mixed-mode machines follow, emphasizing the particular mechanisms for implementing mode-switching during the execution of the application

program. Readers can refer to the references provided for each system for detailed descriptions of the hardware organization and related issues.

5.2.2. PASM

PASM is a PArtitionable-SIMD/MIMD system concept being developed as a design for a large-scale dynamically reconfigurable parallel processing system [SiS87, SiS95]. The PASM design concept is a distributed memory machine and can support at least 1024 PEs in the computational engine. A small-scale proof-of-concept prototype (30 processors, 16 PEs in the computational engine) has been built at Purdue University, in the USA. The prototype is a constantly evolving tool for validating design concepts and studying issues related to the use of reconfigurable parallel processing systems.

As a partitionable mixed-mode system, PASM can be dynamically reconfigured to form submachines of various sizes. Each submachine can independently perform mixed-mode parallelism. PASM uses a flexible multistage interconnection network for inter-PE communication. Thus, PASM is dynamically reconfigurable along three dimensions: partitionability, mode of parallelism, and connections among PEs. To simplify the discussion, the additional hardware needed for partitioning is ignored, and a single control unit will be assumed.

The mechanism used by PASM to switch modes at instruction-level granularity is as follows. In SIMD mode, a PE fetches SIMD instructions by reading an instruction word from the SIMD instruction space of the PE's memory. This is only a logical address space because SIMD instructions are not physically located in the memory of the PEs. Each memory access made by a PE's processor is monitored by the Instruction Broadcast Unit (IBU). The IBU sends an SIMD instruction request to the control unit, and when all enabled PEs have requested a new instruction, it is broadcast from a queue in the control unit. In MIMD, a PE fetches instructions from its local memory. A PE can switch from SIMD mode to an MIMD program located at some address *A* in its local memory by receiving a "branch to *A*" instruction in SIMD mode. Similarly, a PE can change from MIMD mode to SIMD mode by executing a branch to the logical SIMD instruction space. Such flexibility in mode switching allows mixed-mode programs to be written that change modes at instruction-level granularity with generally nominal overhead.

5.2.3. TRAC

The Texas Reconfigurable Array Computer (TRAC) is a partitionable mixed-mode parallel processing system, which was developed at University of Texas at Austin, in the USA [LiM87]. Its resources can be dynamically reconfigured to fit the structures of the applications. TRAC uses a

Banyan interconnection network for inter-processor communication. TRAC 1.1, a shared memory machine, was an experimental prototype of the original paper design of TRAC 1.0. It consisted of four microprocessors that were connected to nine memory modules by an SW-Banyan network with fan-out of three, spread of two, and two levels (see Figure 5.5).

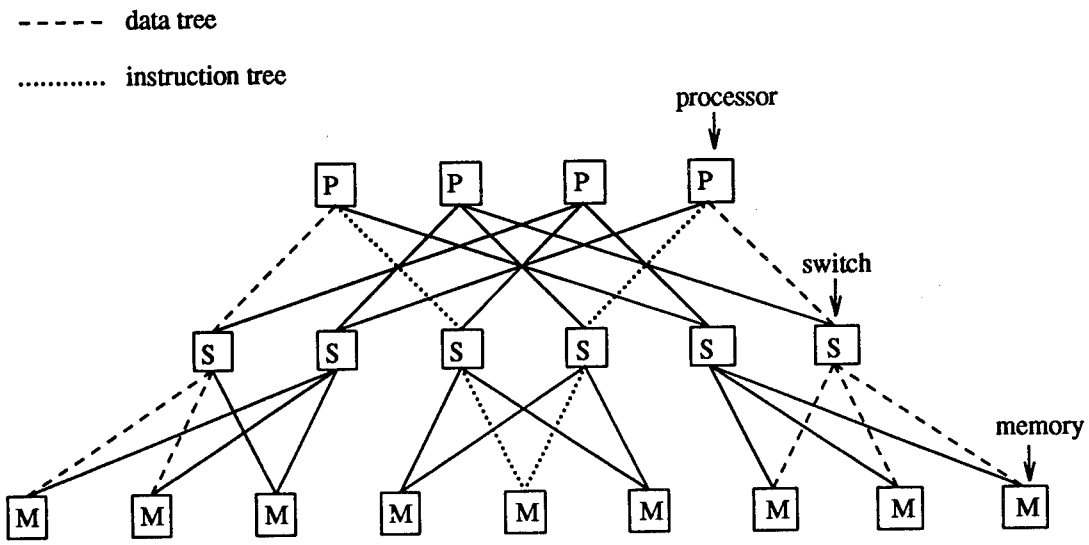


Figure 5.5: A task tree (instruction tree and data tree) of TRAC 1.1 [AlG89].

In TRAC 1.0, after configuring the Banyan network, several data trees connect data memories with their corresponding processors, and an instruction tree connects a specific program memory with processors. As shown in Figure 5.5, the dashed lines in the network illustrate two data trees, each connecting a processor at the top to a number of data memories at the bottom. The dotted lines illustrate an instruction tree that connects a single program memory to two processors that will work together in SIMD mode. In MIMD mode, each processor can independently fetch its own instructions from a memory module associated with it. Mode switching between SIMD and MIMD is implemented by changing the source of the instructions for the processors.

5.2.4. OPSILA

OPSILA is a limited mixed-mode parallel machine built at University of Nice, in France [DuB88]. It runs with two different modes of parallelism, SIMD and SPMD. SPMD (single program - multiple data stream) mode is a special form of MIMD mode where all the PEs execute the same program in an asynchronous fashion, each on its own data [DaG88]. OPSILA is

composed of two parts: a central control unit and a computation unit with 16 PEs. Each PE is a processor associated with a memory bank (MB). A synchronous Omega/Benes interconnection network is used for inter-PE communication.

The central control unit consists of two processors: the scalar processor (SP) and the instruction processor (IP). In SIMD mode, the application program is stored entirely in the scalar memory (SM) of the central control unit and managed by the SP. The data are located in the vector memory (VM) of the computation unit. The IP broadcasts SIMD instructions to each PE. The PEs then execute the same instruction simultaneously, each on data from its own MB.

SPMD mode is initialized by the IP, which provides each PE the starting SPMD code address. In SPMD mode, the same program is duplicated in each MB. PEs cannot exchange information during SPMD mode. Data exchanges can only occur in SIMD mode via the synchronous Omega/Benes interconnection network. The synchronization mechanism for initializing the SPMD mode and for returning to SIMD mode is a fork-join operation executed over the set of PEs. The transition from SPMD to SIMD mode is made in one machine cycle after the end of the execution of the PE with the largest work load.

5.2.5. Triton

Triton is a mixed-mode SIMD/MIMD parallel processing system developed at University of Karlsruhe, in Germany [HeW93, PhW93]. It uses a generalized De Bruijn interconnection network for inter-PE communication. The Triton architecture is scalable up to 4096 nodes. The Triton/1 prototype will consist of 260 nodes (four are for fault tolerance). Each node consists of a processor/memory pair, a memory management unit, a numeric co-processor, a SCSI interface, and a network processor.

In SIMD mode, a single front-end processor produces the instruction stream for all PEs. If a PE is selected not to execute an instruction, a local signal for the instruction stream is turned off and the corresponding PE is disabled. To switch to MIMD mode, the program has to be downloaded to the local memory of the PEs. This is done via load instructions in SIMD mode. The switch from SIMD to MIMD mode is accomplished by two instructions. First, the program counter is set according to the location of the program to be executed in MIMD mode by a branch instruction. Second, the SIMD request bit for each PE is deactivated. Each PE then switches to MIMD mode and starts the execution of the code stored in the local memory. To switch from MIMD to SIMD mode, the SIMD request bit for each PE is activated. The result of a global-wired-or operation of all PEs' SIMD request bits instructs the front-end processor to activate the SIMD mode. Then each PE switches to SIMD mode and the next instruction is from the instruction stream broadcast by the front-end processor.

5.2.6. EXECUBE Chip

The EXECUBE chip is a building block for parallel processing systems that can support both the SIMD and MIMD modes of parallelism [Kog94]. Its current chip design consists of eight PEs. Each PE is a 16-bit CPU, associated with a 64KB memory module. A hypercube interconnection network is used for inter-PE communication. This is all contained on a single chip developed by IBM Federal Systems Division, in the USA. A system with 64 EXECUBE chips (512 CPUs) has been constructed.

In SIMD mode, instructions are sent into each PE's instruction register by a separate controller via the SIMD broadcast bus. In MIMD mode, each PE obtains its own instructions from its local memory. Because the only way for accessing the memory system of each PE is through its CPU, the MIMD instructions are sent and stored into participating PEs' local memory in SIMD mode via the SIMD broadcast bus. Arbitrary collections of PEs can be in either mode simultaneously, with mode switching instructions included for changing modes between SIMD and MIMD. Those mode switching instructions are machine operation codes that activate special hardware functions. The mode switch from SIMD to MIMD is activated by executing an instruction to "switch to MIMD mode" and participating PEs begin execution at a specified address in local memory. After executing a switching instruction, the participating PEs stop fetching instructions from the SIMD broadcast bus and start to execute the instructions stored in local memory. A "switch to SIMD mode" instruction causes PEs to fetch instructions from the SIMD broadcast bus. A collective signal from the PEs is sent to the controller that sends SIMD instructions to each PE's instruction register. If any PE in the PE group that is changing to SIMD mode is still in MIMD execution, then the controller will wait until the collective signal from the PEs is set, at which point the SIMD execution is started.

5.2.7. Conclusions

Mixed-mode machines are one extreme form of HC, where two different modes of parallelism are available in one machine. This is in contrast to mixed-machine HC systems, where a suite of machines can provide different modes of parallelism by having each mode in a different machine. Both types of heterogeneous systems can support tasks that include some subtasks that execute faster in SIMD mode and others that execute faster in MIMD mode. Decomposing a task for mixed-mode execution is easier than mixed-machine because the same PEs are used for both modes and, in general, no data has to be moved as a result of a mode change. This eliminates two major problems in the use of mixed-machine HC: moving data among machines and determining machine loads.

The study of the design and use of mixed-mode machines provides valuable information about the trade-offs between SIMD and MIMD parallelism, explores the advantages and disadvantages of mixed-mode computation as a mode of parallelism, and establishes a relatively simpler environment for developing algorithm mapping techniques that may possibly be adapted to the mixed-machine arena. For example, a block-based mode selection methodology developed for mixed-mode machines, presented in [WaS94], was then extended for use as a heuristic for the mixed-machine case [WaA94].

Thus, mixed-mode machines are important for their advantages over single-mode machines and for their use in developing methodologies that may be adaptable for mixed-machine HC use. The emphasis of this report, however, is on mixed-machine systems. Therefore, for the rest of the report, "HC system" by itself will imply a mixed-machine suite.

5.3. Examples of Uses of Existing HC Systems

5.3.1. Simulation of Mixing in Turbulent Convection at the Minnesota Supercomputer Center

In [KlM93], the usefulness of a "metacomputer" developed at the Minnesota Supercomputer Center is demonstrated through a particular application involving the simulation of mixing in turbulent convection in three dimensions. "Metacomputer" is defined in [KlM93] to be a coordinated set of CPUs, I/O devices, mass storage, and graphical capabilities that are appropriately balanced for solving large-scale computational problems, and is equivalent to the term "HC system" defined in Subsection 5.1. The particular HC system developed consists of Thinking Machines' CM-200 and CM-5, a CRAY 2, and a Silicon Graphics VGX workstation, all interconnected over a high-speed HiPPI (high-performance parallel interface) network.

The underlying physics and mathematics that govern the dynamics associated with simulating mixing in turbulent convection are not included here, but are overviewed in [KlM93]. The required calculations for the simulation were divided into three phases: (1) calculation of velocity and temperature fields, (2) calculation of particle traces, and (3) calculation of particle distribution statistics and refinement of the temperature field. In the following paragraphs, a brief outline of how the required computations were decomposed and assigned to various machines in the system is given.

The velocity and temperature fields associated with the phase 1 calculations are governed by two second order partial differential equations. Three-dimensional cubic splines (over a grid

of size $128 \times 128 \times 64$) were used to approximate the velocity and temperature fields in these equations, resulting in a linear system of equations for the unknown spline coefficients. A conjugate gradient method was applied to solve this system of equations. These computations were done on the CM-5. At each time step, the grid of $128 \times 128 \times 64$ spline coefficients were transferred to the CRAY 2, where the calculation of the particle traces were done.

The particle traces were calculated by solving a set of ordinary differential equations that are dependent on the velocity field solution computed in phase 1. Initially, this computation was attempted on the CM-200 by employing an Eulerian approach. Although this approach worked well for a two-dimensional instance of the problem, the same approach could not be used for the three-dimensional simulations reported in [KlM93] because a prohibitive amount of memory was required. Instead, the three-dimensional simulations were implemented using a vectorized Lagrangian approach on the CRAY 2, which required substantially less memory than the parallel Eulerian scheme. The coordinates of the particles and the spline coefficients of the temperature field were then sent from the CRAY 2 to the CM-200.

The CM-200 was used to calculate statistics of the particle distribution and to assemble a three-dimensional temperature field from the associated spline coefficients (phase 3). A $256 \times 256 \times 128$ point temperature field file was produced from the $128 \times 128 \times 64$ grid of splines, representing a volume of eight million voxels (a voxel is a three-dimensional element). This file of voxels and the coordinates of the particles (one million particles were used in the model) were then sent to an SGI VGX workstation where they were visualized using an interactive volume renderer.

The application was successful in demonstrating the benefits of HC, however, the authors note that there is still much work to be done to improve the environment for developing HC applications. The authors state that there is a need for more vendor involvement, in addition to the need for more basic research in the areas of reliability, I/O software, interactivity, and distributed scheduling.

5.3.2. Interactive Rendering of Multiple Earth Science Data Sets on the CASA Testbed

In 1990, the National Science Foundation (NSF) in conjunction with the Defense Advanced Research Projects Agency (DARPA) established a program to conduct research in the area of networking at gigabit per second speeds [SpR90]. The program established five gigabit testbeds to carry out research in different application areas, each with a different research focus, such as networking protocols, software development, and networking hardware. The research results from this program will contribute to the proposed National Research and Education Network (NREN) and ultimately to the National Information Infrastructure (NII). The NREN will link

government, industry, and higher-education institutions involved in general research areas that can utilize the interconnected computational resources. In this and the next subsection, two applications that utilize the heterogeneous computing resources available on two of the testbeds are overviewed.

The CASA testbed interconnects several remote sites including the California Institute of Technology, San Diego Supercomputer Center, Jet Propulsion Laboratory (JPL), and Los Alamos National Laboratory. In the future, these sites will be interconnected via SONET (synchronous optical network) connections operating at 2.488 gigabits per second; they are currently connected with lower speed connections [BeB93]. The computational resources of the testbed consists of various parallel and vector machines including an Intel Touchstone Delta, Thinking Machines' CM-5 and CM-200, CRAY Y-MP8/864, Y-MP/264, and Y-MP/232, and a number of workstations and specialized visualization engines.

One of the applications developed on the CASA testbed involves interactive three-dimensional rendering of multiple Earth science data sets. Geology can be regarded as a "three-dimensional science," in the sense that both surface and subsurface data from the Earth are collected and studied. In the past, these two types of data were generally collected and analyzed separately. By making effective use of the computing and networking resources of the CASA testbed, researchers can construct a more complete image of the Earth's surface and subsurface, together, by combining multiple sets of data from various sources. The required processing and communication for merging these data sets should be fast enough to enable interactive manipulation of the associated image. According to [BeB93], researchers can rotate, slice, zoom, and "fly over" a full-color view of the Earth's surface and subsurface while sitting at a workstation.

The software for the application is divided into three categories: (1) a collection of functionally distinct two-dimensional image processing modules that generate and/or manipulate color images and elevation data, (2) a rendering process that combines data and creates an electronic rendered image, and (3) the network and control software that coordinate the various processes. The two-dimensional modules are implemented using Network Express, which is a portable, message passing, programming environment developed by the ParaSoft Corporation. Network Express can be used on MIMD machines, vector machines, and other computers. Under Network Express, each machine is considered a node within the network. One node is chosen as a host, which manages a set of other nodes in the network.

As was done for the "simulation of mixing in turbulent convection" application described in the previous subsection, this application was decomposed based on functionality. Functional modules were identified and optimized for specific machines (and executed on those machines).

Thus, when a functional module begins execution, it processes data sets that are completely resident on the machine where the module is executed. Initially, raw data sets are transferred to one of the two-dimensional functional modules for processing. The two-dimensional modules manipulate image and/or elevation data via a number of different algorithms. Most of the two-dimensional modules were developed for the CRAY Y-MP/232 at JPL and the CRAY Y-MP8/864 at the San Diego Supercomputer Center. Two of the two-dimensional modules were implemented on the CM-5 and CM-200 located at Los Alamos. Output from the two-dimensional modules are sent over the network to the three-dimensional rendering process, which was implemented on the Intel Touchstone Delta located at the California Institute of Technology.

In the current implementation of the CASA testbed, there are high-speed HiPPI connections only among machines located at a common geographical site (e.g., the CM-5 and CM-200 located at Los Alamos are both connected to a local HiPPI switch). The current connections among the distributed sites, which utilize lower speed networks, will be upgraded by using HiPPI-SONET gateways to interconnect each site's local HiPPI network to a wide area high-speed SONET network. Future work includes executing the application over this new high-speed HiPPI/SONET network to obtain new benchmark timings that will be compared with those of the current implementation.

5.3.3. Using VISTAnet to Compute Radiation Treatment Planning for Cancer Patients

VISTAnet is another network in the group of five gigabit testbeds mentioned in the last subsection. The VISTAnet testbed consists of several remote sites including the Center for Communications and Signal Processing at North Carolina State University, BellSouth, GTE, and three organizations within the University of North Carolina at Chapel Hill (the Graphics and Image Laboratory in the Department of Computer Science, the Microelectronics Systems Laboratory in the Department of Computer Science, and the Department of Radiation Oncology) [StA93]. The machines connected to the testbed include a CRAY Y-MP, a Pixel-Planes 5, a MasPar MP-1, and Silicon Graphics workstations.

A major application focus for this testbed has been the computation of radiation treatment planning for cancer patients [RoC92]. Recent improvements in the care of cancer patients are due in large part to the effective use of radiation treatment for attacking cancerous cells. Radiation is effective in treating the disease only if it is delivered to the tumorous cells in a high dose while sparing the nontumorous cells. To do this, the physician must determine the number of treatment beams to be used, the beam angles and shapes, the time the beam is to be activated, and which custom filters to use to alter the beam. This process is know as radiation treatment

planning and in the past was carried out in only two spatial dimensions.

Some types of cancer require that the radiation treatment planning take place in three dimensions to achieve maximum effectiveness. This three-dimensional type of planning requires advanced modeling of human anatomy (rendered from tomography scans) as well as three-dimensional modeling of the radiation beam (i.e., the treatment plan). In the application, the treatment plan model is superimposed onto the anatomical model. One of the objectives is to provide a visualization of these models that can be rotated, zoomed, and/or modified interactively.

The computational requirements of the application were decomposed in an attempt to take advantage of the strengths of the machines available in the testbed. The CRAY Y-MP was demonstrated to be ideal for radiation dose calculation and interpolation throughout the entire model. The Pixel-Planes 5 machine (which contains a quarter-million custom one-bit processors) is designed for rendering images and is used for shading and merging large amounts of image data.

The physician interacts with the system via a medical workstation hosted on a Silicon Graphics 340 VGX. >From this workstation, the physician can modify the treatment plan based on the current dosage patterns and can adjust the view by rotating the image. When an image viewpoint is adjusted, the new viewpoint information is sent to the Pixel-Planes 5, which renders the otherwise unchanged data according to the new viewing angle and presents the new image to the physician at the workstation. If the treatment plan is modified, the new treatment plan information is sent to the CRAY Y-MP, which computes the new three-dimensional dose distribution and sends the information to the Pixel-Planes 5 for rendering.

In the future, a MasPar MP-1 will be integrated into the application and will receive the three-dimensional dose distribution generated by the CRAY Y-MP. With this information, the MP-1 will be used to compute a statistical analysis of the treatment plan in relation to the anatomical data. This computed information will provide the physician with a quantitative measure of merit for each treatment plan.

5.4. Examples of Existing Software Tools and Environments

5.4.1. Overview

A variety of software tools and environments have been implemented to assist programmers in developing applications to execute across a heterogeneous suite of computers. A common feature among most of the existing tools is that they create a layer of abstraction between

programmers and the suite of machines. Some also provide explicit constructs needed to express synchronization and communication among tasks within the application. The following subsections discuss examples of software tools that exist and/or are being developed for HC systems. The functionalities of most of the tools described in this section tend to evolve and change rapidly; the descriptions here are based on the references given. A survey of distributed queueing and clustering systems, some of which can be applied to HC, is given in [KaN93].

5.4.2. Linda

Linda was originally implemented for *homogeneous* computing environments such as shared memory parallel computers (e.g., the Sequent Symmetry), distributed memory computers (e.g., the Intel iPSC/2), and local area networks (e.g., a network of workstations). However, as suggested in [CaG92], the tuple space abstraction of Linda makes it an attractive choice for HC systems as well. The tuple space acts to loosely connect processes that communicate via persistent objects called tuples, and not through transient events such as message passing or procedure calls. A process can generate a tuple and place it in a globally shared collection of tuples, which is called the *tuple space*. Additionally, tuples can be removed, read, and evaluated from the tuple space. There are two types of tuples: process tuples that incorporate executable code and data tuples that are passive, ordered collections of data items [BuL93]. Although the current version of Linda does not support concurrent utilization (i.e., interaction) among machines in an HC system, Linda programs are portable across a range of architecture types. Issues that must be resolved in order to extend the present version of Linda for concurrent use among machines in an HC system are outlined and discussed in [CaG92].

5.4.3. p4

p4 is a set of parallel programming tools designed to support portability across a wide range of multicomputer/multiprocessor architectures [BuL92, BuL93, BuL94]. p4 includes high-level operations built on top of low-level system-dependent primitives. These high-level operations allow certain procedure calls for a given system to be replaced with the equivalent p4 calls. The p4 functions are implemented by utilizing the lower-level system-specific set of procedures. The long term goal of this project is to allow a single program to be written for an entire class of systems (e.g., message passing) without requiring the explicit utilization of constructs of the specific system (e.g., Intel Paragon versus nCUBE 2) in the source code. The p4 function library is linked with the source code to provide functions for message passing, shared memory monitoring, process management, debugging, and language interfacing.

The architectures supported by p4 can be divided into three distinct classes. The first class is shared memory multiprocessors (e.g., the Alliant FX/8). In general, the method of communication for shared memory architectures is through the use of the global memory space. Using this method of communication requires that shared data be protected from unsafe concurrent access. p4 provides monitor data types for encapsulating shared data and controlling access. The second class of architectures supported by p4 is the class of distributed memory systems that implement communication through message passing. The members of this class are distributed memory multiprocessor machines and groups of workstations that communicate over a network [BuG93]. The third class of architectures supported by p4 is the class consisting of called "communicating clusters," which can include multiprocessor machines that communicate via shared-memory and/or through the exchange of messages. Therefore, p4 can support communication within and among both shared-memory and message-passing machines.

The process of executing a p4 program begins with the user compiling the code for the desired set of machines. The configuration of the system is then defined by creating a procgroup file, which defines how many programs are to be executed, the names of the programs, and where they are to be executed. The procgroup file gives the user the flexibility to experiment with different configurations and types of machines.

In addition to facilitating code portability in an HC environment, p4 also helps the user understand and analyze the behavior of the program's execution. This is accomplished using a utility called ALOG, which creates a log of time-stamped events captured during program execution. ALOG consists of a set of macros that can be used to instrument C or FORTRAN programs. These macros record various events during execution and then dump the associated information to a file on disk (i.e., log) upon program completion or memory exhaustion. This event log can then be used as an input file for a graphical tool called Upshot [HeL91]. With Upshot, the log file can be examined in detail to detect computational and/or communication bottlenecks.

The developers of p4 stress that it is not an "abstract tool" and that various components of p4 evolved through the development of real applications. As an example, p4 was used in developing a piezoelectric crystal simulation program. In this particular application, p4 was used to coordinate the computations and communications among an Intel Touchstone Delta, the graphical output on a Stardent Titan, and a Solbourne workstation (which was used as an I/O server). Current and future research directions for p4 include the implementation of Linda with p4 to provide a single high-level programming model.

5.4.4 Mentat

Overview

Mentat is an object-oriented parallel processing system designed to provide a layer of abstraction between the user's application and the hardware and system software used to execute the application. Mentat consists of run time support facilities and language abstractions that provide a clear separation between the user and the physical systems [GrW94]. This separation is achieved by using an object-oriented language to specify parallelism within the application and compiler technology to handle many of the tedious and time consuming bookkeeping tasks. Mentat combines a medium-grain dataflow computation model with the object-oriented programming paradigm to produce a system that facilitates hierarchies of parallelism [Gri93]. In this medium-grain dataflow model, programs are characterized as directed graphs. The vertices of the graph represent computational elements (e.g., class member functions) and the edges model data dependencies between these elements. The idea behind Mentat is to allow the programmer to express the problem in a C++ based language, called MPL (Mentat Programming Language), which facilitates data hiding and other popular features of the C++ language. Mentat uses the dataflow model to exploit the inherent medium-grain parallelism of the program; in addition, the programmer can specify those C++ classes which are themselves of sufficient computational complexity to warrant parallel execution [Gri93].

The Mentat system consists of two major parts. The first is the MPL programming language, which is used to express the high-level abstractions of parallelism within the application. The second is Mentat's run time system (RTS).

MPL

The use of object-oriented programming languages, such as MPL, masks much of the underlying complexity from the user and is the basis for ''separating'' the user from the various machines in the HC system. The basic unit of computation in MPL is the Mentat class instance, which is similar to a C structure. The Mentat class instance consists of objects (e.g., local and member variables), their procedures, and a thread of control [GrW94].

In MPL, the standard object-oriented notions of data encapsulation and method encapsulation have been extended to include ''parallelism encapsulation'' [Gri93]. MPL supports two types of parallelism encapsulation: intraobject parallelism encapsulation, where the implementation (i.e., sequential or parallel) of a member function is hidden from the user, and interobject parallelism encapsulation, where the parallelism among member-function invocations is also

hidden from the user. For interobject parallelism encapsulation, it is the responsibility of the MPL compiler to ensure that data dependencies between invocations are satisfied and that communication and synchronization are handled correctly [Gri93]. The MPL compiler maps MPL programs onto the dataflow model by translating the MPL programs into C++ programs with embedded calls to the Mentat run time system. These C++ programs are then compiled by the host C++ compiler resulting in executable object code.

A distinguishing feature of MPL is its implementation of a construct called rtf (return-to-future) [Gri93], which is analogous to the "return" function commonly found in imperative languages such as C. The rtf construct allows Mentat member functions to return values to successor nodes in the macro-dataflow graph. These returned values are forwarded to all member functions (of the successor nodes) that are dependent on the result. The rtf function differs from a standard return in three ways. First, a member function may "rtf a value" from a Mentat-object member function that has not completed execution. Second, the execution of rtf indicates only that the associated values are ready (additional computation may be carried out after the rtf call). Finally, depending on the program's data dependency structure, rtf may not return data to its caller. In particular, if the caller does not use the resulting values locally, then the caller does not receive a copy of the values.

RTS (Run Time System)

The RTS, which initially supported execution on homogeneous parallel machines, has been extended to support HC systems. The RTS supports Mentat's macro-dataflow model via a portable virtual macro-dataflow machine. The virtual *macro-dataflow* machine provides support routines that perform run time data dependence detection, program graph construction, program graph execution, scheduling, communication, and synchronization [Gri93, GrW94]. The virtual macro-dataflow machine contains two inner components: a set of machine-independent components and libraries, and a set of machine-dependent components. One of the important features of the virtual macro-dataflow machine is that it can be ported to any supported machine in the HC system by changing only the machine-dependent components. This low-level portability allows the user to port the application source code to any machine in the supported network and have the code execute without source code changes.

The RTS has been implemented for several platforms including a network of Sun workstations, the Silicon Graphics Iris, and the Intel iPSC/2. Matrix multiplication and Gaussian elimination programs have been coded in MPL and executed on a network of eight Sun workstations and a 32 node iPSC/2. While MPL improved the ease of use of the HC system, it was indicated that the performance may not be as good as hand-coded versions that use send and receive

protocols. Thus, there is a trade-off between ease of use and some performance degradation. Future work includes the implementation of several optimizations for the MPL compiler.

5.4.5. PVM, Xab, and HeNCE

Overview

In this subsection, the PVM (Parallel Virtual Machine) system and two tools that support development of applications using PVM are overviewed. The first of the supporting tools is Xab (X-window Analysis and Debugging), which provides run time monitoring of PVM programs [Beg93]. The second supporting tool is HeNCE (Heterogeneous Network Computing Environment), which provides a high-level PVM-based environment for constructing parallel programs via directed acyclic graphs [BeD93].

PVM

PVM is a software system that enables a collection of heterogeneous computers to be used as a coherent, flexible, and concurrent computational resource [BeD93, Sun90, Sun92]. The PVM package consists of two major parts. The first part includes system level daemons, called pvmds, which reside on each computer in the HC system. The second part is a library of PVM interface routines.

The pvmds provide services to both local processes and remote processes on other platforms in the HC system. Together, the entire collection of pvmds form what is called a "virtual machine" by enabling the HC system to be viewed as a single "meta-computer." Two of the major services provided by the pvmds are communication and synchronization. Processes communicate via the use of messages. The messages are exchanged asynchronously so that a sending process may continue execution without waiting for an acknowledgment from the receiving process. The other major service provided is the synchronization among processes. Synchronizations can be accomplished by using barriers or by using event rendezvous. The synchronizations may be among multiple processes that are executing on a local machine and/or be among processes on different machines.

The second part of the PVM package is a library of interface routines. Applications developed with PVM must be linked with this library. Applications to be executed on one or more computing platforms in the HC system are able to access these platforms via library calls embedded in imperative procedural languages such as C or FORTRAN. The library routines interact with the pvmd (resident on each machine) to provide services such as communication, synchronization, and process management. The pvmd may provide the requested service alone

or in cooperation with other pvmds in the HC system.

From the user's point of view, the PVM system can be conceptualized as a three-level hierarchy. At the uppermost layer, which is the interface to the programmer, is the concept of an instance (or process), which is the basic unit of computational abstraction in PVM. Applications developed with PVM generally consist of several instances (possibly executing concurrently) that cooperate across machine boundaries. The middle layer is defined as the virtual machine layer. The virtual machine layer consists of the pvmds that reside on the machines of the HC system. The lowest layer is the actual set of machines in the HC system.

The computational resources in the HC system may be accessed using three different modes: 1) the transparent mode in which instances are automatically located at the most appropriate sites based upon a user-specified cost matrix, 2) the architecture-dependent mode in which the user can indicate specific architecture types on which particular instances are to execute, and 3) the low-level mode in which particular machines may be specified by the user. The supporting tools described in the next two subsections (Xab and HeNCE) aid the user in monitoring and developing PVM applications based on any of these access modes.

Xab

Xab is a tool developed for the run time monitoring of PVM programs [BeD93, Beg93]. The Xab tool gives the user direct feedback on what PVM functions the program is executing and how the program is performing in a heterogeneous environment. Xab consists of three parts: the Xab library, which contains instrumented PVM routines that are linked to the user's code, a special monitoring process called admon, which receives trace messages from the library routines, and a front-end process, which graphically displays trace events.

Xab monitors a user's program by instrumenting calls to the PVM library. The instrumented calls generate events that can be displayed during program execution. The instrumentation takes place by replacing PVM calls with instrumented Xab calls. Each instrumented call not only performs its intended PVM function, but also sends an Xab event message to the admon process. (The Xab event message is itself a PVM message.) An Xab event message generally includes an event type, a time stamp, and event-specific information.

The admon process receives event messages from the instrumented PVM calls and formats them into human-readable form. These formatted event messages can be sent either to a file or to the Xab display process. At the display process, formatted messages are received from admon and displayed in an X-window. The window displays each event captured during the execution of the program. The user can single-step through these events or allow Xab to replay the events

continuously in real-time.

HeNCE

HeNCE aids users of PVM in decomposing their application into subtasks and deciding how to allocate these subtasks onto the available machines in the HC system [BeD92, BeD93, Sun92]. In HeNCE, the programmer explicitly specifies the parallelism for an application by drawing a directed graph, where nodes in the graph represent subtasks written in either FORTRAN or C. The arcs in the graph represent dependencies and flow control. In addition to subtask nodes and dependency arcs, there are four types of control constructs: conditional, looping, fan-out, and pipelining.

The user must specify a cost matrix, which represents the cost of executing each subtask on each machine in the HC system. Each cost entry is a positive integer; the higher the value the higher the cost of executing a subtask on the associated machine. The meaning of the cost parameters are defined by the user (e.g., estimated execution times or utilization costs in terms of dollars). At run time, HeNCE uses the cost matrix to estimate the most cost effective machine on which to execute each subtask.

Once the graph has been specified and the cost matrix has been defined, the HeNCE tool configures a "virtual machine" using PVM constructs. The machines that make up this virtual machine are a subset of those defined in the cost matrix. After the virtual machine is configured, HeNCE begins execution of the program. Each node in a HeNCE graph is realized by a distinct process on some machine. The nodes communicate with each other by sending parameter values needed for execution of a given node, which are specified by the user for each node (subtask). The subtasks execute in three phases. First they obtain those parameter values needed to begin execution. These parameters are obtained from predecessors of each node. If the immediate predecessors do not have all the required parameters for a node, earlier predecessors are checked until all required parameters are found. The second phase is the actual execution of the subtask. Finally, a node finishes execution and passes the needed parameters onto descendant nodes before exiting.

HeNCE can trace the execution of the heterogeneous application. The captured trace information can be displayed in real-time or replayed later. The trace tool displays active machines in the network as icons whose colors change depending on whether they are computing or communicating. The tool also displays the user's directed graph and dynamically illustrates paths of execution. These visualizations can be used in different ways. They can enable the programmer to detect bottlenecks in the application by displaying the states of the application components while the application is executing. Alternatively, the trace animation can be used for

performance tuning. After viewing the program's behavior, the programmer can reallocate subtasks across the machines in the HC system and tune the application's behavior to match the environment for subsequent executions of the application.

5.5. A Conceptual Model for Heterogeneous Computing

A conceptual model for the automatic assignment of subtasks to machines in an HC environment is shown in Figure 5.6. This model builds on the one presented in [FrS93]. In Figure 5.6, the rectangles contain actions or procedures to be performed as part of the conceptual model. The ellipses show the information used and/or created by action blocks. Figure 5.6 is referred to as a "conceptual" model because no complete automatic implementation currently exists. As stated earlier, automatic decomposition and assignment is a long-term goal in the field of HC.

In stage 1 of the conceptual model in Figure 5.6, a set of descriptive parameters is generated that is represented as the general characteristics of both the computational requirements of the applications and the machine capabilities of the HC system. These parameters define the multi-dimensional decision space to be used for describing and matching subtasks and machines. Information about the expected types of application tasks to be executed and about the machines that currently exist in the heterogeneous suite are used to generate these parameters. For each parameter, a corresponding computational requirement and a corresponding machine architecture feature are derived. For example, considering the parameter "floating point operations," the computational requirements of the application tasks to be quantified are the number and types of the floating point operations needed to perform the calculation. The architecture feature of the machines in the heterogeneous suite to be quantified is the speed for these different types of floating point operations.

A particular parameter is included for further consideration in the following stages of this conceptual model only if both the related computational requirements and the architecture features exist. For example, if the given applications have no floating point operations, then it is not necessary to evaluate the machine capabilities for executing floating point operations in stage 2. As another example, if there is no vector machine available in the heterogeneous suite, vectorizable code may be excluded from the set of the computational requirements that needs to be considered.

After stage 1, a collection of corresponding features of the application tasks and machines in the heterogeneous suite can be enumerated. As stated above, these features determine the

dimensions of this automatic assignment problem for the given applications and the given HC system. Each of these dimensions represents a specific parameter, which characterizes computational requirements and the related machine capabilities, that needs to be considered in the rest of the stages of this conceptual model. The total number of features enumerated determines the complexity of this automatic assignment problem. An important aspect of the chosen parameters is that they evolve dynamically when new types of applications and/or new types of machines are added.

In stage 2, two characterization steps, task profiling and analytical benchmarking, are used to quantify these corresponding features and transform them into concrete quantitative data. Task profiling is a method used to identify the types of computational requirements that are actually present in a specific application program. The task is decomposed into computationally homogeneous subtasks, and the computational requirements for each subtask are determined. The term often used for this characterization step in the existing literature is code profiling. The reason for using task profiling in this section instead is that, to identify the types of computational requirements present in a specific task, both the code and data upon which the specified HC system will operate must be profiled. Analytical benchmarking is a procedure that provides a measure of how effectively each of the available machines in the heterogeneous suite performs on each of the types of computations being considered.

Only the computational requirements and the machine capabilities that are included in the collection of corresponding features from stage 1 are identified and evaluated by task profiling and analytical benchmarking. Recall the example above, if no vector machine is available, then task profiling does not need to search for vectorizable code in each application program. If no floating point operations are performed, then it is not necessary for analytical benchmarking to estimate the machine capabilities for those types of operations. Existing literature that presents explicit methodologies for performing task profiling and analytical benchmarking in the context of HC is reviewed in Subsection 5.6 of this section.

One of the functions of stage 3 is to use the information from stage 2 to derive, for a given application, the estimated execution time of each subtask on each machine in the heterogeneous suite and the inter-machine communication overhead associated with each possible assignment of subtasks to machines. In stage 3, these results and the information about the current loading and "status" of the machines and inter-machine network are used to generate an assignment of the subtasks to machines in the HC system based on certain cost metrics. The "status" could include such items as whether the machines/network are fully or partially functioning due to faults, and when other tasks using the machines/network are expected to complete. The most common cost metric for HC is to minimize the overall execution time (including the inter-

machine communication time) of a given application task on a particular HC system. Another interesting problem is to find the most appropriate suite of heterogeneous machines for a given collection of applications, such that the cost of the corresponding HC system is minimized for a given set of execution time constraints [Fre89]. Subsection 5.7 of this section presents a variety of techniques available in the existing literature for selecting a machine for each subtask based on certain cost metrics.

Stage 4 of this conceptual model is the execution of the given applications on the heterogeneous suite of machines in the HC system. Because the loading of the machines and network in the HC system may change and some faults may occur, sometimes it is necessary to reselect machines for certain subtasks of the application program. Under such circumstances, the current loading and status of the machines and network are updated and stage 3 is reactivated to decide the new assignment of subtasks. Finding techniques for the actual migration of a subtask from one type of machine to another in the middle of execution is a difficult problem; one approach is described in [ArS94].

It is important to note that the mathematical formulation and automation of the intelligent assignment of subtasks to a heterogeneous suite of machines connected by high-speed links are two relatively new fields in HC. Thus, most of the automatic methods that have been proposed for stages 2 and 3 of the conceptual model are frameworks that require further research before they are completely working systems. The task profiling, analytical benchmarking, and matching and scheduling techniques discussed in Subsections 5.6 and 5.7 of this section are representative frameworks.

5.6. Task Profiling and Analytical Benchmarking

5.6.1. Overview

Executing a given task by using an HC system requires identifying and profiling the subtasks in the application code. The basic approach for this, as is described in the literature, is to decompose the overall task into a collection of subtasks, where each subtask is a homogeneous code block, such that the computations within a given code block have similar processing requirements (e.g., [ChE93, FrC90, Fre89, KhP93, Sun92, WaK92]). That is, the concept of a subtask discussed in Subsection 5.5 is represented as a homogeneous code block when considering the actual implementation of the applications. These homogeneous code blocks are then assigned to different types of machines to minimize the overall execution time. In general, the goal is to assign each homogeneous code block to the best-matched machine type. In some cases, it is better not to use the best matched machine because of the overhead involved in any

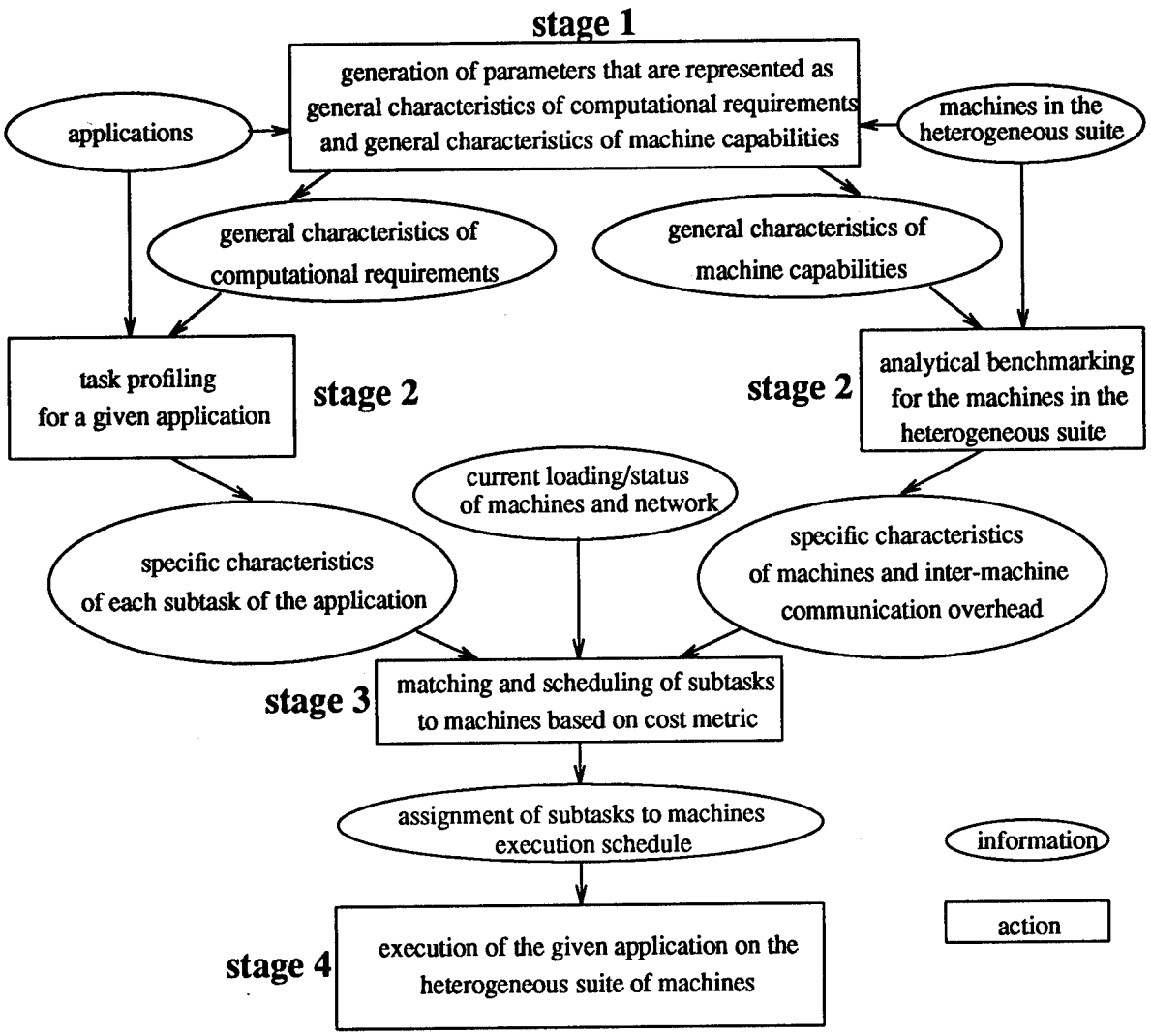


Figure 5.6: Conceptual model of the assignment of subtasks to machines in an HC environment

inter-machine data transfer that may be needed. Thus, it is important to know how well a code block and machine match with each other even when they do not form the optimal pairing. Also, communication overhead must be considered, as indicated as an input to stage 3 of the conceptual model in Subsection 5.5. This section presents example methodologies for task profiling and analytical benchmarking.

5.6.2. Definitions of Task Profiling and Analytical Benchmarking

Task profiling is a method used to identify the types of computations that are actually present in the application program and quantify how effectively each type can be executed on a particular kind of machine [Fre89]. Task profiling divides the source program into homogeneous code blocks based on the types of computations required. The definition of the set of code-types is based on the features of the machine architectures available and the computational requirements of the applications being considered for execution on the HC system. This is done in stage 1 of the conceptual model, as discussed in Subsection 5.5.

Analytical benchmarking is a procedure that provides a measure of how well each of the available machines in the heterogeneous suite performs on each of the given code-types [Fre89]. Together, the task profiling and analytical benchmarking steps provide the information needed for the matching and scheduling step, which is described in Subsection 5.7. The performance of a particular kind of machine on a specific code-type is a multivariable function. The parameters (i.e., variables) for this performance function can include the problem domain, the requirements (e.g., data precision) of the application, the size of the data set to be processed, the algorithm to be applied, the programmer's and compiler's efforts to optimize the program, and the operating system and architecture of the machine that will execute the specific code-type [GhY93].

There are a variety of mathematical formulations, collectively called selection theory, that have been proposed to choose the appropriate machine for each code block of the application program. Many of these mathematical formulations (e.g., [ChE93, KhP92, WaK92]) define analytical benchmarking as a method of measuring the optimal speedup a particular kind of machine can achieve compared to a baseline system when the best matched code-type for that machine is executed. The ratio between the actual speedup and the optimal speedup defines how well a code block is matched with each machine type, and the actual speedup, in general, is less than the optimal speedup.

5.6.3. Methodologies for Performing Task Profiling and Analytical Benchmarking

Overview

There are only a few papers in the literature that provide specific methodologies for performing task profiling and analytical benchmarking in the context of HC. These papers are the focus of this subsection.

A Comparison between Traditional Benchmarking and Analytical Benchmarking

There are a variety of benchmarking techniques used today for evaluating and comparing the performance of different computers. One of the most widely used methods is to execute a set of well-studied programs on a machine (e.g., [CoH91, DoM87]), using the total execution time as the final measure to compare that specific machine's performance with that of others. But in the context of HC, only code blocks, rather than a whole program, are executed on a specific type of computer. The overall execution time cannot illustrate the true comparative performance of a given machine when it is used for applications suited for an HC environment [Fre89]. Such traditional benchmarking techniques do not reflect the individual contributions of several underlying factors to the performance of a particular kind of machine on a specific code-type. These factors can include the mode of the parallelism, hardware architecture, compiler, operating system, I/O capacity, etc. [GhY93]. The problem with these traditional benchmarking techniques is that they are not *analytical*.

The techniques for analytical benchmarking should not only be able to show the overall execution time of a specific kind of machine on a certain type of code, but should also be able to predict future capabilities of an HC environment when new types of machines and/or new types of applications are added [Fre91]. As introduced in [Fre91], the goal of analytical benchmarking is to construct a class of relatively basic benchmarking programs for each type of computer available in the heterogeneous suite. A set of benchmarking programs can be used to derive the performance metrics of the system for a range of conditions. Thus, each performance metric is a function associated with a set of parameters, such as the size of the input data file and the type of calculations required. This is in contrast to the usual benchmarking program, whose result is just the execution time.

Parallel Assessment Window System

Parallel Assessment Window System (PAWS) is an experimental platform capable of performing machine and application evaluations for task profiling and analytical benchmarking. It consists of four tools: the application characterization tool, the architecture characterization tool, the performance assessment tool, and the interactive graphical display tool [PeG91].

Through the application characterization tool, PAWS transforms a given program written in Ada into a graph that illustrates the program's data dependencies. IF1, an acyclic graphical language, is used to generate the intermediate graphical form of the program. In IF1, basic operations, such as addition and multiplication, are represented by simple nodes, and complex constructs, such as conditional branches and loops, are represented by compound nodes. By

grouping sets of nodes and edges into functions and procedures, the application characterization tool can describe the execution behavior of a given program at various levels.

The architecture characterization tool in PAWS partitions the architecture of a specific type of machine into four categories: computation, data movement and communication, I/O, and control. Each category can be further partitioned into subsystems until the subsystems in the lowest level are fine enough to be enumerated and characterized by raw timing information. PAWS stores this hierarchical organization of subsystems in a tree data structure. The raw timing information of each leaf node of the tree can be obtained by low-level benchmarking. This hierarchical organization of architectural parameters for a specific machine provides a detailed model for determining the operational behavior of each subsystem. This facilitates analytical benchmarking in evaluating the execution time of a particular kind of machine when it is used to execute a specific type of code.

The performance assessment tool obtains information from the architecture characterization tool and generates timing information for operations on a given machine upon request. Timings for primitive operations are stored within the architecture characterization tool; the performance assessment tool uses these to determine timings for more complicated operations (e.g., complex floating point multiplication). The user provides the machine performance data for the architecture characterization tool and the parameters that define the primitive operations to be used by the performance assessment tool.

Two sets of performance parameters for an application, parallelism profiles and execution profiles, are generated by the performance assessment tool using the information provided by the application characterization tool. Parallelism profiles represent the applications' theoretical upper bounds of performance (e.g., the maximal number of operations that can be parallelized). Execution profiles represent the estimated performance of the applications after they have been partitioned and mapped onto one particular machine. Both parallelism and execution profiles are produced by traversing the applications' task-flow graph and then computing and recording each node's performance and statistically based execution time estimates.

The interactive graphical display tool is the user interface for accessing all the other tools in PAWS. It has been implemented as a hierarchical menu-driven system. The main menu allows the user to select the other three PAWS tools. Windows containing information for each of these three tools can be opened simultaneously.

The terms "task profiling" and "analytical benchmarking" are not used in PAWS. However, the objectives of parallelism and execution profiles are very similar with those of these two characterization steps.

Distributed Heterogeneous Supercomputing Management System

In [GhY93], a framework called the Distributed Heterogeneous Supercomputing Management System (DHSMS) is proposed for managing an HC environment. DHSMS introduces a systematic methodology for performing both task profiling and analytical benchmarking. The basic approach in DHSMS is to generate a Universal Set of Codes (USC) for task profiling. The USC can also be viewed as a standardized set of benchmarking programs used in analytical benchmarking. Because the method of generating USC is architecture-driven, the benchmarking programs based on USC can provide information about the hardware features of machines in an HC system.

The construction of a USC in DHSMS is based on an architecture-dependent hierarchical structure. This hierarchical structure is a detailed architectural characterization of machines available in an HC system and is similar to the hardware organization generated by the architectural characterization tool in PAWS. At the highest level of this hierarchical structure, the modes of parallelism for classifying machine architectures are selected. At the second level, finer architectural characteristics, such as the organization of the memory system, can be chosen. This hierarchical structure is organized in such a way that the architectural characteristics at any level are choices for a given category, e.g., type of interconnection network used.

To generate a USC, DHSMS assigns a code-type to each path from the root of the hierarchical structure to a leaf node. Every such path defines a set of architectural features corresponding to the nodes traversed by that path. Mathematically, a USC is defined as a set of code-types $\{C_i\}$, where $1 \le i \le K$ and K is the total number of paths from the root of the hierarchical structure to a leaf node. In this proposed framework, conceptually each C_i represents the type of code ideally suited for the architectural features indicated by the i-th path of the hierarchical structure. Thus, K is also the number of code-types available in C. A task profiling vector V_j for a given code block S_j is defined as $V_j = [v_0(j), v_1(j), v_2(j), ..., v_K(j)]$. $v_0(j)$ is the size of the parallelism (e.g., maximum possible number of concurrent threads of execution) in the given code block S_j. $v_i(j)$ $(1 \le i \le K)$ is a real number between 0 and 1 that indicates how well the code block S_j is matched with the code type C_i. The objective of task profiling in DHSMS is to estimate V_j for each S_j.

There are two points that need to be emphasized in this methodology for performing task profiling. First, in the task profiling vector V_j, the element $v_0(j)$ that quantifies the size of parallelism for code block S_j is very important. Benchmarking results for supercomputers show that the size of parallelism can affect the choice of machines used to achieve the best performance on certain programs [CoH91, DoM87]. As an example, consider the study in [Fre91] where the per-

formances of a SIMD machine and a vector machine on SAXPY code (i.e., matrix-vector calculation of the form $S = AX + Y$) are evaluated and compared. Even for a code block that is perfectly matched with the vectorizable code-type, the SIMD machine outperforms vector machine on vectors with length longer than the optimal length for the vector machine. Task profiling must, therefore, consider the size of the parallelism (in the above case, vector length) for each code block with inherent parallelism. Hence, the suggestion in Subsection 5.5 that the term "task profiling" be used instead of code profiling is very appropriate, because both code and data must be considered.

Second, the task profiling process must be repeated for each given application. A fine-grained task profiling, with all levels of architectural features incorporated into the hierarchical structure of machine characteristics mentioned above, will certainly generate a more accurate task profiling vector V_j, but the overhead associated with it increases significantly. Alternatively, a coarse-grained task profiling, which chooses only a few levels of architectural features in the corresponding hierarchical structure, can result in relatively low overhead, but the information obtained from task profiling may not be accurate enough for the subsequent procedures of matching and scheduling. Thus, there is a trade-off between the accuracy of the task profiling and the complexity of the overhead incurred [YaG93]. This trade-off is largely dependent on the number of levels of the hierarchical structure being selected in DHSMS, and this choice can be user-specified.

In DHSMS, the proposed USC is not only used as a set of code-types, but can be viewed as a standard set of architecture-dependent benchmarking programs in the following sense. Analytical benchmarking can be formally defined as a vector $B(n) = [b_q(n)]$, $q = 1, 2, ..., M$, where M is the number of machines available in the heterogeneous suite. The variable $b_q(n)$ is the speedup that machine q can achieve compared to a baseline system by executing optimally matched benchmarking programs with the size of parallelism equal to n. Conceptually, this optimally matched benchmarking program belongs to one of the code-types C_i in USC. Thus, C_i is associated with a benchmarking program that optimally matches the i-th path of the machine architecture hierarchical structure.

Because $B(n)$ only estimates the execution time that each machine spends on its best matched code-type, the inter-machine communication overheads of the application program are not evaluated. This kind of benchmarking technique is categorized as computation benchmarking in DHSMS. There are two other kinds of benchmarking techniques in DHSMS, I/O benchmarking and network-interface profiles. I/O benchmarking estimates the I/O overhead of a given architecture as a performance metric that is a function of the amount of data being transmitted through the I/O subsystem. *Network-interface* profiles estimate the overhead of the

network due to the protocols for communication and media access. Both types of benchmarking techniques are necessary for accurate matching and scheduling in an HC system discussed in stage 3 of the conceptual model.

I/O benchmarking and network-interface profiles are defined by a vector of length M, which is called the communication overhead vector $D(a_m) = [d_1(a_m), d_2(a_m), \ldots, d_M(a_m)]$. Each element $d_q(a_m)$ of $D(a_m)$ represents the destination-independent expected I/O and network-interface overhead of machine q, when there are a_m units of data transmitted through the m-th edge of the data dependence graph of the original program. In reality, the amount of data being transmitted through the network may not be deterministic, in which case some stochastic performance measures are required.

By systematically applying the task profiling and analytical benchmarking techniques described above, DHSMS can generate a code-flow graph (CFG) for the subsequent procedures of matching and scheduling. The process begins with a task-flow graph (TFG), which provides the execution time of each code block S_j on a baseline system and the amount of data transferred between code blocks due to data-dependencies. By using the information generated by task profiling, a task profiling vector V_j is assigned to each code block S_j in TFG, forming an intermediate CFG. The length of V_j and the complexity of task profiling each depend on the number of levels of the hierarchical structure selected by the user. In the final CFG, each code block S_j in the intermediate CFG is associated with an estimated computation time vector $E_j = [e_1, e_2, \ldots, e_M]$, where e_q $(1 \le q \le M)$ is the estimated computation time of code block S_j on machine q and is a function of V_j and $B(n)$.

In the resulting CFG, each communication link m between two code blocks in the original TFG is associated with a communication overhead matrix $D^*(a_m) = \{d^*_{p,q}(a_m)\}$, $1 \le p, q \le M$ (in [GhY93], an asterisk is used to distinguish the communication overhead matrix D from the communication overhead vector D). The element $d^*_{p,q}(a_m)$ represents the expected I/O and network-interface overhead, when there are a_m units of data transmitted between machine p and machine q. The data format conversion overhead also can be added to $d^*_{p,q}(a_m)$. The $M \times M$ matrix $D^*(a_m)$ is assumed to be symmetric along the diagonal. Each element $d^*_{p,q}(a_m)$ is a function of both $d_p(a_m)$ and $d_q(a_m)$. where $1 \le p, q \le M$, and $p \ne q$. The resulting CFG contains detailed information about machine-dependent execution time, I/O performance, and the inter-machine communication overhead associated with each code block in the TFG. The final CFG, can be used in matching and scheduling.

The USC introduced in DHSMS is machine-dependent (i.e., depends on the characteristics of the machines in the HC system), but is not application-dependent because there is no characterization of the given applications involved during the construction of the USC.

However, the efficient management of an HC system requires a detailed analysis of both the architectures of the machines and the structures of the applications. In [YaG93], two techniques called augmented task profiling and augmented analytical benchmarking, are proposed to characterize the applications as well as the machines available in the corresponding HC system. The new augmented approach is a two level framework that combines both fine-grained and coarse-grained characterization techniques. This framework of task profiling and analytical benchmarking is based on generating a Representative Set of Templates (RST) that can characterize the execution behavior of the programs at variant levels of details.

Parametric Task Profiling and Parametric Analytical Benchmarking

In the above two methodologies for performing task profiling and analytical benchmarking, a task profiling vector is defined as a function that maps each combination of the subtasks in the application program and the elements in the set of code-types to a real number in the range [0, 1]. This real number quantifies the degree of the match between the specific subtask and the code-type. Analytical benchmarking is defined as a method of measuring the optimal speedup a certain kind of machine can achieve compared to a baseline system when the best matched code-type for that machine is executed. By combining the results from the above two characterization steps as discussed in DHSMS, the estimation of the execution times of the subtasks on the available machines in the HC system can be obtained. Most of the selection theories of HC adopt the above mathematical formulation for task profiling and analytical benchmarking (e.g., [ChE93, WaK92, NaY94]). Subsection 5.6.4 presents that mathematical formulation in detail.

The parametric task profiling and parametric analytical benchmarking proposed in [YaK94] adopt different mathematical formulations for these two characterization steps. The goal of [YaK94] is to predict the execution of a task on a single machine. At first, a set of parameters is defined such that each parameter represents a distinct category of low-level operations performed in a task. This step corresponds to stage 1 of the conceptual model for HC presented in Subsection 5.5. Then formally, in parametric task profiling, the computational task profiling of stage 2 is defined as a parametric task profiling vector $V_t = [v_1, v_2, \ldots, v_P]$ for an application task *t*. The size of V_t is *P*, where *P* is the cardinality of the parameter (operation) set. Each v_i ($1 \leq i \leq P$) of V_t represents the operation count for parameter *i*. The handling of data-dependent loop parameters and conditionals is not included in this formulation.

In parametric analytical benchmarking, a parametric computation benchmarking vector $B^m = [b_{m1}, b_{m2}, \ldots, b_{mP}]$ is also defined, where *P* is the cardinality of the parameter set also. Each b_{mi} ($1 \leq i \leq P$) represents the execution time of machine *m*, when that specific kind of machine is used to execute one occurrence of parameter *i*.

A computation estimation vector for a given application task t is defined as $E_t^{comp} = [e_1^{comp}, e_2^{comp}, \ldots, e_M^{comp}]$, where M is the number of machines available in the HC system. The element e_m^{comp} $(1 \leq m \leq M)$ represents the estimated computational time of task t on machine m, where $e_m^{comp} = \sum_{i=0}^{P} v_i b_{mi}$. v_i and b_{mi} are obtained from parametric task profiling and parametric analytical benchmarking, respectively.

Although parametric task profiling and parametric analytical benchmarking adopt a mathematical formulation that is different from the one presented in Subsection 5.6.4, this methodology for performing these two characterization steps is still compatible with the conceptual model presented in Figure 5.6. Parametric task profiling is defined as a procedure to estimate the computational requirement of the application task and parametric analytical benchmarking is defined as a method to evaluate the machine capability of the specific HC system as discussed in Subsection 5.5.

In [DiC93], a prototype software system called Automatic Heterogeneous Supercomputing (AHS) is introduced. AHS uses a method similar to the V_t and B^m vectors in [YaK94] to predict execution time. It differs from [YaK94] in several ways. Data-dependent loop parameters and conditional branch probabilities are approximated by constant values. AHS can use information about the current load on a machine to appropriately weight the expected execution time. AHS can estimate the execution time of a specific application program on a group of networked sequential UNIX machines. The inter-machine data transfers are handled by asynchronous communication through a UDP socket. AHS can generate the code for inter-machine communication automatically. A proof-of-concept functioning AHS prototype has determined the usefulness of this approach.

5.6.4. A Mathematical Formulation for Task Profiling and Analytical Benchmarking

A mathematical formulation for task profiling and analytical benchmarking can now be presented in unambiguous terms. Let CS be a code space spanned by C, where $C = \{C_i\}$ $(1 \leq i \leq K)$ is a set of code-types generated as dimensions for task profiling and analytical benchmarking. CS is a K-dimensional space, where K is the number of code-types in C. The contents of C depend on the characteristics of the applications as well as the machine architectures in a given HC system. For example, in DHSMS [GhY93], a USC is generated to be C, where C is a set of code-types for characterizing the architectures of machines in the corresponding HC system. As an example, in [YaK94] and [DiC93], the code types are individual machine instructions.

Let $S = \{S_j\}$ be a set of computationally homogeneous code blocks generated by decomposing a given application program. After task profiling, for each code block S_j, a K-dimensional vector $\Omega(j) = [\Omega_1(j), \Omega_2(j), \ldots, \Omega_K(j)]$ is generated, where $\Omega_i(j)$ is a real number in the interval [0, 1] that quantifies the degree of match between S_j and the i-th dimension of the code space CS.

Let $R = \{m_k\}$ be a set of machines in the HC system. A computation cost-coefficient vector $T = \{t_k\}$ can also be defined, where t_k is the maximal speedup a machine k can achieve compared to a baseline system when it executes the best matched code-type. The purpose of analytical benchmarking is to estimate t_k as a function of a set of parameters, such as types of operations and length of data vectors.

The amount of communication overhead depends on many factors, such as the bandwidth of the memory channels of the source and destination machines, the topology and bandwidth of the interconnection network, and the complexity of the data format conversion. A communication cost-coefficient matrix $B^*(a) = \{\delta^*_{r,s}(a)\}$, where the variable $\delta^*_{r,s}(a)$ represents the expected communication overhead incurred when there are a units of data transmitted from machine r to machine s [KhP92], is also part of analytical benchmarking. It is possible for $B^*(a)$ to be impacted during execution time due to network usage by other tasks.

The above formulation is based on the ideas presented in several papers [ChE93, Fre89, KhP92, NaY94, WaK92]. Methods for automatically determining C, S, Ω, T, and B^* are still largely open problems.

5.6.5. Summary

Definitions and example methodologies for performing task profiling and analytical benchmarking were presented in this section. Also, a mathematical formulation for these two characterization steps was given. As mentioned earlier, these formulations make many simplifying operating assumptions. Further research is needed before these formulations are practical tools that can provide the quantitative results needed in subsequent matching and scheduling techniques, examples of which are presented in the next section.

5.7. Matching and Scheduling for HC Systems

5.7.1. Overview

For HC systems, matching involves deciding on which machine(s) each code block should be executed and scheduling involves deciding when to execute a code block on the machine to which it was mapped. Mapping and scheduling problems for parallel and distributed computing systems, which are closely related to matching and scheduling problems for HC systems, have been studied extensively in the past. Much of the work in mapping and scheduling for parallel and distributed systems has focused on how to effectively execute multiple subtasks across a network of sequential processors (e.g., see [AtB92, CaK88, NiH81]). In such an environment, load balancing can be an effective way to improve response time and throughput. Although some of these existing mapping and scheduling concepts and techniques can be (and have been) applied to matching and scheduling for HC systems, there is a fundamental distinction between mapping and scheduling subtasks for a network of sequential processors (e.g., a network of workstations) and matching and scheduling subtasks for an HC system consisting of various types of parallel computers (e.g., MIMD, SIMD, and vector). In the latter case, the subtasks can be characterized based on "type of parallelism" present in each subtask to account for the fact that certain types of subtasks may execute most effectively on a particular type of parallel architecture. In general, matching subtasks to machines of the appropriate type(s) is a more important factor than merely balancing the load among all machines in the suite. This section describes some basic characteristics of matching and scheduling for HC systems and overviews some existing techniques and formulations for matching and scheduling.

5.7.2. Characterizing Matching and Scheduling for HC Systems

In HC systems, the total execution time of a task depends on the matching and scheduling techniques used as well as the local mapping and local scheduling employed on each machine in the HC system. Local mapping involves the assignment of a code block and its associated data onto the processors/memories of a given parallel architecture. Formulating and solving local mapping problems for specific types of parallel architectures is a subject of extensive research within the parallel processing community ([NoT93] is a recent thorough review on this subject). The choice of the local mapping will impact the execution time of a block, which influences matching/scheduling decisions [ChE93, NaY94]. Local scheduling is typically performed by the individual operating system of each machine in the HC system to decide when to execute multiple jobs that are assigned to run on that machine. Matching/scheduling techniques for HC systems often assume that load information such as start time and percentage of cycles available can

be obtained from local schedulers [AtB92].

In a broad sense, matching and scheduling problems can be viewed as resource management problems consisting of three main components: consumers, resources, and policy [CaK88]. In the context of HC systems, the consumers are represented by the code blocks, which are identified by task profiling. The resources include the suite of computers, the network(s) that interconnect these computers, and the I/O devices. The policy is the set of rules used by the matcher/scheduler to determine how to allocate resources to consumers based on knowledge of the availability of the resources and the suitability of the available resources for each consumer.

Matching/scheduling policies are generally designed to optimize an objective function subject to a set of constraints. Minimizing the overall execution time under a cost constraint or minimizing cost under a performance constraint are two commonly used formulations for HC systems [ChE93, Fre89, WaK92]. Cost can be defined in different ways, including as a weighted sum of execution times for each machine in an existing HC system, or as the total system price (in terms of dollars) for prospective purchases. Execution time can be estimated through the analytical benchmarking and task profiling techniques discussed in Subsection 5.6, or from empirical measurements based on typical input data sets. The I/O time and network delay among machines can also be incorporated in the formulation, e.g., see [GhY93, WaA94]. Once the objective function and constraints are defined, the associated matching/scheduling problem can be solved. In many cases, matching and scheduling problems are NP-complete, thus heuristics and approximation algorithms are often used in practice to obtain solutions (e.g., [TaN93]).

Matching/scheduling techniques (i.e., policies) can be classified as either static or dynamic. Static refers to the case where the decisions of where/when to execute the various code blocks of the given task are made at compile time, and information about the code blocks (e.g., code types and execution time estimates) are available. Either no information on the load of the machines in the HC system is used, or statistically-based models and/or assumptions for these loads may be incorporated. Dynamic matching/scheduling decisions are made at run time, utilizing static information as well as information available only at run time, such as measured load. Dynamic techniques can either be non-preemptive assignments or can allow dynamic reassignments. They can be adaptive or non-adaptive, depending on whether feedback on the effectiveness of the matching/scheduling policy is used to modify the policy itself.

In the next subsection, some of the existing matching and scheduling techniques and formulations for HC systems are reviewed. This is not a complete review of research done in the area; it is presented to demonstrate the range of issues involved and overview some of the approaches proposed for solving matching and scheduling problems for HC systems.

5.7.3. Examples of Techniques and Formulations for Matching and Scheduling for HC Systems

Block–Based SIMD/SPMD Mode Selection Technique and its Extension

An SIMD/SPMD environment, such as a single mixed-mode machine (e.g., PASM [SiS95]) or an SIMD/SPMD mixed-machine system (i.e., a network of SIMD and MIMD machines), represents a special class of HC systems. In [WaS94], a block-based mode selection (BBMS) technique is proposed that uses static source code analysis of data-parallel program behavior to assign each code block to SIMD mode or SPMD mode in a mixed-mode machine. BBMS is used as a basis for a heuristic for machine selection for SIMD/SPMD mixed-machine systems in [WaA94]. In the remainder of this subsection, the application of BBMS for mixed-mode machines is overviewed first, followed by its extension to mixed-machine systems.

In the framework developed in [WaS94], the application program is assumed to be written in a mode-independent language. In a mode-independent language, operations represent the most explicit level at which program representation is identical for each mode of parallelism. Mode-independent languages make it possible to utilize the most appropriate parallel execution mode (machine) for each block of a given program.

In the BBMS framework, task profiling is done by dividing the program into code blocks. Code blocks are identified by their leading statements, called leaders. The first statement in a program is a leader, any statement that is a target of a branch at the machine code level is a leader, any statement following a conditional branch at the machine code level is a leader, and, in addition, any statement requiring a synchronization or an inter-PE data transfer and the statement that follows it are leaders. After the code blocks are defined, the program is transformed into a flow analysis tree, whose structure represents the scope levels within the program. The root of the tree represents the scope of the whole program. The non-leaf nodes represent control and data-conditional constructs. Code blocks are represented by leaf nodes of the tree. An example program segment and its associated flow analysis tree are shown in Figure 5.7.

It is assumed that leaf blocks (i.e., code blocks) are executed either completely in SIMD or completely in SPMD mode, and mode changes are allowed only at inter-block boundaries. Also, the leaf blocks are executed in an ordered sequence (from left to right) as they appear in the flow analysis tree. Thus, the schedule for executing the code blocks is static and is defined by the program itself. If a block is to be executed more than once, such as in a loop, then the mode of parallelism for that block is the same for all loop iterations. Each iteration of a loop body must begin and end execution in the same mode of parallelism (but can change modes

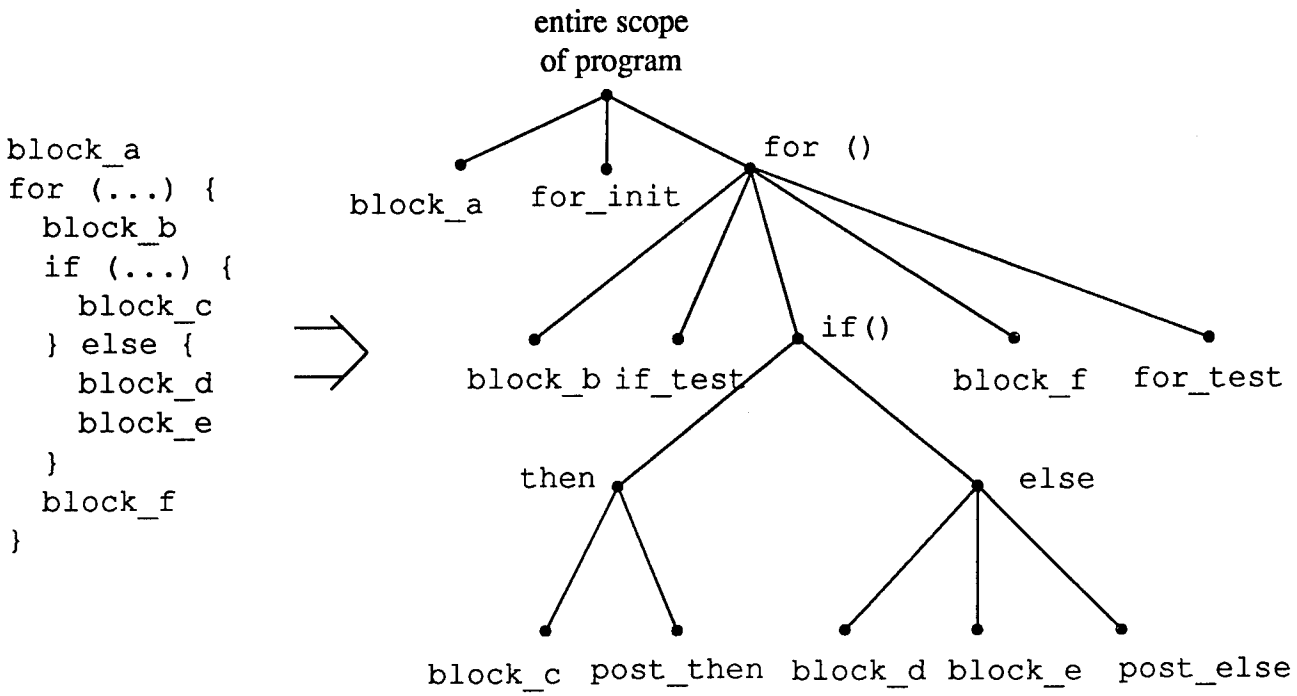


Figure 5.7: Example program segment and its associated flow-analysis tree [WaS94].

within the body). All blocks that are part of (i.e., descendants of) a data-conditional construct are implemented in the same mode of parallelism.

Execution time estimates are assumed to be known (e.g., based on the results of analytical benchmarking) for the leaf blocks in both SIMD and SPMD modes, and are denoted by T_l^{SIMD} and T_l^{SPMD} for the l-th leaf block. It is also assumed that the number of iterations for each looping construct and the probability that a PE executes the ''then'' clause of each data conditional construct are known or estimated at compile time (e.g., through compiler directives). In general, the information associated with sibling nodes at each level of the tree is combined to determine the minimum execution times for starting and ending in SIMD, starting and ending in SPMD, starting in SIMD and ending in SPMD, and starting in SPMD and ending in SIMD. These four times are determined by using a multistage optimization algorithm. Traversing the flow analysis tree using a depth first traversal, the deepest levels of the tree are combined first, and higher levels are combined until only the root node remains. Then the parallel mode for each segment of the program is assigned.

Figure 5.8 shows how the problem of selecting the best modes of execution for a sequence of sibling code blocks is transformed into a multistage optimization graph. The parameters

C^{SIMD} and C^{SPMD} represent the times for switching to SIMD and SPMD modes, respectively. From the multistage optimization graph, four shortest (in terms of time) paths, corresponding to the four minimum execution times mentioned earlier, are determined. The algorithm for the multistage optimization problem reduces a sequence of three stages to two stages by determining the shortest four paths associated with all possible starting and ending mode choices (starting at the first stage and ending at the third stage). This is repeated until only the initial and final stages remain.

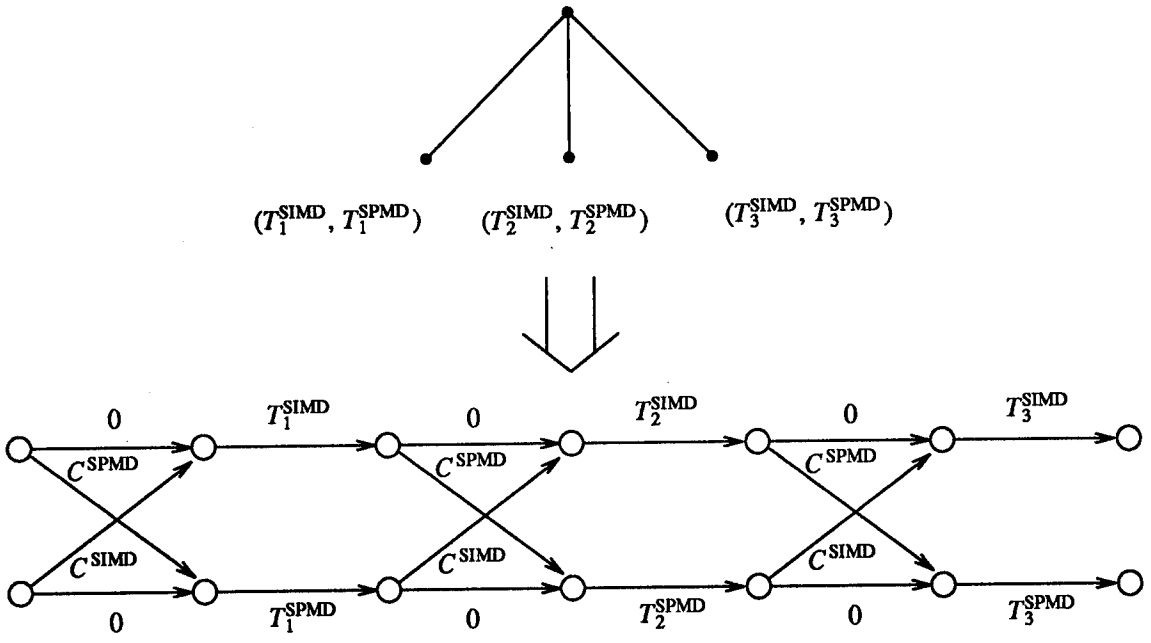


Figure 5.8: Transformation from flow-analysis tree to multistage optimization graph [WaS94].

If the parent node is a looping construct, then the (assumed) information for the number of iterations is utilized to estimate the total time for the loop. If the parent node is a data conditional construct, then the (assumed) information for the probability of executing the "then" clause is used to estimate the total time for the data conditional. The time of the shortest of the four paths at the root is the optimal mixed-mode execution time. The mode assignments corresponding to this path are then made. For more details, refer to [WaS94].

Optimal machine selection in a mixed-machine system consisting of two machines is considered in [WaA94]. The time to switch execution from one machine to the other is assumed to depend on the time to transfer the required data between machines. Thus, in contrast to the assumed constant time associated with switching modes in a mixed-mode machine, the time of switching execution from one machine to another is dependent on which machine(s) contain the data sets that are required to execute the next block, which depends on the machine choices made for executing the previous blocks, and the size of the data set to be transferred. A given machine may contain a data set because it was initially loaded there, it was received from another machine, or it was generated by that machine.

Consider a program segment consisting of a sequence of blocks ($S_0, S_1, S_2, \cdots$), where each block is to be executed on one of the two machines. For each machine, there is an associated execution time that is assumed to be known for each block. It is assumed that a collection of data structures are used to execute the sequence of blocks and for each block, a subset of these data structures is used. The data structure requirements for each block are assumed to be known and are stored in a data use (*DU*) table denoted by DU_i for block S_i. For each data structure listed in a DU table, one of three *usage types* is tabulated: read, create, or modify.

Each data structure is assigned a *cost* attribute, which corresponds to the time required to transfer the data structure between the two machines (for clarity of presentation, this cost is assumed to be independent of the source of the transfer). A *location* attribute is used to track the availability of each data structure for each machine. A data location (*DL*) table stores these as the cost and location attributes for each data structure. The value of the tabulated cost attribute depends on the location(s) of the data structure: if the data structure is on one machine only, then the cost to transfer the data structure to the other machine is tabulated; if the data structure is located on both machines, then a cost of zero is used. DL_i is used to denote the state of the data location table just before executing block S_i. Figure 5.9 shows example DU and DL tables for a program segment consisting of three blocks. In the figure, blocks S_0 and S_1 are assigned to machine Y and block S_2 is assigned to machine X (this assignment is arbitrary). T_i^X is the time required to execute block S_i on machine X.

Given the information specified above, the goal is to find an assignment of blocks to machines that results in the minimum overall execution time. In [WaA94], this problem is transformed into a multistage optimization problem similar to the one used in [WaS94]. Each time the graph is reduced, a separate DL table is kept for each of the four aggregate paths generated in the reduction step (see Figure 5.10). Because the time to switch between machines depends on past machine selections, the proposed approach may not always produce optimal assignments. For example, the algorithm may make a machine assignment for a given block that

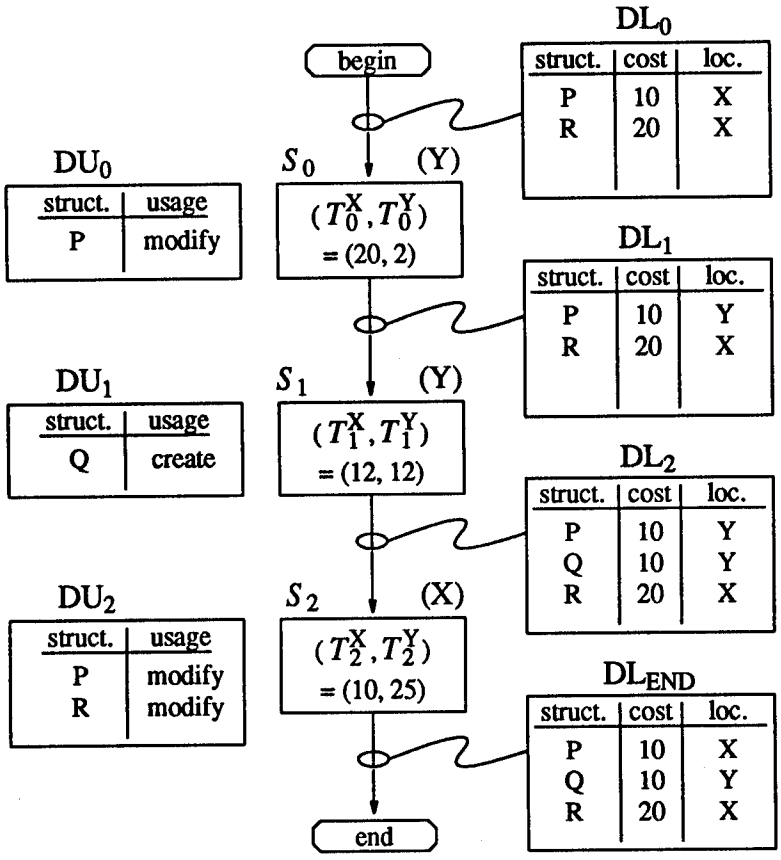


Figure 5.9: Simplified model of parallel program behavior with an arbitrary choice of machine for each code block [WaA94].

will either require a later block to read a large data structure from the other machine or use a machine that is not well suited for that block. However, simulation studies of program behaviors indicate that the proposed approach, which has a polynomial time complexity, typically produces assignments with overall execution times that are less than 1% more than the optimal assignments, which are determined using an exhaustive search that has an exponential time complexity. This research is currently being extended to more than two machines.

Optimal Selection Theory and its Extensions

A mathematical programming formulation for selecting an optimal heterogeneous configuration of machines for a given set of problems under a fixed cost constraint, known as Optimal Selec-

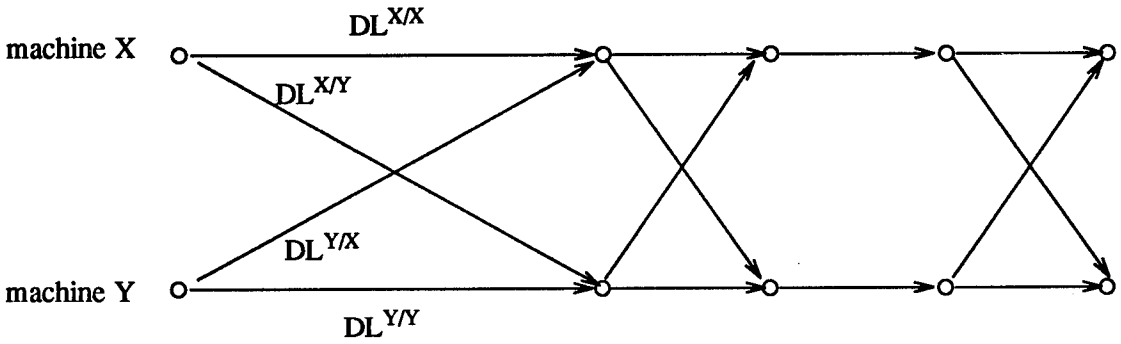


Figure 5.10: Heuristic building on the multistage technique [WaA94].

tion Theory (OST) [Fre89, Fre91], is overviewed in this subsection. An extension of OST, called Augmented Optimal Selection Theory (AOST) [WaK92], is presented (in considerable detail) to illustrate the various components of the mathematical model. Two other extensions of OST, Heterogeneous Optimal Selection Theory (HOST) [ChE93] and Generalized Optimal Selection Theory (GOST) [NaY94] are also reviewed.

In the OST framework, the application is assumed to consist of a set of non-overlapping code segments that are totally ordered in time. Thus, the total execution time of the application is equal to the sum of the execution times of all its code segments. These code segments are identified by task profiling such that each segment is homogeneous in computational requirements. A code segment is defined to be decomposable if it can be partitioned into different code blocks that can be executed on different machines of the same type concurrently. A nondecomposable code segment is a code block. The OST formulation assumes for simplicity linear speedup when a decomposable code segment is executed on multiple copies of a best matched machine type and there are always a sufficient number of machines of each type available. Various information about the code blocks and machines is assumed known, as was the case for the methodologies described in Subsection 5.6. It is noted in [Fre91] that integer programming techniques can be used with the OST formulation to solve the problem of minimizing the execution time of the application under a fixed dollar cost constraint to purchase the machines that will compose the HC suite, or minimizing the cost under a fixed execution time constraint. The solution from the OST framework shows the existence of an optimal suite of heterogeneous super-

computers for a given problem set under a fixed cost constraint.

AOST augments OST by incorporating the performance of code segments for all available machine choices (not just the best matched machine type) and by considering non-uniform decompositions of code segments. The issue of considering all available choices of machines is important in practice because the best matched machine may be unavailable.

In the formulation of AOST, five machine types are considered: vector, SIMD, MIMD, scalar, and special purpose. Each machine type may include different models (e.g., the SIMD machine type may include multiple copies of Thinking Machine's CM-2 and/or MasPar's MP-1.) Unlike the OST formulation, the number of available machines for each type is limited. For ease of presentation and without loss of generality, the case of having only one model (perhaps multiple copies) for every machine type is considered here. The details of dealing with more than one model per machine type are described in [WaK92].

The optimal speedup $\theta[\tau]$ with respect to a baseline sequential system (e.g., a VAX machine), is assumed to be estimated by analytical benchmarking based on the best matched code type for each machine type τ. For each code segment j, a five-tuple is assumed to be known from task profiling: $\omega[j] = (\pi[\text{vector}, j], \pi[\text{SIMD}, j], \pi[\text{MIMD}, j], \pi[\text{scalar}, j], \pi[\text{special},j])$, where $0 \le \pi[\tau,j] \le 1$ is an indicator of how well code segment j can be matched with machine type τ. Let S be the set of |S| non-overlapping code segments of the application task. Let μ be the number of different machine types to be considered.

The maximum number of independent code blocks into which code segment j can be decomposed for concurrent execution on machines of type τ is defined as $v[\tau,j]$, and is assumed to be known. Let $\beta[\tau]$ = number of machines of type τ available (or possible to purchase). Therefore, the actual number of code blocks into which code segment j can be decomposed is defined by $\gamma[\tau,j] = \min(v[\tau,j], \beta[\tau])$. Assume on the baseline system, p[j] = fraction of time spent executing code segment j relative to the overall execution time of S, and p[j,i] = fraction of time spent executing code block i relative to the execution time of code segment j, thus $\sum_{j=1}^{|S|} p[j]=1$ and $\sum_{i=1}^{\gamma[\tau,j]} p[j,i]=1$, for all τ, j.

The available parallelism of a code segment is defined to be the minimum number of processors that results in the optimal execution time with respect to its assumed machine model. Let $\Lambda[\tau,j]$ denote the utilization factor when running a code segment (or block) j on a machine of type τ. $\Lambda[\tau,j] = 1$ if the available parallelism of code segment j with respect to machine type τ is not less than the number of processors within machine type τ; otherwise $\Lambda[\tau,j]$ = (available parallelism) / (total number of processors). Thus, the expected actual speedup of code segment j on

machine τ is $\theta[\tau] \times \pi[\tau,j] \times \Lambda[\tau,j]$. The execution time of a decomposable code segment is the longest execution time among all its code blocks executing on the selected machines. The relative execution time for code segment j on machine type τ is given by:

$$\lambda[\tau,j] = \max_{1 \le i \le \gamma[\tau,j]} \{(p[j] \times p[j,i]) / (\theta[\tau] \times \pi[\tau,j] \times \Lambda[\tau,i])\}.$$

Code segment j is assumed to be executed on machines of type $\tau[j]$, $1 \le \tau[j] \le \mu$, for each $1 \le j \le |S|$. Thus, for a given matching of code segments to machine types (i.e., $\tau[j]$'s), the relative execution time of S is given by:

$$ET\{\tau[1], \tau[2], ..., \tau[|S|]\} = \sum_{j=1}^{|S|} \lambda[\tau[j],j].$$

Given the overall cost constraint, H, and the cost of a machine of type τ, h[t], AOST is formulated as:

$$\min_{\substack{1 \le \tau[j] \le \mu \\ 1 \le j \le |S|}} ET\{\tau[1], \tau[2], ..., \tau[|S|]\}$$

$$\text{subject to } \sum_{\tau=1}^{\mu} \left(\max_{1 \le j \le |S|} \gamma[\tau,j] \right) \times h[\tau] \le H.$$

HOST extends AOST by incorporating the effects of various local mapping techniques and allowing concurrent execution of mutually independent code segments on different types of machines. The "Hierarchical Cluster-M" model [EsF92] is discussed in [ChE93] as a way to simplify the matching process by exploiting the hierarchically clustered structure of both the system architecture and the application's communication graph.

In the formulation of HOST, it is assumed that a particular application task is divided into subtasks. Subtasks are executed serially. Each subtask may consist of a collection of code segments (as defined earlier) that can be executed concurrently. A code segment consists of homogeneous parallel instructions. Each code segment is further decomposed into several code blocks that can be executed concurrently on machines of the same type. The execution time of a subtask is equal to the longest execution time among all code segments in that subtask. Similarly, the execution time of a code segment is equal to the longest execution time among all code blocks in that segment. The underlying mathematical formulation of HOST is similar to (and a natural generalization of) that of AOST.

GOST generalizes OST and its extensions to include tasks modeled by general dependency graphs. In GOST, it is assumed that there are ω different machine types and an unlimited number of machines in each type. Different machine models are treated as different types.

In GOST, the most basic code element is a process, which corresponds to a block or a non-decomposable code segment (as defined by AOST). It is assumed that an application task consists of several processes modeled by a dependency graph, which could be generated by task profiling. Each node η_i of the graph represents a process and has a number of weights corresponding to the execution times of that process on each machine type for each mapping available on that machine. An edge of the graph represents dependencies between two processes that require communication. Each edge (η_i, η_j) has a number of weights (communication times), one for each reasonable communication path between each possible pair of host machines for processes η_i and η_j. The weights for nodes and edges are assumed to be derivable from analytical benchmarking. The objective is to determine the optimal matching/scheduling in which each process node in the dependency graph is assigned one machine type and a start time, and the completion time of the whole application is minimized using polynomial time algorithms.

Other Formulations and Solution Techniques

In [TaN93], the problem of mapping interacting code blocks of a given application task to machines in an HC system is studied. The HC system is represented by an architecture graph, in which the nodes represent the machines and the edges represent the interconnections among the machines. The application task, which is also modeled with a graph, uses nodes to represent the interacting code blocks and edges to represent data communication dependencies among the code blocks. It is assumed that the bandwidth of each link and the interface overhead between each pair of machines are known. It is also assumed that the computation time of each code block on each machine and the amount of communication required between each pair of code blocks are known. Mapping is done by assigning each code block to a machine (i.e., node in the architecture graph). The objective is to minimize the completion time of the whole program. An initial mapping is assumed at the beginning of the search. The basic actions of the proposed graph-based search are called moves. An example of a move is swapping the current locations of two code blocks. Three types of heuristics are used for attempting to find the optimal mapping. Simulations on randomly generated models are conducted to compare the solution quality and execution times among the three approaches.

In [LeP93], another graph-based method for representing problems for automatically matching code blocks to machines in an HC environment is presented. In this work, a "generalized virtual fully-connected architecture graph" is proposed as the machine abstraction and a "Meta Graph" is proposed as the abstraction for the task. In the architecture graph, each node represents a machine in the HC system and contains various machine characteristics. Each edge

represents the virtual communication link between every pair of machines, and includes information such as connectivity (i.e., direct versus indirect), connection bandwidth, physical distance, and node-pair heterogeneity (i.e., data-reformatting requirements). In the Meta Graph, the nodes represent code blocks, and edges represent control and data flows between code blocks. Classical list scheduling [Pol88] is augmented to utilize the node-pair heterogeneity representation and is used in simulations on randomly generated problems to match code blocks to machines. Based on several hundred simulations, an average improvement of approximately 70% is obtained from this implementation over the regular weighted graph implementation (i.e., without the node-pair heterogeneity information).

In [Lil93], a crossover strategy for assigning tasks on a simple HC system consisting of two machines is proposed. It is assumed that the two machines work in a client/server mode. The proposed strategy is used by the client to decide when the speedup of running a subtask on the server can compensate for the communication/interface overhead involved. When deemed to be beneficial, a remote procedure call is used to execute this subtask on the server. Two experiments were conducted on an actual HC system consisting of a Sun workstation, which functioned as the client, and a Thinking Machines CM-200, which operated as the server. The first experiment was an implementation of the "maximum subvector problem," which involves finding the maximum sum of elements of any contiguous subvector of a given real input vector. The second experiment was based on an implementation of the shallow weather prediction benchmark [Swa84]. The proposed crossover strategy was shown to make the correct choice for executing these applications (i.e., executing entirely on the client or using both the client and the server). In the first application, using both the client and the server was shown to be the proper choice provided that the vector size was larger than a critical value. For the second application, the choice was to always use (only) the client because of high communication requirements.

5.7.4. Summary

Some existing matching and scheduling techniques for HC systems were overviewed in this section. All of these frameworks, which are applicable to stage 3 of the conceptual model of Subsection 5.5, assume that information from stage 2 of the conceptual model is available and given. Although some of the proposed techniques make simplifying assumptions that may be difficult to justify in practice, the body of work reviewed represents solid research that is being conducted as important first steps in a relatively new field. More research is needed to integrate all of the stages of the conceptual model into a practical system. Specific research challenges for HC are discussed in the next section.

5.8. Conclusions and Future Directions

Although the underlying goal of HC is straightforward — to support computationally intensive applications with diverse computing requirements — there are a great many open problems that need to be solved before heterogeneous computing can be made available to the average applications programmer in a transparent way. Many (possibly even most) need to be addressed just to facilitate near-optimal practical use of real heterogeneous suites in a "visible" (i.e., user specified) way. Below is a brief informal discussion of some of these open problems; it is far from exhaustive, but it will convey the types of issues that need to be addressed. Others may be found in [KhP93, Sun92].

Implementation of an automatic HC programming environment, such as envisioned in Subsection 5.5, will require a great deal of research for devising practical and theoretically sound methodologies for each component of each stage. A general open question that is particularly applicable to stages 1 and 2 of the conceptual model is: "What information should (must) the user provide and what information should (can) be determined automatically?" For example, should the user specify the subtasks within an application or can this be done automatically? Future HC systems will probably not completely automate all of the steps in the conceptual model. A key to the future success of HC hinges on striking a proper balance between the amount of information expected from the user (i.e., effort) and the level of performance delivered by the system.

To program an HC system, it would be best to have one or more machine-independent programming languages that allow the user to augment the code with compiler directives. The programming language and user specified directives should be designed to facilitate (a) the compilation of the program into efficient code for any of the machines in the suite, (b) the decomposition of tasks into homogeneous subtasks, and (c) the use of machine-dependent subroutine libraries.

Along with programming languages, there is a need for debugging and performance tuning tools that can be used across an HC suite of machines. This involves research in the areas of distributed programming environments and visualization tools.

Operating system support for HC is needed. This includes techniques applicable at both the local machine level and at the system-wide network level.

Ideally, information about the current loading and status of the machines in the HC suite and the network that is linking these machines should be incorporated into the matching and scheduling decisions. Many questions arise here: what information to include in the status (e.g., faulty or not, pending tasks), how to measure current loading, how to effectively incorporate current loading information into matching and scheduling decisions, how to communicate and

structure the loading and status information in the other machines, how often to update this information, and how to estimate task/transfer completion time.

There is much ongoing research in the area of inter-machine data transport. This research includes the hardware support required, the software protocols required, designing the network topology, computing the minimum time path between two machines, and devising rerouting schemes in case of faults or heavy loads. Related to this is the data reformatting problem, involving issues such as data type storage formats and sizes, byte ordering within data types, and machines' network-interface buffer sizes.

Another area of research pertains to methods for dynamic task migration between different parallel machines at execution time. This could be used to rebalance loads or if a fault occurs. Current research in this area involves how to move an executing task between different machines and determining how and when to use dynamic task migration for load balancing.

Lastly, there are policy issues that require system support. These include what to do with priority tasks, what to do with priority users, what to do with interactive tasks, and security.

In conclusion, there is clearly a gap between the state-of-the-art in practical HC computing (briefly illustrated in Subsections 5.2 through 5.4) and automating all of the steps characterized by the conceptual model of Subsection 5.5 (and discussed in Subsections 5.5 through 5.7). In particular, stages 1 through 3 of the conceptual model are typically done entirely by the user, while some aid is provided for the user for stage 4 by existing tools and environments. Thus, although the uses of existing HC systems demonstrate the significant potential benefit of HC, the amount of effort currently required to implement an application on an HC system can be substantial. Future research on the above open problems will improve this situation and make HC more viable.

6. Estimating the Distribution of Execution Times for SIMD/SPMD Mixed-Mode Programs

6.1. Introduction

A heterogeneous computing (HC) system provides a variety of architectural capabilities, orchestrated to perform an application whose subtasks have diverse execution requirements [SiA95]. Two types of HC systems are mixed-mode machines and mixed-machine systems. A mixed-mode machine is defined here as a single parallel processing machine that is capable of operating in either the synchronous SIMD or asynchronous MIMD [Fly66] mode of parallelism and can dynamically switch between modes at instruction-level granularity [SiA95]. A mixed-machine system is a suite of independent machines of different types interconnected by a high-speed network.

The SPMD (single program - multiple data) mode of parallelism is a special case of MIMD in which the processors execute the same program asynchronously on their own data [DaG88]. Applications for this study are assumed to be data parallel programs written in a mode-independent language for execution on an SIMD/SPMD mixed-mode machine. Examples of mixed-mode machines include PASM [SiS95], TRAC [LiM87], Triton [PhW93], OPSILA [DuB88], and EXECUBE [Kog94]. The model of a mixed-mode machine assumed here is a distributed memory machine, in which each processor is paired with a memory module to form a processing element (PE). When PEs switch mode, all that changes is the source of their instructions. For SIMD mode, PEs receive their instructions from a common control unit (CU), while in SPMD mode, each PE fetches its instructions from its own memory module.

Studies on how to make effective use of the heterogeneity present within mixed-mode machines can provide useful insights for how to make effective use of mixed-machine systems. For example, in [WaS94], an optimal mode selection technique was developed for mixed-mode machines that was later generalized for use with mixed-machine systems in [WaA94]. Similarly, much of the work presented here for predicting execution times for mixed-mode machines may be applicable and/or adaptable to mixed-machine systems.

To effectively utilize the computational resources in an HC system (i.e., modes of paral-

---

The co-authors of this section were Yan Alexander Li, John K. Antonio, Howard Jay Siegel, Min Tan, and Daniel W. Watson.

This work was supported by Rome Laboratory under contract number F30602-94-C-0022.
This material will appear in the *Proceedings of the Fourth Heterogeneous Computing Workshop (HCW '95)*, sponsored by the IEEE Computer Society, April 1995.

lelism and/or machines) for executing a task consisting of a set of subtasks, it is very important to be able to predict the execution times of different subtasks for each mode/machine in the system, because performance prediction is the basis of matching and scheduling for HC systems. Many matching and scheduling algorithms make the simplifying assumption that the execution time for each subtask is a known constant for each mode/machine in the system (e.g., [Fre89, WaA94, WaS94]). However, there are elements of uncertainty, such as the uncertainty in input data values or in inter-machine communication time, which can impact the execution times. Mode/machine choices for executing subtasks can also affect the execution time and its degree of uncertainty. For example, in a mixed-mode machine during a MIMD to SIMD mode switch, all PEs must wait for the last one to finish its MIMD execution before entering the synchronous SIMD mode. Thus, the amount of time a particular PE waits depends not only on when it finishes MIMD execution, but also on when the last PE finishes MIMD execution.

Consider the execution of the loop in Fig. 6.1 on a mixed-mode machine. The loop body contains subtasks A and B. Assume the execution times of each subtask vary among the PEs (because the execution time of each subtask depends on input data values that vary across all PEs) and the loop control overhead time is ignored. Assume that, when executed independently, the average execution time of subtask A is minimal in SIMD mode and the average execution time of subtask B is minimal in MIMD mode. In the figure, the wide rectangles spanning horizontally across all PE labels represent the execution time of a subtask in SIMD mode. The thin rectangles under each PE label represent the execution time of a subtask in MIMD mode on each PE. The rectangles are shaded differently to represent the execution times of subtask A and subtask B.

Intuitively, executing subtask A in SIMD mode and subtask B in MIMD mode should be faster than any other combination (because subtask A is fastest in SIMD and subtask B is fastest in MIMD). This intuition is correct provided that the variation in execution time across the PEs is sufficiently small, as shown in Fig. 6.1(a). However, as shown in Fig. 6.1(b), if the variation is large across the PEs, then it is possible that executing everything in MIMD mode is fastest due to the effect of block juxtaposition, even if the average execution of subtask A in MIMD mode is larger [BeK91]. Therefore, incorporating only information for average execution times may lead to incorrect mode selections.

This study introduces a methodology for statically estimating the distribution of execution times for a given data parallel program to be executed in an SIMD/SPMD mixed-mode computing environment. The program is assumed to contain operations whose execution

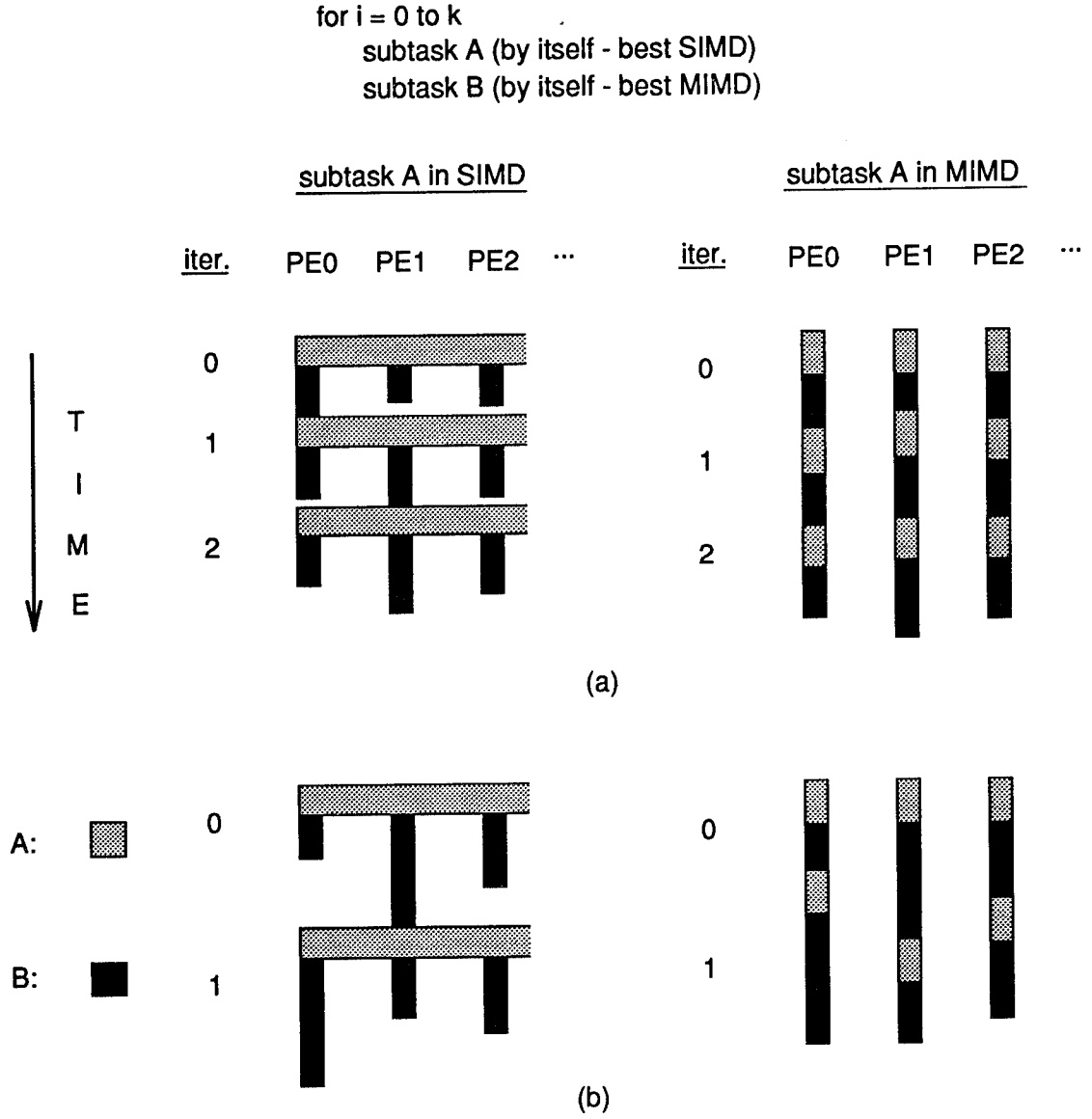


Figure 6.1: Execution of a loop with variable execution time subtasks: (a) low variance for execution time of subtask B; (b) high variance for execution time of subtask B.

time behaviors depend on input data values that cannot be perfectly predicted at compile time. For instance, in the example shown in Fig. 6.1, the number of iterations executed by the looping construct may be data dependent. Also, each subtask within the loop body may themselves contain looping constructs where the number of iterations to be executed is uncertain. Probabilistic models are constructed to model these types of uncertainties, e.g., a probability distribution function is used to represent the number of iterations executed by each looping construct. The aggregate effect of these elements of uncertainty on total execution time of the program is captured by computing the probability distribution for the total execution time.

In the proposed methodology, a block-based approach is used to transform the application program into a flow analysis tree in which the internal nodes represent control or data conditional constructs and the leaf nodes represent basic code blocks [AhS86]. The methodology takes as input the structure of the flow analysis tree, the mode in which each node in the flow analysis tree is to be executed (SIMD or SPMD), execution time distributions for all operations for both SIMD and SPMD modes, and an appropriate probabilistic model for each control and data conditional construct. Based on this information, the distribution of execution times for the entire program is computed. Deriving this proposed methodology for combining statistical information about a SIMD/SPMD mixed-mode program is the focus of this section.

Subsection 6.2 presents the basic assumptions and a brief overview of the proposed approach. Subsection 6.3 reviews some basic probability theory that is used in later subsections. Methods for computing the execution time distribution of a single code block in either SIMD or SPMD mode are discussed in Subsection 6.4. The methods for estimating execution time distributions of an entire program in single and mixed-mode are described in Subsections 6.5 and 6.6, respectively. A numerical example is given in Subsection 6.7 to demonstrate the effect of mode choices on the distribution of total execution time.

6.2. Overview of the Approach

Applications for this study are assumed to be data parallel programs written in a mode-independent language for execution on an SIMD/SPMD mixed-mode machine. A mode-independent language (e.g., ELP [NiS93]) is a language whose syntactic elements have interpretations under more than one mode of parallelism. In a mode-independent language, operations represent the most explicit level at which the program representation is identical for each mode of parallelism. Mode-independent languages make it possible to utilize the

most appropriate parallel execution mode for each block of a given program.

As in the BBMS (block-based mode selection) framework introduced in [WaS94], a flow-analysis tree is used to represent the application program. The application program is divided into code blocks, identified by their leading statements called leaders [AhS86]. The first statement in a program is a leader, any statement that is a target of a branch at the machine-code level is a leader, any statement following a conditional branch at the machine-code level is a leader, and any statement requiring or following a synchronization or an inter-PE communication is a leader. After the code blocks are defined, the program is transformed into a flow analysis tree, whose structure represents the scope levels within the program. The root of the tree represents the scope of the entire program. The non-leaf nodes represent control and data-conditional constructs. Code blocks are represented by the leaf nodes of the tree. An example program and its associated flow analysis tree are shown in Fig. 6.2. A simple model for the language is assumed here, as in [WaS94].

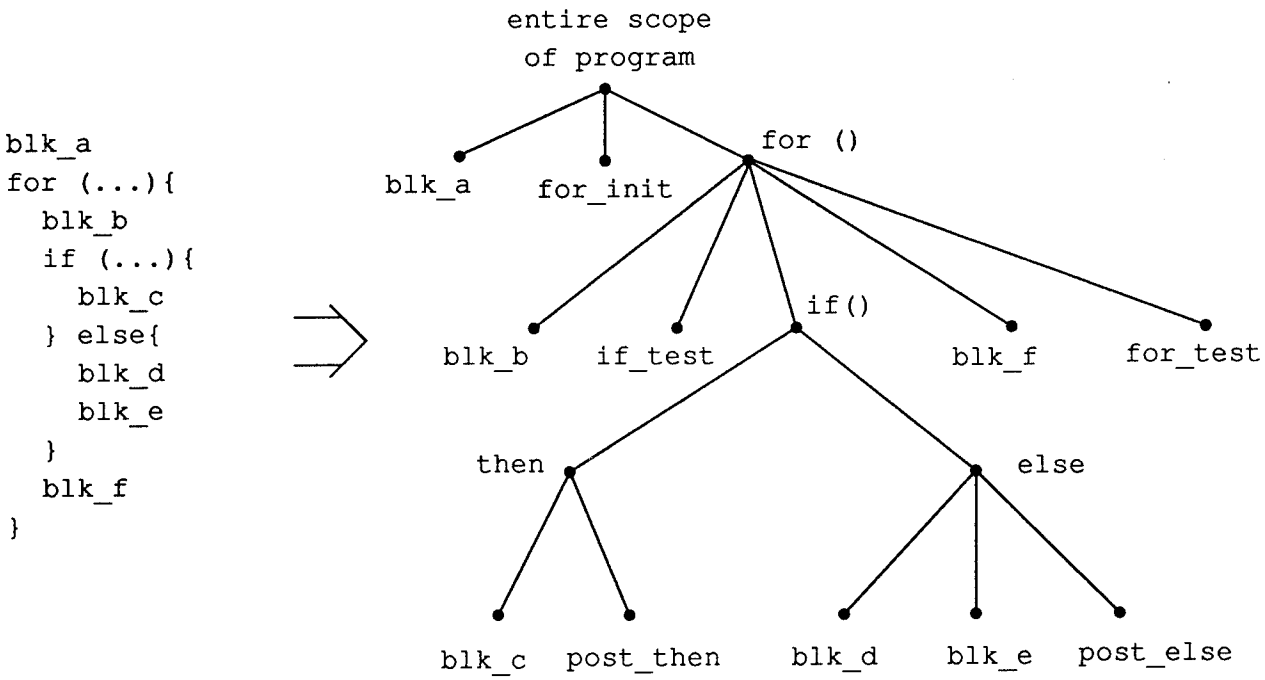


Figure 6.2: Example program and its associated flow analysis tree [WaS94].

It is assumed that leaf blocks (i.e., code blocks) are executed completely in either SIMD or SPMD mode, and mode changes are allowed only at inter-block boundaries. It is also assumed that the sibling nodes are executed in an ordered sequence (from left to right) as they appear in the flow analysis tree. Thus, the schedule for executing the code blocks is

static and is defined by the program itself. If a block is to be executed more than once, such as in a loop, then the mode of parallelism for that block is the same for all loop iterations. Each iteration of a loop must begin and end execution in the same mode of parallelism (or else a mode switch would need to be added to make this true). All blocks that are part of (i.e., descendants of) a data-conditional construct are executed in the same mode of parallelism (e.g., this is a requirement in the operation of PASM to avoid complex and costly bookkeeping overhead).

The execution time of each basic operation within a code block in each mode on a single PE can either be deterministic (i.e., have a constant value) or can assume different values, each with a specified probability. In both cases, a discrete random variable is used to model the execution time, where the former is a special case in which the random variable assumes the constant value with probability 1 (i.e., it is deterministic). Examples of such operations are architecture dependent and may include inter-PE communications and floating point operations. Estimates for the execution time distribution of each operation in each mode is assumed to be known. This information is assumed to be application independent, thus it can be measured (i.e., empirically estimated) for each mode and stored in a database for future reference.

It is assumed that the branching probability of each data conditional construct and the distribution for the number of iterations each loop will execute are application/data dependent and are available from the application programmer (e.g., in the form of compiler directives). In general, the more accurate the information the application programmer provides, the better the prediction that can be made on the distribution of the execution time of the entire program. For example, a program may contain an exception-handling branch that takes a long time to execute. If the application programmer can indicate that this exception occurs only rarely (i.e., with low probability), then it can be predicted that the total execution time distribution is affected only slightly. Otherwise, an arbitrary assumption, such as using a branching probability of approximately one-half, gives a more pessimistic estimate for the total execution time distribution. In addition to getting information directly from the application programmer, empirical information can be derived based on a number of measured execution times.

Given the above information and the mode of parallelism (i.e., SIMD or SPMD) in which each code block is to be executed, the distribution of the execution time of each block—which corresponds to a leaf node in the flow analysis tree—is computed. After that, traversing the flow-analysis tree in depth-first order, each lowest level subtree is pruned. Repeating this

step, the entire flow-analysis tree is pruned, and the execution time distribution of the whole program is computed. In the next subsection, basic probability theory, which forms the basis of our analysis, is reviewed.

6.3. Basic Probability Theory

The purpose of this subsection is to provide an overview of relevant concepts from basic probability theory. Additional definitions and derivations can be found in textbooks on the subject (e.g., [MoG74]).

Define a sample space to be the collection of all possible outcomes of a conceptual experiment. Define an event to be a subset of the sample space. Define a probability function, denoted by $\Pr[\cdot]$, to be a function that maps each event to a real number in [0, 1], which represents the likelihood that a given event occurs. In the context of this section, all possible execution times for a SIMD/SPMD program represent events within a sample space.

Define a random variable, denoted by X or $X(\cdot)$, to be a function that maps each event to a real number. For instance, suppose the execution time of a program takes on one of several possible values. The random variable X is used to map the event "program execution time = x seconds," to the real number x. A random variable X is defined to be discrete if the range of X is countable. Throughout this section, only discrete random variables are used, and the discrete values from the range of the random variable X shall be denoted by $x_i, i \geq 0$. The notation $X = x_i$ is used to denote the event that is mapped to the value x_i. The notation $X \leq x_i$ is used to denote the union of all events that are mapped to values less than or equal to the value x_i.

The density function of X is defined as:

$$f_X(x) = \begin{cases} \Pr[X = x_i], & \text{if } x = x_i,\ i = 0, 1, \ldots \\ 0, & \text{otherwise.} \end{cases}$$

The distribution function of X is defined as:

$$F_X(x) = \Pr[X \leq x].$$

Three basic properties of a distribution function are: (1) $F_X(-\infty) = 0$, (2) $F_X(\infty) = 1$, and (3) $\forall x_j \geq x_i,\ F_X(x_j) \geq F_X(x_i)$. Also, as shown below, the distribution function can be derived from the density function and vice versa:

$$F_X(x) = \sum_{\{i\ :\ x_i \leq x\}} f_X(x_i),$$

$$f_X(x_i) = \begin{cases} F_X(x_i) - F_X(x_{i-1}) & i \geq 1 \\ F_X(x_0) & i = 0. \end{cases}$$

Consider two random variables X_0 and X_1, and let $X_0 = x_{0j}$ and $X_1 = x_{1k}$ denote arbitrary events associated with these random variables. The random variables X_0 and X_1 are defined to be independent if and only if for all x_{0j} and x_{1k}

$$\Pr[X_0 = x_{0j} \cap X_1 = x_{1k}] = \Pr[X_0 = x_{0j}] \Pr[X_1 = x_{1k}].$$

Let $X_0, X_1, \ldots, X_{k-1}$ be a collection of k independent random variables defined on the same probability space and assume that the range for each of these random variables is a subset of $\{\Delta \cdot i \ : \ i = 0, 1, \ldots\}$, for some real constant Δ. Without loss of generality, assume $\Delta = 1$ for the remainder of this section.

Let the random variable Y denote the sum of the independent random variables X_0, X_1, $\ldots, X_{k-1}$, i.e., $Y = \sum_{i=0}^{k-1} X_i$. For $k = 2$, the density function of Y is the convolution of $f_{X_0}(\cdot)$ and $f_{X_1}(\cdot)$, denoted by $f_Y(\cdot) = f_{X_0}(\cdot) * f_{X_1}(\cdot)$, which is defined by:

$$f_Y(j) = \sum_{i=0}^{\infty} f_{X_0}(j-i) f_{X_1}(i), \ j = 0, 1, \ldots.$$

In general, the density function of Y is given by

$$f_Y(\cdot) = f_{X_0}(\cdot) * f_{X_1}(\cdot) * \cdots * f_{X_{k-1}}(\cdot). \tag{6.1}$$

To illustrate, consider two independent random variables X_0 and X_1 that have identical density functions defined by:

$$f_{X_0}(i) = f_{X_1}(i) = \begin{cases} \frac{1}{3}, & i = 0, 1, 2 \\ 0, & \text{otherwise.} \end{cases} \tag{6.2}$$

Therefore, the random variable $Y = X_0 + X_1$ has the following density function:

$$f_Y(i) = \begin{cases} \frac{1}{9}, & i = 0, 4, \\ \frac{2}{9}, & i = 1, 3, \\ \frac{1}{3}, & i = 2, \\ 0, & \text{otherwise.} \end{cases}$$

Let the random variable Z denote the maximum over the set of independent random variables X_0, X_1, ..., X_{k-1}, i.e., $Z = \max\{X_0, X_1, \ldots, X_{k-1}\}$. Because of the independence assumption, the distribution of Z is derived as follows:

$$\begin{aligned} F_Z(i) &= \Pr[Z \leq i] \\ &= \Pr[X_0 \leq i] \Pr[X_1 \leq i] \cdots \Pr[X_{k-1} \leq i]. \end{aligned}$$

Therefore,

$$F_Z(i) = F_{X_0}(i)F_{X_1}(i)\cdots F_{X_{k-1}}(i). \tag{6.3}$$

To illustrate, consider the two identical and independent random variables X_0 and X_1 defined in Equation (6.2). The distribution functions for these random variables are defined by:

$$F_{X_0}(i) = F_{X_1}(i) = \begin{cases} 0, & i < 0, \\ \frac{i+1}{3}, & i = 0, 1, 2, \\ 1, & i > 2. \end{cases}$$

Thus, the random variable $Z = \max\{X_0, X_1\}$ has a distribution function given by:

$$F_Z(i) = \begin{cases} 0, & i < 0, \\ (\frac{i+1}{3})^2, & i = 0, 1, 2, \\ 1, & i > 2. \end{cases}$$

In the next subsection, the approach to compute the execution time distribution of a code block using the probability theory reviewed in this subsection is introduced. Throughout the discussion, it is assumed that time is measured based on a given discrete unit and thus assumes non-negative integer values. All execution times are assumed to be modeled as discrete random variables. The case of a constant execution time is regarded as a special case where the random variable is equal to the specified constant with probability 1.

6.4. Execution Time Distribution of a Code Block

It is assumed that the execution times of an operation on all PEs are independent and identically distributed (i.i.d.), and the execution times of different operations of the program are assumed to be mutually independent. There are N PEs in the mixed-mode machine. Because a code block contains no data conditional statements, any given PE must either be enabled to execute the whole block or disabled for all operations of this block. Thus, the number of enabled PEs does not change during the execution of a code block. It should be noted that the number of enabled PEs may change during the execution of data conditional or looping constructs in SIMD mode, or during the execution of mixed-mode loops (see Subsections 6.5 and 6.6). The computational complexity for keeping track of the number of enabled PEs is very high, thus N is used instead as an approximation in this section.

The code block associated with the ℓ-th leaf node in the flow analysis tree is called code block ℓ. Let k_ℓ denote the number of operations in code block ℓ, and label the operations

in code block ℓ as $0, 1, \ldots, k_\ell - 1$. For each code block ℓ, define two arrays of random variables $I_\ell^{i,j}$ and $P_\ell^{i,j}$, $i \in \{0, 1, \ldots, k_\ell - 1\}$. The values of the random variables $\underline{I_\ell^{i,j}}$ and $\underline{P_\ell^{i,j}}$ correspond to the execution time of operation i of block ℓ executed on PE j in S*IMD* and S*PMD* modes, respectively. It is assumed that for each operation i, $I_\ell^{i,j}$ are i.i.d. for all PEs j. Likewise, for each PE j, $I_\ell^{i,j}$ are independent for all operations i. The same is assumed for $P_\ell^{i,j}$.

For each code block ℓ executed in SIMD mode, define a random variable I_ℓ, and for each operation i in this block, define a random variable $\tilde{I}_\ell^i$. The value of $\underline{\tilde{I}_\ell^i}$ corresponds to the execution time of operation i of code block ℓ in SIMD mode, and the value of $\underline{I_\ell}$ corresponds to the execution time of code block ℓ in SIMD mode. Because every operation in the block is synchronized among all enabled PEs, the execution time of an operation is the maximum of the execution times of this operation among all enabled PEs. The random variables $\tilde{I}_\ell^i$ and I_ℓ are determined from $I_\ell^{i,j}$ as follows:

$$\tilde{I}_\ell^i = \max_{j \in \{0,1,\ldots,N-1\}} \{I_\ell^{i,j}\}$$
$$I_\ell = \sum_{i=0}^{k_\ell - 1} \tilde{I}_\ell^i.$$

Thus, the distribution function of $\tilde{I}_\ell^i$ and density function of I_ℓ are:

$$F_{\tilde{I}_\ell^i}(\cdot) = \prod_{j=0}^{N_\ell - 1} F_{I_\ell^{i,j}}(\cdot) = (F_{I_\ell^{i,0}}(\cdot))^N$$
$$f_{I_\ell}(\cdot) = f_{\tilde{I}_\ell^0(\cdot)} * f_{\tilde{I}_\ell^1}(\cdot) * \cdots * f_{\tilde{I}_\ell^{k_\ell - 1}}(\cdot)$$

where $F_{I_\ell^{i,j}}(\cdot) = F_{I_\ell^{i,0}}(\cdot)$ for all j because of the i.i.d. assumption.

For each code block ℓ executed in SPMD mode, define two random variables P_ℓ and $\tilde{P}_\ell^j$. The value of $\underline{\tilde{P}_\ell^j}$ corresponds to the execution time of code block ℓ in SPMD mode on PE j, and the value of $\underline{P_\ell}$ corresponds to the execution time of code block ℓ in SPMD mode with all PEs synchronized at the beginning and end of the block. The random variables $\tilde{P}_\ell^j$ and P_ℓ are determined from $P_\ell^{i,j}$ as follows:

$$\tilde{P}_\ell^j = \sum_{i=0}^{k_\ell - 1} P_\ell^{i,j}$$
$$P_\ell = \max_{j \in \{0,1,\ldots,N-1\}} \{\tilde{P}_\ell^j\}.$$

The density function of $\tilde{P}_\ell^j$ is the convolution of the density functions of $P_\ell^{i,j}$ for all operations i, thus:

$$f_{\tilde{P}_\ell^j}(\cdot) = f_{P_\ell^{0,j}}(\cdot) * f_{P_\ell^{1,j}}(\cdot) * \cdots * f_{P_\ell^{k_\ell - 1,j}}(\cdot).$$

The distribution function of P_ℓ can be computed from the distribution function of $\tilde{P}_\ell^j$ as follows:

$$F_{P_\ell}(\cdot) = \prod_{j=0}^{N-1} F_{\tilde{P}_\ell^j}(\cdot) = (F_{\tilde{P}_\ell^0}(\cdot))^N.$$

6.5. Single-Mode Execution of a Program

6.5.1. Overview of Single-Mode Execution

This subsection presents a methodology for computing the execution time distribution of an entire program in a single mode. A method for reducing a series of blocks to a single block having equivalent execution time characteristics is introduced first. Then, methods for reducing two basic program structures, pure loop and pure data conditional construct, are presented. A pure loop is a loop whose body is currently represented as a series of one or more leaf blocks. Similarly, a pure data conditional construct is a data conditional construct for which each clause is currently represented as a series of one or more leaf blocks. It is shown that pure loop and pure data conditional construct can each be reduced to a single equivalent block. Finally, it is shown how these methods can be combined in an divide-and-conquer way to handle arbitrary program structures.

In rest of this section, it is assumed that among all PEs, the distributions for the number of iterations for each looping construct are i.i.d., and the probability that a PE executes the "then" clause of each data conditional construct is identical with and independent from that of any other PE. These distributions and probabilities are assumed to be known or estimated at compile time (e.g., through compiler directives).

6.5.2. A Series of Blocks

Consider the single-mode execution of a series of L blocks, $B_0, \ldots, B_{L-1}$. It will be shown that they can be reduced to a single block having an equivalent execution time distribution. The case of two blocks is studied first and the case of more than two blocks follows by induction.

Suppose $L = 2$ and B_0 and B_1 are both executed in SIMD mode. Because each operation is synchronized in SIMD mode, the total execution time of B_0 and B_1 is the sum of execution times of both blocks. Thus, B_0 and B_1 can be reduced to an equivalent block B with $I_B = I_{B_0} + I_{B_1}$ and $f_{I_B}(\cdot) = f_{I_{B_0}}(\cdot) * f_{I_{B_1}}(\cdot)$. By induction, for $L \geq 2$, a series of L SIMD

blocks $B_0, \ldots, B_{L-1}$ are equivalent to one SIMD block B with

$$I_B = \sum_{\ell=0}^{L-1} I_{B_\ell},$$
$$f_{I_B}(\cdot) = f_{I_{B_0}}(\cdot) * f_{I_{B_1}}(\cdot) * \cdots * f_{I_{B_{L-1}}}(\cdot).$$

Suppose $L = 2$ and B_0 and B_1 are both executed in SPMD mode. If no synchronization among the PEs at the block boundaries is explicitly specified in the source code, each PE will execute operations of both code blocks as one contiguous block. Therefore, B_0 and B_1 are equivalent to a SPMD block B whose SPMD execution time on PE j is

$$\tilde{P}_B^j = \tilde{P}_{B_0}^j + \tilde{P}_{B_1}^j,$$

Thus,

$$f_{\tilde{P}_B^j}(\cdot) = f_{\tilde{P}_{B_0}^j}(\cdot) * f_{\tilde{P}_{B_1}^j}(\cdot).$$

If all PEs are synchronized before B_0 and after B_1, then

$$P_B = \max_{j \in \{0,1,\ldots,N-1\}} \{\tilde{P}_B^j\},$$
$$F_{P_B}(\cdot) = \prod_{j=0}^{N-1} F_{\tilde{P}_B^j}(\cdot) = (F_{\tilde{P}_B^0}(\cdot))^N.$$

By induction, for $L \geq 2$, a series of L SPMD blocks $B_0, \ldots, B_{L-1}$ are equivalent to one SPMD block B with

$$\tilde{P}_B^j = \sum_{\ell=0}^{L-1} \tilde{P}_{B_\ell}^j = \sum_{\ell=0}^{L-1} \sum_{i=0}^{k_\ell - 1} P_{B_\ell}^{i,j}$$
$$f_{\tilde{P}_B^j}(\cdot) = f_{\tilde{P}_{B_0}^j}(\cdot) * f_{\tilde{P}_{B_1}^j}(\cdot) * \cdots * f_{\tilde{P}_{B_{L-1}}^j}(\cdot)$$
$$P_B = \max_{j \in \{0,1,\ldots,N-1\}} \{\tilde{P}_B^j\}$$
$$F_{P_B}(\cdot) = \prod_{j=0}^{N-1} F_{\tilde{P}_B^j}(\cdot) = (F_{\tilde{P}_B^0}(\cdot))^N.$$

6.5.3. Pure Loop

For each pure looping construct, it is assumed that the given distribution for the number of iterations to be executed by each PE is i.i.d. across all PEs. If the pure loop body consists of a series of blocks, it can be reduced to one equivalent block using the techniques discussed in the previous subsection. Therefore, the focus here is on determining the execution time distribution of a loop whose body is a single block.

Suppose node n of the flow analysis tree corresponds to a loop whose equivalent body is denoted by code block $\underline{B_{n-\text{body}}}$, and the random variable for the number of iterations to be executed by PE j is $\underline{\tilde{R}_n^j}$, where $\forall r \leq 0$, $f_{\tilde{R}_n^j}(r) = 0$. For each possible (i.e., non-zero probability) number of iterations r, unroll the loop into a series of r blocks (i.e., repeat loop body block $B_{n-\text{body}}$ r times). The density for the execution time of this series can be estimated using the summation method discussed in the previous subsection. The density for the execution time of the entire loop is the weighted sum of the densities of execution times for the unrolled loop for all possible number of iterations (the densities are weighted by the probability of executing the associated number of iterations).

Because the number of iterations to be executed by each PE can be distinct, bounds for looping iterations are assumed to be based on local data from each PE. The case where loop bounds are defined based on data from the central CU (in SIMD mode) is a simpler case, and is not considered here.

When executing a loop in SIMD mode, r iterations of the loop body $I_{n-\text{body}}$ are equivalent to a single SIMD block, whose execution time is $I_{r-\text{iteration}}$, where its density function is given by:

$$f_{I_{r-\text{iteration}}}(\cdot) = \underbrace{f_{I_{n-\text{body}}}(\cdot) * \cdots * f_{I_{n-\text{body}}}(\cdot)}_{r \text{ times}}.$$

Because the number of iterations to be executed can be different among the PEs, the CU must broadcast instructions for the PE(s) that executes the largest number of iterations (and the other PEs must wait until this PE(s) finishes its last iteration before executing operations that follow the loop). Let $\underline{R_n}$ correspond to the largest number of iterations to be executed by all PEs, then

$$R_n = \max_{j \in \{0,1,\ldots,N-1\}} \{\tilde{R}_n^j\}$$

$$F_{R_n}(\cdot) = \prod_{j=0}^{N-1} F_{\tilde{R}_n^j}(\cdot) = (F_{\tilde{R}_n^0}(\cdot))^N.$$

By the total probability theorem [MoG74]:

$$f_{I_n}(\cdot) = \sum_{r=1}^{\infty} f_{R_n}(r) f_{I_{r-\text{iteration}}}(\cdot).$$

In SPMD mode, r iterations of the loop body $\tilde{P}^j_{n-\text{body}}$ for PE j can be represented by a block $\tilde{P}^j_{r-\text{iteration}}$, with density function:

$$f_{\tilde{P}^j_{r-\text{iteration}}}(\cdot) = \underbrace{f_{\tilde{P}^j_{n-\text{body}}}(\cdot) * \cdots * f_{\tilde{P}^j_{n-\text{body}}}(\cdot)}_{r \text{ times}}.$$

Therefore, the density function for the execution time of the loop on PE j is:

$$f_{\tilde{P}_n^j}(\cdot) = \sum_{r=1}^{\infty} f_{\tilde{R}_n^j}(r) f_{\tilde{P}^j_{r-\text{iteration}}}(\cdot).$$

If there are synchronizations prior to and after completion of the loop, the random variable and its associated distribution function for the execution time across all PEs are given by:

$$P_n = \max_{j \in \{0,1,\ldots,N-1\}} \{\tilde{P}_n^j\}$$

$$F_{P_n}(\cdot) = \prod_{j=0}^{N-1} F_{\tilde{P}_n^j}(\cdot) = (F_{\tilde{P}_n^0}(\cdot))^N.$$

6.5.4. Pure Data Conditional Construct

For a pure data conditional construct, if either clause consists of a series of two or more blocks, then the series of blocks can be reduced to one equivalent block using the techniques discussed in Subsection 6.5.2. Hence, the focus of this subsection is on data conditional constructs in which each of the "then" and "else" clauses is represented by a single block.

Assume node n of the flow analysis tree corresponds to a pure data conditional construct. It is assumed that PE j executes the "then" clause with probability p_n^j, and this branching probability is independent across the PEs. Thus, PE j executes the "else" clause with probability $1 - p_n^j$. The whole data conditional construct can be represented by a single block having an equivalent execution time distribution, which is computed as follows.

In SIMD mode, instructions are broadcast by the CU. During the execution of a data conditional construct, when instructions of the "then" clause are broadcast, PEs for which the condition is false are disabled; when instructions of the "else" clause are broadcast, PEs for which the condition is true are disabled. If all PEs are to execute the same clause, then the other clause is skipped (i.e., the instructions for the other clause are not broadcast). Thus, there are three cases to consider: all PEs execute the "then" clause (with probability $(p_n^j)^N$); all PEs execute the "else" clause (with probability $(1 - p_n^j)^N$); or some PEs execute "then" and the others execute "else" (with probability $1 - (p_n^j)^N - (1 - p_n^j)^N$). Therefore, the density function of the execution time of node n is:

$$f_{I_n}(\cdot) = (p_n^j)^N f_{I_{n-\text{then}}}(\cdot) + (1 - p_n^j)^N f_{I_{n-\text{else}}}(\cdot) + [1 - (p_n^j)^N - (1 - p_n^j)^N] f_{I_{n-\text{then}}}(\cdot) * f_{I_{n-\text{else}}}(\cdot).$$

In SPMD mode, each PE fetches instructions from its own memory, thus during the execution of a data conditional construct, some PEs may execute the "then" clause while

the others are executing the "else" clause. Therefore, for each PE j,

$$f_{\tilde{P}_n^j}(\cdot) = p_n^j f_{\tilde{P}_{n-\text{then}}^j}(\cdot) + (1 - p_n^j) f_{\tilde{P}_{n-\text{else}}^j}(\cdot).$$

If all PEs are synchronized prior to and after completion of node n, the random variable and its associated distribution function for the execution time of node n are given by:

$$P_n = \max_{j \in \{0,1,\ldots,N-1\}} \{\tilde{P}_n^j\},$$
$$F_{P_n}(\cdot) = \prod_{j=0}^{N-1} F_{\tilde{P}_n^j}(\cdot) = (F_{\tilde{P}_n^0}(\cdot))^N.$$

6.5.5. Arbitrary Program Construct

It was shown in the previous two subsections that any subtree corresponding to a pure loop or a pure data conditional construct can be reduced to one equivalent leaf node. Thus a divide-and-conquer algorithm can be used to calculate the execution time distribution of an arbitrary program.

Traversing the flow-analysis tree in depth-first order, each non-leaf node traversed can only have leaf nodes as its children. Therefore, each such non-leaf node corresponds to one of the following program structures: (1) a pure loop; (2) a pure data conditional construct; or (3) a clause of a data conditional construct that contains a series of blocks. The applicable technique is then used to reduce the subtree under this node into an equivalent leaf node. This procedure is repeated until only the root node remains. If the program is executed in SIMD mode, then the execution time distribution is $f_{I_{\text{root}}}(\cdot)$, otherwise the execution time distribution is $f_{P_{\text{root}}}(\cdot)$.

6.6. Mixed-Mode Execution of a Program

6.6.1. Overview of Mixed-Mode Execution

In this subsection, a methodology is presented to estimate the execution time distribution of a program executed in mixed-mode. Many of the concepts developed in the previous subsection are used for the mixed-mode analysis.

Compared with single mode execution, during the mixed-mode execution of a program, a significant factor that affects the overall execution time distribution is the possible difference in execution time distribution for each code block (associated with SIMD and SPMD modes). In addition, mode switching overheads and the synchronization required at a mode switch

from SPMD to SIMD also play important roles in shaping the distribution. The calculation of the execution time distribution for individual code blocks is the same as in single mode execution. Also, the calculations for data conditional constructs are the same as the single mode case (because all descendants of a data conditional construct must be executed in the same mode). In contrast to single mode execution, where a series of blocks or a loop can be reduced to a single equivalent block, in mixed-mode execution, a series of blocks or a pure loop can be reduced to an equivalent series of at most three blocks. A method for combining these techniques to model arbitrary mixed-mode program execution is presented.

6.6.2. A Series of Mixed-Mode Blocks

It was demonstrated earlier that the single mode execution of a series of blocks can be modeled by an equivalent single block. Thus, a series of blocks executed in mixed-mode can be transformed into an equivalent series of alternating SIMD and SPMD blocks. A series of SIMD and SPMD blocks can begin and end with either mode, so there are four cases to consider: (1) begin with SIMD and end with SIMD; (2) begin with SIMD and end with SPMD; (3) begin with SPMD and end with SIMD; (4) begin with SPMD and end with SPMD.

Recall from Subsection 6.2 that cases (2) and (3) are possible with the root node only. For case (1), it is shown below that the series can be reduced to one single block (in SIMD mode). This result enables cases (2) and (3) to be reduced to a series of at most two blocks and case (4) to be reduced to a series of at most three blocks. For case (1), it will be shown next how a three-block series is reduced to an equivalent single block, and the case of more than three blocks follows by induction.

Let $\underline{T_{\text{to-SPMD}}}$ and $\underline{T_{\text{to-SIMD}}}$ denote the random variables whose values represent the mode-switching time from SIMD to SPMD and from SPMD to SIMD, respectively. Let B_0, B_1, and B_2 be a series of blocks executed in the order of $B_0 \rightarrow B_1 \rightarrow B_2$. Assume that B_0 and B_2 are executed in SIMD mode and B_1 in SPMD mode. Although the PEs can execute block B_1 in an asynchronous fashion, they must begin execution at the same time because block B_0 is in SIMD mode, and they must wait for the last PE to finish before they begin to execute block B_2, which is executed in SIMD mode. Thus, they can be reduced to a block B whose execution time can be calculated as follows:

$$
\begin{aligned}
I_B &= I_{B_0} + T_{\text{to-SPMD}} + P_{B_1} + T_{\text{to-SIMD}} + I_{B_2} \\
f_{I_B}(\cdot) &= f_{I_{B_0}}(\cdot) * f_{T_{\text{to-SPMD}}}(\cdot) * f_{P_{B_1}}(\cdot) * f_{T_{\text{to-SIMD}}}(\cdot) * f_{I_{B_2}}(\cdot).
\end{aligned}
$$

By induction, a series of blocks that begins and ends with SIMD mode can be reduced to one equivalent SIMD block by repeatedly applying this method and the methods from Subsection 6.5. This is referred to as an equivalent SIMD block because the PEs must be synchronized at the end of this composite block.

From the above discussion, the following conclusions are made.

1. If the series begins and ends with SIMD blocks, then it can be reduced to an equivalent single SIMD (composite) block.

2. If the series begins with an SIMD block and ends with an SPMD block, then it can be reduced to an equivalent series of an SIMD (composite) block followed by an SPMD (composite) block.

3. If the series begins with an SPMD block and ends with an SIMD block, then it can be reduced to an equivalent series of an SPMD (composite) block followed by an SIMD (composite) block.

4. If the series begins with an SPMD block and ends with an SPMD block, then it can be reduced to either a single equivalent SPMD (composite) block (if all blocks in the original series are in SPMD mode), or an equivalent series of three (composite) blocks executed in the order of SPMD $\rightarrow$ SIMD $\rightarrow$ SPMD.

For case (2), (3), and (4), the beginning and/or the ending SPMD (composite) blocks in the resulting series are the equivalent of the longest subseries of SPMD blocks from the beginning and/or the end of the original series. The key is that each series of composite blocks must begin and end with the same type of beginning and ending blocks that the original series had to be able to correctly merge with other nodes (due to synchronous nature of SIMD and asynchronous nature of SPMD).

6.6.3. Pure Loop

Recall from Subsection 6.2, it is assumed that the different blocks inside a loop may use different modes of parallelism, but must be the same for all iterations of the loop, and each iteration of a loop must begin and end execution in the same mode. Therefore, the body of a mixed-mode loop contains a series of blocks that begin and end with the same mode. From the discussion in the previous subsection, if it begins and ends with SIMD mode, then the loop body can be reduced to a single block and the single mode technique can be applied to reduce the loop to a single SIMD block. Otherwise, if it begins and ends in SPMD mode,

then the loop body can be represented as a series of three blocks executed in the order of SPMD→SIMD→SPMD. This fact will be demonstrated in the remainder of this subsection.

Suppose the loop body contains a series of three blocks B_0, B_1, and B_2, where B_0 and B_2 are in SPMD mode and B_1 is in SIMD mode. Recall that $\tilde{R}^j$ corresponds to the number of iterations that PE j is to execute the loop body. Because SIMD instructions must be broadcast by the CU, the number of iterations the CU will go through is:

$$R = \max_{j \in \{0,1,\ldots,N-1\}} \{\tilde{R}^j\}.$$

As in the single mode case, the loop is unrolled. For each possible number of iterations r, the resulting series of blocks from unrolling the loop is a series consisting of SIMD and SPMD blocks beginning with B_0 (SPMD) and ending with B_2 (SPMD). From the discussion in the previous subsection, this series can be reduced to a series of three blocks executed in the order of $B_0 \rightarrow Q_r \rightarrow B_2$, in which B_0 and B_2 are in SPMD mode and Q_r is in SIMD mode, and

$$I_{Q_r} = I_{B_1} + \sum_{k=1}^{r-1} (T_{\text{to-SPMD}} + \max_{j \in \{0,1,\ldots,N-1\}} \{(\tilde{P}^j_{B_2} + \tilde{P}^j_{B_0})\} + T_{\text{to-SIMD}} + I_{B_1}).$$

Therefore, the loop is equivalent to the series B_0, Q_r, and B_2 with probability $f_R(r)$ ($f_{R_n}(r)$ in Subsection 6.5.3). Using the total probability theorem [MoG74], an equivalent SIMD block Q can be used to represent all of the Q_r's for all possible numbers of iterations r. The density function for Q is given by:

$$f_{I_Q}(\cdot) = \sum_{r=1}^{\infty} f_R(r) f_{I_{Q_r}}(\cdot).$$

Thus, the loop is reduced to a series of three blocks: $B_0 \rightarrow Q \rightarrow B_2$.

6.6.4. Arbitrary Program Construct

As stated earlier, because all descendents of a data conditional construct must be executed in the same mode, each data conditional construct can be reduced to one block using the single mode reduction technique. It was shown in the previous subsections that a series of blocks or a pure loop can be reduced to a series of at most three blocks. As in the single mode case, a divide-and-conquer algorithm can be used to calculate the execution time distribution of the whole program. Traversing the flow-analysis tree in depth-first order, each non-leaf node n traversed can only have leaf nodes as its children. Therefore, each such non-leaf node corresponds to one of the following program structures: (1) a pure loop; (2) a pure data conditional construct; or (3) a series of blocks that is a clause of a data conditional

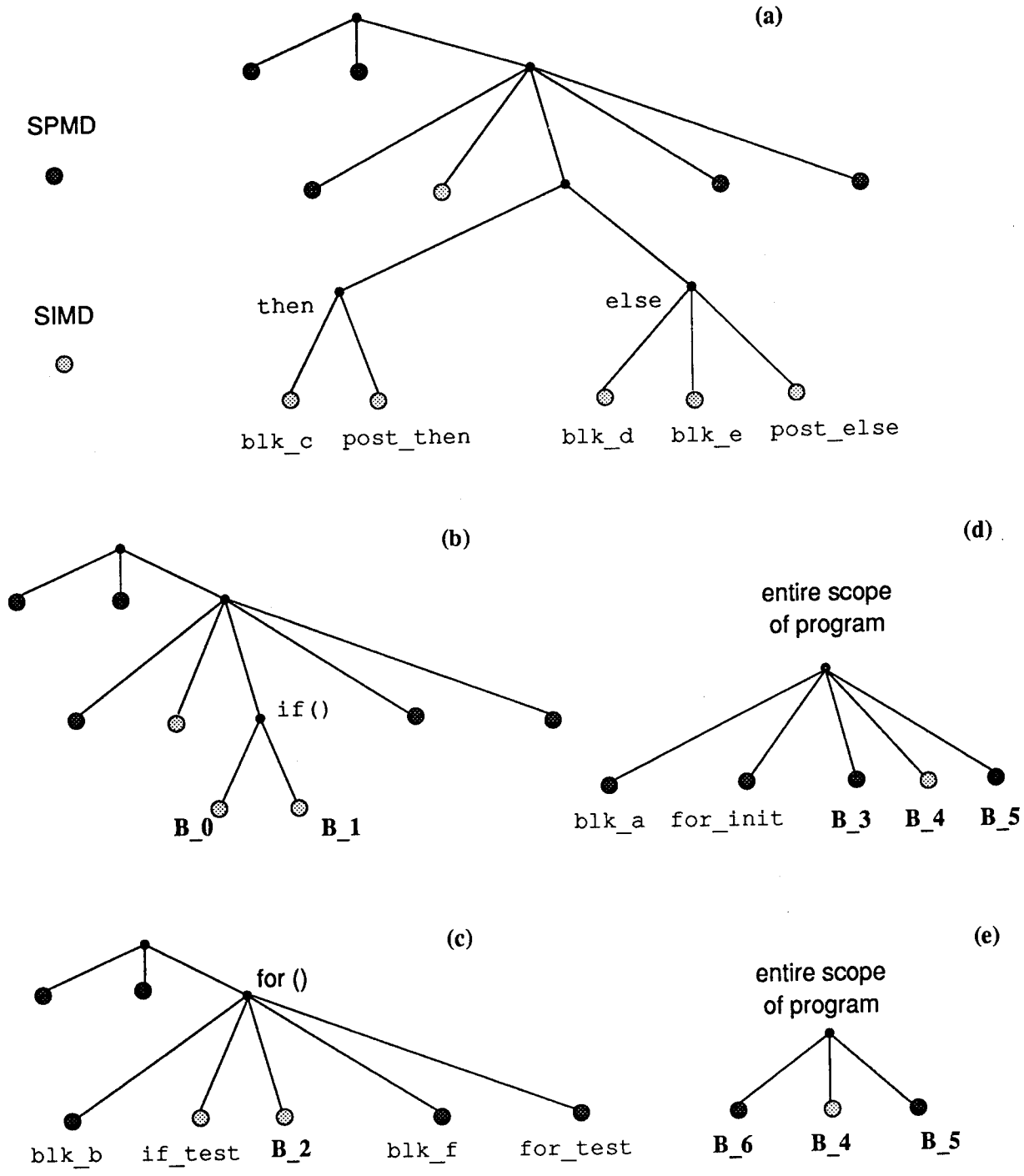


Figure 6.3: An example of mixed-mode tree reduction.

construct. Using the techniques introduced above, the subtree under node n is reduced to a equivalent series of at most three leaf nodes (a single leaf node for case (2) and (3)). Node n is then replaced with this series of leaf node(s). This procedure is repeated until only the root node is represented by a series of at most three blocks. There are four possible cases.

1. The root is represented by a SIMD node $B_{\text{root}-0}$, in which case the execution time is $I_{B_{\text{root}-0}}$.

2. The root is represented by a series of two blocks, $B_{\text{root}-0}$ and $B_{\text{root}-1}$. $B_{\text{root}-0}$ is executed in SIMD mode and $B_{\text{root}-1}$ in SPMD mode. The execution time of the program is $I_{B_{\text{root}-0}} + T_{\text{to-SPMD}} + P_{B_{\text{root}-1}}$.

3. The root is represented by a series of two blocks, $B_{\text{root}-0}$ and $B_{\text{root}-1}$. $B_{\text{root}-0}$ is executed in SPMD mode and $B_{\text{root}-1}$ in SIMD mode. The execution time of the program is $P_{B_{\text{root}-0}} + T_{\text{to-SIMD}} + I_{B_{\text{root}-1}}$.

4. The root is represented by a series of three blocks, $B_{\text{root}-0}$, $B_{\text{root}-1}$, and $B_{\text{root}-2}$. $B_{\text{root}-0}$ is executed in SPMD mode, $B_{\text{root}-1}$ in SIMD mode, and $B_{\text{root}-2}$ in SPMD mode. The execution time of the program is $P_{B_{\text{root}-0}} + T_{\text{to-SIMD}} + I_{B_{\text{root}-1}} + T_{\text{to-SPMD}} + P_{B_{\text{root}-2}}$.

Fig. 6.3 illustrates how the flow-analysis tree of Fig. 6.2 is reduced when mixed-mode execution is assumed. In the figure, part (a) shows one possible assignment of modes to leaf nodes. In part (b), each clause of the `if` statement is reduced to an equivalent SIMD block (blocks **B_0** and **B_1**). In part (c), the `if` statement itself is reduced to an equivalent SIMD block **B_2**, and the loop becomes a pure loop. In part (d), the pure loop is reduced to a series of three blocks **B_3**, **B_4**, and **B_5**, where **B_3** and **B_5** are executed in SPMD and **B_4** is executed in SIMD. The entire tree is reduced to an equivalent series of three blocks **B_6**, **B_4**, and **B_5** (part (e)), where **B_6** and **B_5** are executed in SPMD and **B_4** is executed in SIMD.

6.7. Numerical Example

To demonstrate how mode choices can affect the distribution of total execution time, numerical parameters are assumed in the flow analysis tree of Fig. 6.2 to construct a very simple example. The program is assumed to be executed on an 8-PE SIMD/SPMD mixed-mode machine. It is assumed that in each mode, the execution time for each basic operation is a constant (i.e., deterministic). Therefore, the execution time of each of the original leaf

blocks for each mode is deterministic. It is also assumed that the execution time of each operation is identical in SIMD and SPMD mode except for inter-PE communication operations, in which case SIMD mode is assumed to execute faster than SPMD mode. Only `blk_f` is assumed to contain inter-PE communications, and so it executes faster in SIMD mode than in SPMD mode. Execution times of each block is listed in Table 6.1. For each PE, the loop is assumed to execute 8, 9, 10, 11, or 12 iterations with equal probability, and the data conditional statement is assumed to execute the "then" clause with probability 0.8.

mode	blk_a	blk_b	blk_c	blk_d	blk_e	blk_f	overhead*
SIMD	12	15	10	29	23	10	1
SPMD	12	15	10	29	23	40	1

(*assumed overhead for: mode switching, `for_init`, `for_test`, `if_test`, `post_then`, and `post_else`)

Table 6.1: Assumed execution times of each block in SIMD and SPMD modes.

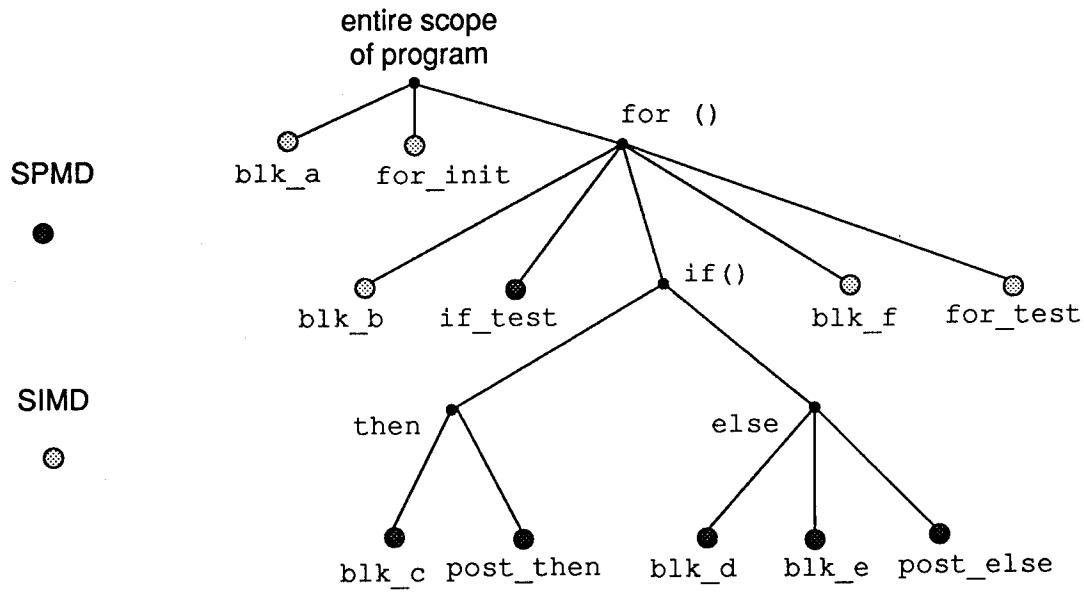


Figure 6.4: Mixed-mode execution for the numerical example.

The execution time distribution of the whole program is computed for three cases: executing the program in SIMD mode, SPMD mode, and in mixed-mode. For the mixed-mode case, the `if` statement (and its descendents) and the `if_test` are executed in SPMD mode and the rest of the program is executed in SIMD mode (Fig. 6.4). The expected values and standard deviations of the resulting distributions for program execution times are listed in Table 6.2. These tabulated statistics were computed after evaluating the density function

for each case considered. From the tabulated statistics, pure SPMD execution provides a smaller expected value and smaller standard deviation for the total execution time than pure SIMD execution (SPMD's advantages associated with effectively executing the data conditional construct and juxtaposition of SPMD blocks without inter-block synchronization exceed its disadvantage—compared to SIMD—for inter-PE communication time). However, the mixed-mode execution scheme combines the advantage of SPMD mode in executing the data conditional construct and the advantage of using SIMD mode for inter-PE communication to obtain a significantly smaller expected execution time (with a comparable standard deviation) than either pure SIMD or pure SPMD execution.

statistics of execution time	SIMD	SPMD	mixed-mode
expected value	983.1	947.6	898.6
standard deviation	76.8	61.3	62.9

Table 6.2: Expected value and standard deviation of execution times of the whole program in SIMD, SPMD, and mixed-mode.

6.8. Summary and Future Work

A methodology was introduced for estimating the distribution of execution times for a given data parallel program that is to be executed on an SIMD/SPMD mixed-mode heterogeneous system. A block-based approach was used to transform the application program into a flow analysis tree in which the internal nodes represent control and data conditional constructs and the leaf nodes represent basic code blocks. Given the mode in which each node in the flow analysis tree is to be executed (SIMD or SPMD), execution time distribution for each operation for both SIMD and SPMD modes, and an appropriate probabilistic model for each control and data conditional construct, the methodology computes the distribution of execution times for the program. A numerical example was given to illustrate the utility of the proposed technique by comparing execution time characteristics for pure SIMD, pure SPMD, and mixed-mode execution of a sample program structure. Future work includes developing optimal mode selection techniques for mixed-mode machines in which the optimality criteria can be defined in terms of various combinations of execution time statistics (e.g., a weighted combination of expected execution time and variance in execution time). Extensions of the proposed framework to compute execution time distributions for mixed-machine systems are currently under development.

7. Scheduling and Data Relocation for Sequentially Executed Subtasks

7.1. Introduction

A single application program often requires many different types of computation that result in different needs for machine capabilities. Heterogeneous computing (HC) is the effective use of the diverse hardware and software components in a heterogeneous suite of machines connected by a high-speed network to meet the varied computational requirements of a given application [FrS93, KhP93, SiA95].

The goal of HC is to decompose an application program into subtasks, and then assign each computationally homogeneous subtask to the machine where it is best suited for execution. In general, each subtask is assigned to one of the machines in the heterogeneous suite such that the total execution time (computation time and inter-machine communication time) of the application program is minimized. This subtask assignment problem is referred to as matching in HC.

There are a variety of mathematical formulations for matching, collectively called selection theory, that have been proposed to choose the appropriate machine for each subtask of an application program (e.g., [ChE93, Fre89, NaY94, WaK92]). A collection of algorithms, called graph-based algorithms in this section (e.g. [Bok81, NaY94, Sto77, Tow86]), have been developed to solve matching-related problems based on a subtask flow graph that describes the data dependencies among subtasks of an application program. As shown in Figure 7.1(a), each vertex of the subtask flow graph represents a subtask. Let $S[k]$ denotes the k-th subtask. There is an edge from $S[k]$ to $S[j]$ labeled with the variable name of the data that $S[k]$ transfers to $S[j]$ during execution. An extra vertex labeled Source denotes the locations where the initial data elements of the program are stored. The purpose of selection theory formulations and graph-based algorithms is to find the matching scheme that minimizes the total execution time of the application program. For this section, it is assumed that matching has already been done.

This co-authors of this section were Min Tan, John K. Antonio, Howard Jay Siegel, and Yan A. Li.

This research was supported by Rome Laboratory under contract number F30602-94-C-0022.
This material will appear in the *Proceedings of the Fourth Heterogeneous Computing Workshop (HCW '95)*, sponsored by the IEEE Computer Society, April 1995. Versions have appeared in (1) Purdue University Technical Report TR-EE-95-2 January 1995, and (2) MSEE thesis by Min Tan, December 1994.

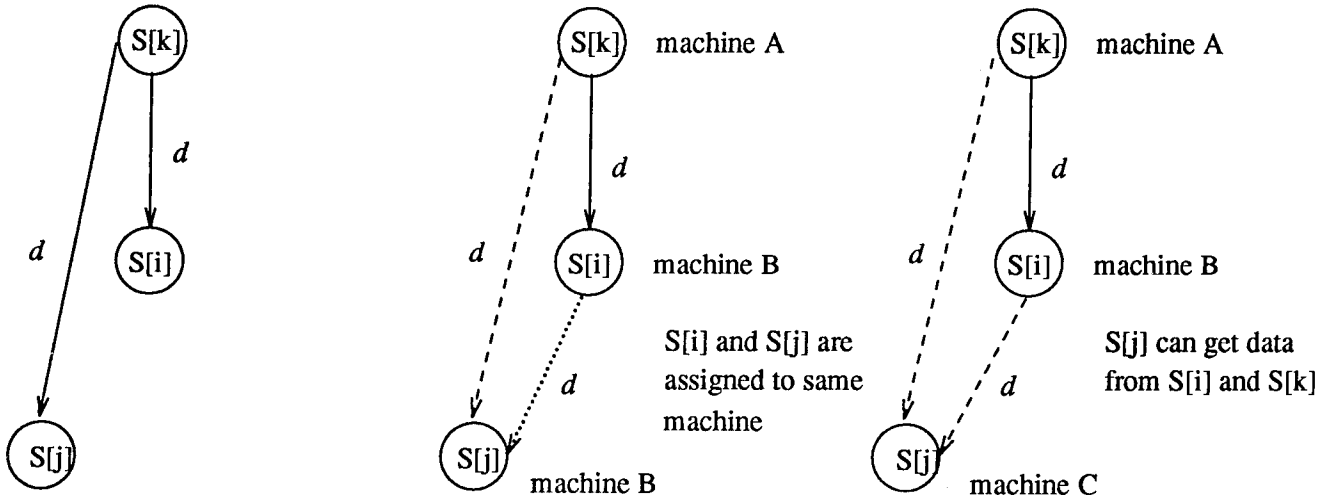


Figure 7.1: Data-distribution situations in HC. (a) Subtask flow graph. (b) Data-reuse. (c) The multiple data-copies situation.

Let a data item be a block of information that can be transferred between subtasks. For example, a data item can be an integer, an array of characters, or a large file, such as a multispectral image. Based on static (compile time) analysis, a given subtask may need as input one or more data items generated (or modified) by one or more other subtasks. Using information from the subtask flow graph, a data item is denoted by the two-tuple (s, d), where $s \geq 0$ is the number of the subtask that generates the needed value of d upon completion of execution of that subtask. For example, $(3, x)$ represents the value of variable x generated by subtask $S[3]$ upon completion of its execution. In (s, d), $s = -1$ if the needed value of d is an initial input to the program. Two data items are the same if and only if they are both associated with the same variable name in an application program and the corresponding value of the data is generated by the same subtask (which implies that the two data items have the same value).

Sequential execution of subtasks implies that at any instant in time during the execution of a specific application program P, only one subtask of P is being executed on any of the machines in the heterogeneous suite. In practice, concurrent execution of subtasks is possible, however, the simplifying assumption of sequentiality is made here as a step toward solving the more

general problem. This simplifying assumption is used by many other researchers as well (e.g., [Bok81, Sto77, Tow86]).

In general, most of the graph-based algorithms for matching-related problems assume that the pattern of data transfers among subtasks is known a priori and can be illustrated using a subtask flow graph (e.g., [Bok81, Lo88, NaY94, Sto77, Tow86]). Thus, no matter which machine is used for executing each subtask of a specific application program, the locations (subtasks) from which each subtask obtains its corresponding input-data items are determined by the subtask flow graph and independent of any particular matching scheme between machines and subtasks.

The above assumption generally needs refinement in the case of HC. Two data-distribution situations arise, namely data-reuse and multiple data-copies. It is assumed that each subtask $S[i]$ keeps a copy of each of its individual input-data items and output-data items on the machine to which $S[i]$ is assigned by the matching scheme. Data-reuse arises when two subtasks, $S[i]$ and $S[j]$, need the same data item from $S[k]$ (as in the example subtask flow graph in Figure 7.1(a)). For any data item $e = (k, d)$, $\underline{e}$ represents the value of the associated data and $|e|$ (or $|d|$) represents the size of the associated data. As shown in Figure 7.1(b), suppose the particular matching scheme is the one that assigns $S[k]$ to machine A, and both $S[i]$ and $S[j]$ to machine B. Furthermore, assume for this example that the subtasks are executed in the order k, i, then j. In this case, there is no need to transfer data item e from $S[k]$ to $S[j]$ as shown by the dashed line in Figure 7.1(b), because e is already on machine B due to the data transfer of e from $S[k]$ to $S[i]$ completed earlier (solid line in Figure 7.1(b)). If a subtask flow graph is used to compute inter-subtask communication cost, then without considering machine assignments, the impact of data-reuse is ignored.

The multiple data-copies situation arises when two subtasks, $S[i]$ and $S[j]$, need the same data item $e = (k, d)$ from $S[k]$, where $S[i]$, $S[j]$, and $S[k]$ are assigned to different machines in the HC system. In the example in Figure 7.1(c), the matching scheme assigns $S[k]$ to machine A, $S[i]$ to machine B, and $S[j]$ to machine C. Therefore, $S[j]$ can get data item e from either machine A or machine B (shown by the two dashed lines). The choice that results in the shortest time should be selected. Retrieving the needed data item from the selected source is referred to as data relocation. In general, when using information only from the subtask flow graph, the possibility of multiple sources of a needed data item due to a specific matching scheme is not considered.

When a subset of subtasks can be executed in any order and the multiple data-copies situation is considered, varying the order of the execution of these subtasks (while maintaining the data dependencies among all subtasks) can impact the execution time of the application program. Determining the sequence of execution for the subtasks is referred to as scheduling in

this section. Thus, matching determines on which machine each subtask should be executed, while scheduling determines when to execute a subtask on the machine to which it is assigned [SiA95].

The inter-machine communication time between subtasks can be substantial in an HC system. Thus, this inter-machine communication time can be a major factor in degrading the performance of an HC system. Taking the effects produced by data-reuse and multiple data-copies into account can potentially decrease this time and hence the total execution time of the application program. This section focuses on methods for minimizing the communication time of an application program with a known matching scheme. In particular, the impact of scheduling and data relocation schemes on the communication time of the subtasks executed in sequence are examined.

In Subsection 7.2, a mathematical model for matching, scheduling, and data relocation in HC is introduced. Subsection 7.3 presents a theorem, which states that, when multiple data-copies are not considered but data-reuse is taken into account, the execution time of a given application program depends only on the specific matching scheme (i.e., it is independent of scheduling). In Subsection 7.4, an extension to the usual scheduling methodology is introduced. Specifically, temporally interleaved execution of the atomic input operations of different subtasks (TIE) is considered. When considering multiple data-copies, this extension to scheduling can decrease the execution time of an application program. A minimum spanning tree based algorithm (referred to as the TIE algorithm) is described in Subsection 7.5 that finds, for a given matching, the optimal scheduling scheme for the execution of subtasks and the optimal data relocation scheme for each subtask. Both data-reuse and multiple data-copies are considered in the TIE algorithm. The correctness of the TIE algorithm is proved and an example is given. Based on this TIE algorithm, a two-stage approach for matching, scheduling, and data relocation in HC is proposed in Subsection 7.6.

7.2. A Mathematical Model for Matching, Scheduling, and Data Relocation in HC

A mathematical model for matching, scheduling, and data relocation in HC is formalized in this subsection. The model serves as the mathematical basis for Theorem 1 presented in Subsection 7.3. The TIE algorithm in Subsection 7.5 is based on this mathematical model.

(1) An application program P is composed of a set of subtasks $\underline{S} = \{ S[0], S[1], ..., S[n-1] \}$, where $\underline{n}$ is the number of subtasks in P.

(2) Suppose that $\underline{NI[i]}$ is the number of input-data items required by $S[i]$ and $\underline{NG[i]}$ is the number of output-data items generated by $S[i]$. There are two sets of data items associated

with each $S[i]$. One is the input-data set $I[i] = \{ Id[i, 0], Id[i, 1], ..., Id[i, NI[i] - 1] \}$, the other is the generated output-data set $G[i] = \{ Gd[i, 0], Gd[i, 1], ..., Gd[i, NG[i] - 1] \}$. Each $Id[i, j]$ and $Gd[i, j]$ is a data item (i.e., a two-tuple as defined in Subsection 7.1). The program structure of P is specified by a subtask flow graph. In this section, the subtask flow graph of any application program P is assumed to be acyclic. To satisfy this assumption, a combination of following two approaches is used: (i) a cycle is unrolled (conceptually) and the number of times a cycle repeats is known or estimated (as is typically done by optimizing compilers) or (ii) the whole looping construct is viewed as part of a single subtask and the boundaries for decomposing an application program into subtasks are not allowed to be in the middle of a loop.

(3) An HC system consists of a heterogeneous suite of machines $\underline{M} = \{ M[0], M[1], ..., M[m - 1] \}$, where $\underline{m}$ is the number of machines in the system.

(4) Each $S[i]$ of the application program P can be executed by any of the machines $M[j]$ in the HC system. There is a computation matrix $\underline{C} = \{ C[i, j] \}$ associated with S and M, where $C[i, j]$ denotes the computation time of $S[i]$ on machine $M[j]$ [GhY93, YaK94]. The computation matrix C is assumed to be known. It can be computed from empirical information or by applying two characterization techniques in HC, namely task profiling and analytical benchmarking (see [SiA95] for a survey of these techniques).

(5) Suppose that a set of initial data elements $\{ d_0, d_1, ..., d_{Q-1} \}$ are required for executing the application program P, where Q is the number of initial data elements for P. A set of initial-data functions $\underline{H} = \{ H[0], H[1], ..., H[Q - 1] \}$ is defined, where $H[k](j)$ ($0 \le k < Q$ and $0 \le j < m$) represents the *smallest* communication time for machine $M[j]$ to obtain the initial data element d_k from one of the devices where d_k is stored before the execution of P. Initial data element d_k is also denoted as data item $(-1, d_k)$.

(6) The communication function matrix $\underline{D(|e|)} = \{ D[s, r](|e|) \}$, for $0 \le s,r < m$, where $D[s, r](|e|)$ denotes the communication time for transferring data item e (of size $|e|$) from machine $M[s]$ to machine $M[r]$ [GhY93, KhP92]. It is assumed that $D[s, s](|e|) < D[s, r](|e|)$ for $r \neq s$, i.e., the communication time for machine $M[s]$ to fetch any data item from its local storage (denoted by $D[s, s](|e|)$) is smaller than the communication time required to fetch the same data item from any other machine in the heterogeneous suite (denoted by $D[s, r](|e|)$ and $r \neq s$). Having $s = -1$ indicates that $e = (-1, d)$ and d is one of the initial data elements of P and there exists k ($0 \le k < Q$) such that $d = d_k$ and $D[s, r](|e|) = H[k](r)$.

(7) An assignment function *Af* is associated with the application program P, such that $Af: S \rightarrow M$. If $Af(i) = j$, then $S[i]$ is assigned to be executed on machine $M[j]$. The assignment function *Af* corresponds to the matching problem discussed in Subsection 7.1.

(8) Given that sequential execution of the subtasks for the application program P is assumed, a scheduling function Sf is associated with the application program P. $Sf(i) = k$ means that $S[i]$ is the k-th subtask to be executed. The scheduling function Sf corresponds to the scheduling problem discussed in Subsection 7.1. Sf is a bijection from the set S onto itself (i.e., a permutation).

Sf is defined as a valid scheduling function if and only if for all $S[i_1]$ and $S[i_2]$ such that $G[i_1] \cap I[i_2] \neq \emptyset$, $Sf[i_1] < Sf[i_2]$. For the rest of the section, all scheduling functions considered will be valid. Therefore, if one of the input-data items required by $S[i_2]$ is one of the output-data items generated by $S[i_1]$, then, with respect to a valid scheduling function, $S[i_2]$ must be executed after $S[i_1]$ is executed. If two subtasks have no data dependency between them, then either can be executed before the other.

(9) The set of data-source functions is $DS = \{ DS[0], DS[1], ..., DS[n-1] \}$, where $DS[i](j) = k$ ($0 \leq i, k < n$) means that $S[i]$ obtains the input-data item $Id[i, j]$ from $S[k]$. If $DS[i](j) = -1$, then $Id[i, j] = (-1, d_x)$ and $S[i]$ obtains the associated data from the "closest" device where d_x is initially stored. The set of data-source functions DS corresponds to the data relocation problem discussed in Subsection 7.1. When data-reuse is considered, a restriction on DS is made. For any two subtasks $S[i_1]$ and $S[i_2]$, if $Af(i_1) = Af(i_2) = k$, $Sf(i_2) < Sf(i_1)$, and if there exists j_1 and j_2 such that $Id[i_1, j_1] = Id[i_2, j_2]$, then $DS[i_1](j_1) = i_2$. That is, if $S[i_1]$ and $S[i_2]$ are assigned to the same machine $M[k]$ by Af, and if $S[i_2]$ is executed before $S[i_1]$ according to Sf, and if $S[i_2]$ and $S[i_1]$ have a common input-data item (possibly generated from third different subtask), then $S[i_1]$ should take advantage of data-reuse and obtain the common data item from $S[i_2]$ that was executed previously on the same machine $M[k]$. Because each machine in the HC system can fetch any data item from its local storage faster than fetching it from other machines in the HC suite (see definition (6)), this restriction on DS when considering data-reuse is justified.

For different scheduling functions (as well as assignment functions), with consideration of the data-reuse and multiple data-copies situations, there are different sets of choices for the data-source functions. Thus, the communication time of an application program P depends on both Sf and DS.

(10) For a given computation matrix C and communication function matrix $D(|e|)$, the total execution time of the application program P associated with an assignment function Af, a

valid scheduling function *Sf*, and a set of data-source functions *DS* is defined by the following formula:

Execution_time$_P$(*Af*, *Sf*, *DS*) =

Computation_time$_P$(*Af*, *Sf*, *DS*) + Communication_time$_P$(*Af*, *Sf*, *DS*),

such that,

$$\text{Computation\_time}_P(Af, Sf, DS) = \sum_{i=0}^{n-1} C[i, Af(i)] = \text{Computation\_time}_P(Af)$$

and

$$\text{Communication\_time}_P(Af, Sf, DS) = \sum_{i=0}^{n-1} \sum_{j=0}^{NI[i]-1} D[Af(DS[i](j)), Af(i)](|Id[i, j]|).$$

Although the dependence of Communication_time$_P$ on *Sf* is not explicitly shown in the above equation, the possible sets of data-source functions *DS* depend on *Sf* (see definition (9)). Thus, Communication_time$_P$ does indeed depend on *Sf*. The objective of matching, scheduling, and data relocation for HC is to find an assignment function Af^*, a valid scheduling function Sf^*, and a set of data-source functions DS^* such that

$$\text{Execution\_time}_P(Af^*, Sf^*, DS^*) = \min_{Af,Sf,DS} \{\text{Execution\_time}_P(Af, Sf, DS)\}.$$

7.3. Impact of Data-Reuse Without Considering Multiple Data-Copies

The following lemma and theorem use the mathematical model described in the previous subsection for the case where there is data-reuse. The multiple data-copies situation is not considered in this subsection.

Lemma 1: When data-reuse is considered and the multiple data-copies situation is not, *DS* = $f'(Af, Sf)$, where f' is a function of *Af* and *Sf*.

Proof: Without considering both data-reuse and multiple data copies, *DS* is uniquely determined by the underlying given subtask flow graph and *Af*. When the data-reuse (only) is considered, then by definition, *DS* is uniquely determined by the conditions imposed by *Af* and *Sf* (see definition (9) in Subsection 7.2). Thus, without considering multiple data-copies, *DS* is a function of *Af* and *Sf*. □

Theorem 1: If data-reuse is considered but the multiple data-copies situation is not, then the execution time of an application program *P* is a function of *Af* only, i.e.,

$$\text{Execution\_time}_P(Af, Sf, DS) = f(Af).$$

Proof: From Lemma 1, DS is a function of Af and Sf. Thus, Execution_time$_P$ is only a function of Af and Sf. It needs to be shown that Execution_time$_P$ is independent of Sf.
As shown in definition (10), because Computation_time$_P$ is independent of Sf and DS, it needs to be shown that Communication_time$_P$ is independent of Sf and DS. Recall that the formula for Communication_time$_P$ is

$$\text{Communication\_time}_P(Af, Sf, DS) = \sum_{i=0}^{n-1} \sum_{j=0}^{NI[i]-1} D[Af(DS[i](j)), Af(i)](|Id[i, j]|).$$

Case 1: Subtask $S[i]$ cannot receive its data item $Id[i, j]$ from a subtask on machine $M[Af(i)]$ according to any valid scheduling function Sf (i.e., there is no opportunity for data-reuse for the particular data item $Id[i, j]$ of $S[i]$): As stated in the proof of Lemma 1, with no data-reuse, the value of $DS[i](j)$ (and hence the value of $D[Af(DS[i](j)), Af(i)]$ ($|Id[i, j]|$) is independent of Sf.
Case 2: Subtask $S[i]$ can receive its data item $Id[i, j]$ from a subtask on the same machine $M[Af(i)]$ according to an arbitrary scheduling function Sf (i.e., there is opportunity for data-reuse for the particular data item $Id[i, j]$ of $S[i]$): Let $Id[i, j] = (x, d)$, where d is the corresponding variable name and $S[x]$ is the subtask that generates d. Af determines which subtasks are executed on machine $M[Af(i)]$. Let $S' \subseteq S$ be the subset of subtasks that are executed on $M[Af(i)]$ and need the unique input-data item (x, d). The data item (x, d) must be moved from $M[Af(x)]$ to $M[Af(i)]$ just once, and then can be used by all $S[k] \in S'$. Thus, the communication time for all $S[k] \in S'$ to receive (x, d) from $M[Af(x)]$ is equal to the time for any one of the subtasks in S' to receive (x, d) from $M[Af(x)]$ and the time for other subtasks in S' to fetch (x, d) from local storage with the consideration of data-reuse. Mathematically, if $|S'|$ is the size of the subset S', and $S[q]$ is an arbitrary element of S', then the time for all $S[k] \in S'$ to receive (x, d) is:

$$D[Af(x), Af(q)](|(x, d)|) + (|S'| - 1)D[Af(q), Af(q)](|(x, d)|).$$

Therefore, the communication time for all $S[k] \in S'$ to obtain data item (x, d) is independent of the order of execution of the subtasks in S', and hence is independent of Sf. Thus, as with Case 1, it is shown that Communication_time$_P$ is independent of Sf.

Therefore, both Computation_time$_P$ and Communication_time$_P$ are independent of Sf. Thus, Execution_time$_P$ depends on Af only. □

7.4. Impact of Multiple Data-Copies and Temporally Interleaved Execution of Atomic Input Operations for Different Subtasks (TIE)

In this subsection, both the data reuse and multiple data-copies situations are considered. Furthermore, data reuse is viewed as a special case of having multiple data copies.

It was shown in Theorem 1 that, when data-reuse is considered and multiple data-copies are not, the execution time of any application program P depends only on the assignment function Af. But when one considers the multiple data-copies situation, the execution time of an application program P also depends on the scheduling function Sf and the set of data-source functions DS. Each scheduling function Sf defines a set of possible choices for DS.

Recall from Subsection 7.1 that the size of a data item $e = (k, d)$ is denoted by $|e|$ (or $|d|$). To show the effect of utilizing the multiple data-copies, consider an HC system with four machines connected by the network illustrated in Figure 7.2. The number with each link represents the communication cost for obtaining the corresponding data item. The order for executing the data transfers is indicated by the numbers in the circles. An initial data element d_0 is stored on machine $M[0]$, and $(-1, d_0)$ is the only required input-data item for both $S[0]$ and $S[1]$ (thus, there is no data dependency between $S[0]$ and $S[1]$). Assume that $Af(0) = 1$ and $Af(1) = 2$; thus, $\text{Computation\_time}_P$ is determined. If $S[0]$ is scheduled for execution before $S[1]$, by the data transfers illustrated by the arrows in Case 1 of Figure 7.2, $\text{Communication\_time}_P = 205$. If $S[0]$ is executed after $S[1]$, by the data transfers illustrated by the arrows in Case 2 of Figure 7.2, $\text{Communication\_time}_P = 305$. Hence, depending on which scheduling function (and, in general, which set of data-source functions) is chosen, the execution time of an application program P may be different.

It is assumed, without loss of generality, that all input-data items are received for a subtask prior to that subtask's computation. For an arbitrary $S[i]$, there are $NI[i]$ necessary operations for obtaining the input-data items in $I[i]$. These operations are defined as the atomic input operations of $S[i]$. The scheduling function Sf only represents the order for executing the subtasks, *not* the order for executing the atomic input operations. Most of the existing algorithms for matching, scheduling, and data relocation in HC only allow consecutive execution of the atomic input operations of each subtask. This means that if $Sf(i_1) < Sf(i_2)$, then all atomic input operations of $S[i_1]$ must be executed *before* the atomic input operations of $S[i_2]$ are executed. The temporally interleaved execution of atomic input operations for different subtasks (TIE) allows some of the atomic input operations of $S[i_1]$ to be executed *after* some atomic input operations of $S[i_2]$ are executed even if $Sf(i_1) < Sf(i_2)$. The effective use of TIE can result in a smaller execution time than that associated with considering the sequence of $NI[i]$

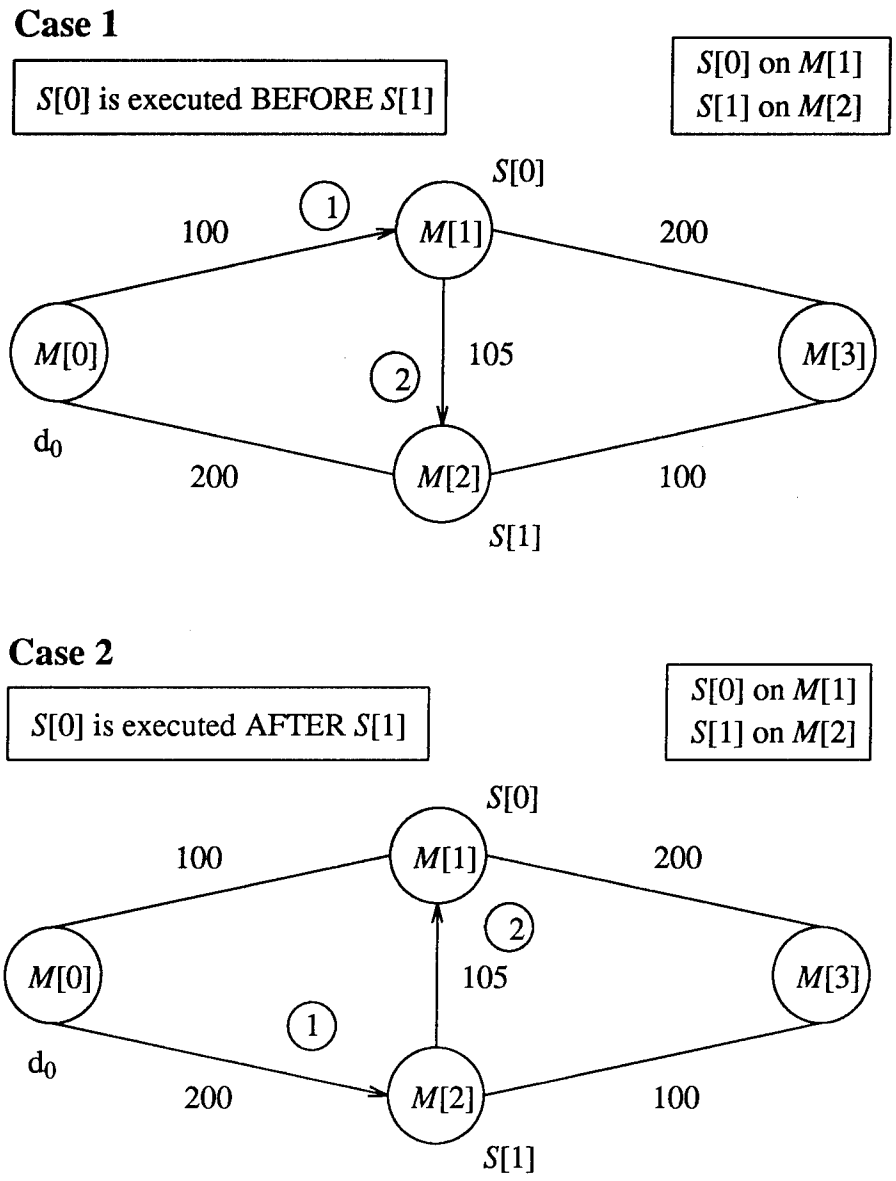


Figure 7.2: Network of four machines with the initial data element d_0 on $M[0]$.

atomic input operations of $S[i]$ to be indivisible. This is true because TIE gives more options for choosing the set of data-source functions for $S[i]$.

As an example, for the same HC system and the same assignment function *Af* described in Figure 7.2, assume that $(-1, d_0)$ and $(-1, d_1)$ are the only required input-data items of both $S[0]$ and $S[1]$ (initially stored on $M[0]$ and $M[3]$, respectively, and with $|d_0| = |d_1|$). If $S[0]$ is executed

before $S[1]$, by the data transfers illustrated by the arrows in Case 1 of Figure 7.3, $\text{Communication\_time}_P = 505$. If $S[0]$ is executed after $S[1]$, by the data transfers illustrated by the arrows in Case 2 of Figure 7.3, $\text{Communication\_time}_P = 505$. But if TIE is allowed, suppose the atomic input operation for $S[0]$ to obtain d_0 is executed first, then the atomic input operation for $S[1]$ to obtain d_1 is executed second, followed by the atomic input operation for $S[0]$ to obtain d_1 and the atomic input operation for $S[1]$ to obtain d_0. In this case, $\text{Communication\_time}_P = 410$. For all three cases mentioned, the same Af is used, and hence $\text{Computation\_time}_P$ is the same.

A set of ordering functions $\underline{\text{Order}} = \{ Order[i] \mid 0 \le i < n \}$ is associated with P. If $Order[i](j) = k$, where $0 \le j < NI[i]$ and $0 \le k < \sum_{i=0}^{n-1} NI[i]$, then the j-th atomic input operation of $S[i]$ (to obtain the input-data item $Id[i, j]$) is the k-th atomic input operation to be executed during the execution of P.

The usual definition of scheduling implicitly assumes that the atomic input operations (corresponding to communication) and computation of $S[i]$ are executed indivisiblely. Suppose the execution steps of two or more subtasks are interleaved and the concept of sequential execution of subtasks (i.e., no concurrent execution of different subtasks across different machines in the HC suite) is still enforced. Given the mathematical model presented in Subsection 7.2, the interleaved computation of subtasks cannot change the total computation time (which is determined by Af). However, interleaved communication (i.e., the atomic input operations of subtasks) may result in smaller total communication time. Thus, extending the definition of scheduling function Sf to allow TIE can potentially enhance the performance of the HC system. The set of ordering functions, $Order$, defines the interleaving of the execution of atomic input operations for the subtasks in a program and is an extension to the regular scheduling function Sf.

In the following Steps 1 to 4, a graph (denoted as $\underline{Gr[Af, DS]}$) corresponding to a particular DS (with respect to an arbitrary assignment function Af) is generated. When using TIE, the concept of a *valid* set of data-source functions DS for the atomic input operations of the application program P can be defined according to the properties of $Gr[Af, DS]$. There may be many such valid sets, each corresponding to a unique graph, and each resulting in a $\text{Communication\_time}_P$ that may be different from the others. An invalid DS would correspond to a set of data-source functions that does not result in an operational program (e.g., in Figure 7.3, the case where $S[0]$ receives d_0 from $S[1]$, $S[1]$ receives d_0 from $S[0]$, and neither receives d_0 from $M[0]$ is not valid).

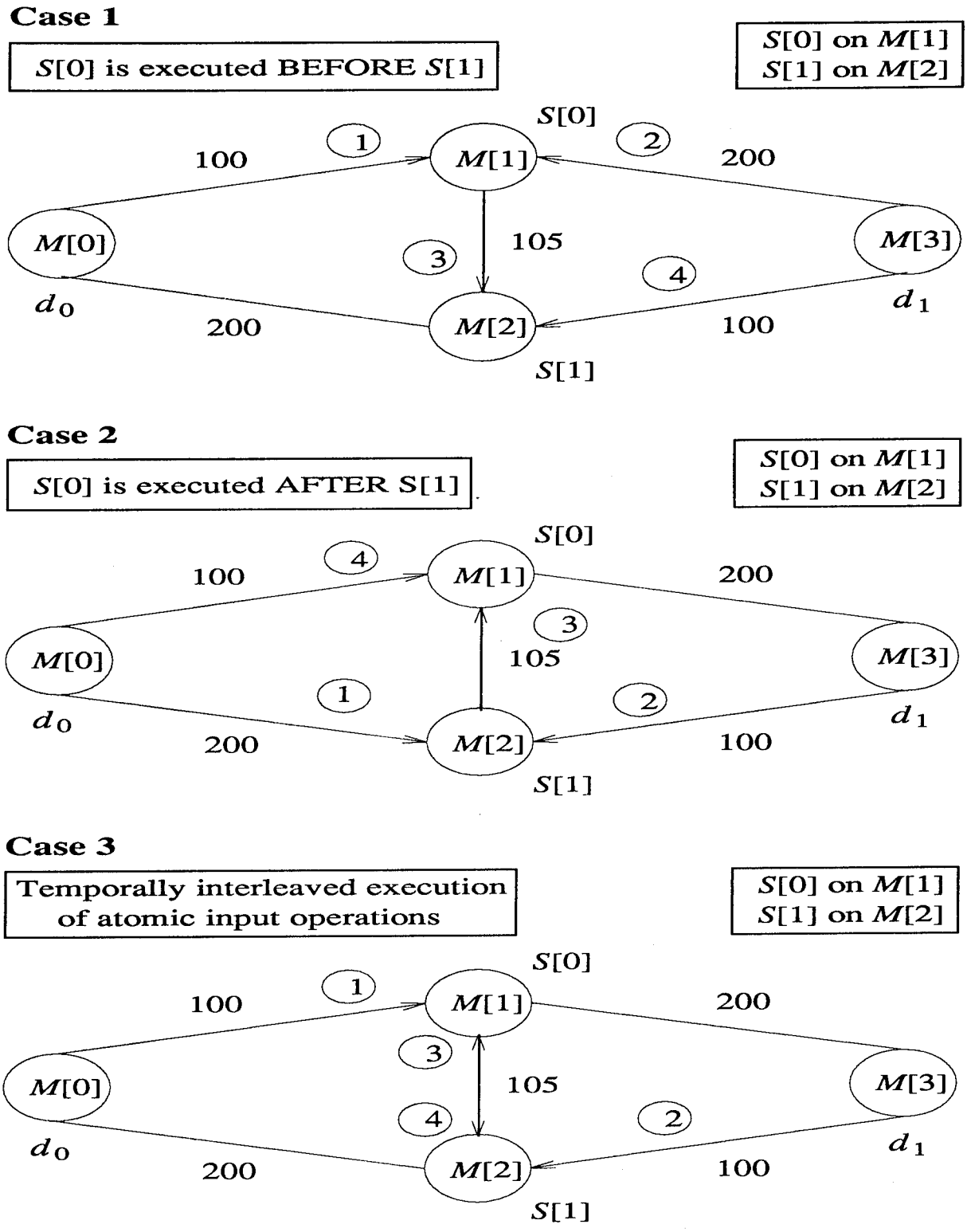


Figure 7.3: Network of four machines with the initial data on $M[0]$ and $M[3]$.

Step 1: A *Source* vertex is generated that represents the locations for all the initial data elements (which may be on different devices/machines).

Step 2: For each $S[i]$, $NI[i] + 1$ vertices, one for each of the $NI[i]$ atomic input operations and one for all of the generated output data items of $S[i]$, are created. These are the set of input-data vertices, labeled $V[i, j]$ ($0 \le j < NI[i]$) and the output-data vertex $V_g[i]$ (as shown in Figure 7.4). $\underline{V}$ is a set that contains all the above vertices associated with the application program P in Steps 1 and 2.

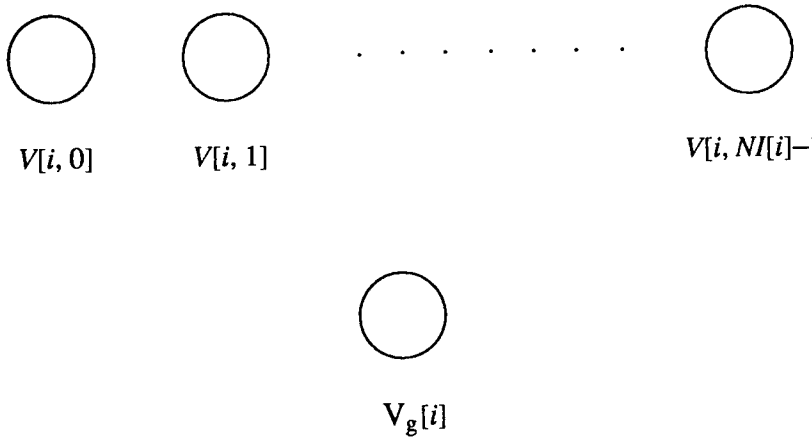


Figure 7.4: The generation of the vertices for the atomic operations of $S[i]$.

Step 3: Let $\underline{W}$ denote the maximum communication time necessary to transfer any data item from an initial source or machine in the heterogeneous suite to any other machine (this can be determined from H and D defined in Subsection 7.2).

Step 4: For any input-data vertex $V[i_1, j_1]$, suppose that $DS[i_1](j_1) = i_2$, where $-1 \le i_2 < n$.
Case A: If $0 \le i_2 < n$, then for all j_2 such that $Id[i_1, j_1] = Id[i_2, j_2]$, a directed edge with weight $D[Af(i_2), Af(i_1)](|Id[i_1, j_1]|)$ is added from $V[i_2, j_2]$ to $V[i_1, j_1]$.
Case B: If $0 \le i_2 < n$, then for all j_2 such that $Id[i_1, j_1] = Gd[i_2, j_2]$, a directed edge with weight $D[Af(i_2), Af(i_1)](|Id[i_1, j_1]|)$ is added from $V_g[i_2]$ to $V[i_1, j_1]$.
Case C: If $i_2 = -1$, then there exists k ($0 \le k < Q$), such that $Id[i_1, j_1] = (-1, d_k)$, and a directed edge with weight $H[k](Af(i_1))$ is added from the *Source* vertex to $V[i_1, j_1]$.

For any input-data vertex $V[i_1, j_1]$ ($0 \le i_1 < n$ and $0 \le j_1 < NI[i_1]$), one and only one case of A, B, or C can occur. That is, $S[i_1]$ can obtain its required input-data item $Id[i_1, j_1]$ either from copying $S[i_2]$'s input-data item (Case A), or from the subtask that generates $Id[i_1, j_1]$ (Case

B), or from the *Source* vertex if $Id[i_1, j_1] = (-1, d_k)$ and d_k is one of the initial data elements (Case C). Thus, any vertex $V[i_1, j_1]$ has one and only one parent vertex. Also, the weight of the edge between $V[i_1, j_1]$ and its unique parent vertex is the communication time for $S[i_1]$ to obtain $Id[i_1, j_1]$ with respect to a given Af and DS.

As an example, suppose that a specific application program P is illustrated by the subtask flow graph shown in Figure 7.5 and the sizes of the data items are shown as follows (d_0 and d_1 are the variable names of initial data elements of P; X_0, X_1, Y, Z_0, and Z_1 are the variable names of generated data items of P; a is an arbitrary constant).

$S[0]$: $NI[0] = 1$, $Id[0, 0] = (-1, d_0)$, $|d_0| = 2a$;
$NG[0] = 2$, $Gd[0, 0] = (0, X_0)$, $Gd[0, 1] = (0, X_1)$, $|X_0| = 8a$, $|X_1| = 3a$.

$S[1]$: $NI[1] = 2$, $Id[1, 0] = (-1, d_0)$, $Id[1, 1] = (0, X_0)$;
$NG[1] = 1$, $Gd[1, 0] = (1, Y)$, $|Y| = 5a$.

$S[2]$: $NI[2] = 2$, $Id[2, 0] = (0, X_0)$, $Id[2, 1] = (-1, d_1)$, $|d_1| = 6a$;
$NG[2] = 2$, $Gd[2, 0] = (2, Z_0)$, $Gd[2, 1] = (2, Z_1)$, $|Z_0| = 4a$, $|Z_1| = a$.

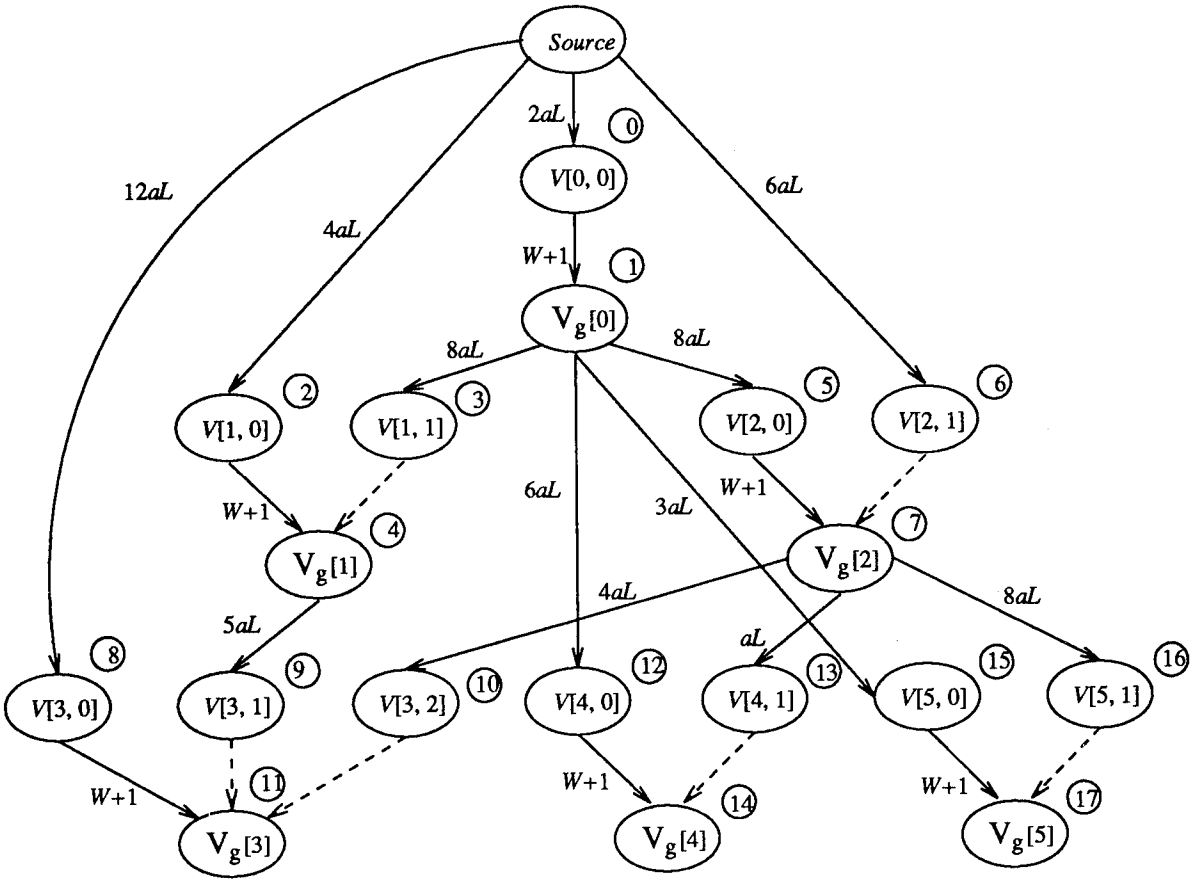
$S[3]$: $NI[3] = 3$, $Id[3, 0] = (-1, d_1)$, $Id[3, 1] = (1, Y)$, $Id[3, 2] = (2, Z_0)$; and $NG[3] = 0$.

$S[4]$: $NI[4] = 2$, $Id[4, 0] = (0, X_1)$, $Id[4, 1] = (2, Z_1)$; and $NG[4] = 0$.

$S[5]$: $NI[5] = 2$, $Id[5, 0] = (0, X_1)$, $Id[5, 1] = (2, Z_0)$; and $NG[5] = 0$.

Consider, for ease of presentation, an HC system with four machines connected by the very simple linear array network illustrated in Figure 7.6(c). Here, $D[s, r](|d|) = |s - r||d|L$, where $0 \leq s, r < 4$ and L is the length of the physical link between the neighboring machines in the linear array network. This equation for D is an oversimplified example; any appropriate equation that represents the communication costs of the network in the HC system can be used. The result of applying the set of data-source functions defined by the subtask flow graph in Figure 7.5 is shown in Figures 7.6(a) and 7.6(b). The solid lines in Figure 7.6(a), except the lines with weight $W + 1$, show the direct edges added by applying Step 4. The assignment function Af for this current example is shown in Figure 7.6(c): $Af(0) = 1$, $Af(1) = 2$, $Af(2) = 2$, $Af(3) = 1$, $Af(4) = 3$, and $Af(5) = 0$. W is $24aL$ according to Step 3.

Step 5: For every $0 \leq i < n$, a directed edge with weight $W + 1$ (i.e., a weight greater than any possible communication time) is added from $V[i, 0]$ to $V_g[i]$. $V_g[i]$ also has one and only one parent vertex, i.e., $V[i, 0]$. These directed edges are shown by the solid lines with weight $W + 1$ in Figure 7.6(a).



(a)

$DS[0](0) = -1;$

$DS[1](0) = -1,$ $DS[1](1) = 0;$

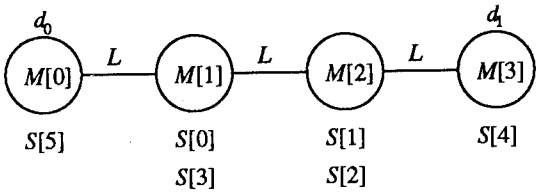
$DS[2](0) = 0,$ $DS[2](1) = -1;$

$DS[3](0) = -1,$ $DS[3](1) = 1,$ $DS[3](2) = 2;$

$DS[4](0) = 0,$ $DS[4](1) = 2;$

$DS[5](0) = 0,$ $DS[5](1) = 2.$

(b)



(c)

Figure 7.5: Subtask flow graph for the example application program.

If $Gr[Af, DS]$ generated above is a tree (denoted as $Tree[Af, DS]$) with the *Source* vertex being the root of the tree, then the corresponding DS is defined as a valid set of data–source functions for atomic input operations of the application program P. The DS defined in Figure 7.6(b) is a valid set of data-source functions.

The reason for this definition is that, for a *valid* set of data-source functions DS, $Gr[Af, DS]$ must be an acyclic graph. Otherwise deadlock arises in the application program P, which makes P unschedulable (recall the earlier example of an invalid DS). Because a $Gr[Af, DS]$ generated with respect to a *valid* DS is acyclic and each vertex (except the *Source* vertex) of $Gr[Af, DS]$ has one and only one parent vertex, from basic graph theory [BoM76], $Gr[Af, DS]$ is a tree with the *Source* vertex as the root of the tree. Thus, the validity of the corresponding DS can be determined according to whether the $Gr[Af, DS]$ generated by above Steps 1 to 5 is a tree or not. Furthermore, with an arbitrary assignment function Af and a valid set of data-source functions DS, the weight of the edge between $V[i_1, j_1]$ ($0 \le i_1 < n$ and $0 \le j_1 < NI[i_1]$) and its unique parent vertex is the communication time for $S[i_1]$ to obtain $Id[i_1, j_1]$ with respect to the given Af and DS. Thus, the communication time for the application program P is only a function of Af and DS (DS must be valid) and

$$\text{Communication\_time}_P(Af, DS) = \text{Weight}(Tree[Af, DS]) - n(W + 1),$$

where Weight(x) is the sum of the weights on all edges of tree x. For the application program P specified by Figure 7.5, with respect to the given assignment function Af and the given valid data-source functions DS as defined in Figure 7.6(b), $\text{Communication\_time}_P(Af, DS) = 67aL$.

To determine a set of ordering functions *Order* corresponding to a valid DS for executing the atomic input operations of different subtasks, a directed edge with weight zero from $V[i_1, j_1]$ to $V_g[i_1]$ is added to the $Tree[Af, DS]$ for every i_1 and j_1 except $j_1 = 0$ (i.e., $0 \le i_1 < n$ and $1 \le j_1 < NI[i_1]$). These directed edges are illustrated by the dashed lines shown in Figure 7.6(a) for the example application program P. After adding these zero-weight edges, the tree becomes a directed acyclic graph (DAG). One possible set of ordering functions *Order* corresponding to DS can be determined by applying a topological sort algorithm [CoL92] to this generated DAG. For the example application program P, the numbers in the circles in Figure 7.6(a) indicate one ordering for the execution of the corresponding atomic input operations and subtask computation of P as determined by one particular topological sort.

It is stated in part (10) of the mathematical model presented in Subsection 7.2 that $\text{Communication\_time}_P$ is a function of Af, Sf, and DS. If TIE is allowed, because *Order* is an extended version of Sf, $\text{Communication\_time}_P$ is a function of Af, *Order*, and a valid DS. *Order*

must be one of the sets of ordering functions, generated by the topological sort described above, corresponding to a valid *DS*. If not, the scheduling scheme and the data relocation scheme are incompatible with one another (i.e., *Order* and *DS* collectively cannot result in an operational program). If $Order_1$ and $Order_2$ are two sets of ordering functions, then, because $Communication\_time_P(Af, DS) = Weight(Tree[Af, DS]) - n(W + 1)$, $Communication\_time_P(Af, Order_1, DS) = Communication\_time_P(Af, Order_2, DS)$. Thus, if TIE is allowed and the corresponding *DS* is a valid set of data source functions for the atomic input operations of the application program *P*, $Communication\_time_P$ is a function of *Af* and *DS* only. Because the computation time for *P* is a function of only *Af*, the total execution time for *P* is a function of *Af* and *DS*. The objective of matching, scheduling, and data relocation for HC is to find an assignment function Af^* and a valid set of data-source functions DS^*, such that

$$Execution\_time_P(Af^*, DS^*) = \min_{Af,\ DS} \{Execution\_time_P(Af, DS)\}.$$

7.5. A Minimum Spanning Tree Based Algorithm for Finding the Optimal Set of Data-Source Functions and the Corresponding Set of Ordering Functions

7.5.1. Description of the Algorithm

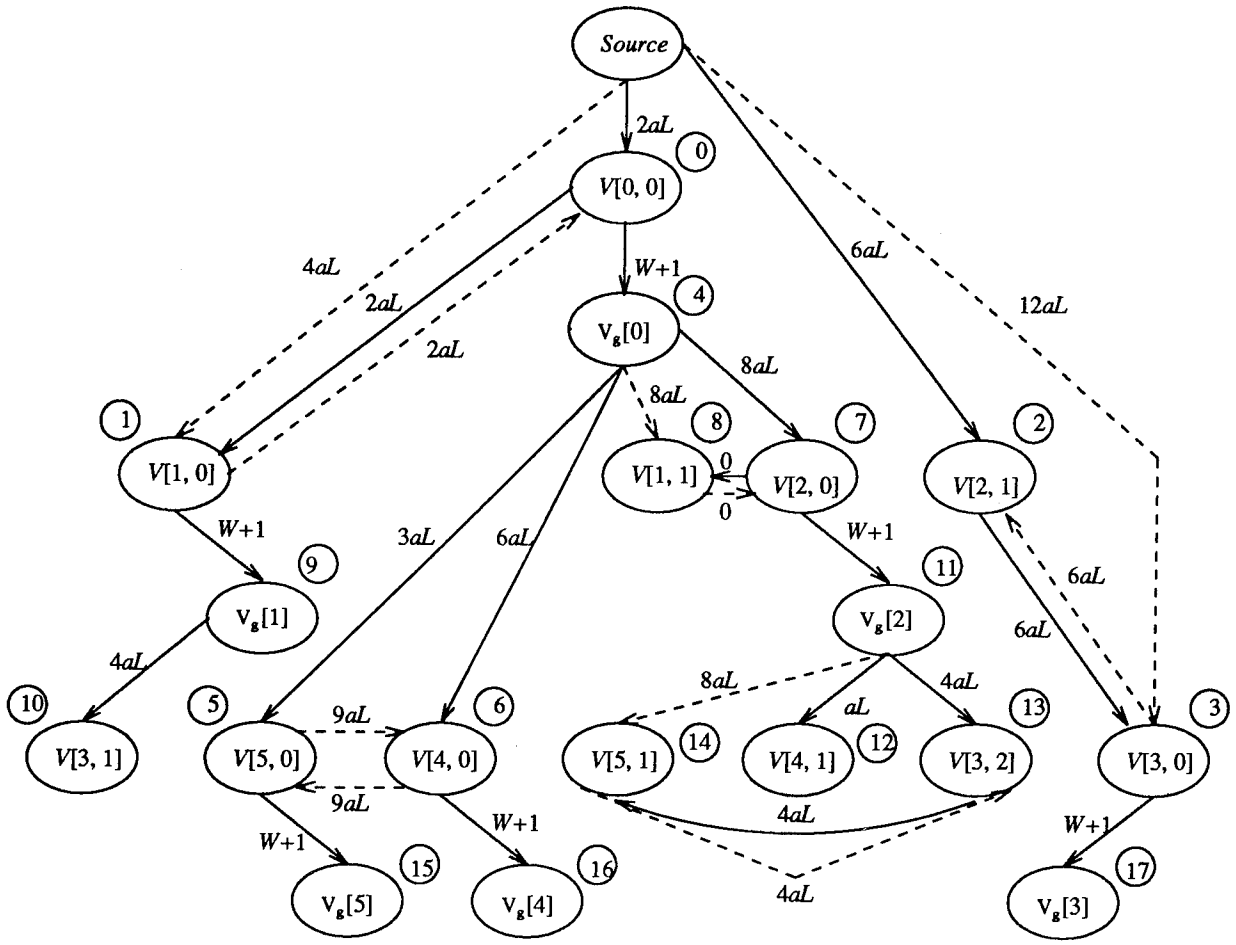
For an arbitrary assignment function *Af*, a minimum spanning tree based algorithm is presented for finding a corresponding optimal valid set of data-source functions DS^*, such that for any other valid set of data-source functions *DS*,

$$Execution\_time_p(Af, DS^*) \leq Execution\_time_p(Af, DS).$$

A directed graph Dg (see Figure 7.7(a)) corresponding to a specific assignment function *Af* can be generated by connecting the vertices in *V* as follows (recall that *V* is a set that contains all the vertices generated for any specific application program *P* according to Steps 1 and 2 described in Subsection 7.4). Figure 7.7, which is based on the example program shown in Figure 7.5, uses the same machine and assignment function as in Figure 7.6(c), and has all the same vertices as in Figure 7.6(a).

(a) For every i_1, j_1, i_2, and j_2, where $0 \leq i_1, i_2 < n$, $0 \leq j_1 < NI[i_1]$, $0 \leq j_2 < NI[i_2]$, and $i_1 \neq i_2$, such that $Id[i_1, j_1] = Id[i_2, j_2] = e$, a directed edge from $V[i_1, j_1]$ to $V[i_2, j_2]$ with weight $D[Af(i_1), Af(i_2)](|e|)$ and a directed edge from $V[i_2, j_2]$ to $V[i_1, j_1]$ with weight $D[Af(i_2), Af(i_1)](|e|)$ are added.

(b) For every i_1, j_1, i_2, and j_2, where $0 \leq i_1, i_2 < n$, $0 \leq j_1 < NG[i_1]$, and $0 \leq j_2 < NI[i_2]$, such that $Gd[i_1, j_1] = Id[i_2, j_2] = e$, a directed edge from $V_g[i_1]$ to $V[i_2, j_2]$ with weight $D[Af(i_1),$



(a)

$DS^*[0](0) = -1;$

$DS^*[1](0) = 0,$ $DS^*[1](1) = 2;$

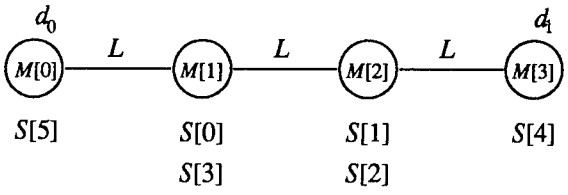
$DS^*[2](0) = 0,$ $DS^*[2](1) = -1;$

$DS^*[3](0) = 2,$ $DS^*[3](1) = 1,$ $DS^*[3](2) = 2;$

$DS^*[4](0) = 0,$ $DS^*[4](1) = 2;$

$DS^*[5](0) = 0,$ $DS^*[5](1) = 3.$

(b)



(c)

Figure 7.6: Generating a spanning tree with respect to the set of data-source functions associated with the subtask flow graph. (a) The spanning tree (solid lines). (b) The set of data-source functions. (c) The linear array network and the matching scheme.

$Af(i_2)](|e|)$ is added.

(c) For every i, j, and k, such that $Id[i, j] = (-1, d_k)$, where $0 \leq i < n$, $0 \leq j < NI[i]$, and $0 \leq k < Q$, a directed edge from the *Source* vertex to $V[i, j]$ with weight $H[k](Af(i))$ is added.

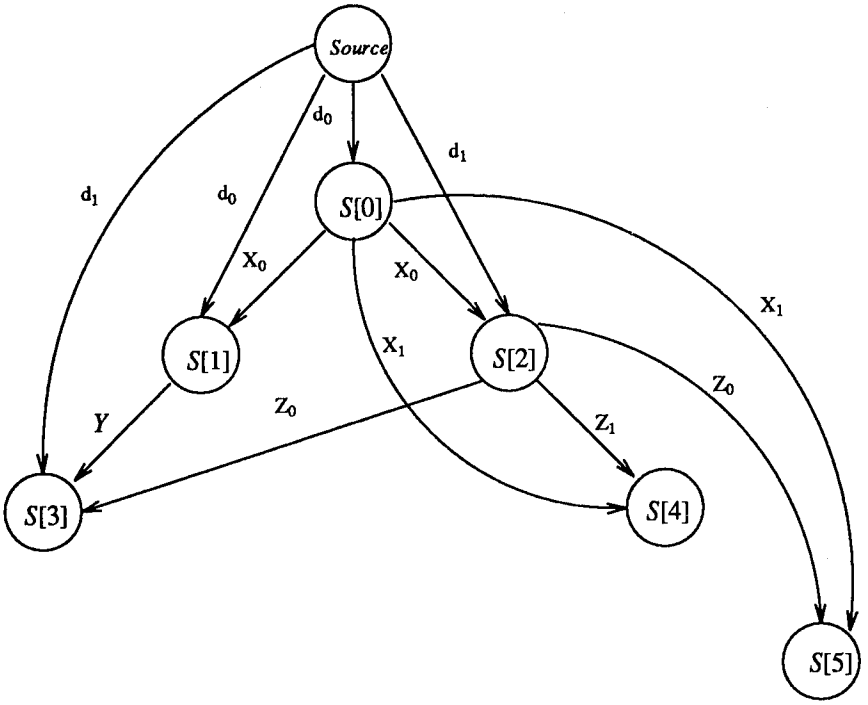


Figure 7.7: Generating a minimum spanning tree for the example application program and its corresponding valid data-source functions. (a) The minimum spanning tree (solid lines). (b) The set of data-source functions. (c) The linear array network and the matching scheme.

All the edges generated in (a), (b), and (c) are called fetch edges. For the example application program P illustrated by the subtask flow graph in Figure 7.5, with the linear network of four machines as the heterogeneous suite and the assignment function defined in Figure 7.7(c) (and Figure 7.6(c)), the edges (both solid lines and dashed lines) of Dg in Figure 7.7(a) (except the ones with weight $W + 1$) are fetch edges.

(d) For every $0 \leq i < n$, a directed edge from $V[i, 0]$ to $V_g[i]$ with weight $W + 1$ is added.

All the edges generated in (d) are called activate edges. There are a total of n activate edges with total weight $n(W + 1)$. Notice that the weight of an activate edge is larger than the weight of any fetch edge because of the definition of W. The edges of Dg shown in Figure 7.7(a) with weight $W + 1$ are activate edges.

For given system parameters D and H, the directed graph Dg can be generated by knowing only P and Af. After generating Dg corresponding to a specific Af, a modified version of Prim's algorithm [CoL92], referred to as the TIE algorithm in this section is applied, to find a minimum spanning tree $MST[Af]$ of Dg. The *Source* vertex is the root of the minimum spanning tree. Suppose A is a set that contains the vertices that have been added to the tree, and T is the tree partially generated during the execution of the TIE algorithm. The order of execution for the atomic input operation that corresponds to any vertex $V[i, j]$ ($0 \le i < n$ and $0 \le j < NI[i]$) in V is $\text{Order}^*[i](j)$. The TIE algorithm is described as follows.

Step 1: Let $A = \{Source\}$, $T = \{Source\}$, and $Counter = 0$.

Step 2: Case A: If the set of cut edge(s) between A and $V - A$ (a cut edge is an edge that connects a vertex in A and a vertex in $V - A$) contains fetch edge(s), then find a cut edge that has the smallest weight (there might be several, in which case an arbitrary minimum weight edge is chosen). Include that edge in T and move the corresponding vertex $V[i, j]$ that is currently in $V - A$ into A and T. Increment *Counter* by 1 and set $\text{Order}^*[i](j) = Counter$. Because the set of cut edges between A and $V - A$ contains fetch edges, then no activate edge can be chosen, because the weight of an activate edge is greater than the weight of any fetch edge.

Case B: If the set of cut edges between A and $V - A$ contains only activate edges, these edges will connect to a subset of the V_g vertices. Let this subset be denoted as $\{ V_g[i_0], V_g[i_1], ..., V_g[i_j], ..., V_g[i_{u-1}] \}$, where $1 \le u \le n$, $0 \le j < u$, $0 \le i_j < n$, and $\underline{u}$ is the number of activate edges in that set. It can be shown that there always exists at least one j ($0 \le j < u$) such that all $V[i_j, k]$ is contained in A already ($0 \le k < NI[i_j]$) by previous iterations of the TIE algorithm. Any such $V_g[i_j]$ is defined as a ready-to-execute vertex. Given that the application program is valid and that the set of cut edge(s) between A and $V - A$ only contains activate edges, there is at least one subtask $S[i_j]$ such that all of its input-data vertices $V[i_j, k]$ are already in A. Otherwise, P is not a valid program because it would allow deadlock. Include a ready-to-execute vertex $V_g[i_j]$ in A and T (if there are several, any one of the ready-to-execute vertices is chosen), and its corresponding activate edge (i.e., the edge from $V[i_j, 0]$ to $V_g[i_j]$) in T. Because all $V_g[i]$ ($0 \le i < n$) are included in the $MST[Af]$ after they become ready-to-execute vertices, $S[i]$ generates

all of its output-data items after it obtains all of its input-data items. Unlike Prim's algorithm, the TIE algorithm uses two classes of edges and places a ready-to-execute vertex into T and A. In all other respects, the algorithms are the same. Because each activate edge is the only edge entering a computation vertex (i.e., V_g vertex), all activate edges will eventually become part of the minimum spanning tree. Hence, this modification to Prim's algorithm to create the TIE algorithm still generates a minimum spanning tree.

Step 3: If $A = V$, terminate the algorithm, otherwise execute Step 2 again.

For the application program P illustrated by the subtask flow graph in Figure 7.5, with the linear network of four machines as the heterogeneous suite and the same assignment functions defined in Figure 7.7(c), the solid lines in Figure 7.7(a) show the $MST[Af]$ corresponding to Af after applying the TIE algorithm to Dg. This $MST[Af]$ was generated by knowing only Af, $I[i]$, and $G[i]$ (for given system parameters D and H).

The optimal valid set of data-source functions DS^* for atomic input operations of the application program P that corresponds to the minimum spanning tree $MST[Af]$ generated above can be determined as follows:

(a) If, in $MST[Af]$, the parent vertex of $V[i_1, j_1]$ is $V[i_2, j_2]$, then $DS^*[i_1](j_1) = i_2$.

(b) If, in $MST[Af]$, the parent vertex of $V[i_1, j_1]$ is $V_g[i_2]$, then $DS^*[i_1](j_1) = i_2$.

(c) If, in $MST[Af]$, the parent vertex of $V[i_1, j_1]$ is the *Source* vertex, then $DS^*[i_1](j_1) = -1$.

Because $MST[Af]$ is a tree, every vertex except the *Source* vertex has one and only one parent vertex, and the value of $DS^*[i_1](j_1)$ for any $0 \le i_1 < n$ and $0 \le j_1 < NI[i_1]$ is unique. The optimal set of data-source functions DS for the application program P illustrated by the subtask flow graph in Figure 7.5 is derived and given in Figure 7.7(b) according to the procedures described above. The numbers in the circles in Figure 7.7(a) indicate the order in which vertices were added to the minimum spanning tree, which is the order for executing their corresponding atomic input operations and subtask computation. The set of ordering functions, $Order^*[i](j)$, generated by the TIE algorithm corresponds to this order except that the computation vertices (i.e., V_g's) are not included.

For the complexity analysis of the TIE algorithm, suppose that $|E|$ is the number of edges in Dg and $|V|$ is the number of vertices in Dg. If a Fibonacci heap is used to implement the priority queue in the TIE algorithm, as was done in Prim's algorithm [CoL92], the worst case asymptotic complexity of the algorithm for finding DS^* is $O(|E| + |V|\lg|V|)$. For Dg, $|V| = \sum_{i=0}^{n-1}(NI[i] + 1) + 1$

$= \sum_{i=0}^{n-1} NI[i] + n + 1$. Each vertex $V[i, j]$ is connected to at most n other vertices in Dg. This corresponds to the case where $S[i]$ can obtain its required input-data item $Id[i, j]$ from all the other subtasks in P and from the source where the initial data elements are stored. Each vertex $V_g(i)$ is connected only to $V[i, 0]$. Thus, $|E| \leq n\sum_{i=0}^{n-1} NI[i] + n$. If $A = \sum_{i=0}^{n-1} NI[i]$, then $|V| = A + n + 1$ and $|E| \leq nA + n$. The worst case asymptotic complexity of the TIE algorithm in terms of A and n is $O[nA + (n + A)\lg(n + A)]$, where n is the number of subtasks in P.

7.5.2. Proof of Correctness of the Algorithm

It is shown in Subsection 7.4 that with an arbitrary assignment function Af, any valid set of data-source functions DS for atomic input operations of the application program P corresponds to a spanning tree of Dg (denoted as $\text{Tree}_P[Af, DS]$). The weight of $\text{Tree}_P[Af, DS]$ (denoted as $\text{Weight}(\text{Tree}_P[Af, DS])$) is $\text{Communication\_time}_P(Af, DS) + n(W + 1)$.
Thus,

$$\text{Execution\_time}_P(Af, DS) = \text{Computation\_time}_P(Af) + \text{Communication\_time}_P(Af, DS)$$
$$= \text{Computation\_time}_P(Af) + \text{Weight}(\text{Tree}_P[Af, DS]) - n(W+1).$$

Because

$$\text{Execution\_time}_P(Af, DS^*) = \text{Computation\_time}_P(Af) + \text{Weight}(\text{Tree}_P[Af, DS^*]) - n(W+1)$$
$$= \text{Computation\_time}_P(Af) + \text{Weight}(MST[Af]) - n(W+1)$$

and

$$\text{Weight}(MST[Af]) \leq \text{Weight}(\text{Tree}_P[Af, DS]),$$

it is true that

$$\text{Execution\_time}_P(Af, DS^*) \leq \text{Execution\_time}_P(Af, DS).$$

For the application program P illustrated by the subtask flow graph in Figure 7.5, if the set of data-source functions DS is determined directly from the subtask flow graph provided (as shown in Figure 7.6(b)), then Execution_time_P is $C[0, 1] + C[1, 2] + C[2, 2] + C[3, 1] + C[4, 3] + C[5, 0] + 67aL$. After applying the algorithm presented in Subsection 5.1 and using DS^*, then Execution_time_P is $C[0, 1] + C[1, 2] + C[2, 2] + C[3, 1] + C[4, 3] + C[5, 0] + 47aL$.

7.6. Two-Stage Approach for Matching, Scheduling, and Data Relocation in HC

In Subsections 7.4 and 7.5, it was shown how to calculate an Order* and a DS^* given a fixed Af (allowing both data-reuse and multiple data-copies). The algorithm presented in Subsection 7.5 can be used to do this in polynomial time. However, as was stated in Subsection

7.4, the objective of matching, scheduling, and data relocation (with the assumption that TIE is allowed) is to find Af^* and DS^* (with one of its corresponding Order*) for a specific application program P, such that, for any assignment function Af and any valid set of data-source functions DS, Execution_time$_P(Af^*, DS^*) \leq$ Execution_time$_P(Af, DS)$. The problem of finding Af^* is, in general, NP-complete with an arbitrary heterogeneous suite of m machines and an arbitrary application program P with n subtasks [Fer89].

One approach to matching, scheduling, and data relocation in HC is to find some (possibly suboptimal) assignment functions Af and some valid sets of data-source functions DS using a heuristic algorithm. Another approach, called the two-stage approach for matching, scheduling, and data relocation in HC, is introduced as follows:

Stage 1: Any existing heuristic (e.g., [Lo88, WaA94, WaS94]) for finding a (possibly suboptimal) assignment function, Af_{sub}, can be applied in the first stage.

Stage 2: Once a specific assignment function Af_{sub} is found, the TIE algorithm can be applied to find the optimal set of data-source functions DS^* and the corresponding set of ordering functions Order* with respect to Af_{sub}. The tuple (Af_{sub}, Order*, DS^*) is a suboptimal solution for matching, scheduling, and data relocation problems in HC.

One of the advantages of the two-stage approach is that efforts for deriving the heuristic can be concentrated solely on finding a "good" assignment function Af_{sub}. After Stage 1, the separate provably optimal TIE algorithm for finding Order* and DS^* with respect to Af_{sub} can be applied.

7.7. Summary

In an HC system, the subtasks of an application program P must be assigned to a suite of heterogeneous machines to utilize computational resources effectively (the matching problem). The execution time of P is impacted by the order of execution of subtasks (the scheduling problem), and the scheme for distributing the initial data elements and the generated data items of P to different subtasks (the data relocation problem).

The inter-machine communication time in an HC system can have a significant impact on overall system performance, so any techniques that can be used to reduce this time are important. This section focuses on scheduling schemes and data relocation schemes to minimize inter-machine communication time for a given matching scheme.

In this section, a mathematical model for matching, scheduling, and data relocation in HC was presented. The assignment function Af, the scheduling function Sf (including the set of ord-

ering functions *Order*), and the set of data-source functions *DS* were used to quantify the matching, scheduling, and data relocation problems respectively. Two data-distribution situations were identified, namely data-reuse and multiple data-copies. A theorem was presented, which states that if only data-reuse is considered (and not the multiple data-copies situation), then the execution time of P is independent of *Sf* and *DS*. Subsection 7.4 introduced an extension to scheduling, called temporally interleaved execution of the atomic input operations for different subtasks (TIE). Examples were provided to show that both multiple data-copies and TIE have an impact on the execution time of the application program P. A minimum spanning tree based algorithm with polynomial complexity was described for finding an optimal set of ordering functions $Order^*$ and an optimal set of data-source functions DS^* for an arbitrary assignment function *Af*. Based on this algorithm, a two-stage approach for matching, scheduling, and data relocation in HC was proposed.

To limit the scope of this section, sequential execution of subtasks for a specific application program P was assumed. Data-reuse and multiple data-copies will also occur when concurrent execution of subtasks across different machines in the HC is allowed, but this sequential work is a necessary step in solving the more general situation involving concurrency. Future research includes applying the concepts developed here to the more general problem.

8. Publication List

This is a list of publications that were supported in whole or part by Rome Laboratory under contract number F30602-94-C-0022.

[LiA95] Yan Alexander Li, John K. Antonio, Howard Jay Siegel, Min Tan, and Daniel W. Watson, "Estimating the Distribution of Execution Times for SIMD/SPMD Mixed-Mode Programs," *Fourth Heterogeneous Computing Workshop (HCW '95)*, sponsor: IEEE Computer Society, accepted and to appear, Apr. 1995.

[SiA94a] Howard Jay Siegel, John K. Antonio, Richard C. Metzger, Min Tan, and Yan Alexander Li, "The Goals of and Open Problems in High-Performance Heterogeneous Computing," *The 23rd Applied Imagery Pattern Recognition Workshop - Image and Information Systems: Applications and Opportunities,* sponsor: SPIE (Society of Photo-Optical Instrumentation Engineers), pp. 205-217, Oct. 1994. Invited.

[SiA94b] Howard Jay Siegel and John K. Antonio, "Views of Mixed-Mode Computing and Network Evaluation," *International Symposium on Parallel Architectures, Algorithms, and Networks (ISPAN '94),* sponsor: Japan Advanced Institute of Science and Technology, pp. 1-8, Dec. 1994. Invited.

[SiA94c] Howard Jay Siegel, John K. Antonio, Richard C. Metzger, Min Tan, and Yan Alexander Li, "Heterogeneous Computing," Purdue University, School of Electrical Engineering, Technical Report No. TR-EE 94-37, Dec. 1994, 80 pp.

[SiA95a] Howard Jay Siegel, John K. Antonio, Min Tan, Richard C. Metzger, Richard F. Freund, and Yan A. Li, "Heterogeneous Computing: One Approach to Sustained Petaflops Performance," *The Petaflops Frontier Workshop at the 5th Symposium on the Frontiers of Massively Parallel Computation,* sponsor: IEEE Computer Society, proceedings to appear, Feb. 1995.

[SiA95b] Howard Jay Siegel, John K. Antonio, Richard C. Metzger, Min Tan, and Yan Alexander Li, "Heterogeneous Computing," in *Handbook of Parallel and Distributed Computing,* edited by Albert Y. Zomaya, McGraw-Hill, to appear, 1995.

[TaA95] Min Tan, John K. Antonio, Howard Jay Siegel, and Yan Alexander Li, "Scheduling and Data Relocation for Sequentially Executed Subtasks in a Heterogeneous Computing System," *Fourth Heterogeneous Computing Workshop (HCW '95),* sponsor: IEEE Computer Society, accepted and to appear, Apr. 1995.

[Tan94] Min Tan, "Aspects of Scheduling and Data Relocation for Subtasks Executed on a Heterogeneous Computing System," Purdue University, School of Electrical Engineering, Master's Thesis, Dec. 1994.

[UlM94] Renard R. Ulrey, Anthony A. Maciejewski, and Howard Jay Siegel, "Parallel Algorithms for Singular Value Decomposition," *8th International Parallel Processing Symposium (IPPS '94),* sponsor: IEEE Computer Society, pp. 524-533, Apr. 1994.

[WaA94] Daniel W. Watson, John K. Antonio, Howard Jay Siegel, and Mikhail Atallah, "Static Program Decomposition Among Machines in an SIMD/SPMD Heterogeneous Environment with Non-Constant Mode Switching Costs," *3rd Workshop on Heterogeneous Computing (HCW '94),* sponsor: IEEE Computer Society, pp. 58-65, Apr. 1994.

[WaS94] Daniel W. Watson, Howard Jay Siegel, John K. Antonio, Mark A. Nichols, and Mikhail J. Atallah, "A Block-Based Mode Selection Model for SIMD/SPMD Parallel Environments," *Journal of Parallel and Distributed Computing,* Special Issue on Heterogeneous Processing, Vol. 21, No. 3, pp. 271-288, June 1994.

9. References

[AhS86] A. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools,* Addison-Wesley, Reading, MA, 1986.

[AlG89] G. S. Almasi and A. Gottlieb, *Highly Parallel Computing*, Benjamin/Cummings, Redwood City, California, 1989.

[AnT91] J. K. Antonio, W. K. Tsai, and G. M. Huang, "A highly parallel algorithm for multistage optimization problems and shortest path problems," *J. Parallel and Distributed Computing,* Vol. 12, No. 3, July 1991, pp. 213-222.

[ArN91] J. B. Armstrong, M. A. Nichols, H. J. Siegel, and L. H. Jamieson, "Examining the Effects of CU/PE Overlap and Synchronization Overhead when Using the Complete Sums Approach to Image Correlation," *Proceedings of the Third IEEE Symposium on Parallel and Distributed Processing*, Dec. 1991, pp. 224-232.

[ArS94] J. B. Armstrong, H. J. Siegel, W. E. Cohen, M. Tan, H. G. Dietz, and J. A. B. Fortes, "Dynamic Task Migration from SPMD to SIMD Virtual Machines," *Proceedings of the 1994 International Conference on Parallel Processing*, Vol. II, Aug. 1994, pp. 160-169.

[ArW93] J. B. Armstrong, D. W. Watson, and H. J. Siegel, "Software issues for the PASM parallel processing system," in *Software for Parallel Computation,* J. Kowalik and L. Grandinetti, eds., Springer-Verlag, Berlin, Germany, 1993, pp. 134-148.

[AtB92] M. J. Atallah, C. L. Black, D. C. Marinescu, H. J. Siegel, and T. L Casavant, "Models and Algorithms for Coscheduling Compute-Intensive Tasks on a Network of Workstations," *J. Parallel and Distributed Computing*, Vol. 16, No. 4, Dec. 1992, pp. 319-327.

[AuB86] M. Auguin and F. Boeri, "The OPSILA computer," in *Parallel Languages and Architectures,* M. Consard, ed., Elsevier Science, Holland, 1986, pp. 143-153.

[AuB87] M. Auguin and F. Boeri, "Experiments on a parallel SIMD/SPMD architecture and its programming," *France-Japan Artificial Intelligence and Computer Science Symp. 87,* Nov. 1987, pp. 385-411.

[Bat68] K. E. Batcher, "Sorting Networks and Their Applications," *Proceedings of the AFIPS 1968 Spring Joint Computer Conference,* 1968, pp. 307-314.

[Bat77] K. E. Batcher, "The multidimensional access memory in STARAN," *IEEE Trans. on Computers,* C-26(2), Feb. 1977, pp. 174-177.

[BeB93] L. Bergman, H-W. Braun, B. Chinoy, A. Kolawa, A. Kuppermann, P. Lyster, C. R. Mechoso, P. Messina, J. Morrison, D. Stanfill, W. St. John, and S. Tenbrink, "CASA Gigabit Testbed 1993 Annual Report: A Testbed for Distributed Supercomputing," *Technical Report CCSF-33*, Caltech Concurrent Supercomputing Facilities, California Institute of Technology, Pasadena, California, May 1993, 68 pp.

[BeD91] A. Beguelin, J. Dongarra, G. A. Geist, R. Manchek, and V. Sunderam, "A User's Guide to PVM: Parallel Virtual Machine," *Technical Report ORNL/TM-11826*, Oak Ridge National Laboratory, Engineering Physics and Mathematics Division, Oak Ridge, Tennessee, July 1991, 13 pp.

[BeD92] A. Beguelin, J. Dongarra, G. A. Geist, R. Manchek, K. Moore, R. Wade, J. Plank, and V. Sunderam, "HeNCE: A User's Guide, Version 1.2," Oak Ridge National Laboratory, Engineering Physics and Mathematics Division, Dec. 1992, 28 pp.

[BeD93] A. Beguelin, J. Dongarra, A. Geist, R. Manchek, and V. Sunderam, "Visualization and Debugging in a Heterogeneous Environment," *IEEE Computer*, Vol. 26, No. 6, June 1993, pp. 88-95.

[Beg93] A. L. Beguelin, "Xab: A Tool for Monitoring PVM Programs," *Proceedings of the Workshop on Heterogeneous Processing*, Apr. 1993, pp. 92-97.

[BeK91] T. B. Berg, S.-D. Kim, and H. J. Siegel, "Limitations Imposed on Mixed-Mode Performance of Optimized Phases Due to Temporal Juxtaposition," *J. Parallel and Distributed Computing*, Vol. 13, No. 2, Oct. 1991, pp. 154-169.

[BeS91] T. B. Berg and H. J. Siegel, "Instruction Execution Trade-Offs for SIMD vs. MIMD vs. Mixed-Mode Parallelism," *Proceedings of the Fifth International Parallel Processing Symposium,* May 1991, pp. 301-308.

[Bok81] S. H. Bokhari, "A shortest tree algorithm for optimal assignments across space and time in a distributed processor system," *IEEE Trans. on Software Engineering*, Vol. SE-7, No. 6, Nov. 1981, pp. 583-589.

[BoM76] J. A. Bondy and U. S. R. Murty, *Graph theory with applications*, Elsevier Science Publishing Co., Inc., New York, NY, 1976.

[BrA89] J. C. Browne, M. Azam, and S. Sobec, "CODE: A unified approach to parallel programming," *IEEE Software,* Vol. 6, No. 4, July 1989, pp. 10-18.

[BrC90] E. Bronson, T. Casavant, and L. Jamieson, "Experimental application-driven architecture analysis of an SIMD/MIMD parallel processing system," *IEEE Trans. Parallel and Distributed Systems,* Vol. 1, No. 2, Apr. 1990, pp. 195-205.

[BrH75] A. E. Bryson and Y.-C. Ho, *Applied Optimal Control,* Hempshire, Washington, DC, 1975.

[BrL85] R. P. Brent, F. T. Luk, C. Van Loan, "Computation of the singular value decomposition using mesh-connected processors," *J. VLSI and Computer Systems,* 1(3), 1985, pp. 242-270.

[BrT82] J. Browne, A. Tripathi, S. Fedak, A. Adiga, and R. Kapur, "A language for specification and programming of reconfigurable parallel computation structures." *Proc. 1982 Int'l Conf. Parallel Processing.* Aug. 1982, 142-149.

[BuG93] R. Butler, W. Gropp, and E. Lusk, "Developing Applications for a Heterogeneous Computing Environment," *Proceedings of the Workshop on Heterogeneous Processing*, Apr. 1993, pp. 77-83.

[BuL92] R. M. Butler and E. L. Lusk, "User's Guide to the p4 Parallel Programming System," *Technical Report ANL-92/17*, Argonne National Laboratory, Mathematics and Computer Science Division, Argonne, Illinois, Oct. 1992, 34 pp.

[BuL93] R. M. Butler, A. L. Leveton, and E. L. Lusk, "p4-Linda: A Portable Implementation of Linda," *Proceedings of the Second International Symposium on High Performance Distributed Computing*, July 1993, pp. 50-58.

[BuL94] R. M. Bulter and E. L. Lusk, "Monitors, Messages, and Clusters: The p4 Parallel Programming System," *Parallel Computing*, Vol. 20, Apr. 1994, pp. 547-564.

[CaG92] N. Carriero, D. Gelernter, and T. G. Mattson, "Linda in Heterogeneous Computing Environments," *Proceedings of the Workshop on Heterogeneous Processing*, Mar. 1992, pp. 43-46.

[CaK88] T. L. Casavant and J. G. Kuhl, "A Taxonomy of Scheduling in General-Purpose Distributed Computing Systems," *IEEE Trans. Software Engineering*, Vol. 14, No. 2, Feb. 1988, pp. 141-154.

[ChC89] H. Chuang, L. Chen, "Efficient computation of the singular value decomposition on cube connected SIMD machine," *Supercomputing '89,* Nov. 1989, pp. 276-282.

[ChC92] M. Chen, and J. Cowie, "Prototyping Fortran-90 compilers for massively parallel machines." *Proc. ACM SIGPLAN 1992 Conf. Programming Language Design and Implementation.* June 1992, 94-105.

[ChE93] S. Chen, M. M. Eshaghian, A. Khokhar, and M. E. Shaaban, "A selection theory and methodology for heterogeneous supercomputing," *Workshop on Heterogeneous Processing*, Apr. 1993, pp. 15-22.

[CoH91] T. M. Conte and W. W. Hwu, "Benchmark Characterization," *IEEE Computer*, Vol. 24, No.1, Jan. 1991, pp. 48-56.

[CoL92] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms*, MIT Press, Cambridge, Mass., 1992.

[DaG88] F. Darema, D. A. George, V. A. Norton, and G. F. Pfister, "A Single-Program-Multiple-Data Computational Model for EPEX/FORTRAN," *Parallel Computing*, Vol. 7, Apr. 1988, pp. 11-24.

[Dav70] H. A. David, *Order Statistics*. John Wiley & Sons, New York, NY, 1970.

[DiC93] H. G. Dietz, W. E. Cohen, and B. K. Grant, "Would You Run It Here... Or There? (AHS: Automatic Heterogeneous Supercomputing)," *Proceedings of 1993 International Conference on Parallel Processing*, Vol. II, Aug. 1993, pp. 217-221.

[Dij59] E. Dijkstra, "A note on two problems in connexion with graphs," *Numerische Mithematik,* Vol. 1, 1959, pp. 269-271.

[DiK85] H. Dietz and D. Klappholz, "Refined C: a sequential language for parallel programming," *1985 Int'l Conf. Parallel Processing,* Aug. 1985, pp. 442-449.

[DiS89] H. G. Dietz, T. Schwederski, M. T. O'Keefe, A. Zaafrani, "Static synchronization beyond VLIW," *Supercomputing '89,* Nov. 1989, pp. 416-425.

[DiZ92] H. Dietz, A. Zaafrani, and M. O'Keefe, "Static scheduling for barrier MIMD architectures," *J. Supercomputing,* Vol. 5, 1992, pp. 263-289.

[DoM87] J. Dongarra, J. L. Martin, and J. Worlton, "Computer Benchmarking: Paths and Pitfalls," *IEEE Spectrum*, July 1987, pp. 38-43.

[DuB88] P. Duclos, F. Boeri, M. Auguin, and G. Giraudon, "Image Processing on a SIMD/SPMD Architecture: OPSILA," *Proceedings of the 9th International Conference on Pattern Recognition*, Nov. 1988, pp. 14-17.

[EsF92] M. M. Eshaghian and R. F. Freund, "Cluster-M Paradigms for High-Order Heterogeneous Procedural Specification Computing," *Proceedings of the Workshop on Heterogeneous Processing*, Mar. 1992, pp. 47-49.

[Exp90] Parasoft Corporation, *Express User's Guide Version 3.0*, Parasoft Corporation, Pasadena, California, Feb. 1990.

[Fer89] 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.

[FiC91] S. A. Fineberg, T. L. Casavant, and H. J. Siegel, "Experimental Analysis of a Mixed-Mode Parallel Architecture Using Bitonic Sequence Sorting," *J. Parallel and Distributed Computing*, Vol. 11, No. 3, Mar. 1991, pp. 239-251.

[Fly66] M. J. Flynn, "Very High-Speed Computing Systems," *Proceedings of the IEEE*, Vol. 54, No. 12, Dec. 1966, pp. 1901-1909.

[FrC90] R. F. Freund and D. S. Conwell, "Superconcurrency: a Form of Distributed Heterogeneous Supercomputing," *Supercomputing Review*, Vol. 3, No. 10, Oct. 1990, pp. 47-50.

[Fre89] R. F. Freund, "Optimal Selection Theory for Superconcurrency," *Proceedings of Supercomputing '89*, Nov. 1989, pp. 699-703.

[Fre91] R. F. Freund, "SuperC or Distributed Heterogeneous HPC," *Computing Systems in Engineering*, Vol. 2, No. 4, 1991, pp. 349-355.

[FrS93] R. F. Freund and H. J. Siegel, "Guest Editors' Introduction: Heterogeneous Processing," *IEEE Computer*, Vol. 26, No. 6, June 1993, pp. 13-17.

[GeH90] G. A. Geist, M. T. Heath, B. W. Peyton, and P. H. Worley, "User's Guide to PICL: A Portable Instrumented Communication Library," *Technical Report ORNL/TM-11616*, Oak Ridge National Laboratory, Engineering Physics and Mathematics Division, Oak Ridge, Tennessee, Aug. 1990, 22 pp.

[GhY93] A. Ghafoor and J. Yang, "Distributed Heterogeneous Supercomputing Management System," *IEEE Computer*, Vol. 26, No. 6, June 1993, pp.78-86.

[GiW92] N. Giolmas, D. W. Watson, D. M. Chelberg, and H. J. Siegel, "A Parallel Approach to Hybrid Range Image Segmentation," *Proceedings of the 6th International Parallel Processing Symposium*, Mar. 1992, pp. 334-342.

[GoV83] G. H. Golub, C. F. Van Loan, *Matrix Computations,* Johns Hopkins University Press, Baltimore, MD, 1983.

[Gri93] A. S. Grimshaw, "Easy-to-Use Object-Oriented Parallel Processing with Mentat," *IEEE Computer*, Vol. 26, No. 5, May 1993, pp. 39-51.

[GrW94] A. S. Grimshaw, J. B. Weissman, E. A. West, and E. Loyot, "Meta Systems: An Approach Combining Parallel Processing and Heterogeneous Distributed Computing Systems," *J. Parallel and Distributed Computing*,

[GuB92] M. Gupta and P. Banerjee, "Compile-time estimation of communication costs on multicomputers," *6th Int'l Parallel Processing Symp.*, Mar. 1992, pp. 470-475.

[Had93] E. Haddad, "Load Distribution Optimization in Heterogeneous Multiple Processor Systems," *Proceedings of the Workshop on Heterogeneous Processing*, Apr. 1993, pp. 42-47.

[HeL91] V. Herrarte and E. Lusk, "Studying Parallel Program Behavior with Upshot," *Technical Report ANL-91/15*, Argonne National Laboratory, Mathematics and Computer Science Division, Aug. 1991.

[HeW93] C. G. Herter, T. M. Warschko, W. F. Tichy, and M. Philippsen, "Triton/1: A Massively-Parallel Mixed-Mode Computer Designed to Support High Level Languages," *Proceedings of the Workshop on Heterogeneous Processing*, Apr. 1993, pp. 65-70.

[HiK91] S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, and C. Tseng, "An overview of the Fortran D programming system." *Preliminary Proc. 4th Workshop on Languages and Compilers for Parallel Computing*. Aug. 1991, e1-e17.

[HiS86] W. D. Hillis and G. L. Steele, Jr. "Data parallel algorithms," *Communications of the ACM*, Vol. 29, No. 12, Dec. 1986, pp. 1170-1183.

[HPF92] High Performance Fortran Forum. "Draft: high performance Fortran language specification." *High Performance Fortran Forum*. Sep. 1992.

[Jam87] L. H. Jamieson, "Characterizing Parallel Algorithms," in *The Characteristics of Parallel Algorithms*, L. H. Jamieson, D. B. Gannon, and R. J. Douglass, eds., MIT Press, Cambridge, Massachusetts, 1987, pp. 65-100.

[KaJ93] J. K. Karpovich, M. Judd, W. T. Strayer, and A. S. Grimshaw, "A Parallel Object-Oriented Framework for Stencil Algorithms," *Proceedings of the Second International Symposium on High Performance Distributed Computing*, July 1993, pp. 34-41.

[KaN93] J. A. Kaplan and M. L. Nelson, "A Comparison of Queueing, Cluster and Distributed Computing Systems," *NASA Technical Memorandum 109025*, National Aeronautics and Space Administration, Langley Research Center, Hampton, Virginia, Oct. 1993, 47 pp.

[KhP92] A. Khokhar, V. Prasanna, M. Shaaban, and C. Wang, "Heterogeneous Supercomputing: Problems and Issues," *Proceedings of Workshop on Heterogeneous Processing*, Mar. 1992, pp. 3-12.

[KhP93] A. Khokhar, V. Prasanna, M. Shaaban, and C. Wang, "Heterogeneous Computing: Challenges and Opportunities," *IEEE Computer*, Vol. 26, No. 6, June 1993, pp. 18-27.

[KiN91] S.-D. Kim, M. A. Nichols, and H. J. Siegel, "Modeling Overlapped Operation between the Control Unit and Processing Elements in an SIMD Machine," *J. Parallel and Distributed Computing*, Vol. 12, No. 4, Aug. 1991, pp. 329-342.

[KlH83] C. A. Klein, C. H. Huang, "Review of pseudoinverse control for use with kinematically redundant manipulators," *IEEE Trans. on Systems, Man, and Cybernetics,* 13(2), 1983, pp. 245-250.

[KlM93] A. E. Klietz, A. V. Malevsky, and K. Chin-Purcell, "A Case Study in Metacomputing: Distributed Simulations of Mixing in Turbulent Convection," *Proceedings of the Workshop on Heterogeneous Processing*, Apr. 1993, pp. 101-106.

[Kog94] P. M. Kogge, "EXECUBE - a New Architecture for Scalable MPPs," *Proceedings of 1994 International Conference on Parallel Processing*, Vol. I, Aug. 1994, pp. 77-84.

[Law75] D. H. Lawrie, "Access and alignment of data in an array processor," *IEEE Trans. on Computers,* C-24(12), Dec. 1975, pp. 1145-1155.

[LeP93] C. Leangsuksun and J. Potter, "Problem Representations for an Automatic Mapping Algorithm on Heterogeneous Processing Environments," *Proceedings of the Workshop on Heterogeneous Processing*, Apr. 1993, pp. 48-53.

[LiA95] Y. A. Li, J. K. Antonio, H. J. Siegel, M. Tan, D. W. Watson, "Estimating the Distribution of Execution Times for SIMD/SPMD Mixed-Mode Programs," *Heterogeneous Computing Workshop*, April 1995, to appear.

[Lil93] D. J. Lilja, "Experiments with a Task Partitioning Model for Heterogeneous Computing," *Proceedings of the Workshop on Heterogeneous Processing*, Apr. 1993, pp. 29-35.

[LiM87] G. J. Lipovski and M. Malek, *Parallel Computing: Theory and Comparisons*, John Wiley & Sons, New York City, New York, 1987.

[Lo88] V. M. Lo, "Heuristic algorithm for task assignment in distributed systems," *IEEE Trans. on Computers*, Vol. 37, No. 11, Nov. 1988, pp. 1384-1397.

[Luk80] F. T. Luk, "Computing the singular-value decomposition on the ILLIAC IV," *ACM Trans. on Mathematical Software,* 6(4), Dec. 1980, pp. 524-539.

[Luk86] F. T. Luk, "A triangular processor array for computing singular values," *Linear Algebra and its Applications,* 77(5), 1986, pp. 259-273.

[MaK89] A. A. Maciejewski, C. A. Klein, "The singular value decomposition: Computation and applications to robotics," *Int'l J. Robotics Research,* 8(6), Dec. 1989, pp. 63-79.

[MiP90] S. P. Midkiff, and D. A. Padua, "Issues in the optimization of parallel programs." *Proc. 1990 Int'l Conf. Parallel Processing*. **2**, Aug. 1990, 105-113.

[MoG74] A. M. Mood, F. A. Graybill, and D. C. Boes, *Introduction to the Theory of Statistics*, McGraw-Hill, 1974.

[Moo57] E. F. Moore, "The shortest paths through a maze," *Int'l Symp. Theory of Switching*, 1957, pp. 285-292.

[Mot87] Motorola, Inc., *MC68881/MC68882 Floating-Point Coprocessor User's Manual*, MC68881UM/AD, Motorola, Inc., 1987.

[Nas79] J. C. Nash, *Compact Numerical Methods for Computers: Linear Algebra and Function Minimisation*, John Wiley & Sons, NY, 1979.

[NaY94] B. Narahari, A. Youssef, and H. A. Choi, "Matching and Scheduling in a Generalized Optimal Selection Theory," *Proceedings of the Heterogeneous Computing Workshop*, Apr. 1994, pp. 3-8.

[NiH81] L. M. Ni and K. Hwang, "Optimal Load Balancing Strategies for a Multiple Processor System," *Proceeding of the 1981 International Conference on Parallel Processing*, Aug. 1981, pp. 352-357.

[NiS93] M. A. Nichols, H. J. Siegel, and H. G. Dietz, "Data management and control-flow aspects of an SIMD/SPMD parallel language/compiler," *IEEE Trans. Parallel and Distributed Systems*, Vol. 4, No. 2, Feb. 1993, pp. 222-234.

[NoT93] M. G. Norman and P. Thanisch, "Models of Machines and Computation for Mapping in Multicomputers," *ACM Computing Surveys*, Vol. 25, No. 3, Sep. 1993, pp. 263-302.

[Pap84] A. Papoulis, *Probability, Random Variables, and Stochastic Processes*. McGraw-Hill, New York, NY, 1984.

[PeG91] D. Pease, A. Ghafoor, I. Ahmad, D. L. Andrews, K. Foudil-Bey, T. E. Karpinski, M. A. Mikki, and M. Zerrouki, "PAWS: a Performance Evaluation Tool for Parallel Computing Systems," *IEEE Computer*, Vol. 24, No. 1, Jan. 1991, pp. 18-29.

[PhW93] M. Philippsen, T. Warschko, W. F. Tichy, and C. Herter, "Project Triton: Towards Improved Programmability of Parallel Machines," *Proceedings of the 26th Hawaii International Conference on System Sciences*, Jan. 1993, pp.192-201.

[Pol88] C. D. Polychronopoulos, *Parallel Programming and Compilers*, Kluwer Academic Publishers, Norwell, Massachusetts, 1988.

[QiS91] B. Qin, H. Sholl and R. Ammar, "Micro time cost analysis of parallel computations," *IEEE Trans. Computers,* Vol. 40, No. 5, May 1991, pp. 613-628.

[RoC92] J. Rosenman and T. Cullip, "High-Performance Computing in Radiation Cancer Treatment," *CRC Critical Reviews in Biomedical Engineering*, Vol. 20, Nos. 5-6, 1992, pp. 391-402.

[SaS93] G. Saghi, H. J. Siegel, and J. L. Gray, "Predicting Performance and Selecting Modes of Parallelism: a Case Study Using Cyclic Reduction on Three Parallel Machines," *J. Parallel and Distributed Computing,* Vol. 19, No. 3, Nov. 1993, pp. 219-233.

[ScL86] D. E. Schimmel, F. T. Luk, "A new systolic array for the singular value decomposition," in *Advanced Research in VLSI,* C. E. Leiserson, ed., MIT Press, Cambridge, MA, 1986, pp. 205-217.

[SiA92a] H. J. Siegel and S. Abraham, et al., "Report of the Purdue Workshop on Grand Challenges in Computer Architecture for the Support of High Performance Computing," *J. Parallel and Distributed Computing,* Vol. 16, No. 3, Nov. 1992, pp. 199-211.

[SiA92b] H. J. Siegel, J. B. Armstrong, and D. W. Watson, "Mapping Computer-Vision-Related Tasks onto Reconfigurable Parallel Processing Systems," *IEEE Computer*, Vol. 25, No. 2, Feb. 1992, pp. 54-63.

[SiA95] H. J. Siegel, J. K. Antonio, R. C. Metzger, M. Tan, and Y. A. Li, "Heterogeneous computing," in *Handbook of Parallel and Distributed Computing*, edited by A. Y. Zomaya, McGraw-Hill, 1995, to appear (also Purdue University, School of Electrical Engineering, Technical report TR-EE 94-37, Dec. 1994).

[Sie90] H. J. Siegel, *Interconnection Networks for Large-Scale Parallel Processing: Theory and Case Studies, Second Edition,* McGraw-Hill, New York, NY, 1990.

[SiS87] H. J. Siegel, T. Schwederski, J. T. Kuehn, and N. J. Davis IV, "An Overview of the PASM Parallel Processing System," in *Computer Architecture,* D. D. Gajski, V. M. Milutinovic, H. J. Siegel, and B. P. Furht, eds., IEEE Computer Society Press, Washington, DC, 1987, pp. 387-407.

[SiS95] H. J. Siegel, T. Schwederski, W. G. Nation, J. B. Armstrong, L. Wang, J. T. Kuehn, R. Gupta, M. D. Allemang, D. G. Meyer, and D. W. Watson, "The Design and Prototyping of the PASM Reconfigurable Parallel Processing System," to appear in *Parallel Computing: Paradigms and Applications*, A. Y. Zomaya, ed., Chapman and Hall, London, U.K., 1995.

[SiW95] H. J. Siegel, L. Wang, J. E. So, and M. Maheswaran, "Data Parallel Algorithms," to appear in *Handbook of Parallel and Distributed Computing*, A. Y. Zomaya, ed., McGraw-Hill, 1995.

[SpR90] "Special Report: Gigabit Network Testbeds," *IEEE Computer*, Vol. 23, No. 9, Sept. 1990, pp. 77-80.

[StA93] D. Stevenson and K. Antell, eds., "VISTAnet Annual Report," May 1993, 138 pp.

[Sto77] H. S. Stone, "Multiprocessor scheduling with the aid of network flow algorithms," *IEEE Trans. on Software Engineering*, Vol. SE-3, No. 1, Jan. 1977, pp. 85-93.

[Sun90] V. S. Sunderam, "PVM: A Framework for Parallel Distributed Computing," *Concurrency: Practice and Experience*, Vol. 2, No. 4, Dec. 1990, pp. 315-339.

[Sun92] V. S. Sunderam, "Design Issues in Heterogeneous Network Computing," *Proceedings of the Workshop on Heterogeneous Processing*, revised edition, Mar. 1992, pp. 101-112.

[Swa84] P. N. Swartzrauber, "The Shallow Benchmark Weather Prediction Program," *National Center for Atmospheric Research*, Oct. 1984.

[TaA95] M. Tan, J. K. Antonio, H. J. Siegel, and Y. A. Li, "Scheduling and Data Relocation for Sequentially Executed Subtasks in a Heterogeneous Computing System," *Heterogeneous Computing Workshop*, April 1995, to appear.

[TaN93] L. Tao, B. Narahari, and Y. C. Zhao, "Heuristics for Mapping Parallel Computations to Heterogeneous Parallel Architectures," *Proceedings of the Workshop on Heterogeneous Processing*, Apr. 1993, pp. 36-41.

[Tow86] D. Towsley, "Allocating programs containing branches and loops within a multiple processor system," *IEEE Trans. on Software Engineering*, Vol. SE-12, No. 10, Oct. 1986, pp. 1018-1024.

[TuR88] L. Tucker and G. Robertson, "Architecture and applications of the Connection Machine," *Computer,* Vol. 21, No. 8, Aug. 1988, pp. 26-38.

[UlM94] R. R. Ulrey, A. A. Maciejewski, and H. J. Siegel, "Parallel Algorithms for Singular Value Decomposition," *Proceedings of the 8th International Parallel Processing Symposium*, Apr. 1994, pp. 524-533.

[UlM??] R. R. Ulrey, A. A. Maciejewski, H. J. Siegel, *Parallel Approaches for Singular Value Decomposition on Systems with a Multistage Network,* Tech. Rep. in preparation, EE School, Purdue.

[WaA94] D. W. Watson, J. K. Antonio, H. J. Siegel, and M. J. Atallah, "Static Program Decomposition Among Machines in an SIMD/SPMD Heterogeneous Environment with Non-Constant Mode Switching Costs," *Proceedings of the Heterogeneous Computing Workshop*, Apr. 1994, pp. 58-65.

[WaK92] M. Wang, S.-D. Kim, M. A. Nichols, R. F. Freund, H. J. Siegel, and W. G. Nation, "Augmenting the Optimal Selection Theory for Superconcurrency," *Proceedings of the Workshop on Heterogeneous Processing*, Mar. 1992, pp. 13-22.

[WaS93] D. W. Watson, H. J. Siegel, J. K. Antonio, M. A. Nichols, and M. J. Atallah, "A framework for compile-time selection of parallel modes in an SIMD/SPMD heterogeneous environment," *Workshop on Heterogeneous Processing (WHP '93)*, Apr. 1993, pp. 57-64.

[WaS94] D. W. Watson, H. J. Siegel, J. K. Antonio, M. A. Nichols, and M. J. Atallah, "A Block-Based Mode Selection Model for SIMD/SPMD Parallel Environments," *J. Parallel and Distributed Computing*, Vol. 21, No. 3, June 1994, pp. 271-288.

[WeW94] C. C. Weems, G. E. Weaver, and S. G. Dropsho, "Linguistic Support for Heterogeneous Parallel Processing: a Survey and an Approach," *Proceedings of the Heterogeneous Computing Workshop*, Apr. 1994, pp. 81-88.

[Whi69] D. E. Whitney, "Resolved motion rate control of manipulators and human prostheses," *IEEE Trans. on Man-Machine Systems*, 10(2), 1969, pp. 47-53.

[YaG93] J. Yang, I. Ahmad, and A. Ghafoor, "Estimation of Execution Times on Heterogeneous Supercomputer Architecture," *Proceedings of the 1993 International Conference on Parallel Processing*, Vol. I, Aug. 1993, pp. 219-225.

[YaK94] J. Yang, A. Khokhar, S. Sheikh, and A. Ghafoor, "Estimating Execution Time for Parallel Tasks in Heterogeneous Processing (HP) Environment," *Proceedings of the Heterogeneous Computing Workshop*, Apr. 1994, pp. 23-28.

[Yos85] T. Yoshikawa, "Manipulability of robotic mechanisms," *Int'l J. Robotics Research*, 4(2), 1985, pp. 3-9.

[ZiB88] H. P. Zima, H.-J. Bast, and M. Gerndt, "SUPERB: a tool for semi-automatic MIMD/SIMD parallelization," *Parallel Computing*, Vol. 6, No. 1, Jan. 1988, pp. 1-18.

Rome Laboratory

Customer Satisfaction Survey

RL-TR-_____

Please complete this survey, and mail to RL/IMPS, 26 Electronic Pky, Griffiss AFB NY 13441-4514. Your assessment and feedback regarding this technical report will allow Rome Laboratory to have a vehicle to continuously improve our methods of research, publication, and customer satisfaction. Your assistance is greatly appreciated.
Thank You

_____

_____

Organization Name:_____(Optional)

Organization POC: _____(Optional)

Address:_____

1. On a scale of 1 to 5 how would you rate the technology developed under this research?

5-Extremely Useful 1-Not Useful/Wasteful

Rating_____

Please use the space below to comment on your rating. Please suggest improvements. Use the back of this sheet if necessary.

2. Do any specific areas of the report stand out as exceptional?

Yes___ No_____

If yes, please identify the area(s), and comment on what aspects make them "stand out."

3. Do any specific areas of the report stand out as inferior?

Yes___ No___

If yes, please identify the area(s), and comment on what aspects make them "stand out."

4. Please utilize the space below to comment on any other aspects of the report. Comments on both technical content and reporting format are desired.

*U.S. GOVERNMENT PRINTING OFFICE: 1995-710-126-20060

MISSION

OF

ROME LABORATORY

Mission. The mission of Rome Laboratory is to advance the science and technologies of command, control, communications and intelligence and to transition them into systems to meet customer needs. To achieve this, Rome Lab:

a. Conducts vigorous research, development and test programs in all applicable technologies;

b. Transitions technology to current and future systems to improve operational capability, readiness, and supportability;

c. Provides a full range of technical support to Air Force Materiel Command product centers and other Air Force organizations;

d. Promotes transfer of technology to the private sector;

e. Maintains leading edge technological expertise in the areas of surveillance, communications, command and control, intelligence, reliability science, electro-magnetic technology, photonics, signal processing, and computational science.

The thrust areas of technical competence include: Surveillance, Communications, Command and Control, Intelligence, Signal Processing, Computer Science and Technology, Electromagnetic Technology, Photonics and Reliability Sciences.