Copyright Warning & Restrictions

The copyright law of the United States (Title 17, United States Code) governs the making of photocopies or other reproductions of copyrighted material.

Under certain conditions specified in the law, libraries and archives are authorized to furnish a photocopy or other reproduction. One of these specified conditions is that the photocopy or reproduction is not to be "used for any purpose other than private study, scholarship, or research." If a, user makes a request for, or later uses, a photocopy or reproduction for purposes in excess of "fair use" that user may be liable for copyright infringement,

This institution reserves the right to refuse to accept a copying order if, in its judgment, fulfillment of the order would involve violation of copyright law.

Please Note: The author retains the copyright while the New Jersey Institute of Technology reserves the right to distribute this thesis or dissertation

Printing note: If you do not wish to print this page, then select "Pages from: first page # to: last page #" on the print dialog screen

INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

UMI

A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor MI 48106-1346 USA
313/761-4700 800/521-0600

ABSTRACT

THEORY AND DESIGN OF PORTABLE PARALLEL PROGRAMS FOR HETEROGENEOUS COMPUTING SYSTEMS AND NETWORKS

by
Ying-Chieh Wu

A recurring problem with high-performance computing is that advanced architectures generally achieve only a small fraction of their peak performance on many portions of real applications sets. The Amdahl's law corollary of this is that such architectures often spend most of their time on tasks (codes/algorithms and the data sets upon which they operate) for which they are unsuited. Heterogeneous Computing (HC) is needed in the mid 90's and beyond due to ever increasing superspeed requirements and the number of projects with these requirements. HC is defined as a special form of parallel and distributed computing that performs computations using a single autonomous computer operating in both SIMD and MIMD modes, or using a number of connected autonomous computers. Physical implementation of a heterogeneous network or system is currently possible due to the existing technological advances in networking and supercomputing. Unfortunately, software solutions for heterogeneous computing are still in their infancy. Theoretical models, software tools, and intelligent resource-management schemes need to be developed to support heterogeneous computing efficiently. In this thesis, we present a heterogeneous model of computation which encapsulates all the essential parameters for designing efficient software and hardware for HC. We also study a portable parallel programming tool, called Cluster-M, which implements this model. Furthermore, we study and analyze the hardware and software requirements of HC and show that Cluster-M satisfies the requirements of HC environments.

THEORY AND DESIGN OF PORTABLE PARALLEL PROGRAMS FOR HETEROGENEOUS COMPUTING SYSTEMS AND NETWORKS

by
Ying-Chieh Wu

A Dissertation
Submitted to the Faculty of
New Jersey Institute of Technology
in Partial Fulfillment of the Requirements for the Degree of
Doctor of Philosophy

Department of Computer and Information Science

May 1996

APPROVAL PAGE

THEORY AND DESIGN OF PORTABLE PARALLEL PROGRAMS FOR HETEROGENEOUS COMPUTING SYSTEMS AND NETWORKS

Ying-Chieh Wu

Dr. Mary M. Eshaghian, Dissertation Advisor — Date
Director of Advanced Computer Architecture and Parallel Processing Laboratory
Assistant Professor of Computer and Information Science and
Electrical and Computer Engineering, NJIT

Dr. John D. Carpinelli, Committee Member — Date
Director of Computer Engineering
Acting Associate Chair of Electrical and Computer Engineering
Associate Professor of Electrical and Computer Engineering and
Computer and Information Science, NJIT

Dr. Peter A. Ng, Committee Member — Date
Professor and Chairperson of Computer and Information Science, NJIT

Dr. Alice C. Parker, Committee Member — Date
The Vice Provost for Research and Dean of Graduate Studies
Professor of Electrical Engineering-Systems,
University of Southern California, Los Angeles, CA

Dr. Richard B. Scherl, Committee Member — Date
Assistant Professor of Computer and Information Science, NJIT

Dr. Sotirios G. Ziavras, Committee Member — Date
Associate Professor of Electrical and Computer Engineering and
Computer and Information Science, NJIT

BIOGRAPHICAL SKETCH

Author: Ying-Chieh Wu

Degree: Doctor of Philosophy

Date: May 1996

Undergraduate and Graduate Education:

- Doctor of Philosophy in Computer Science,
  New Jersey Institute of Technology, Newark, NJ, 1996
- Master of Science in Computer Science,
  University of Missouri, Columbia, MO, 1992
- Bachelor of Science in Computer Science,
  Tunghai University, Taichung, Taiwan, 1989

Major: Computer Science

Journal Publications:

1. A Suboptimal Heterogeneous Mapping, with M. M. Eshaghian and A. C. Parker, accepted for publication in *Journal of High Performance Computing*, 1996.
2. Evaluation of Two Programming Paradigms for Heterogeneous Supercomputing, with S. Chen, M. M. Eshaghian, R. Freund and J. L. Potter, *Journal of Parallel and Distributed Computing*, 31(1), pp. 41-55, Nov., 1995.
3. Mapping Arbitrary Nonuniform Task Graphs onto Arbitrary Nonuniform System Graphs, with S. Chen and M. M. Eshaghian, in revision for publication in *IEEE Transactions on Parallel and Distributed Systems.*
4. The Heterogeneous Optimal Selection Theory, with S. Chen, M. M. Eshaghian, A. Khokhar and M. E. Shaaban, in revision for publication in *Parallel Processing Letters.*
5. On Estimating The Resource Requirements of Heterogeneous Tasks, with M. M. Eshaghian and A. C. Parker, submitted to *Future Generations Computer Systems.*

Book Chapters and Magazines

1. A Portable Programming Model for Network Heterogeneous Computing, in M. Eshaghian (ed.) *Heterogeneous Computing*, Artech House, Norwood, MA, 1996.

2. Mapping and Resource Estimation in Network Heterogeneous Computing, in M. Eshaghian (ed.) *Heterogeneous Computing*, Artech House, Norwood, MA, 1996.

3. A Portable Parallel Programming Tool, with M. Eshaghian, submitted to *IEEE Computer*, Oct., 1995.

Conference Publications:

1. A Suboptimal Algorithm for Mapping Parallel Tasks onto Heterogeneous Systems, with M. M. Eshaghian and A. C. Parker, submitted to *Fifth IEEE International Symposium on High Performance Distributed Computing*, January 1996.

2. Mapping Arbitrary Non-Uniform Task Graphs onto Arbitrary Non-Uniform System Graphs, with S. Chen and M. M. Eshaghian, *Proc. of the International Conference on Parallel Processing*, Vol. II, pp. 191-195, Oconomowoc, WI, August, 1995.

3. On Estimating The Resource Requirements of Heterogeneous Tasks, with M. M. Eshaghian and A. C. Parker, *Proc. of the IPPS Workshop on Heterogeneous Computing*, pp. 47-52, Santa Barbara, CA, April, 1995.

4. A Sub-Optimal Assignment of Application Tasks onto Heterogeneous Systems, with J. Desouza-Batista, M. M. Eshaghian, A. C. Parker, and S. Prakash, *Proc. of the IPPS Workshop on Heterogeneous Computing*, pp. 9-16, Cancun, Mexico, April, 1994.

5. Scalable Heterogeneous Programming Tools, with S. Chen, M. M. Eshaghian, R. F. Freund, and J. L. Potter, *Proc. of the IPPS Workshop on Heterogeneous Computing*, pp. 89-96, Cancun, Mexico, April, 1994.

To my lovely wife and my parents

ACKNOWLEDGMENT

The author wishes to express his sincere gratitude to his advisor, Professor Mary M. Eshaghian, for her guidance, friendship, and moral support throughout this research.

Special thanks to Professor John D. Carpinelli, Professor Peter A. Ng, Professor Alice C. Parker, Professor Richard B. Scherl, and Professor Sotirios G. Ziavras for serving as members of the committee and offering invaluable suggestions to this dissertation.

The author appreciates the consistent help from the Cluster-M project team members: Geetha Chitti, Ajitha Gadangi, Javier G. Vasquez, and especially Dr. Song Chen.

Lastly, the author wants to thank his dear wife, Shiu-Ling Chen, for her love, understanding and help without which he simply can not complete this dissertation.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Figure **Page**

CHAPTER 1

INTRODUCTION TO HETEROGENEOUS COMPUTING

In this chapter, we introduce heterogeneous computing in Section 1.1 and discuss the three layers of heterogeneous computing in Sections 1.2 to 1.4. The organization of this thesis is then presented in Section 1.5.

1.1 Introduction

Today's supercomputing applications are characterized by a high level of diversity in terms of the type of embedded parallelism and by an ever-increasing demand for computational performance. Conventional parallel supercomputing systems utilize a number of homogeneous processors to cooperate on solving parallel tasks. These systems are usually classified according to the multiplicity of data and instruction streams [31].

Such homogeneous systems provide efficient solutions to tasks with embedded parallelism matching that offered by the system (i.e. SIMD, MIMD, vector). If more than one type of parallelism is present in a task, the system performance is greatly degraded. If greater computational power is needed, the whole system needs to be replaced by a more powerful homogeneous system, a costly solution.

Heterogeneous computing is a novel approach that overcomes several shortcomings of conventional homogeneous parallel systems. Heterogeneous computing (HC) is defined as a special form of parallel and distributed computing that performs computations using a single autonomous computer operating in both SIMD and MIMD modes, or using a number of connected autonomous computers. This approach aims at providing high performance by executing portions of code on machines offering similar types of parallelism.

The HC environment is comprised of several hardware and software components that manage the suite of heterogeneous machines in the system, thus enabling applications to run efficiently. The hardware and software requirements for HC can be classified into three layers: network layer, communication layer, and intelligent layer. In this thesis, we concentrate on issues related to the intelligent layer. We next describe each of these layers.

1.2 Network Layer

The network layer in HC includes the physical aspects of interconnecting the autonomous high performance machines in the system. This includes low level network protocols and machine interfaces. Current Local Area Networks (LANs) can be used to connect existing machines but this approach is not suitable for HC. In order to realize a HC environment, higher bandwidth and lower latency networks are essential. The bandwidth of commercially-available LANs is limited to about 10 Mbits/sec. However, in HC, assuming machines operating at 25 MHz clock with 40 MIPS instruction rate and 16 bits word length, a bandwidth in the order of 1 Gbits/sec is required to match computation and communication.

Recent advances in network technology have made it feasible to build *gigabit LANs*. Links in these networks are capable of operating on the order of 1 Gbits/sec or higher rates. Thus having at least 100 more bandwidth than today's 10 Mbits/sec Ethernets. Gigabit LAN standards are emerging. The High Performance Parallel Interface (HIPPI), whose physical layer has been approved as an ANSI standard, will likely become the backbone for interconnecting machines in HC. HIPPI-based LANS support data rates of 800 Mbits/sec and 1.6 Gbit/sec. Such networks have been used to interconnect CRAY-2 and CM-2 at the Minnesota Supercomputer Center [70]. A similar project using A CRAY Y-MP and CM-2 was undertaken at the Pittsburgh Supercomputing Center [47].

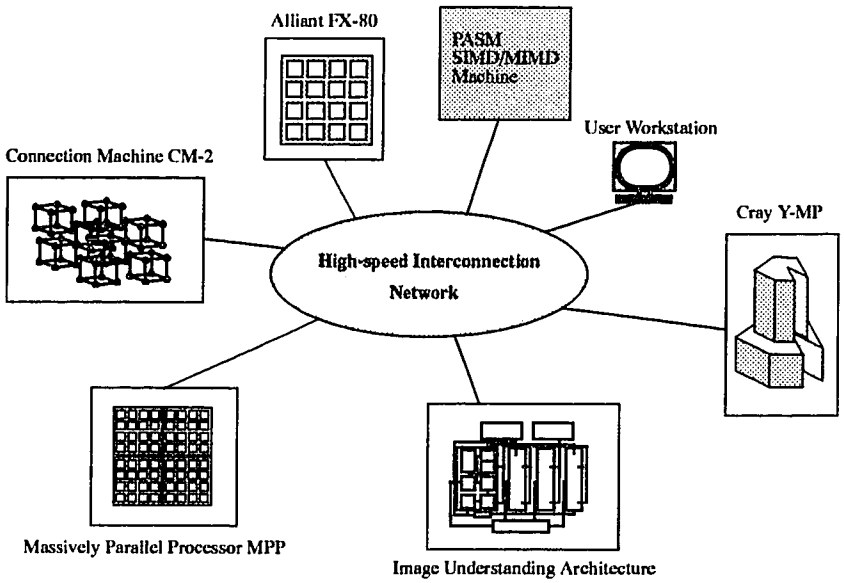


Figure 1.1 A heterogeneous network-based parallel computing system.

Even with high bandwidth networks, there are three main sources of inefficiency in current network implementations. First, existing application interfaces incur excessive overhead due to context switching and data copying between the user process and the machine's operating system. Secondly, each machine must incur overhead of executing high-level protocols that ensure reliable communication between tasks. Also, the network interface burdens the machine with interrupt handling and header processing for each packet.

Nectar [5] is an example of a network backplane for heterogeneous multicomputers. It consists of a high-speed fiber-optic network, large crossbar switches and powerful network interface processors. Protocol processing is off-loaded to these interface processors.

In HC, modules from various vendors share physical interconnections. Since different manufacturers usually use different communication protocols, the network management problem becomes more complex [52]. The following three general approaches in dealing with network heterogeneity are given in [72]:

1. To treat the heterogeneous network as a partitioned network, each partition employs a uniform set of protocols,

2. to have only a single "visible" network management console, and

3. to integrate the heterogeneous management functions at a single management console.

1.3 Communication Layer

The HC environment achieves efficient execution of parallel tasks by decomposing the task into several modules which are assigned to machines in the system with a similar mode of embedded parallelism. The task modules run on assigned machines as local processes. These processes need to exchange intermediate results and process synchronization information, either from processes residing in the same machine or from processes residing on other machines using the network. Since each machine on the system may utilize different process communication and synchronization primitives, a uniform system-wide communication mechanism operating above native operating systems is needed to facilitate this exchange of information. Due to the networked nature of HC and the lack of shared memory, such a communication mechanism must support message passing.

An example of a communication tool suitable for HC is the parallel virtual machine (PVM) [66]. The PVM system emulates a virtual concurrent computing machine on a suite of networked machines by executing system-level processes on each machine. A process that runs on a local machine can access the virtual machine via library routines embedded in imperative procedural languages, such as C. Communication support is provided for process management via stream-oriented message-passing, synchronization based on barriers or variants of rendezvous and/or auxiliary tasks. These library routines interact with the PVM system process on each machine,

which then provides the requested actions in cooperation with PVM system processes running on other machines in the system. Other examples of networking communication tools are Portable Programs for Parallel Processors (P4) and Message Passing Interface (MPI). MPI includes a number of utilities for supporting message passing while P4 can handle both message passing and shared memory. MPI is a message passing interface for MIMD distributed memory concurrent computers. MPI includes point-to-point and collective communication routines, as well as support for process groups, communication contexts, and application topologies.

1.4 Intelligent Layer

The intelligent layer of the HC environment provides system-wide tools and techniques necessary to manage the suite of heterogeneous machines and to insure proper and efficient execution of tasks. Such tools operate over the native operating systems of the individual machines and use the process communication primitives provided by the communication layer. The services provided by this layer are the most challenging ones in HC and include programming environments, language support, application task decomposition, mapping and scheduling, and load balancing, as illustrated in Figure 1.4. We next briefly describe two functions which are essential for designing and supporting these various services. These functions are used in the Heterogeneous Optimal Selection Theory (HOST) presented in Section 1.4.2.

1.4.1 Code Profiling and Analytical Benchmarking

Traditional *program profiling* involves testing a program assumed to be comprised of several modules, by running it on some test data. The *profiler* monitors the execution of the program and gathers statistics including the running time of each program module. This information is then utilized to modify different modules

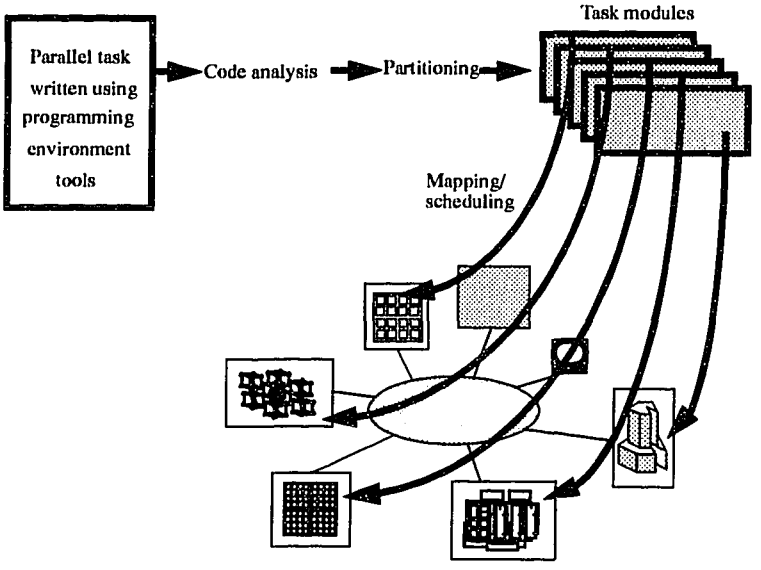


Figure 1.2 Intelligent layer services.

improving the overall execution time. In HC, profiling is not done only to estimate the execution time of code, but the type of the code according to the execution mode is also considered. This is achieved by *code-type profiling*. The code-type profiling introduced in [35] is a code-specific function to determine the code-type (e.g. SIMD, MIMD, vector, scalar, etc.).

Analytical benchmarking provides a mean to measure how well the available machines perform on a given code-type [35]. While code-type profiling identifies the type of code, analytical benchmarking ranks machines in terms of efficiency in executing a given code. Thus, analytical benchmarking techniques determine the relative effectiveness of a given parallel machine on various computation types.

1.4.2 Heterogeneous Optimal Selection Theory

In Freund's Optimal Selection Theory (OST), it can be assumed that the number of machines available is unlimited and that an application task is comprised of several uniform and non-overlapping code segments. Code segments are considered to be

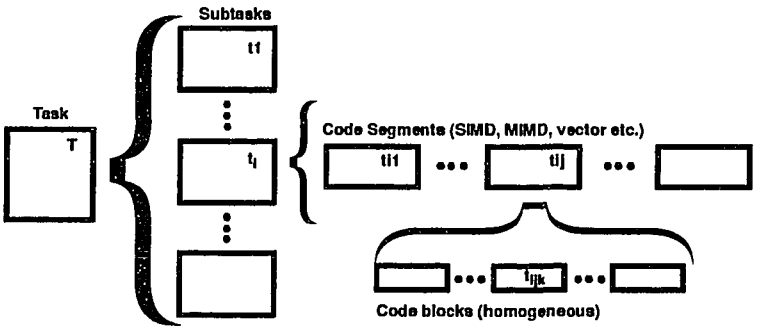


Figure 1.3 Input format of HOST.

executed in a series. Each code segment has homogeneous parallelism embedded in its computations. A code segment is decomposed into code blocks. All code blocks within a code segment have the same type of parallelism and can be executed concurrently. This type is determined by the process of task profiling. The goal of OST is to assign the code blocks, within each code segment, to the available matching machine types so that it may be optimally executed. Information about how fast a given machine type can execute a code type is assumed to be known as a result of analytical benchmarking. Augmented Optimal Selection Theory (AOST) [71] extended OST to incorporate the performance of code segments on non-optimal machine choices, assuming that the number of available machines for each type is limited. Based on this assumption, a code segment most suitable for one type of machine may have to be assigned to another type.

The Heterogeneous Optimal Selection Theory (HOST) [16] is an extension of AOST in two ways: it incorporates the effects of various, fine-grain, mapping techniques available on individual machines and it assumes heterogeneous embedded parallelism. The input format of HOST, as shown in Figure 1.3, allows concurrent execution of mutually independent code segments. An application task is decomposed into several subtasks which are then executed in series. Each subtask may contain a collection of code segments which can be executed in parallel. A code segment consists of a set of code blocks and a code block consists of a number of instructions.

All the code blocks within a code segment are of the same type and are to be executed concurrently or sequentially on the machines of the same type, depending on their interdependencies. A machine type is identified according to the underlying architecture, such as SIMD, MIMD, vector or scalar. Each machine type may have more than one model, for example, the Ncube and Mesh are two models of an SIMD machine type. In HOST, heterogeneous code blocks of different code segments can be executed concurrently on different machine types, thus exploiting the heterogeneous parallel computations embedded in a given application. Narahari et al.[51] extended HOST to the *Generalized Optimal Selection Theory (GOST)*. GOST allows non-optimal selections of machines, as in AOST, and heterogeneous code blocks, as in HOST. It further incorporates data communication time, system reconfiguration time and data conversion time [51].

To express the formulation of HOST, some parameters must be defined. Table 1.1 contains a complete listing of this notation. For a more detailed description of these terms, see [16]. HOST is formulated as follows:

For any subtask, there exists a τ *with* $\min_{\tau} \chi[\tau]$ *subject to* $\sum_{j=1}^{S} (\gamma[Y[j], j] \times c[Y[j]]) \leq C$

Based on HOST, an optimal machine selection leading to a minimum execution time exists. To find such an optimal solution, however, is not computationally feasible. Therefore, we present an overview of a set of sub-optimal solutions in the next section.

1.5 Organization of the Dissertation

This thesis focuses on the design issues of the intelligent layer in HC. In this section, we briefly present an overview of these issues. They are presented in detail later in this thesis.

S	the number of code segments of the given task
M	the number of different machine types to be considered
$\eta[y]$	the number of machine models of type y
$\alpha[y]$	the number of mappings available on machine type y
$\beta[y,l]$	the number of available machines of model l of type y
$v[y,j]$	the maximum number of code blocks code segment j can be decomposed
$\gamma[y,j]$	the number of machines of type y actually used to execute code segment j
$m[y,k]$	mapping technique used for a code block k on machine type y
$\theta[y,m]$	the optimal speedup for a particular mapping m on machine type y
$\pi[y,j]$	how well a code segment j can be matched with machine type y
$\Lambda[y,k]$	utilization factor when running code block k on a machine of type y
$p[j]$	the percentage of execution time of code segment j within a given subtask
$p[j,k]$	the percentage of execution time of block k within code segment j
$\mu[y,j]$	mapping vector for code segment j on machine type y
$\delta[y,j,\mu]$	execution time of segment j with mapping μ on machine type y
$\lambda[y,j]$	minimum execution time of segment j among all possible mappings on type y
τ	machine type selection vector
$\chi[\tau]$	execution time of the given subtask with machine type selection τ
$Y[j]$	the type of machine selected to execute code segment j
$c[Y[j]]$	the cost of machine selected to execute code segment j
C	the total cost constraint

Table 1.1 Notations used in HOST formulation

1.5.1 Portable Programming Model

A programming paradigm suitable for the intelligent layer must allow portable software to be shared and/or distributed among various computers in the heterogeneous suite. Furthermore, it must support architecturally independent programming which does not include any architecturally specific details. Since homogeneous programming tools are not suitable to heterogeneous computing, we need to develop a new tool based on a heterogeneous programming model. We present a heterogeneous parallel programming model, called Cluster-M, in Chapter 2. This model is proposed to bridge between software and hardware for heterogeneous computing.

It acts as an intermediate medium based on which portable parallel programs are specified and can be mapped onto dynamically reconfigured heterogeneous organizations. The implementation of this model as a portable programming tool is also presented. Using Cluster-M, a single program can be ported among various heterogeneous architectures or suite of computers.

1.5.2 Partitioning, Mapping and Scheduling

In HC, similar to homogeneous systems, the problems of partitioning a parallel task into several modules, mapping resulting modules into various machines and scheduling the execution of each module are pertinent. In the past, the partitioning and mapping problems for homogeneous parallel environments have been investigated extensively [9, 10, 18, 30, 43, 44, 61, 63, 64]. However, HC poses new constraints. In the following, we define partitioning and mapping as two different problems and also differentiate between the contexts of these terms in homogeneous and heterogeneous environments.

In a homogeneous environment, the partitioning problem addressed in [12, 36, 39] can be divided into two sub-problems. *Parallelism detection* determines the parallelism in a program. *Clustering* combines several operations into tasks and thus partitions the application into several tasks. Each cluster is then assigned to a processor. Both of these sub-problems can be performed by the user, the compiler or by the machine at run time.

The mapping/allocation of program modules to processors has been addressed by many researchers in the past [9, 18, 30, 43, 44, 61]. Informally, in homogeneous environments the mapping problem can be defined by assigning program modules to processors. Thus, the number of pairs of communicating modules that fall on pairs of directly connected processors is maximized [9]. In HC, machines are globally connected through a high-bandwidth network, and therefore the assignment of

communicating modules to directly-connected machines is not an issue. However, other objective functions for mapping, such as matching the code-type to the machine-type, add additional constraints. If such mapping has to be performed at run time, for load balancing purposes or due to failure of a machine, the mapping becomes more complicated.

In homogeneous environments, the scheduling process assigns each task to a processor in order to achieve better processor utilization and high throughput. Three levels of scheduling are generally employed. *High-level scheduling* selects a subset of all submitted jobs competing for the available resources. *Intermediate-level scheduling* responds to short-term fluctuations in the system load by temporarily suspending and activating processes to achieve smooth system operation. *Low-level scheduling* determines the next ready process to be assigned to a processor for a certain duration.

In HC, while all of the above three levels of scheduling may reside in each machine, a fourth level of scheduling is needed. This level deals with scheduling at the system level. The scheduler maintains a balanced system-wide workload by monitoring the progress of all the tasks in the system. The scheduler needs to know the different task-types and available machine-types (i.e., SIMD, MIMD, Mixed-mode, etc.) in the system, since tasks may be reassigned due to changes in the system configuration or due to overload problems. Communication bottlenecks and queueing delays incurred due to the heterogeneity of hardware add additional constraints on the scheduler. The scheduler also needs to use information from code-type profiling and analytical benchmarking.

In Chapter 3, we extend the algorithms of Chapter 2 to incorporate the "type heterogeneity" (i.e. SIMD and MIMD) of tasks and systems in HC. The augmented mapping algorithm presented maps tasks to processors of similar computation type and proceeds with an enhanced fine-grain mapping technique. Since the expected

number of clusters at every level of the fine-grain mapping is constant, we propose to use an optimal matching strategy to enhance the algorithm. Therefore, we formulate and solve each step of the fine-grain cluster mapping by using an Integer Linear Programming (ILP) model.

1.5.3 Hardware Estimation

Once the information provided by code-type profiling is available, it is desirable to know how many processors are needed for each of the code types. In Chapter 4, we propose two methods for estimating the minimum number of processors required for each of these code types in HC. The first method involves making use of task compatibility graphs. We show that a task compatibility graph can be generated by analyzing certain compatible relations between task module pairs of a given task flow graph. We define the resource (processor) minimization problem therefore to be equivalent to finding the minimal number of cliques that cover the task compatibility graph, or to finding the minimal number of colors that color the vertices of its complement graph, called task conflict graph. We solve this problem using a greedy approach in $O(|V| \log |V| + |E|)$ time, where $|V|$ and $|E|$ are the number of vertices and edges of the task compatibility graph. We further show that for special types of task compatibility graphs, the optimal solution can be obtained in polynomial time. The second method studied in Chapter 4 uses the Cluster-M methodology for estimating the minimum number of processors. Examples are shown to compare the estimated results obtained using different techniques.

1.5.4 Software Environments

In HC, machine-independent and portable parallel programming languages and tools are required. Also, a HC software package should be portable among and executable on various architectures. Certain tools are needed to act as intermediate media based on which machine-independent algorithms can be designed using a single

programming language. These are then mapped onto the desired architecture. One such programming model, *Linda* [13, 11] defines a logically shared data structuring memory mechanism called tuple space. However, Linda is difficult to implement on architectures not supporting a shared memory structure. In contrast to Linda, the programming tool *Express* supports a distributed-memory system organization. However, algorithms coded using Express are machine dependent, and therefore are not fully portable. Other candidate parallel programming environments for HC are: the *Actors* Programming model [1, 2, 3] and *Tool for Large-Grained Concurrency (TLC)*. TLC, developed by BBN, employs implicitly parallel constructs to specify the dependencies among a set of coarse-grained remote computations. The Actors model, on the other hand, allows massively parallel execution of algorithms. At extra cost of implementing such a system, Actors is machine independent: it can be executed on shared memory computers and over distributed networks.

Cluster-M, presented in Chapter 2, is a model which provides an environment for porting various tasks onto the machines in a heterogeneous suite, so that resource utilization is maximized and the overall execution time is minimized. In Chapter 5, we formally define the scalability of heterogeneous programming paradigms. Also, we present another portable and scalable programming paradigm, called Heterogeneous Associative Computing (HAsC)[54]. HAsC models a heterogeneous network as a coarse-grained associative computer and is designed to optimize the execution of problems where the size of the program is small compared to the amount of data processed. It uses broadcasting to avoid the mapping problem. Ease of programming and execution speed, not the utilization of idle resources are the primary goals of HAsC. We show that both Cluster-M and HAsC are scalable. We then illustrate how these two paradigms can be used together to provide an efficient medium for heterogeneous programming.

CHAPTER 2

A PORTABLE PARALLEL PROGRAMMING MODEL FOR HETEROGENEOUS COMPUTING

We present a heterogeneous parallel programming model called Cluster-M. This model is proposed to bridge between software and hardware for heterogeneous computing. It acts as an intermediate medium based on which portable parallel programs are specified and then can be mapped onto dynamically reconfigured heterogeneous organizations. The implementation of this model as a portable programming tool is presented in this chapter. Using Cluster-M, a single software can be ported among various heterogeneous architectures or suite of computers.

2.1 Introduction

A programming paradigm suitable for the intelligent layer should allow portable software to be shared and/or distributed among various computers in the heterogeneous suite. Furthermore, it should support architecturally independent programming that does not include any architecturally specific details. A number of homogeneous programming tools have been developed that take a high-level program as the input and map it onto the underlying systems. The question is whether or not these homogeneous programming tools can be directly used for heterogeneous computing. Examples of these tools include Linda, Prep-P, Oregami, Hypertool, PARSA, and PYRROS [13, 8, 45, 74, 75]. Linda [13] defines a logically shared memory mechanism called tuple space. Tuple space holds two kinds of tuples: process tuples, which are under active evaluation, and data tuples, which are passive. Ordinarily, building a Linda program involves dropping a process tuple into tuple space and then spawning other process tuples. This pool of process tuples, all executing simultaneously, exchange data by generating, reading, and consuming data tuples. Once a process tuple has finished executing, it turns into a data tuple that is indistinguishable from

other data tuples. Linda requires large volumes of data to be exchanged to and from the shared memory. For this reason, Linda has been mostly used for coarse-grain computations.

Prep-P, Oregami, Hypertool, and PYRROS, however, all include an architecturally independent mapping component that can map a fine-grain given parallel program onto either a special or an arbitrary system. However, the mapping components of Prep-P [8] and Oregami [45] are basically libraries of specialized mapping algorithms that only map regularly structured programs onto regularly structured systems. Their mappings for irregularly structured programs or systems that are not found in the libraries may be slow and ineffective. Hypertool [74] and PYRROS [75] generate fast and near-optimal mappings for arbitrary programs by using a clustering method. However, they can only be mapped onto fully connected systems. Therefore, they are not suitable for a heterogeneous network that may have arbitrary interconnections. This chapter will only focus on the tools that can efficiently map arbitrary program tasks onto arbitrary computer systems. Since homogeneous programming tools are not suitable to heterogeneous computing, we need to develop a new tool based on a heterogeneous programming model. An essential component of such a tool will be an efficient mapping algorithm, which maps an arbitrary task onto an arbitrary system.

A program task can be represented by a task graph, with each node representing a task module and each edge representing data communication between two modules. Each node is associated with a weight representing the time needed to execute the instructions contained in the node on a baseline computer, while the weight of an edge represents the communication amount. Similarly, a parallel computer system can be modeled as a weighted undirected system graph, whose weights represent processor speeds and transmission rates of communication links. If the task graphs and the system graphs are known before program execution, then mapping of the task

graphs onto the system graphs is called static mapping. Here, we consider only static mapping. In static mapping, the assignments of the nodes of the task graphs onto the system graphs are determined prior to the execution and are not changed until the end of the execution. Static mapping can be classified in two general ways. The first classification is based on the topology of task and/or system graphs [15]. Based on this, the mappings can be classified into four groups: (1) mapping specialized tasks onto specialized systems, (2) mapping specialized tasks onto arbitrary systems, (3) mapping arbitrary tasks onto specialized systems and (4) mapping arbitrary tasks onto arbitrary systems. The second classification can be based on the uniformity of the weights of the nodes and the edges of the task and/or the system graphs. Based on this, the mappings can be categorized into the following four groups: (1) mapping uniform tasks onto uniform systems [7, 9, 15, 24, 43], (2) mapping uniform tasks onto nonuniform systems, (3) mapping nonuniform tasks onto uniform systems [22, 48, 59, 74, 76] and (4) mapping nonuniform tasks onto nonuniform systems [44, 60].

Two of the earlier static mapping algorithms that can map arbitrary nonuniform task graphs onto arbitrary nonuniform system graphs are Lo's Max Flow/Min Cut algorithm [44], and El-Rewini and Lewis' mapping heuristic (MH) algorithm [22]. The time complexity of these two algorithms are $O(M^4 N \log M)$ and $O(M^2 N^3)$ respectively, where M is the number of task modules and N is the number of processors. In this chapter we present a mapping technique that is used in the mapping module of an implemented tool, which is based on a portable programming model for heterogeneous computing called Cluster-M. Using this paradigm, we can produce near-optimal mapping of arbitrary nonuniform architecture-independent task graphs onto arbitrary nonuniform system graphs in $O(MP)$ time, where $P = max(M, N)$. Similar to BSP and LogP, the Cluster-M model serves as an intermediate layer between software and hardware. Therefore, it supports portable

machine-independent programming. BSP and LogP support portable programming for a set of uniform (homogeneous) processing units, while the Cluster-M model allows the processing units to be nonuniform (heterogeneous).

The rest of this chapter is organized as follows. In Section 2 we present the Cluster-M heterogeneous model of computation. In Section 3, the components of the Cluster-M tool are presented. The efficiency of the Cluster-M mapping module is discussed in Section 4. Concluding remarks are in Section 5.

2.2 Cluster-M Portable Parallel Programming Tool

A tool implementing the Cluster-M model, presented in the last section, must support portable parallel algorithm design and programming. It must provide a mechanism so that both set of parameters can be extracted from any given problem and any underlying heterogeneous organization. Furthermore, this tool must provide an efficient mechanism for mapping these portable programs onto heterogeneous systems using these two sets of parameters. The Cluster-M tool, presented below, is an implementation of the model satisfying these conditions.

2.2.1 Cluster-M

Cluster-M is a programming tool that facilitates the design and mapping of portable parallel programs [15]. Cluster-M has three main components: the specification module, the representation module and the mapping module. In the specification module, machine-independent algorithms are specified and coded using the program composition notation (PCN) [34] programming language [25]. Cluster-M specifications are represented in the form of a multilayer clustered task graph called a Spec graph. Each clustering layer in the Spec graph represents a set of concurrent computations, called Spec clusters. A Cluster-M representation represents a multilayer partitioning of a system graph called a Rep graph. At every partitioning layer

of the Rep graph, there are a number of clusters called Rep clusters. Each Rep cluster represents a set of processors with a certain degree of connectivity. Given a task (system) graph, a Spec (Rep) graph can be generated using one of the Cluster-M clustering algorithms. The clustering is done only once for a given task (system) graph, independent of any system (task) graphs. It is a machine-independent (application-independent) clustering, therefore it is not necessary to repeat it for different mappings. For this reason, the time complexities of the clustering algorithms are not included in the time complexity of the Cluster-M mapping algorithm. In the mapping module, a given Spec graph is mapped onto a given Rep graph. This process is shown in Figure 2.1. In an earlier publication [15], two Cluster-M clustering algorithms and a mapping algorithm were presented for uniform graphs. Next, the basic concepts used in Cluster-M clustering and mapping will be explained. In Section 3, we will show how uniform Cluster-M algorithms can be extended and applied to nonuniform task and system graphs.

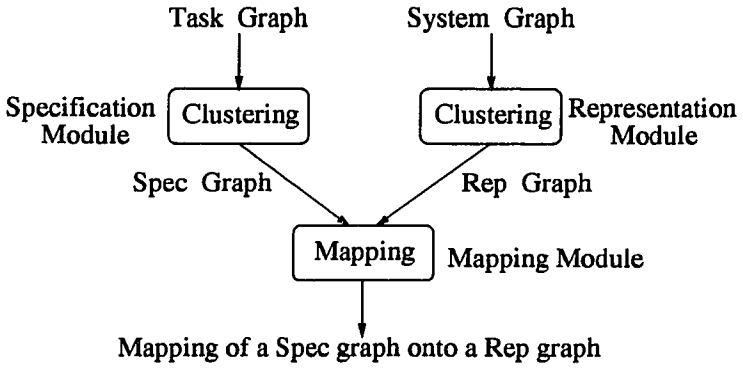


Figure 2.1 Cluster-M mapping process.

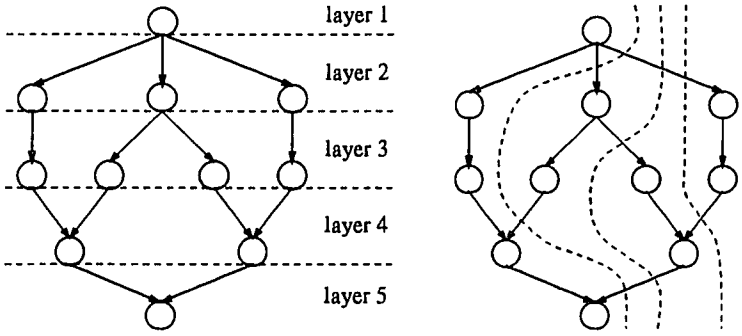
2.2.2 Basic Concepts

There are a number of reasons and benefits in clustering task and system graphs in the Cluster-M fashion. Basically, Cluster-M clustering causes both task and system graphs be partitioned so that the complexity of the mapping problem is simplified

and good mapping results can be obtained. In clustering an undirected graph, completely connected nodes are grouped together forming a set of clusters [15, 25]. Clusters are then grouped together again if they are completely connected. This is continued until no more clustering is possible. When an undirected graph is a task graph, then doing this clustering essentially identifies and groups communication-intensive sets of task nodes into a number of clusters called Spec clusters. Similarly for a system graph, doing the clustering identifies well-connected sets of processors into a number of clusters called Rep clusters. In the mapping process, each of the communication intensive sets of task nodes (Spec clusters) is to be mapped onto a communication-efficient subsystem (Rep cluster) of suitable size. Note that the mapping of undirected task graphs onto undirected system graphs is referred to as the allocation problem. An earlier publication [15] showed that Cluster-M clustering and mapping algorithms can lead to good allocation results. It compared its results with Bokhari's $O(N^3)$ algorithm and showed that its algorithm has a lower time complexity of $O(MN)$, where M and N are the number of nodes in the task and system graphs, respectively.

Clustering directed graphs (i.e., directed task graphs) produces two types of graph partitioning: horizontal and vertical. Horizontal partitioning is obtained because, as part of clustering, we divide a directed graph into a layered graph such that each layer consists of a number of computation nodes that can be executed in parallel and a number of communication edges incoming to these nodes. This is shown in Figure 2.2(a). The layers are to be executed one at a time. Therefore, the mapping is done one layer at a time. This significantly reduces the complexity of the mapping problem since the entire task graph need not be matched against the entire system graph.

Vertical graph partitioning is obtained because, as part of the clustering, the nodes from consecutive layers are merged or embedded. All the nodes in a layer are



(a) Horizontal (b) Vertical

Figure 2.2 Horizontal and vertical partitioning of a task graph.

merged to form a cluster if they have a common parent node in the layer above or a common child node in the layer below. Doing this traces the flow of data. This information will be used later as part of the mapping so that the tasks are placed onto the processors in a way that total communication overhead is minimized. For example, to avoid unnecessary communication overhead, the task nodes along a path may be embedded into one another so that they are assigned to the same processor. The effect of this type of partitioning is shown in Figure 2.2(b).

Both horizontal and vertical graph partitionings are accomplished by performing the clustering in a bottom-up fashion. The Cluster-M mapping will then be performed in a top-down fashion by mapping the Spec clusters one layer at a time onto the Rep clusters. The next two sections show how these clustering and mapping ideas work for nonuniformly weighted graphs. The nonuniform algorithms shown in this chapter are nontrivial extensions of the Cluster-M uniform algorithms presented in an earlier publication [15].

2.3 A Portable Parallel Programming Model

A computational model is designed such that it can be an efficient bridge between software and hardware; high-level languages can be compiled efficiently on to the

model; yet it can be efficiently implemented in hardware [69]. The von Neumann model is a computational model that successfully bridges the gap for sequential computations. For parallel computing, a number of models have been introduced. One of the earliest and most widely used parallel models is the parallel random access machine (PRAM) model [32]. This model is unrealistic because it assumes that all processors work synchronously and that interprocessor communication is free [19]. Several variations of the PRAM model have been proposed to identify restrictions that would make it more practical while preserving the unrealistic assumption that communication is free. Algorithms that are designed based on PRAM and its variations perform very poorly once mapped onto parallel machines with electrical interconnects. If the electrical interconnects are to be replaced with optical ones, however, the PRAM algorithms can be implemented efficiently [29, 26]. The optical model of computation (OMC) is a computational model for parallel architectures with unit-delay optical interconnects.

The bulk-synchronous parallel model (BSP) developed by Valiant [69] attempts to bridge theory and practice for all types of parallel computations. It assumes processors work synchronously, and it models latency and limited bandwidth. It requires few machine parameters as long as a certain programming methodology is followed. An improvement over the BSP model is the LogP model proposed by Culler et al. [19]. LogP allows algorithm designers to address key performance issues without specifying unnecessary details. It allows machine designers to give a concise performance summary of their machines, against which algorithms can be evaluated. Using LogP, portable parallel algorithms can be designed, if processors are all assumed to be identical (homogeneous).

Heterogeneous computing is defined as a special form of parallel and distributed computing that performs computations using a single autonomous computer operating in both SIMD and MIMD modes, or using a number of connected

autonomous computers. Furthermore, the heterogeneous architectures may be changed at every step of computation as new resources become available or occupied. Because of the nonuniformity and the unpredictability in the availability of the processing units, the LogP model will not be suitable as a model for heterogeneous computing [58]. The following presents the portable programming model called Cluster-M, which can efficiently bridge the software and hardware in a heterogeneous environment. This model allows software portability without imposing any restrictions on the hardware. The Cluster-M model consists of two sets of parameters, one for representing a portable parallel program and the other for specifying the organization of the underlying heterogeneous architecture or suite. In addition, the Cluster-M model consists of an evaluation function for predicting the time performance of any two sets of parameters being considered.

2.3.1 Machine-Independent Program Parameters

A given parallel program consists of a sequence of steps such that in each step a number of computations can be done concurrently. Each step is called a layer. These concurrent computations for a given step (layer) can each be presented by a cluster called a Spec cluster. The mth Spec cluster at layer u is denoted by S_m^u and associated with the following parameters.

σS_m^u The size of S_m^u, which is the maximum number of nodes in this cluster that can be computed in parallel.

δS_m^u The maximum sequential computation amounts (i.e., the maximum number of clock cycles required to execute all the instructions sequentially using a baseline computer) in S_m^u.

ΠS_m^u The total amount of communication from layer 1 to layer u of S_m^u.

πS_m^u The average communication amount at the layer u in S_m^u.

ρS_m^u The computational type of S_m^u. Its value is set to 0 for a single instruction, multiple data (SIMD) type and 1 for a multiple instruction, multiple data (MIMD) type.

2.3.2 Program-Independent Machine Parameters

Any heterogeneous architecture can be similarly represented in a multilayered format such that each layer presents a set of processing units that are completely connected. Each processing unit is represented by a cluster called a Rep cluster. The nth Rep cluster at layer v is denoted by R_n^v and associated with the following parameters.

σR_n^v The number of processors contained in R_n^v.

δR_n^v The average computation speed of the processors in R_n^v.

ΠR_n^v The total data transmission rate including the transmission rate over the links (communication bandwidth) and over the nodes (switching latency) from layer 1 to layer v in R_n^v.

πR_n^v The average data transmission rate at layer v of R_n^v.

ρR_n^v The computational type of the Rep cluster. Its value is set to 0 for a SIMD type and 1 for an MIMD type.

2.3.3 Evaluation Function

In heterogeneous computing, the structure of the underlying heterogeneous organization may be changed dynamically. Therefore, it is desirable to be able to compute an estimated total execution time for mapping a program onto the heterogeneous architecture at every step of the computation. We denote the estimated total execution time of mapping the Spec cluster S_i^u onto the Rep cluster R_j^v by $\tau(S_i^u, R_j^v)$, which includes computation time and communication time. The total computation amount of S_i^u is estimated to be $\sigma S_i^u \times \delta S_i^u$, and the total computation power of R_j^v

can be calculated as $\sigma R_j^v \times \delta R_j^v$. Therefore, the computation time for executing S_i^u on R_j^v is estimated to be $(\sigma S_i^u \times \delta S_i^u)/(\sigma R_j^v \times \delta R_j^v)$. Similarly, the total communication requirement of S_i^u is ΠS_i^u and the total communication capacity of R_j^v is ΠR_j^v, hence the estimated communication time for mapping S_i^u on R_j^v will be $\Pi S_i^u / \Pi R_j^v$. A slow-down factor, d, is defined that indicates the factor of slow down due to mismatch of the computation type between S_i^u and R_j^v. This leads to an estimated execution time in (2.1). Note that the estimated execution time does not take into consideration the memory requirements of a given problem and the memory space available in the underlying organization. This is mainly due to the fact that the model does not contain any parameters for memory size requirements and availabilities.

$$\tau(S_i^u, R_j^v) = d \times \frac{\sigma S_i^u \times \delta S_i^u}{\sigma R_j^v \times \delta R_j^v} + \frac{\Pi S_i^u}{\Pi R_j^v}, \quad d = \begin{cases} \sigma S_i^u & \text{if } \rho S_i^u = 1 \text{ and } \rho R_j^v = 0 \\ 1 & \text{otherwise} \end{cases} \tag{2.1}$$

The Cluster-M tool presented in the previous section is an implementation of this model. We will show that using the clustering algorithms presented in Section 2.4 as part of the tool, the above two set of parameters can be extracted from any given task or system graph.

2.4 Non-Uniform Clustering

In this section we first present a clustering algorithm to be used for directed task graphs independent of any system graphs and then present another one for undirected system graphs independent of any task graphs. Both algorithms are done only once for any given task or system graph and are not repeated as part of the mapping process.

2.4.1 Clustering Directed Task Graphs

A task can be represented by a directed graph $G_t(V_t, E_t)$, where $V_t = \{t_1, ..., t_M\}$ is a set of task modules to be executed and E_t is a set of edges representing the partial

orders and communication directions between task modules. A directed edge (t_i, t_j) represents that a data communication exists from module t_i to t_j and that t_i must be completed before t_j can begin, where $1 \leq i, j \leq M$. Each edge (t_i, t_j) is associated with D_{ij}, the amount of data required to be transmitted from module t_i to module t_j, where $D_{ij} \geq 1$. Each task module t_i is associated with its amount of computation A_i, that is, the number of clock cycles required to execute all the instructions of t_i on a baseline machine. Note that $A_i \geq 1$ and $D_{ij} \geq 1$ if there exits an edge (t_i, t_j), for $1 \leq i, j \leq M$. If a directed edge (t_i, t_j) exists, t_i is called a parent node (module) of t_j and t_j a child node (module) of t_i. If a node has more than one child, it is called a fork-node. If a node has more than one parent, it is called a join-node. A task graph is divided into a number of layers, so that all nodes in a layer can be executed concurrently.

A clustering algorithm called Clustering Nonuniform Directed Graphs (CNDG) is shown in detail in Figure 2.3. This nonuniform algorithm is designed as an extension to the uniform clustering algorithm presented in an earlier publication [15]. The nonuniform algorithm has been designed in such a way that it is a generalization of the uniform algorithm. For clustering nonuniform directed graphs, a quintuple of parameters (σS_m^u, δS_m^u, ΠS_m^u, πS_m^u, ρS_m^u) from the Cluster-M model described in Section 2.3 is associated with the m-th Spec cluster at layer u denoted by S_m^u. The clustering is done layer by layer. At layer 1, a node with computation amount A_i is a cluster by itself with parameters $(1, A_i, 0, 0, 0)$ for SIMD type or $(1, A_i, 0, 0, 1)$ for MIMD type. Then for other layers, the nodes are clustered as follows. If a node is a join-node, we first embed it onto one of its parent nodes that has the largest weighted edge connecting to this join-node. If multiple parent nodes have edges with the same largest weight, we randomly select one of them. When a node with a computation amount A is to be embedded to S_m^u, then these parameters are updated to σS_m^u, $\delta S_m^u + A_i$, ΠS_m^u, πS_m^u, and ρS_m^u. We then merge all its parent nodes into a

Clustering Nonuniform Directed Graphs (CNDG) Algorithm
Divide the directed graph into a number of layers
for each node at layer 1 do
    Make it into a cluster and calculate its parameters
For each of the other layers do
begin
    for all edges (t_i, t_j) do
    begin if t_i is a fork-node then
        begin Embed the child node with the largest edge weight to t_i
            if the child nodes of t_i are not in a cluster then
            begin Merge them with t_i into a cluster
                Calculate the parameters of the new cluster
            end
        end
        if t_j is a join-node then
        begin Embed the child node with the largest edge weight to t_i
            if the parent nodes of t_j are not in a cluster then
            begin Merge them with t_j into a cluster
                Calculate the parameters of the new cluster
            end
        end
    end
end

Figure 2.3 Clustering Nonuniform Directed Graphs (CNDG) algorithm.

new cluster denoted by S_1^{u+1}. This is shown in Figure 2.4, where a join-node at layer $(u+1)$ with computation amount A has n parent nodes $S_1^u, S_2^u, \cdots, S_n^u$ at layer u. The communication amount between the join-node and one of its parent nodes S_i^u is denoted by D_i, where $1 \leq i \leq n$. Also, $D_1 = \max_{1 \leq i \leq n} D_i$. The new cluster S_1^{u+1} is generated by embedding the join-node to S_1^u and merging it with all the other parent nodes. The first four parameters of S_1^{u+1} can be computed as follows.

$$\sigma S_1^{u+1} = \sum_{i=1}^{n} \sigma S_i^u \tag{2.2}$$

$$\delta S_1^{u+1} = max(\delta S_1^u + A, \delta S_2^u, \cdots, \delta S_n^u) \tag{2.3}$$

$$\Pi S_1^{u+1} = \sum_{i=1}^{n} (\Pi S_i^u + D_i) - D_1 \tag{2.4}$$

$$\pi S_1^{u+1} = \frac{\sum_{i=2}^{n} D_i}{n-1} \tag{2.5}$$

If a node is a fork-node, we will embed one of its child nodes to this fork-node. The child node is selected so that it has the largest weighted edge connecting to the fork-node. If multiple child nodes have edges with the same largest weight, we randomly select one of them. We then merge the rest of the child nodes with the fork-node into a new cluster. As shown in Figure 2.5, a fork-node S_1^u at layer u has n child nodes at layer $(u+1)$. These child nodes have computation amounts $A_1, A_2, \cdots, A_n$, and the communication amounts between the fork-node and each of them are $D_1, D_2, \cdots, D_n$, respectively. Similar to the case of join-node, $D_1 = \max_{1 \le i \le n} D_i$. Then the node with the computation amount A_1 is embedded to the fork-node before we merge the fork-node with all the other child nodes to generate the new cluster S_1^{u+1}. The first four parameters of S_1^{u+1} is then computed as follows.

$$\sigma S_1^{u+1} = \max(\sigma S_1^u, n) \tag{2.6}$$

$$\delta S_1^{u+1} = \max(\delta S_1^u + A_1, A_2, \cdots, A_n) \tag{2.7}$$

$$\Pi S_1^{u+1} = \Pi S_1^u + \sum_{i=2}^{n} D_i \tag{2.8}$$

$$\pi S_1^{u+1} = \frac{\sum_{i=2}^{n} D_i}{n-1} \tag{2.9}$$

For both fork and join nodes, the fifth parameter, ρS_m^u, is determined as follows. As an MIMD cluster is merged with an SIMD or MIMD cluster, the computation type of the new generated cluster is MIMD. When two SIMD clusters are merged then the computation type of the new cluster is decided by their computational form (addition, subtraction, multiplication, etc.). If the two SIMD clusters have exactly the same computation form, then the computational type of the new cluster is SIMD; otherwise, it is MIMD. We denote the computation form of S_m^u by $CF(S_m^u)$. Then the computational type of a new cluster S_m^u generated from embedding or merging

n clusters, $S_1^u, S_2^u, \cdots, S_n^u$, can be formulated as follows.

$$\rho S_m^u = \begin{cases} 0 & \text{if } (\rho S_i^u = 0, \text{ for all } i) \text{ and } [CF(S_1^u) = CF(S_2^u) = \cdots = CF(S_n^u)] \\ 1 & \text{otherwise} \end{cases} \tag{2.10}$$

Note that since our task graphs are independent of any system graphs (unlike [74, 59, 76]), they do not contain the information about computation time and communication delay. Therefore, we can only embed one node into another as part of clustering for reducing communication overhead. The embedding of multiple nodes onto one node is done as part of the mapping, as explained in the next section.

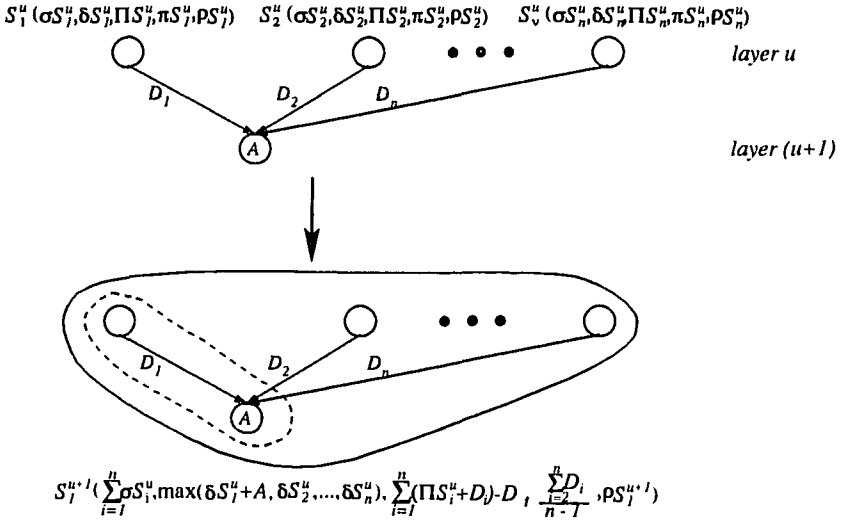


Figure 2.4 Clustering on a join-node: a general case.

The time complexity of the CNDG algorithm is bounded by the number of edges in the task graph, which is $O(|E_t|)$. For the worst case, we have an upper bound for this algorithm, that is $O(M^2)$, where M is the number of nodes. However, note that most graphs are not completely connected, therefore, in practice, the time complexity of this algorithm will be $O(M)$ if the number of edges is proportional to the number of nodes. To illustrate this algorithm, consider the task graph of seven modules and its Spec graph, as shown in Figure 2.6. Each module is labeled with its computation amount and each edge is labeled with the amount of data

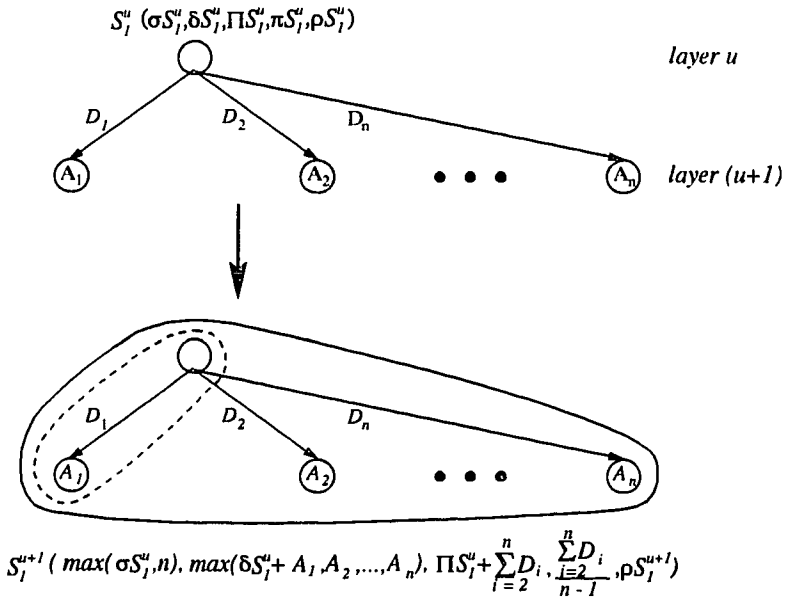


Figure 2.5 Clustering on a fork-node: a general case.

communication. The Spec graph is constructed by embedding/merging the clusters layer by layer and is a multi-layer clustered graph as shown.

2.4.2 Clustering Undirected System Graphs

A parallel system that can be modeled as an undirected system graph $G_p(V_p, E_p)$. In G_p, $V_p = \{p_1, ..., p_N\}$ is a set of processors forming the underlying architecture, while E_p is the set of edges representing the interconnection topology of the parallel system. We assume that the connections between adjacent processors are bidirectional. Therefore, an edge (p_i, p_j) represents that there is a direct connection between processor p_i and p_j. The computation speed of processor p_i is denoted by B_i, and the communication bandwidth between two processors p_i and p_j is denoted by C_{ij}. The transmission rate is a function of the communication bandwidth between p_i and p_j and the node latencies at p_i and p_j. Both the computational speeds of different processors and the transmission rates of different communication links may

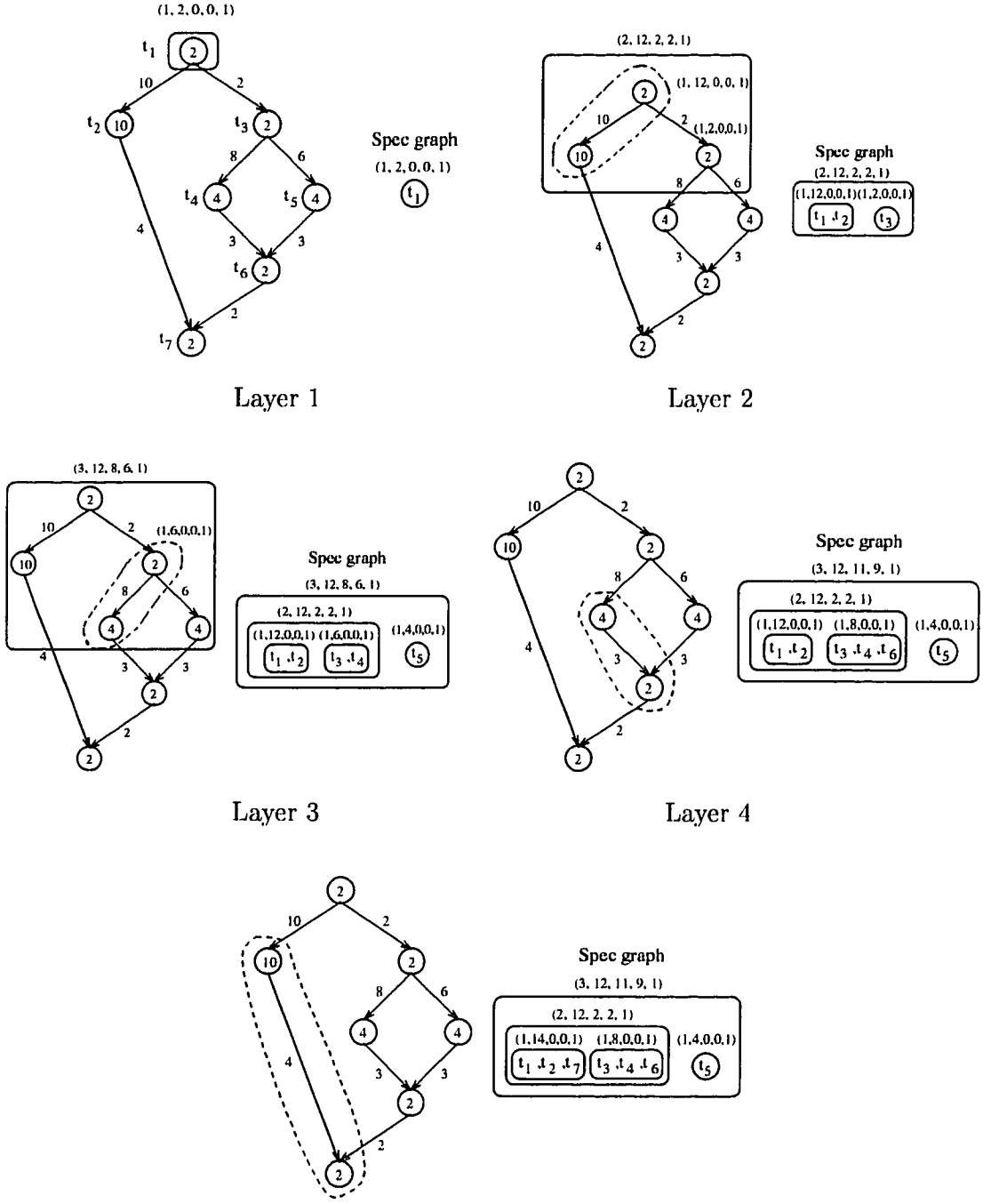


Figure 2.6 A task graph and steps for obtaining the Spec graph.

be nonuniform. This makes the Cluster-M approach more general than approaches such as PYRROS and Hypertool, which assume fully connected uniform systems.

Similar to Spec clusters, the nth Rep cluster at layer v, R_n^v, is associated with the quintuple $(\sigma R_n^v, \delta R_n^v, \Pi R_n^v, \pi R_n^v, \rho R_n^v)$ defined as part of the Cluster-M model in Section 2.3. To construct a Rep graph from an undirected system graph, initially, every node with computation speed of B_i forms a cluster by itself with parameters (1, B_i, 0, 0, 1), assuming that these nodes are all MIMD type. Then clusters that are completely connected are merged to form a new cluster, and the parameters of the new cluster are calculated, as explained below. This process is repeated until no further merging is possible. Three clusters R_x^v, R_y^v, and R_z^v are completely connected if R_x^v contains a node p_x, R_y^v contains a node p_y, and R_z^v contains a node p_z, so that nodes p_x, p_y, and p_z form a clique. This definition can be extended for N completed connected clusters. To calculate the values of the first four parameters for a new cluster, consider a new cluster R_n^{v+1}, which is generated at layer $(v+1)$ by merging N completely connected clusters $R_1^v, R_2^v, \cdots, R_N^v$ at layer v. Then the values of σR_n^{v+1} and δR_n^{v+1} can be easily computed as follows.

$$\sigma R_n^{v+1} = \sum_{i=1}^{N} \sigma R_i^v \tag{2.11}$$

$$\delta R_n^{v+1} = \frac{\sum_{i=1}^{N} \sigma R_i^v \delta R_i^v}{\sigma R_n^{v+1}} = \frac{\sum_{i=1}^{N} \sigma R_i^v \delta R_i^v}{\sum_{i=1}^{N} \sigma R_i^v} \tag{2.12}$$

We denote the transmission rate between R_i^v and R_j^v to be C_{ij}^v, which is defined as the sum of the transmission rate (as a function of communication bandwidth and switching latency) of each pair of processors (subclusters) p_i and p_j such that p_i is in R_i^v and p_j is in R_j^v, that is, $C_{ij}^v = \sum_{p_i \in R_i^v, p_j \in R_j^v} C_{ij}$. Then ΠR_n^{v+1} and πR_n^{v+1} can be calculated as follows.

$$\Pi R_n^{v+1} = \sum_{i=1}^{N} \Pi R_i^v + \sum_{i=1}^{N-1} \sum_{j=i+1}^{N} C_{ij}^v \tag{2.13}$$

$$\pi R_n^{v+1} = \frac{\sum_{i=1}^{N-1} \sum_{j=i+1}^{N} C_{ij}^v}{\frac{N(N-1)}{2}} = \frac{2(\sum_{i=1}^{N-1} \sum_{j=i+1}^{N} C_{ij}^v)}{N(N-1)} \tag{2.14}$$

The algorithm for clustering undirected graphs, called Clustering Nonuniform Undirected Graphs (CNUG)[1], is shown in Figure 2.7. Instead of using an optimal algorithm for finding cliques, we use a heuristic so that, for every cluster, we examine the set of edges connected to it in the following manner. The edges are sorted in descending order based on the value of C_{ij}. The edges are then examined one at a time from this list. If more than one of the edges have the same weight, then an arbitrary one is selected. A simple example is shown in Figure 2.8.

We now analyze the running time of this implementation. For each layer, we first sort all the edges between clusters that take $O(|E_p| \log |E_p|)$, where $|E_p|$ is the number of edges in the system graph. Then, we keep merging clusters into the next layers. Suppose at a certain layer, there are m clusters $c_1, \cdots, c_m$. The time for finding cliques among these clusters is at most $m \times m \leq N^2$, where N is the number of processors in the system graph. The most number of layers there can be is $N-1$. Therefore the total time complexity of this algorithm is $O(N(|E_p| \log |E_p| + N^2))$. Consider the worst case, where the system graph is completely connected (i.e., $|E_p| = O(N^2)$), then the time complexity of this algorithm will be $O(N^3 \log N)$. Note that most system graphs are not completed connected. Therefore, in practice the time complexity of this algorithm will be $O(N^3)$ if the number of edges is proportional to the number of nodes.

2.5 Cluster-M Mapping Algorithm

A Spec graph and a Rep graph can be generated directly from a given task graph and system graph, using the clustering algorithms presented in the previous section. Given a Spec graph and a Rep graph, this section presents an efficient mapping algorithm that produces a suboptimal matching of the two graphs in $O(MP)$ time, where $P = \max(M, N)$. Note that the mapping algorithm maps the Spec graph

[1] Pronounced "see-nudge."

Clustering Nonuniform Undirected Graphs (CNUG) Algorithm

```
for all nodes p_i do
begin Make a cluster for p_i at clustering layer 1
      Set the parameters of the cluster to be (1, B_i, 0, 0)
end
Set cluster layer to be 1
while there is at least one edge linking two clusters do
begin Sort all edges linking any two clusters
      while sorted edge list is not empty, do
      begin Take the first edge (c_i, c_j) from sorted edge list
            Delete the edge from the list
            Merge c_i and c_j into cluster c' at next layer
            Calculate the parameters of c'
            Delete clusters c_i and c_j from current layer
            for each edge (c_x, c_y) in sorted edge list
            if c_x is a sub-cluster of c' and
            c_y is not a sub-cluster of any cluster and
            c_y is connected to all other sub-clusters of c', then
            begin Merge c_y into c'
                  Recalculate the parameters of c'
                  Delete (c_x, c_y) from edge list
            end
            else if c_x and c_y are sub-clusters of
            two different clusters at next layer, then
            begin Add the weight of (c_x, c_y) to
                  the edge between the two super-clusters
                  Delete (c_x, c_y) from edge list
            end
      end
      Increment clustering layer by 1
end
```

Figure 2.7 Clustering Nonuniform Undirected Graphs (CNUG) algorithm.

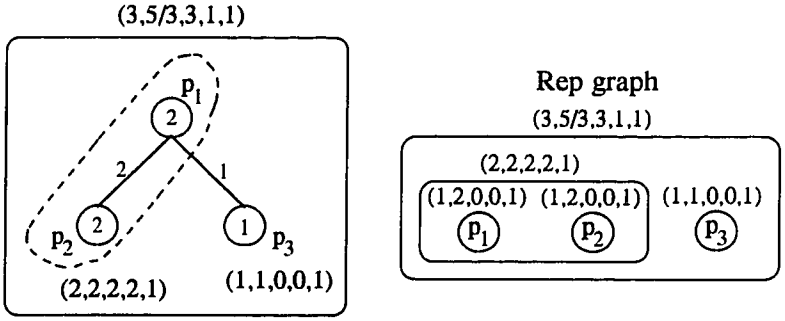


Figure 2.8 A nonuniform system graph and its Rep graph.

one layer at a time as explained in Section 2.2.2. Every layer of the Spec graph represents a computational step in which a number of concurrent computations are represented by a number of Spec clusters. These clusters are formed by tracing the data dependency of other subcomputations from a previous step. We are interested in mapping the Spec clusters at each layer to the appropriate Rep clusters. In the following, we first present a set of preliminaries and then give a high-level description of the mapping algorithm. In Section 2.5.3, a few examples are given to illustrate the mapping algorithm.

2.5.1 Preliminaries

We first define the mapping function $f_m : V_t \xrightarrow{\text{onto}} V_p$. Following the precedence constraints and the computation and communication requirements of the original task graph, a schedule can be obtained by assigning each task module t_i to the processor $f_m(t_i)$. We assume that the communication time for a task graph edge (t_i, t_j) is equal to $\sum_{(p_x,p_y)\in \text{path}(f_m(t_i),f_m(t_j))} \frac{D_{t_i t_j}}{C_{xy}}$, where $\text{path}(p_i, p_j)$ is the shortest path between processor p_i and p_j.

A schedule can be illustrated with a Gantt chart that consists of a list of all processors and a list of all task modules allocated to each of the processors ordered by their execution time [23]. We define the total execution time of a schedule, T_m, to be the latest finishing computation time of the last scheduled task module on any

processor. Obviously, T_m is equal to the total execution time of a given task on a given system. As we consider the shortest execution time of a given task on a system to be the ultimate goal in scheduling, we take T_m as our measure of quality to scale how good a mapping is.

2.5.2 The Algorithm

A detailed description of the mapping algorithm is presented in Figure 2.9. In the following, we give an overview of the algorithm. The mapping is done recursively at each clustering layer, where we try to find the best matching between Spec clusters and Rep clusters. Assume that at a certain step of mapping, m Spec clusters of layer u, $S_1^u, S_2^u, \cdots, S_m^u$, are to be mapped onto n Rep clusters of layer v, $R_1^v, R_2^v, \cdots, R_n^v$. We denote the execution time of mapping the Spec cluster S_i^u onto the Rep cluster R_j^v by $\tau(S_i^u, R_j^v)$ expressed by (2.1). Then the mapping process at each layer can be viewed as an optimization problem, as follows.

$$\min \sum_{i=1}^{m} \tau(S_i^u, f_m(S_i^u)) \tag{2.15}$$

The time complexity of finding an optimal solution to the above formula can be costly. Therefore, we propose the following greedy algorithm for finding a near-optimal solution to the formula for each layer. In this greedy algorithm, we assume that all the computations are MIMD. Therefore, we only deal with four of the five parameters in the process. The greedy algorithm continues as follows. First, the Spec and Rep clusters are sorted in descending order with respect to the order of the four parameters (σ, δ, Π, π). For example, Spec clusters with larger sizes are sorted before those with smaller sizes, and for Spec clusters with the same size, those with larger amount of sequential computation are sorted first.

Secondly, we compute a reduction factor denoted by $f_{(u,v)}$, which is the ratio of the total size of the Rep clusters over the total size of the Spec clusters and is used to estimate how many computation nodes to share a processor. This is essential for

mapping task graphs of size M onto system graphs of size N, where $M > N$. The value of $f_{(u,v)}$ is computed as:

$$f_{(u,v)} = \frac{\sum_{j=1}^{n} \sigma R_j^v}{\sum_{i=1}^{m} \sigma S_i^u} \tag{2.16}$$

Third, we map each of the Spec clusters S_i^u, $1 \leq i \leq m$, as follows. We first search for a Rep cluster R_j^v, $1 \leq j \leq n$, with the best matched size, that is, closest to $f_{(u,v)} \times \sigma S_i^u$. Therefore, we try to minimize the function in Equation (2.17). If multiple Rep clusters with the matching size are found, we select the one with the minimum estimated execution time. If no Rep cluster with a matching size can be found for a Spec cluster, we either merge or split (unmerge) Rep clusters until a matching Rep cluster is found.

$$|f_m| = \sum_{i=1}^{m} |f_{(u,v)} \times \sigma S_i^u - \sigma[f_m(S_i^u)]| \tag{2.17}$$

Finally, for every matched pair of the Spec and Rep clusters, we do the following to embed communication intensive nodes together. This is similar to the clustering process in [74, 59, 76]. However, in this chapter, we only do it in the mapping step so that the clustering of the task graph is kept independent of the system graph, as described in the previous section. Assume that a Spec cluster S_i^u having k subclusters, $S_1^{u-1}, S_2^{u-1}, \cdots, S_k^{u-1}$, is mapped to a Rep cluster R_j^v. If the communication overhead for processing the subclusters in parallel is greater than the computation overhead for processing the subclusters sequentially, then we embed all subclusters into one subcluster having the largest size so that they will be executed sequentially. We then calculate the parameter quadruple for the new cluster. In Inequality (2.18), $\pi S_i^u / \pi R_j^v$ is the communication time if the subclusters are executed in parallel and

$$\frac{1}{f_{(u,v)}} \times \frac{\min(\sigma S_1^{u-1} \delta S_1^{u-1}, \sigma S_2^{u-1} \delta S_2^{u-1}, \cdots, \sigma S_k^{u-1} \delta S_k^{u-1})}{\delta R_j^v}$$

is the computation time for executing the subclusters sequentially on R_j^v. The embedded cluster is inserted back in the proper position in the sorted list of Spec

clusters for mapping, and the matching process is repeated for the remaining Spec clusters in the list. If no embedding is necessary, then the mapping of this Spec cluster onto a Rep cluster is done for this layer, and, therefore, this Spec cluster is removed from the list.

$$\frac{\pi S_i^u}{\pi R_j^v} > \frac{1}{f_{(u,v)}} \times \frac{\min(\sigma S_1^{u-1}\delta S_1^{u-1}, \sigma S_2^{u-1}\delta S_2^{u-1}, \cdots, \sigma S_k^{u-1}\delta S_k^{u-1})}{\delta R_j^v} \tag{2.18}$$

In the above mapping algorithm, the worst case of the time complexity of the mapping algorithm at layer i occurs in one of the following two cases. In case 1, for each Spec cluster, all the remaining Rep clusters have the matching size, thus (2.1) is used to select the best Rep cluster. In case 2, for each Spec cluster, no Rep cluster of matching size is found, thus Rep clusters are merged or split recursively until a Rep cluster of matching size is obtained. Suppose the number of Spec clusters at layer i is K_i. In both cases described above, or in any combination of the two cases, it takes $O(K_iN)$ time to find the best matches for all K_i Spec clusters, as the total number of clusters in the Rep graph is $O(N)$, where N is the number of processors. For each pair of matching Spec and Rep clusters, if Inequality (2.18) is satisfied, then an extra $O(M)$ time for embedding will be needed. The total number of Spec clusters is $O(M)$, that is, $\sum_i K_i = O(M)$, where M is the number of nodes in original task graph. Therefore, the total time complexity of this mapping algorithm is $\sum_i(K_iN + M) = O(MN) + O(M^2) = O(MP)$, where $P = \max(M, N)$.

2.5.3 Mapping Examples

In Section 2.4, we constructed a Spec graph and a Rep graph from the original task graph and system graph, as shown in Figures 2.6 and 2.8. Figure 2.10 shows the snapshot of the mapping process. Figure 2.11 shows the final schedule obtained from the above mapping by following the data and operational precedence of the task graph. As shown in the Gantt chart, $T_m = 10$.

Mapping Algorithm

for each layer of Spec graph do
  Sort all Spec clusters at top layer in descending order of σS_i^u, δS_i^u, ΠS_i^u, and πS_i^u.
  Sort all Rep clusters at top layer in descending order of σR_j^v δR_j^v, ΠR_j^v, and πR_j^v.
  Calculate $f_{(u,v)}$, if $f_{(u,v)} > 1$, let $f_{(u,v)} = 1$.
  Calculate the required size of the Rep cluster matching S_i^u to be $f_{(u,v)} \times \sigma S_i^u$
  for each Spec cluster at top layer sorted list, do
    if the cluster has only one sub-cluster, then
      Go to a lower layer where there are multiple or no sub-clusters
    if at least a Rep cluster of required size is found, then
      Select the Rep cluster of required size with minimum
      estimated execution time according to Equation (2.1)
      Match the Spec cluster to the Rep cluster
      Delete the Spec and Rep clusters from Spec and Rep lists
  for each unmatched Spec cluster, do
    if the size of the first Rep cluster > the required size, then
      Split the Rep cluster into two parts with one part of the required size
      Match the Spec cluster to this part
      Insert the other part to proper position of the sorted Rep cluster list
      Merge Rep clusters until the sum of sizes $\geq$ the required size
      if = then Match the Spec cluster to the merged Rep cluster
      else
        Split the merged Rep cluster into two parts with one of required size
        match the Spec cluster to this part
        Insert the other part to the sorted Rep list
  for each matching pair of Spec cluster and Rep cluster, do
    if the Rep cluster contains only one processor, then
      Map all the modules in the Spec cluster to the processor
    else if Inequality (2.18) is satisfied, then
        Select the sub-cluster of the Spec cluster with the largest size
        Embed the nodes of other sub-clusters
        to the connected nodes of the selected sub-cluster
        Calculate the parameters for the new cluster
        Insert it into the sorted Spec cluster list
      else
        Delete the Spec and the Rep clusters from the cluster lists
        Go to the sub-clusters of the Spec and Rep cluster
        (thus they are pushed to top layer)
        Call the same mapping algorithm for these clusters

Figure 2.9 Mapping algorithm.

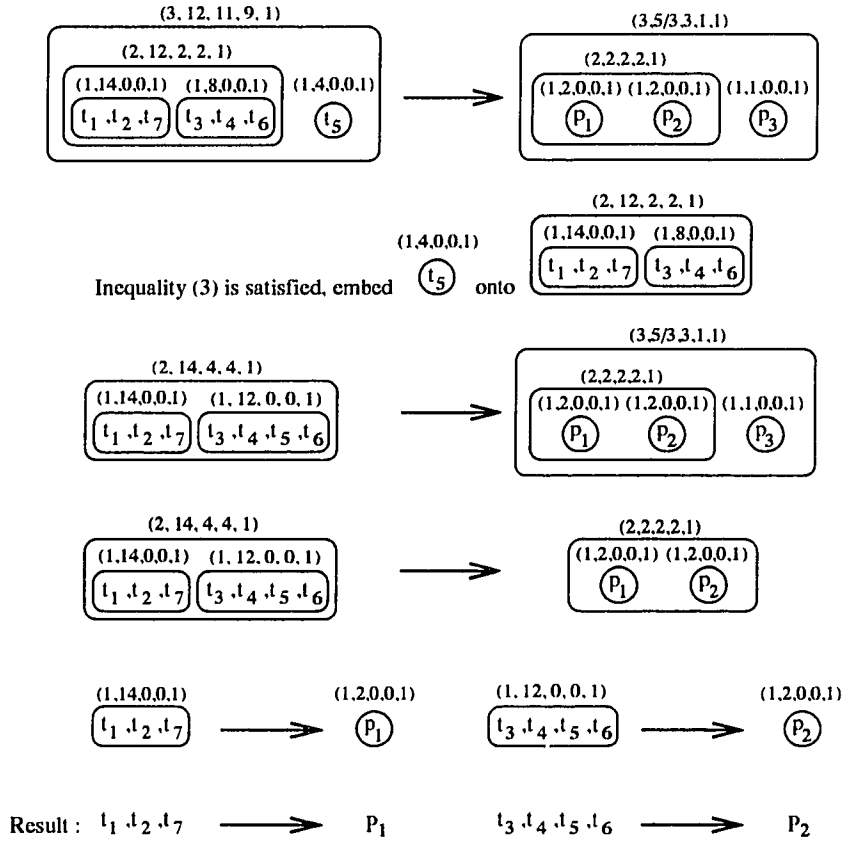


Figure 2.10 A mapping example.

To show that the same task graph can be mapped onto various system graphs, three different system graphs are chosen and shown in Figure 2.12. Figure 2.12(a) is the same task graph as shown in Figure 2.6. Figure 2.12(b) shows a uniform, fully connected system graph and its clustering. The computation speed of each processor and communication bandwidth of each communication link are equal to 2. The result of Cluster-M mapping onto this graph is shown in Figure 2.12(c). In Figure 2.12(d), the system is fully connected with computation speed of 1 at each processor, but the

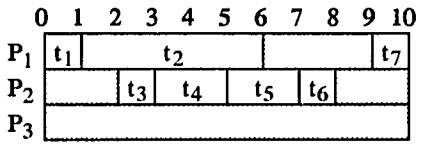
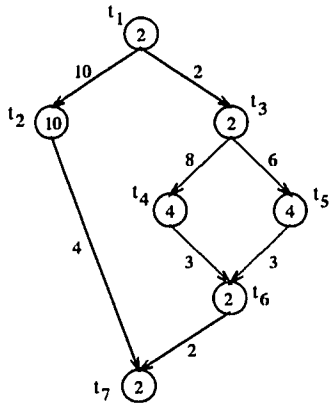
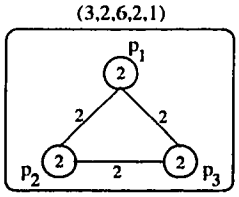


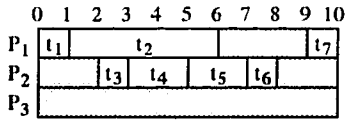
Figure 2.11 Gantt chart of the obtained schedule.



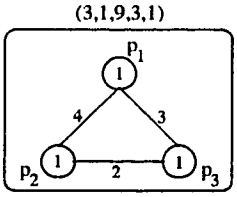
(a) Task graph



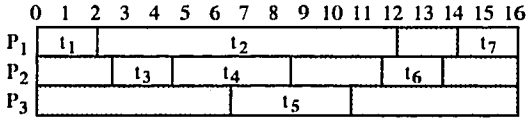
(b) A uniform system graph



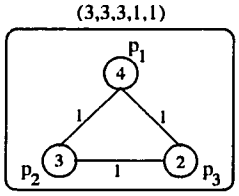
(c) Mapping result on (b)



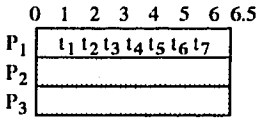
(d) A non-uniform system graph



(e) Mapping result on (d)



(f) A different non-uniform system graph



(g) Mapping result on (f)

Figure 2.12 Mappings on different system graphs.

communication bandwidths are nonuniform. In this case, the Cluster-M algorithm distributes the task modules to all three processors, as shown in Figure 2.12(e), to utilize the relatively high communication bandwidth available. If the system is fully connected with uniform communication bandwidth and nonuniform computation speeds as shown in Figure 2.12(f), however, Cluster-M mapping algorithm maps all the task modules onto the processor with the highest speed to avoid the relatively expensive communication cost. This is shown in Figure 2.12(g).

```
      SUBROUTINE KJI(A,LDA,N)
C
C     SAXPY
C     FORM KJI-SAXPY
C
      REAL A(LDA,N)
      DO 40 K=1,N-1
         DO 10 I=K+1,N
            A(I,K)=-A(I,K)/A(K,K)
 10      CONTINUE
         DO 30 J=K+1,N
            DO 20 I=K+1,N
               A(I,J)=A(I,J)+A(I,K)*A(K,J)
 20         CONTINUE
 30      CONTINUE
 40   CONTINUE
      RETURN
```

Figure 2.13 The Fortran code of the Gaussian elimination on a $N \times N$ matrix.

Finally, we give an example for mapping a real application task. We choose the Gaussian elimination algorithm used in LINPACK. The FORTRAN code is given in Figure 2.13. Suppose using a baseline computer, it takes one clock cycle to perform an addition or subtraction, and it takes two clock cycles to do a multiplication or division of two real numbers. Also, we assume the communication amount on an edge to be the number of real numbers that need to be sent. A task graph for computing the Gaussian elimination of a 5×5 matrix is shown in Figure 2.14(a). In each task module T_j^k, column j is modified by using column k. Suppose that the system running this task contains only two workstations p_1 and p_2. Workstations p_1 and p_2 have speeds of 2 and 1.6, respectively, and are connected with a link of bandwidth 1.

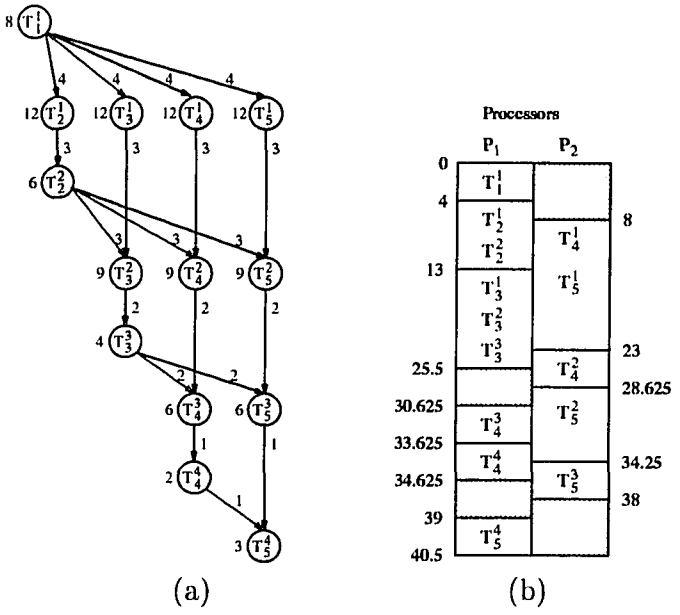


Figure 2.14 (a) The task graph and (b) the mapping result of the Gaussian elimination on a 5 × 5 matrix.

The mapping result using our technique is illustrated in Figure 2.14(b). For a more practical illustration of our algorithms, we performed the following two experiments. Tables 2.1 - 2.3 shows the mapping results of doing Gaussian eliminations on various sizes of matrices using different two-processor systems. The speeds of the processors are 2 and 1.6, 1 and 1, and 0.8 and 0.7, respectively, while the communication bandwidth is assumed to be 1. To illustrate the efficiency of the Cluster-M mapping, we experimented with mapping a 500 × 500 Gaussian elimination problem on 1 to 10 uniformly weighted and fully connected processors. As shown in Figure 2.15, near-optimal speedups have been obtained. These experiments were done manually as we do not yet have an interface which automatically generates a task graph from a given program. However, given a task or a system graph, we can automatically generate a clustered graph, and then run the mapping code for allocating and scheduling the task graph onto the system graph. In the next section we show the mapping generated using the Cluster-M code on randomly generated task and system graphs.

Table 2.1 Gaussian elimination mapping results using two processors with speed 2 and 1.6.

Width of Matrix	100	200	300	400	500
Speedup	1.196	1.180	1.175	1.171	1.170

Table 2.2 Gaussian elimination mapping results using two processors with speed 1 and 1.

Width of Matrix	100	200	300	400	500
Speedup	1.494	1.474	1.468	1.465	1.463

Table 2.3 Gaussian elimination mapping results using two processors with speed 0.8 and 0.7.

Width of Matrix	100	200	300	400	500
Speedup	1.308	1.290	1.285	1.281	1.280

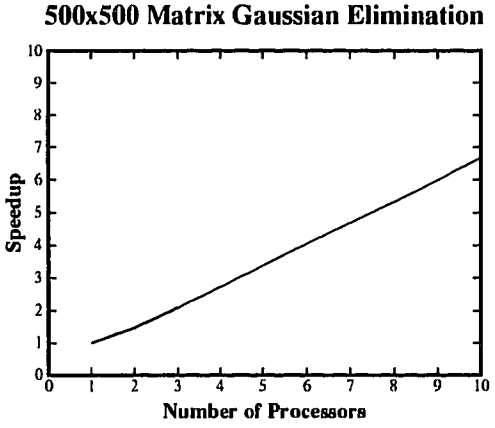


Figure 2.15 More Gaussian elimination mapping results.

2.6 Comparison Results

In this section, we first present our comparison results for the scheduling problem and then for the allocation problem. The following five criteria are used for evaluating the performance of the algorithms examined: (1) the total time complexity of executing the mapping algorithm, T_c; (2) the total execution time of the generated mappings, T_m; (3) the speedup $S_m = T_s/T_m$, where T_s is the sequential execution time of the task; (4) efficiency $\eta = S_m/N_m$, where N_m is the number of processors used; and (5) the actual time of running the mapping algorithm on a certain computer, T_c.

2.6.1 Scheduling

In this section, we present a set of experimental results we have obtained in comparing our algorithm with other leading scheduling techniques. The comparisons presented in this section are classified into two categories: (1) mapping arbitrary nonuniform task graphs onto arbitrary nonuniform system graphs, and (2) mapping arbitrary nonuniform task graphs onto uniform fully connected system graphs. We first present the comparison for the first category and then the second one.

2.6.1.1 Mapping Nonuniform Tasks onto Nonuniform Systems The mapping techniques in this category include El-Rewini and Lewis' mapping heuristic (MH) [22] and Lo's Max Flow/Min Cut (MFMC) algorithm [44]. To the best of our knowledge, they are the only known efficient mapping techniques that can map arbitrary nonuniform task graphs onto arbitrary nonuniform system graphs in polynomial time. The experimental results shown in this section are obtained by running a set of simulations on a SUN SPARCstation 20 workstation, and all running times are measured in second on this machine. The nonuniform task graphs are randomly generated. We map these task graphs onto four different nonuniform

systems[2]: (1) a randomly generated system graph with 100 nodes, where the computation speed of the nodes and the communication bandwidth of the edges range from 1 to 5, (2) a randomly generated system graph with five nodes, where the computation speed of the nodes and the communication bandwidth of the edges range from 1 to 5, (3) a completely connected system graph with four nodes as shown in Figure 2.16, and (4) a hypercube with eight nodes as shown in Figure 2.17.

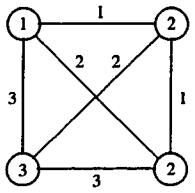


Figure 2.16 System (2): A completedly connected system.

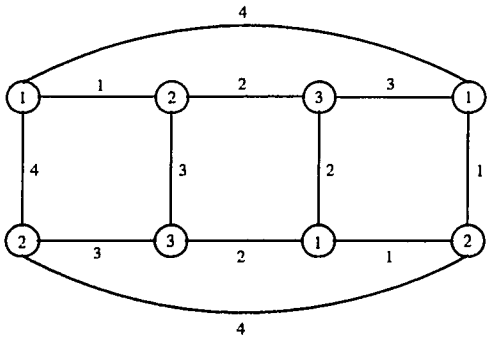


Figure 2.17 System (3): A hypercube system.

Comparison with El-Rewini and Lewis' MH We first compare our algorithm with El-Rewini and Lewis' mapping heuristic (MH) algorithm. MH is an improved list scheduling algorithm. The time complexity of MH is $O(M^2N^3)$, while ours has an $O(MN)$ time complexity. In Table 2.4, comparison results are shown for mapping nonuniform random task graphs ranging from 100 to 1000 nodes onto the random system graph of size 100. The running time of MH grows significantly when the

[2]For comparing against MFMC, we use three system configurations, system (2)-(4). The time complexity of MFMC in practice is too high and for the first system configuration, each experiment takes several days. For more detail, see Section 2.6.1.2.

size of task graph grows. The running time of Cluster-M remains stable. Tables 2.5, 2.6, and 2.7 shows the comparison results obtained on system (2), (3), and (4), respectively. In these three tables, the size of randomly generated task graphs ranges from 10 to 100 nodes. In most cases, Cluster-M obtains better speedup than MH. But in all cases Cluster-M has a significantly lower time complexity. For example, for a random nonuniform task graph of size 1000, and a random nonuniform system graph of size 100, Cluster-M generates a mapping result with the speedup of 3.49 in 0.01 second, while MH produces one with the speedup of 2.73 but in 10753.4 seconds (i.e., Cluster-M is faster by a factor of nearly 1,000,000). Theoretically, Cluster-M is faster by a factor of $O(MN^2)$.

Table 2.4 Comparison of Cluster-M and MH on system (1).

Size of Random Graph	T_s	Cluster-M $[O(MN)]$			MH $[O(M^2N^3)]$		
		T_m	S_m	T_c	T_m	S_m	T_c
100	286	88.80	3.22	0.01	95.80	2.99	128.4
200	630	133.20	4.73	0.01	231.82	2.72	425.9
300	855	345.55	2.47	0.01	240.25	3.56	971.3
400	1162	478.40	2.43	0.01	496.30	2.34	1725.0
500	1514	550.80	2.75	0.01	458.07	3.31	2768.6
600	1793	358.20	5.01	0.01	599.07	3.00	3954.3
700	2075	690.85	3.00	0.01	685.57	3.03	5348.3
800	2376	474.00	5.01	0.01	967.57	2.46	7026.5
900	2653	1113.80	2.38	0.01	1117.67	2.37	8812.2
1000	2966	850.15	3.49	0.01	1087.08	2.73	10753.4

Comparison with Lo's Max Flow/Min Cut Lo's algorithm is based on Stone's work [63], where the mapping problem is transferred into a network flow model and is solved using a Max Flow/Min Cut algorithm. Stone's model provides an optimal solution for two-processor problem only. Lo [44] extended Stone's work to find a suboptimal solution of the mapping problem for general distributed (nonuniform) systems. Lo's algorithm is a heuristic which combines recursive invocation of Max-

Table 2.5 Comparison of Cluster-M and MH on system (2).

Size of Random Graph	T_s	Cluster-M $[O(MN)]$			MH $[O(M^2N^3)]$		
		T_m	S_m	T_c	T_m	S_m	T_c
10	27	7.93	3.40	0.01	11.13	2.43	0.1
20	64	19.00	3.37	0.01	26.33	2.43	0.1
30	73	20.65	3.54	0.01	31.10	2.35	0.2
40	112	23.15	4.84	0.01	29.97	3.74	0.3
50	155	35.57	4.36	0.01	50.93	3.04	0.4
60	183	46.27	3.96	0.01	44.23	4.14	0.6
70	217	86.60	2.51	0.01	55.03	3.94	0.8
80	237	92.33	2.57	0.01	94.17	2.52	1.0
90	260	88.45	2.94	0.01	101.95	2.55	1.3
100	280	75.57	3.71	0.01	93.90	2.98	1.5

Table 2.6 Comparison of Cluster-M and MH on system (3).

Size of Random Graph	T_s	Cluster-M $[O(MN)]$			MH $[O(M^2N^3)]$		
		T_m	S_m	T_c	T_m	S_m	T_c
10	27	9.00	3.00	0.01	17.33	1.56	0.1
20	64	19.00	3.37	0.01	33.83	1.89	0.1
30	73	30.67	2.38	0.01	38.17	1.91	0.2
40	112	47.33	2.37	0.01	43.83	2.56	0.3
50	155	78.17	1.98	0.01	64.67	2.40	0.3
60	183	53.33	3.43	0.01	82.17	2.23	0.6
70	217	78.33	2.77	0.01	107.17	2.02	0.7
80	237	80.67	2.94	0.01	127.17	1.86	0.9
90	260	117.17	2.22	0.01	157.67	1.65	1.2
100	280	109.00	2.57	0.01	137.83	2.03	1.3

Table 2.7 Comparison of Cluster-M and MH on system (4).

Size of Random Graph	T_s	Cluster-M $[O(MN)]$			MH $[O(M^2N^3)]$		
		T_m	S_m	T_c	T_m	S_m	T_c
10	27	9.83	2.75	0.01	17.92	1.51	0.1
20	64	19.00	3.37	0.01	44.83	1.43	0.1
30	73	35.58	2.05	0.01	54.25	1.35	0.3
40	112	47.33	2.37	0.01	42.92	2.61	0.4
50	155	58.17	2.66	0.01	91.58	1.69	0.7
60	183	58.80	3.13	0.01	87.83	2.08	0.9
70	217	91.83	2.36	0.01	93.00	2.33	1.2
80	237	96.67	2.45	0.01	150.25	1.58	1.6
90	260	162.58	1.60	0.01	158.83	1.64	1.8
100	280	122.42	2.29	0.01	151.25	1.85	2.2

Flow/Min-Cut algorithms with a greedy-type algorithm. The time complexity of Lo's algorithm is $O(M^4N \log M)$. Tables 2.8, 2.9, and 2.10 shows the comparison results obtained on system (2), (3), and (4), respectively. In addition to MFMC, the simulations results using MH on these task graphs are also integrated in these tables. We only compare small task graphs here since it takes days for MFMC to run larger task graphs. As shown, Cluster-M produces similarly good results but in significantly less amount of time.

Table 2.8 Comparison of Cluster-M, MFMC, and MH on system (2).

Size of Graph	T_s	Cluster-M $[O(MN)]$			MFMC $[O(M^4N \log M)]$			MH $[O(M^2N^3)]$		
		T_m	S_m	T_c	T_m	S_m	T_c	T_m	S_m	T_c
10	27	7.93	3.40	0.01	8.10	3.33	0.8	11.13	2.43	0.1
12	33	8.23	4.00	0.01	16.85	1.96	4.1	9.03	3.65	0.1
14	45	8.20	5.49	0.01	18.25	2.47	23.9	16.87	2.67	0.1
16	46	12.50	3.68	0.01	23.70	1.94	109.1	14.05	3.27	0.1
18	54	20.33	2.66	0.01	27.90	1.94	556.3	19.98	2.70	0.1
20	64	19.00	3.37	0.01	34.70	1.84	2762.3	26.33	2.43	0.1
22	60	23.40	2.56	0.01	33.20	1.80	13430.0	28.29	2.12	0.1
24	86	16.00	5.38	0.01	39.65	2.17	21323.0	32.75	2.63	0.1

Table 2.9 Comparison of Cluster-M, MFMC, and MH on system (3).

Size of Graph	T_s	Cluster-M $[O(MN)]$			MFMC $[O(M^4N \log M)]$			MH $[O(M^2N^3)]$		
		T_m	S_m	T_c	T_m	S_m	T_c	T_m	S_m	T_c
10	27	9.00	3.00	0.01	15.33	1.76	0.8	17.33	1.56	0.1
12	33	13.50	2.44	0.01	17.83	1.85	3.7	17.00	1.94	0.1
14	45	13.67	3.29	0.01	19.00	2.37	21.8	20.67	2.18	0.1
16	46	21.00	2.19	0.01	22.50	2.04	99.6	20.50	2.24	0.1
18	54	19.33	2.79	0.01	26.83	2.01	503.8	32.00	1.69	0.1
20	64	19.00	3.37	0.01	31.17	2.05	2504.8	33.83	1.89	0.1
22	60	24.50	2.45	0.01	35.83	1.67	13445.3	39.17	1.53	0.1
24	86	26.67	3.23	0.01	39.83	2.16	15225.2	48.17	1.79	0.1

Table 2.10 Comparison of Cluster-M, MFMC, and MH on system (4).

Size of Graph	T_s	Cluster-M $[O(MN)]$			MFMC $[O(M^4N \log M)]$			MH $[O(M^2N^3)]$		
		T_m	S_m	T_c	T_m	S_m	T_c	T_m	S_m	T_c
10	27	9.83	2.75	0.01	18.66	1.45	1.1	17.92	1.51	0.1
12	33	21.33	1.54	0.01	19.33	1.71	5.3	17.08	1.93	0.1
14	45	13.67	3.29	0.01	39.00	1.15	29.3	16.17	2.78	0.1
16	46	21.00	2.19	0.01	45.83	1.00	141.2	25.83	1.78	0.1
18	54	19.33	2.79	0.01	29.50	1.83	715.4	33.58	1.61	0.1
20	64	19.00	3.37	0.01	60.17	1.06	3579.5	44.83	1.43	0.1
22	60	26.00	2.31	0.01	40.83	1.47	17298.8	51.00	1.18	0.2
24	86	26.67	3.23	0.01	71.83	1.20	30081.7	41.17	2.09	0.2

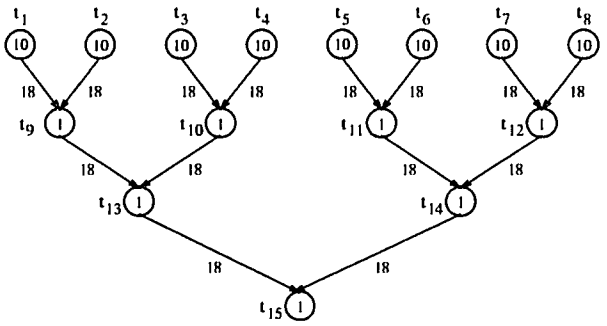
2.6.1.2 Mapping Nonuniform Tasks onto Uniform Systems The mapping techniques in this category include McCreary-Gill's Clan [48], Sarkar's Edge-Zeroing clustering [59], Wu-Gajski's MCP [74], and Yang-Gerasoulis' DSC [76]. These algorithms have proven to be very effective and efficient in mapping arbitrary and nonuniform directed tasks but work only for uniform and fully connected systems. Similar to our algorithm, these algorithms also cluster the task graphs before the mapping. However, they all assume that the target systems are fully connected with unbounded number of uniform processors and communication links. If the number of processors is bounded and smaller than the number of obtained clusters of task

modules, some clusters will be merged until the number of clusters is no less than the number of processors. The examples selected here are not designed by us, rather are those presented and studied by the authors of the papers reporting the leading techniques.

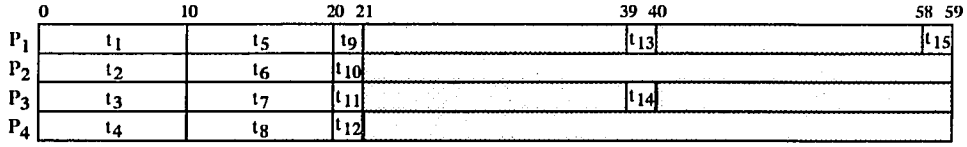
Comparison with McCreary-Gill's Clan We compare Cluster-M with McCreary-Gill's Clan algorithm, which finds suitably sized grain (cluster) of task modules to be assigned to the same processor before scheduling the tasks [48]. A clan is a set of nodes X of the directed task graph G_t if and only if for all $t_x, t_y \in X$ and all $t_z \in G_t - X$ such that t_z is a parent node of t_x if and only if t_z is a parent node of t_y; or t_z is a child node of t_x if and only if t_z is a child node of t_y. Informally, a clan is a subset of nodes where every element outside the set is related in the same way to each member in the set. An $O(M^3)$ parsing algorithm has been proposed that decomposes a task graph into clans. In McCreary-Gill's algorithm, it is also assumed that the underlying system is fully connected and all the processors and communication links are uniform ($S_i = 1$, $R_{ij} = 1$, for all i, j). Using McCreary-Gill's algorithm, the following task modules of the task graph shown in Figure 2.18(a) are clustered together and are assigned to the processors of a fully connected four processor system:

p_1: 1, 2, 9
p_2: 3, 4, 10
p_3: 5, 6, 11
p_4: 7, 8, 12

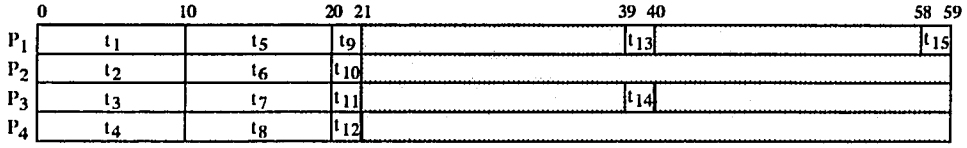
As task module 13 receives data from 9 and 10, it is assigned to p_1. Similarly, 14 is assigned to p_2 and 15 is assigned to p_1. The schedule resulting from this assignment appears in Figure 2.18(c). Even though our clustering and mapping algorithms are different and more generic than Clan, we have obtain similar results, as shown in Figure 2.18(b).



(a) Task graph



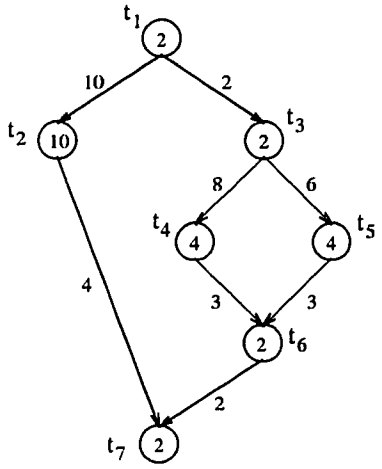
(b) Clan mapping, $T_c = O(M^3)$, $T_m = 59$, $N_m = 4$



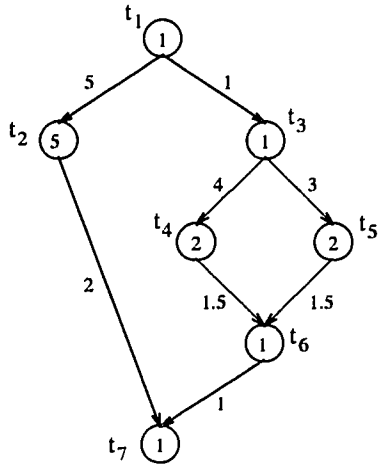
(c) Cluster-M mapping, $T_c = O(MP)$, $T_m = 59$, $N_m = 4$

Figure 2.18 Comparison example with Clan.

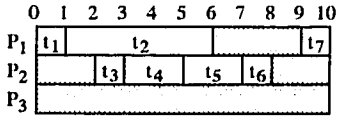
Comparison with Wu-Gajski's MCP The modified critical path (MCP) algorithm [74] is based on critical path introduced by Hu [37]. A critical path in a directed acyclic graph (DAG) is a path of greatest weight from a source node to a sink node, including the weights of all the nodes and edges along this path. The critical paths can be shortened by removing communication weights (zeroing edges) and embedding the nodes on the path. MCP assumes that the weights of task nodes and edges are the actual computation and communication times. Therefore, given the same task graph as shown in Figure 2.6 and the system graph as shown in Figure 2.12(b), a transformed task graph incorporating the information about the system graph has to be generated first, as shown in Figure 2.19(b). The mapping results by our technique and MCP are shown in Figure 2.19(c) and (d), respectively. We have



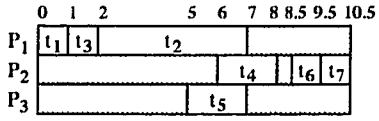
(a) The original task graph



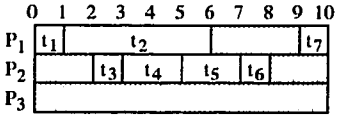
(b) The transformed task graph



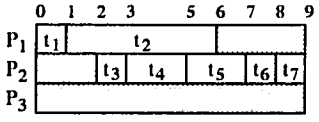
(c) Cluster-M, $T_c = O(MP)$
$T_m = 10$, $N_m = 2$



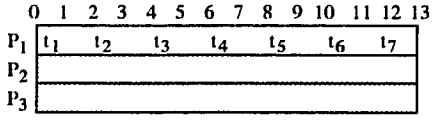
(d) MCP, $T_c = O(M^2 \log M)$
$T_m = 10.5$, $N_m = 3$



(e) Sarkar, $T_c = O(|E_t|(M + |E_t|))$
$T_m = 10$, $N_m = 2$



(f) DSC, $T_c = O((|E_t| + M) \log M)$
$T_m = 9$, $N_m = 2$



(g) Clan, $T_c = O(M^3), T_m = 13$, $N_m = 1$

Figure 2.19 Comparison example with MCP, Sarkar, DSC and Clan.

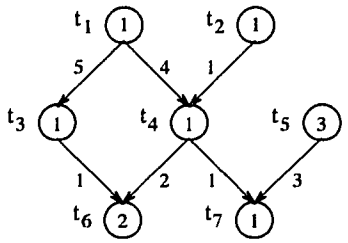
obtained a mapping with $T_m = 10$, while their $T_m = 10.5$. The time complexity of MCP is $O(M^2 \log M)$.

Comparison with Sarkar's Edge-Zeroing Algorithm The basic idea of Sarkar's Edge-Zeroing algorithm is to repetitively zero the highest weighted edge if it does not increase the estimated T_m, until all the edges have been examined. Its time complexity is $O(|E_t|(M+|E_t|))$, where $|E_t|$ is the number of edges in the task graph. Figure 2.19(e) shows the mapping result obtained by the edge-zeroing algorithm on the same example used for MCP in the last section. This result matches ours.

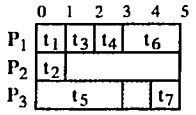
Comparison with Yang-Gerasoulis' DSC Yang-Gerasoulis' dominant sequence clustering (DSC) algorithm [76] is also based on critical path and edge zeroing, and it incorporates several other heuristics for better clustering. DSC can find optimal schedules for some special DAGs such as fork and join. However, the task graphs considered in DSC are not machine-independent and similar to the above three techniques, it cannot map to nonuniform systems such as those shown in Figure 2.12(d) and (f). The time complexity of DSC is $O((|E_t| + M) \log M)$, where $|E_t|$ is the number of edges in the task graph.

Figure 2.19(f) shows the mapping result obtained by DSC for the same example studied in comparison with MCP and Sarkar's algorithms. Among the results for this example, the DSC algorithm produces the best mapping results but does not have the lowest time complexity. In the following, we show several more comparisons with DSC. These examples are taken from [76]. Figure 2.20 and 2.21 show the mapping of two task graphs onto an unbounded number of identical processors fully connected by identical communication links. The task graph in Figure 2.22 was taken from an example studied by El-Rewini and Lewis's $O(M^2N^3)$ MH algorithm [22]. It is to be mapped onto a eight-processor hypercube with unit computation speed and communication bandwidth. The mapping by MH has $T_m = 26$ and $N_m = 7$. An optimal mapping using eight processors and having $T_m = 25$ is given in [15]. (In

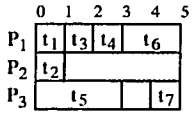
[15], graphs with uniform edges were considered.) The mapping results using our technique and DSC are illustrated in Figure 2.22(b) and (c). If a four-processor hypercube is used, DSC's and our mappings of the same task graph are shown in Figure 2.22(d) and (e).



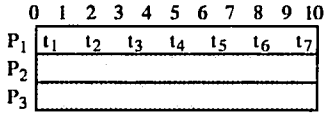
(a) Task graph



(b) Cluster-M, $T_m = 5$, $N_m = 3$



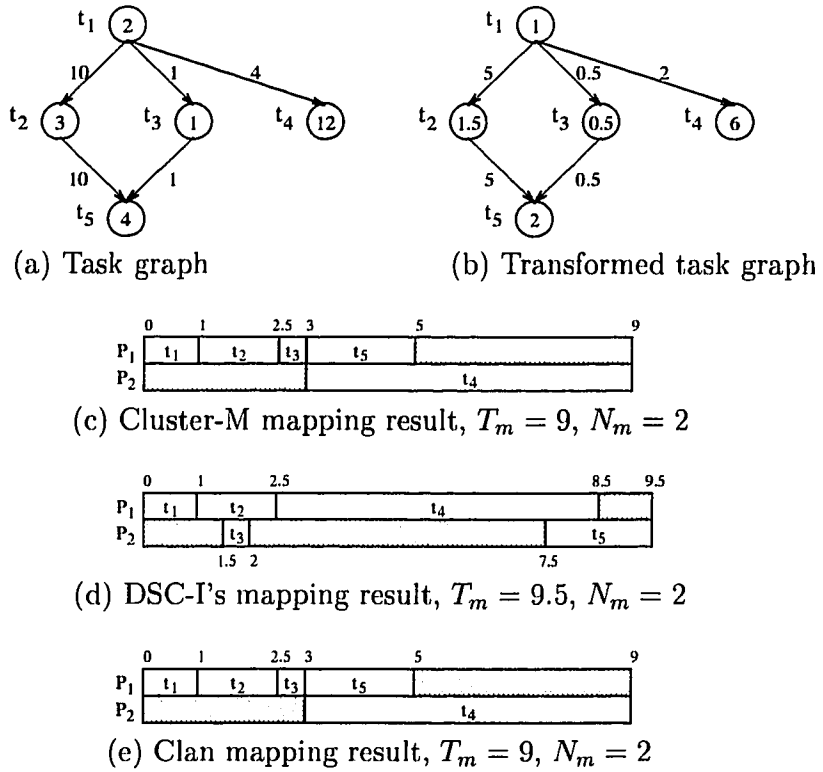
(c) DSC, $T_m = 5$, $N_m = 3$



(d) Clan, $T_m = 10$, $N_m = 1$

Figure 2.20 Comparison example 2 with DSC and Clan.

Comparison with Chaudhary and Aggarwals' Algorithm Next, we compare our mapping results with Chaudhary and Aggarwal. We present two examples. In the first example, the task graph of Figure 2.23 is mapped onto a 2-cube. The mapping results for this example is shown in Figure 2.24. In the second example, the task graph of Figure 2.25 is mapped onto a 2-cube. The mapping results for this example is shown in Figure 2.26. As we see in all the examples in this section, Cluster-M mapping has a superior running time, and the results obtained are similar to or better than those from the other algorithms.



(a) Task graph (b) Transformed task graph

(c) Cluster-M mapping result, $T_m = 9$, $N_m = 2$

(d) DSC-I's mapping result, $T_m = 9.5$, $N_m = 2$

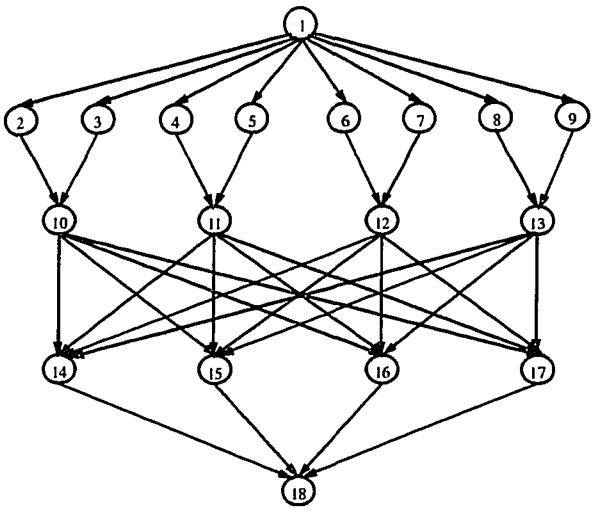
(e) Clan mapping result, $T_m = 9$, $N_m = 2$

Figure 2.21 Comparison example 3 with DSC and Clan.

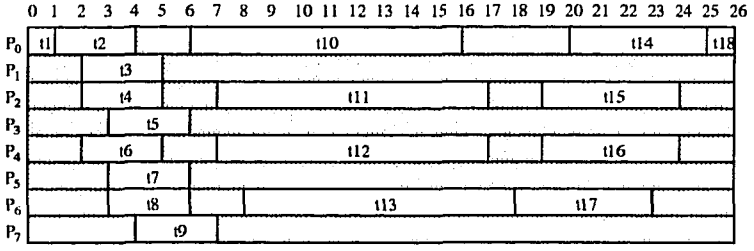
2.6.2 Task Allocation

A generic mapping technique must be able to do both task scheduling as well as task allocation. Cluster-M can efficiently be applied to the both cases. The goal of task allocation is to minimize the communication delay between processors and to balance the load among processors. The problem of task allocation arises when specifying the order of executing the task modules is not required. Therefore, the task graph in task allocation is undirected and the clustering-undirected-graphs algorithm is used to generate the Spec graph in this case. We consider the measure of mapping quality in task allocation to be T_m.

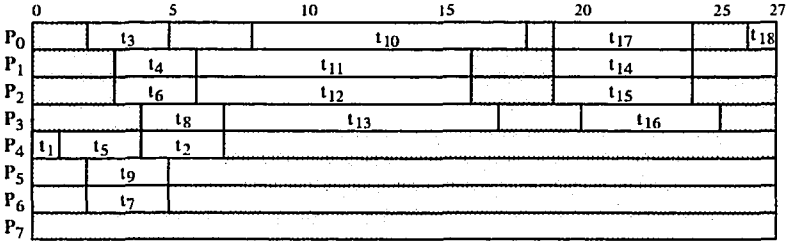
We compare our results to Bokhari's mapping (allocation) algorithm [9] using undirected task graphs. Bokhari's algorithm has the running time complexity of $O(N^3)$, while ours is $O(MN)$. Bokhari's algorithm assumes that the computation



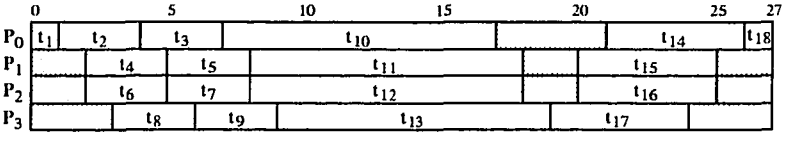
(a) Task graph



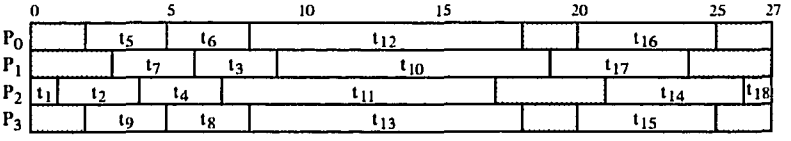
(b)Cluster-M mapping on 8 processors, $T_m = 26$, $N_m = 8$



(c) DSC's mapping on 8 processors, $T_m = 27$, $N_m = 7$



(d) Cluster-M mapping on 4 processors, $T_m = 27$, $N_m = 4$



(e) DSC's mapping on 4 processors, $T_m = 27$, $N_m = 4$

Figure 2.22 Comparison example 4 with DSC.

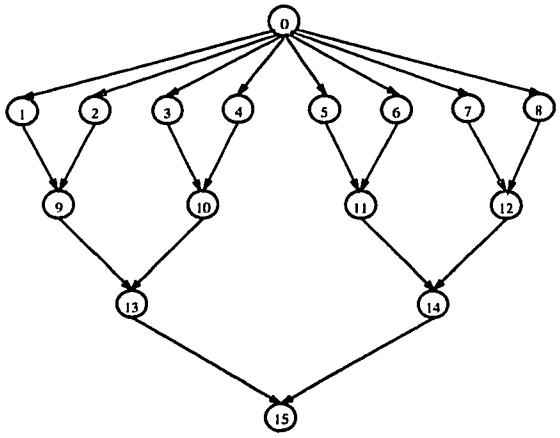
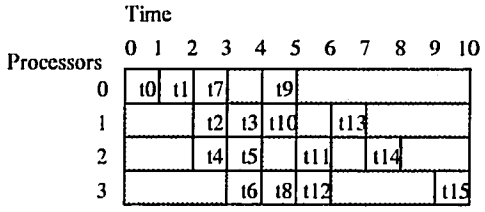
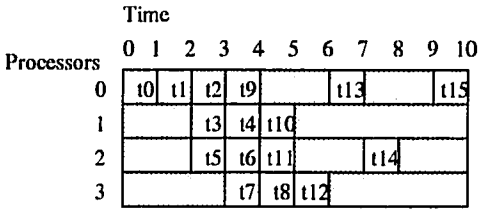


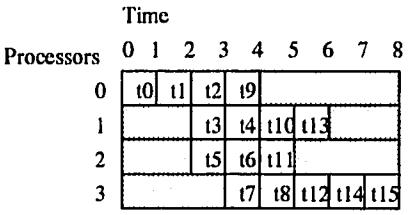
Figure 2.23 Comparison example 1 with Chaudhary and Aggarwal: task graph.



(a) Chaudhary and Aggarwal, $T_c = O(M^4)$, $T_m = 10$, $S_m = 1.6$, $\eta = 0.4$.



(b) Cluster-M, $T_c = O(MN)$, $T_m = 10$, $S_m = 1.6$, $\eta = 0.4$.



(c) Optimal, $T_c = O(2^{MN})$, $T_m = 8$, $S_m = 2$, $\eta = 0.5$.

Figure 2.24 Comparison example 1 with Chaudhary and Aggarwal: mapping results.

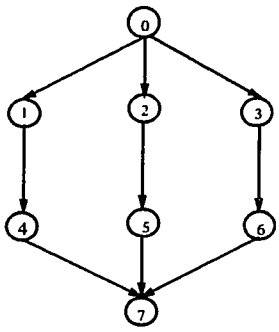
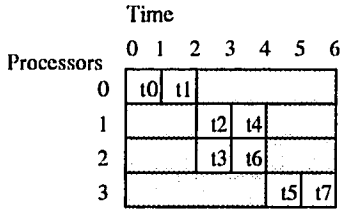
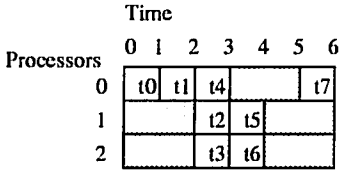


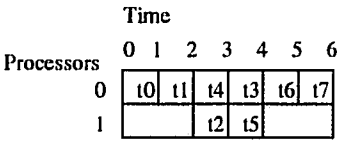
Figure 2.25 Comparison example 2 with Chaudhary and Aggarwal: task graph.



(a) Chaudhary and Aggarwal, $T_c = O(M^4)$, $T_m = 6$, $S_m = 1.3$, $\eta = 0.33$.



(b) Cluster-M, $T_c = O(MN)$, $T_m = 6$, $S_m = 1.3$, $\eta = 0.43$.



(c) Optimal, $T_c = O(2^{MN})$, $T_m = 6$, $S_m = 1.3$, $\eta = 0.65$.

Figure 2.26 Comparison example 2 with Chaudhary and Aggarwal: mapping results.

amount of each task module, the amount of data communication along each task graph edge, the computation speed of each processor, and the data transmission rate along each communication link are all uniform, that is, 1. It further assumes the number of task modules is no greater than the number of processors, so that the mapping can be one-to-one. In this case, a lower bound on T_m can be $\delta + 1$, where δ is the degree of a given task graph.

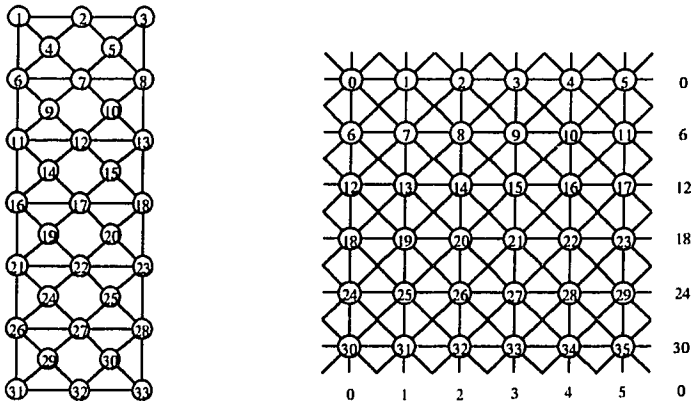


Figure 2.27 Comparison example with Bokhari: task and system graph.

In comparing Cluster-M with Bokhari, we use the example shown in Figure 2.27, which has a 33-node task graph and a 6 × 6 finite element machine (FEM) [9]. A Sun SPARCstation 1 was used for the experiments. The results are shown in Table 2.11. Note that the running time of clustering the task graph and system graph by Cluster-M, which is 0.7 seconds, is not included in T_c, as our clustering is independent of the mapping. However, even if we included it, the running time of Cluster-M would still be 200 times faster than Bokhari's algorithm. The lower bound on T_m as described before is 9, and yet both Cluster-M and Bokhari's algorithms obtained near optimal results of $T_m = 17$ and 13, respectively. The above example uses the same structured task and system graph as in [9]. We have also tested other randomly generated task and system graphs. Table 2.12 shows the mapping results

Table 2.11 Mapping of Bokhari's algorithm and Cluster-M

Task Module	Mapped processor	
	Bokhari	Cluster-M
1	5	0
2	30	1
3	3	2
4	0	6
5	2	3
6	6	4
7	1	7
8	8	8
9	7	9
10	15	5
11	13	12
12	14	10
13	20	11
14	9	13
15	19	19
16	10	18
17	17	14
18	18	15
19	11	26
20	12	20
21	16	27
22	22	32
23	23	21
24	21	16
25	29	28
26	26	17
27	27	22
28	28	33
29	31	24
30	33	23
31	25	25
32	32	30
33	34	31
T_m	13	17
T_c (sec)	152.5	0.05

Table 2.12 Comparisons of mappings of Bokhari's algorithm and Cluster-M

Random Graphs of 10 Nodes	T_m			T_c (sec)	
	Bokhari	Cluster-M	Lower Bound	Bokhari	Cluster-M
1	15	15	8	0.82	0.03
2	9	13	7	1.58	0.03
3	10	11	8	1.20	0.03
4	11	14	8	1.00	0.03
5	11	12	9	1.02	0.03
6	10	12	8	2.35	0.02
7	11	12	8	1.40	0.03
8	10	12	8	1.18	0.03
9	10	13	9	1.20	0.02
10	9	10	7	1.03	0.02

and comparisons for 10 randomly generated task and system graphs of 10 nodes. Similar results were obtained for the set of random graphs.

2.7 Conclusion

This chapter presents a portable parallel programming model called Cluster-M that bridges software and hardware for heterogeneous computing. This model allows software portability without imposing any restrictions on the hardware and provides a mechanism for estimating the performance of a given parallel program on any heterogeneous computers or suite of computers. Using the parameters of this model, portable parallel programs can be specified and then mapped onto dynamically reconfigured heterogeneous organizations. An implementation of this model as a portable programming tool was also presented. Two clustering algorithms were presented that need to be applied only once for each problem (system), independent of any system (problem), and need not be repeated for each mapping. The mapping module of the Cluster-M tool was shown to produce efficient and near-optimal mappings for any given task and system graphs. Using Cluster-M a single software can be ported and shared among various computing units in a heterogeneous suite.

CHAPTER 3

MAPPING AND SCHEDULING FOR HETEROGENEOUS COMPUTING

This chapter consists of two parts. In the first part, we present a brief survey of existing heterogeneous mapping techniques. In the second part, we illustrate a suboptimal Cluster-M-based solution to the problem of mapping application tasks onto heterogeneous computing systems. We propose two clustering algorithms for generating clustered task and system graphs on behalf of mapping. The mapping algorithm employs integer linear programming recursively for mapping clusters of the task graphs onto clusters of the system graphs in order to find a suboptimal solution.

3.1 Introduction

The mapping problem, in its general form, has been known to be NP-complete and has been studied intensively for homogeneous parallel computers during the past two decades [6, 9, 15, 21, 22, 25, 43, 45, 53, 76]. In mapping, an application task and a computing system are usually modeled in terms of a task flow graph and a system graph. The problem, then, is how to map efficiently the task flow graph to the system graph. A task flow graph is a directed acyclic graph (DAG) that consists of a set of vertices and a set of directed edges. A vertex denotes a task module decomposed from the given task. Each vertex is associated with a weight that denotes the computation amount within the corresponding task module. A directed edge joining two task modules denotes that data communication and dependency exist between the two task modules. The weight of an edge represents the amount of data communication. While a task flow graph is usually directed, the system graph is usually an undirected graph. A set of vertices in a system graph denote processors and a set of undirected edges indicate physical communication links for

processor pairs. The weight of a vertex (edge) represents the speed (bandwidth) of the corresponding processor (communication link). We define a graph as nonuniform if and only if the weights of all vertices or the weights of all edges are not the same; otherwise it is uniform.

In recent years, trends in heterogeneous computing (HC) have drawn researchers' attention to the problem of mapping tasks onto a suite of heterogeneous computers [71, 16, 38, 40, 56, 17, 20]. In HC, the task and system graphs can be nonuniform. Therefore, the mapping problem in HC can be viewed as mapping of an arbitrary nonuniform task graph onto an arbitrary nonuniform system graph. This chapter first presents an overview of a number of existing heterogeneous mapping techniques and then illustrates a suboptimal Cluster-M-based heterogeneous mapping algorithm. An essential part of mapping is a way to "cluster" nonuniform task and system graphs. These algorithms are the augmented versions of the clustering algorithms presented in the previous chapter, so that the vertices of the graphs are clustered if and only if they are of the same computational type. For example, all the single instruction, multiple data (SIMD) nodes in a task (system) graph are grouped together. The clustering algorithms are done only once for each task (system) graph, independent of any system (task) graphs, and need not be repeated for every pair of system-task graphs to be mapped.

The Cluster-M mapping algorithm presented in Chapter 2 maps arbitrary clustered task graphs with nonuniform nodes[1] and edges onto arbitrary clustered system graphs with nonuniform nodes and edges. The mapping process of this algorithm is then performed in a recursive fashion by a greedy algorithm matching the clusters of the task graphs (Spec clusters) to the clusters of the system graphs (Rep clusters). In this chapter, we use an extended version of the algorithm which incorporates the *type heterogeneity* [i.e., SIMD and multiple instruction, multiple data

[1]In this chapter, "vertex" and "node" are used interchangeable.

(MIMD)] of tasks and systems in HC. The augmented mapping algorithm presented first maps Spec clusters to Rep clusters of similar computational type and then proceeds with an enhanced fine-grain mapping technique. Since the expected number of clusters at every level of the fine-grain mapping is constant, we propose to use an optimal matching strategy to enhance the algorithm. Therefore, we formulate and solve each step of the fine-grain cluster mapping by using an integer linear programming (ILP) model. We then compare the mapping results of our algorithm with those of some other heterogeneous mapping techniques.

The remainder of this chapter is organized as follows. We first review a number of heterogeneous mapping techniques in Section 3.2. We then present augmented Cluster-M clustering and mapping algorithms in Section 3.3. A comparison study is also included in this section. The conclusion is presented in Section 3.4.

3.2 A Survey of Heterogeneous Mappings

In this section, we present an overview of a number of recently proposed heterogeneous mapping algorithms[27, 28, 65]. We categorize these algorithms into two groups: static and semi-dynamic algorithms. In static mapping, the structure of both task and system are known prior to execution and do not change throughout the computation. In semi-dynamic mapping, the structure of the task is not known prior to execution, but the structure of the system is known, and it is assumed not to change. The rest of this section is organized as follows. The static algorithms are presented in two groups. The first group is a set of nondeterministic mapping algorithms presented in Section 3.2.1. The second group, presented in Section 3.2.2, is a set of graph-based algorithms. Semi-dynamic algorithms are explained in Section 3.2.3

3.2.1 Nondeterministic Algorithms

Tao et al. [38] proposed three static heuristic mapping algorithms based on simulated annealing, tabu search, and stochastic probe approaches. Three types of costs are taken into account: computation, communication, and interference costs.

The computation cost of a processor is the sum of the computation time of tasks on the processor. Communication cost is the time consumed by communication over the interconnection network between two interacting tasks located on two different processors. Interference cost is the time incurred when two tasks compete for the resources available on one processor where the two tasks are assigned. The execution time of a processor under a mapping is estimated as the sum of its computation, communication, and interference costs. The completion cost of a mapping is defined as the maximum execution time of all processors. The objective function of the mapping problem is to find a mapping so that the completion cost is minimized. These algorithms are nondeterministic, hence their time complexities cannot be known in advance. Another disadvantage of these algorithms is that data dependency is not considered. This implies the assumption that there is no interdependent relation between any two tasks. This assumption does not hold, however, in most application tasks.

- Simulated Annealing

  Simulated annealing utilizes occasional uphill moves to avoid entrapment in poor local optimums. To achieve this, a random-number generator and a control parameter called temperature are used. A typical implementation of simulated annealing usually has two nested loops and two other parameters, a cooling ratio r and a temperature length L. The following shows a typical simulated annealing heuristic.

Get a random initial solution π

Get an initial temperature $T > 0$

While stop criterion not met do:

Perform the following loop L times:

Let π' be a random neighbor of π

Let Δ = cost(π) - cost(π')

If $\Delta \geq 0$ (downhill move)

set $\pi = \pi'$

If $\Delta < 0$ (uphill move)

set $\pi = \pi'$ with probability $e^{\Delta/T}$

Set $T = rT$ (reduce temperature)

Return the best π visited

In the implementation of Tao et al., temperature length, L, is set to be $n\times$ *SIZEFACTOR*, where n is the number of task modules and *SIZEFACTOR* is a parameter that must be tuned. The initial temperature T is chosen so that the initial acceptance rate is around another parameter, which needs to be tuned, called *INITPROB*. The stop criterion of their implementation is that: (1) for five temperatures, the acceptance rates are all lower than *MINPERCENT* which is the third parameter that needs to be tuned, and (2) the best visited solution is not improved during this period of time. All three parameters are tuned for each problem instance. This is not an easy task and it is often obtained by trial and error. It has been determined that for most problem instances, the following values are appropriate. $r = 0.95$, *SIZEFACTOR* = 16, *INITPROB* = 0.4 and *MINPERCENT* = 0.02.

- Tabu Search

In typical tabu search, listed below, t is the length of the tabu list. During

each iteration, the algorithm makes an exhaustive search of the solutions in the neighborhood of the current solution that have not been traversed in the last t iterations. The current solution is replaced by the neighboring solution that has the best cost. A circular list is used to implement the tabu list and to maintain the vertices moved in the last t iterations.

Get a random initial solution π

While stop criterion not met do:

Let π' be a neighbor of π maximizing Δ=cost(π)-cost(π') and not visited in the last t iteration

Set $\pi = \pi'$

Return the best π visited

- Stochastic Probe Approach

The stochastic probe algorithm is a combination of the stochastic search process in simulated annealing and the aggressive search process in the tabu search. In the algorithm, $\tilde{S}(\pi, v)$ denotes the subset of moves in $S(\pi)$ that redefines $\pi(v)$, where $S(\pi)$ represents the set of moves applicable to solution π and $\pi(v)$ indicates the processor that v is assigned to under solution π. Given any integer $p \geq 0$, random$(-p)$ denotes a random integer such that $-p \leq$ random$(-p) \leq 0$. The value of β is set to between 10% to 15% depending on problem instance. The stochastic probe algorithm is detailed as follows.

Get a random initial solution π

Let L be a circular list of the vertices in V

Set v to any of the vertices in V (the current vertex)

While stop criterion not met do:

While there is any $\Delta > 0$ in the last k iterations of this loop do:

Let v be the next vertex down the list L

Let $s \in \tilde{S}(\pi, v)$ and $\pi' = s(\pi)$ such that Δ = cost(π) - cost(π') is maximized

If $\Delta >$ random($-p$), set $\pi = \pi', p = p_0$

Perturb randomly the value of $\pi(u)$ for $\beta\%$ of the vertices u in V

Return the best π visited

The algorithm consists of a sequence of probes and each probe searches for a local optimum. The last solution of a probe will be modified randomly to be used as the initial solution of the next probe. In experimental studies, the cost-performance of the stochastic probe heuristic is superior to heuristics based on simulated annealing and tabu search.

3.2.2 Graph-Based Algorithms

The mapping problem can be formulated in a graph theoretic manner. One of the most famous graph-based approaches is Stone's work [63]. Stone transfers the mapping problem into a network flow model and solves this problem using a Max Flow/Min Cut algorithm. Stone's model provides an optimal solution only for the two-processor problem. Lo [44] extended Stone's work to find a suboptimal solution of the mapping problem for general distributed (heterogeneous) systems. Lo's algorithm, called Algorithm A, is a heuristic that combines recursive invocation of Max-Flow/Min-Cut algorithms with a greedy-type algorithm. Algorithm A consists of the following three parts:

1. Grab. For a given processor p_i, the n-processor system graph is converted into a two-processor system with p_i and a supernode $\bar{p}_i$, which represents the other $n - 1$ processors. Apply a Max Flow/Min Cut algorithm to the two-processor system to find those tasks that would be assigned to p_i. These steps are repeated for each processor to yield a partial mapping.

2. Lump. For those tasks that remain unmapped in Grab, map all of them to one processor.

3. Greedy. For those tasks that remain unmapped in Lump, identify clusters of tasks between which communication costs are large. Merge such clusters of tasks, and map all tasks in the same cluster to the processor which could finish executing these tasks earliest.

The major flaw of this algorithm is its high time complexity, which is equal to $\mathcal{O}(M^4 N \log M)$, where M and N are the number of tasks and processors, respectively. Another problem is that it does not take into account data dependencies between tasks.

Another graph-based algorithm is Shen and Tsai's graph-matching approach [60]. The mapping problem is transformed into a graph-matching model based on the weak homomorphism from task graph to system graph. A graph $G_1(V_1, E_1)$ is weakly homomorphic to a graph $G_2(V_2, E_2)$ if there exists a mapping $M : V_1 \rightarrow V_2$ such that if edge $(a, b) \in E_1$, then edge $(M(a), M(b)) \in E_2$. They consider a cost function that represents the total execution and communication time for completing the given task, and a minimax criterion for the minimization of the cost function. The search of optimal weak homomorphism corresponding to optimal mapping is next formulated as a state-space search problem. The problem is then solved using the well-known A* algorithm in artificial intelligence [73]. In a state-space search problem, each state is denoted by a node. Node expansion is an operation for generating successors of nodes. A solution path is a path defined by a sequence of node expansions that leads a start node to one of the goal nodes. A* algorithm is a heuristic that combines branch-and-bound and dynamic programming approaches. In an A* algorithm, an evaluation function is used to decide the order of nodes for examination. It is guaranteed to find a solution path optimal in term of minimized path cost. An evaluation function

is defined as $f(n) = g(n) + h(n)$, where $g(n)$ is the minimum path cost from the start node to node n in the state space and $h(n)$ is an estimate of the minimum path cost from node n to a goal node. The problem with the A*-type algorithms is that if the estimate of $h(n)$ is chosen inappropriately, then the optimal solution path may not be easily found.

Tan et al. [67] propose a minimum spanning tree based algorithm for finding minimal scheduling time of sequentially executed subtasks. Two types of data distributions are considered, namely data reuse and multiple data copies. Data reuse occurs when two subtasks located at one processor need the same data item from a subtask at another processor. Multiple data copies arise when two subtasks need the same data item from another subtask and all three subtasks are located at different processors. They assume that atomic input operations of two subtasks can be executed in an interleaved fashion. This assumption makes it possible to reduce communication delay among interacting subtasks. This algorithm involves the following two steps.

1. Constructing a graph with respect to the given information including subtask flow graph, the representation of the heterogeneous computing system, and an arbitrary matching scheme.

2. Using a modified version of Prim's minimum spanning tree algorithm [4] to find a minimum spanning tree in the graph generated from step 1. The order of the vertices added to the minimum spanning tree corresponds to the executing order of the corresponding atomic input operation, hence the minimal scheduling.

The time complexity of this algorithm is $\mathcal{O}(E + V \log V)$ where E and V are the number of edges and the number of vertices in the graph obtained from step 1.

The main drawback of this technique is that subtasks are assumed to be executed in sequential order.

Leangsuksun and Potter [41] propose a set of heterogeneous mapping algorithms. The first algorithm, *HP greedy*, is the simplest and is used as an initial phase in other algorithms. The HP greedy algorithm is performed as follows.

1. Partition the input task graph into independent subgraphs.
2. For each subgraph (starting from the top to the bottom), sort tasks in the subgraph by their weights.
3. Starting from the heavier node, map each task to the processor leading to the best expected execution time.
4. Remove the chosen processor from the processor list. If the processor list becomes empty, it is reset to include all processors.

Another algorithm, called one level reach-out greedy (OLROG), is similar to HP greedy except that it uses the simple processor list assignment policy and it takes waiting time into account in the processor selection decision. Waiting time includes the previous scheduled task completion time, communication time, and delay time of the current task. The empirical results show that algorithm OLROG performs better but has larger complexity. The main drawback of these techniques is that the communication bandwidth of the links are not taken into account. Therefore, the accurate data communication time cannot be well captured.

Cluster-M mapping, presented in the last chapter, can map arbitrary structured nonuniform task graphs with M task modules onto arbitrary structured nonuniform system graphs with N processors in $\mathcal{O}(MP)$ time, where $P = \max(M, N)$. In Cluster-M, a clustered task (system) graph is a multilayered partitioned graph such that every level contains a number of clusters, each representing a partition

subgraphs[15, 25, 17]. This simplifies the mapping process since at every level independent subgraphs of the task graph are mapped onto the subgraphs of the system graph. An extended version of the Cluster-M clustering and mapping algorithms is presented in the next section. These augmented algorithms are more suitable for HC.

3.2.3 Semi-Dynamic Algorithms

Leangsuksun et al. [42] developed two semidynamic mapping schemes, centralized and distributed, that differ in the extent of system knowledge and location(s) of the task allocator(s). It is assumed that task execution and communication times are not known until execution and that the system condition is invariant. In the distributed mapping algorithm, called K nearest-neighboring algorithm, each computing node has a local mapper that allocates tasks in its local task queue to the most suitable node among itself and its K highest communication capacity neighbors. The algorithm consists of the following steps.

1. K nearest neighbor grouping; for each processor, group K highest communication capacity neighboring nodes for its local mapper.

2. Premapping; each mapper gets the same number of tasks in its local queue.

3. Local queue length equalizing (LQE); each mapper determines the best node among the group of nodes in step 1 to execute tasks.

The complexity of the K nearest-neighboring algorithm is $\mathcal{O}(KN/M)$, where N and M are the total number of tasks and processors.

In the centralized mapping algorithm, the global queue equalizer (GQE) algorithm, there is only one global mapper located in a master host. The host node collects global system information and determines task assignment. The algorithm consists of the following two procedures.

1. Master host selection; selecting a master node as the centralized mapper.

2. GQE-OLROG module; for each task in the global queue, the algorithm determines task allocation by choosing a node which has the most communication bandwidth.

The master host selection module in the GQE algorithm can be carried out prior to execution time, and therefore its complexity can be disregarded. Within the GQE-OLROG module, there are M choices for the best task-machine selection. Considering communication time in order to obtain a better performance, there are, at most, $M-1$ possible machines executing parent nodes of a current task. Therefore, the total complexity of the GQE algorithm is $\mathcal{O}(NM^2)$. Although these two algorithms are proposed to handle dynamic cases, they are not fully dynamic since task rescheduling and migration are not considered.

3.3 An Augmented Cluster-M Mapping

Our proposed technique is based on the Cluster-M paradigm [15, 25, 17] which facilitates the design and mapping of portable parallel programs. A Spec (Rep) graph may be obtained by clustering a given task (system) graph. A graph is called nonunform if the weights of all the nodes are not the same and the weights of the edges also differ. The weight of a node in a task graph (system graph) represents the number of instructions (speed) in that code block (processor). In Chapter 2, two algorithms were proposed for clustering arbitrary nonuniform task graphs and arbitrary nonuniform system graphs. In this section, we extend those algorithms by incorporating the heterogeneity of tasks and systems in HC. The extended task graph clustering takes into account the type of parallelism present in each portion of the task by clustering each code segment independently. The modification to the system graph clustering takes into account the presence of different machines in

the system, which provides a spectrum of computational modes. Furthermore, the mapping algorithm presented in this chapter is an augmented version of the original one presented in Chapter 2. The mapping algorithm uses integer linear programming instead of the greedy algorithm in every step of mapping.

3.3.1 Task Clustering

As defined by the input format of HOST explained in Chapter 1, a task is composed of a number of subtasks. Each of the subtasks contains a number of heterogeneous code segments. Each code segment is further decomposed into several homogeneous code blocks. These correspond to the input format of HOST presented in Chapter 1. The Clustering Nonuniform Directed task Graph (CNDG) algorithm, presented in Chapter 2, clusters the task graph without distinguishing between different layers (i.e., subtask, code segment and code block). We present the Augmented Task Clustering (ATC) algorithm to cluster a subtask graph having such a hierarchical structure. The ATC algorithm first clusters code blocks inside each code segment concurrently; it then clusters code segments at the subtask level.

Algorithm ATC(G)
Input: Subtask graph G consists of code segments $G_i, 1 \leq i \leq n$
Output: Spec graph S
begin
    for each $G_i, 1 \leq i \leq n$ **do in parallel**
    begin
        G'_i = CNDG(G_i)
    end
    $G' = \cup_{i=1}^{n} G'_i$
    S = CNDG(G')
end

Figure 3.1 The Augmented Task Clustering (ATC) algorithm.

The input to the algorithm is a subtask graph G that contains n subgraphs (code segments) $G_i, 1 \leq i \leq n$, and the output is a Spec graph. These code segments are clustered in parallel by calling the CNDG subroutine. Notice that by clustering each code segment independently, we are clustering only code blocks having the same computational type. The returned Spec clusters from these subroutines then form a new subtask graph in which each node (code segment) is a Spec cluster. The new graph is further clustered using CNDG subroutine.

As discussed in Chapter 2, the time complexity of the CNDG algorithm is $\mathcal{O}(M^2)$, where M is the number of nodes on the input graph. To analyze the time complexity of the ATC algorithm, we assume that the number of nodes in the subtask graph is M. Then the number of code segments, n, will be in the range, $1 \leq n \leq M$. When $n = 1$, that is, there is only one code segment, the code segment is exactly the same as the subtask and the time complexity is bound by $\mathcal{O}(M^2)$. If $n = M$, (i.e., each code segment has only one node), it implies that the new subtask graph is the same as the original subtask graph, then the time complexity is still $\mathcal{O}(M^2)$. Therefore the time complexity of the ATC algorithm is also $\mathcal{O}(M^2)$.

To illustrate this algorithm, consider the heterogeneous subtask flow graph, which consists of one MIMD code segment and one SIMD code segment, as shown in Figure 3.2. Each vertex is labeled with its computation amount, and each edge is labeled with its data communication amount. Using the CNDG algorithm, a single Spec graph would have been obtained in which the two code segments were not distinguishable. However using the ATC algorithm, the obtained Spec graph will consist of two subgraphs: one contains MIMD-type clusters and the other contains SIMD-type clusters. The MIMD-type Spec subgraph is illustrated in Figure 3.3. The Spec graph is constructed by merging the clusters when they have communication needs. In our illustration, embedding operations are represented by perforated lines and merging operations are represented by dotted and rounded rectangles.

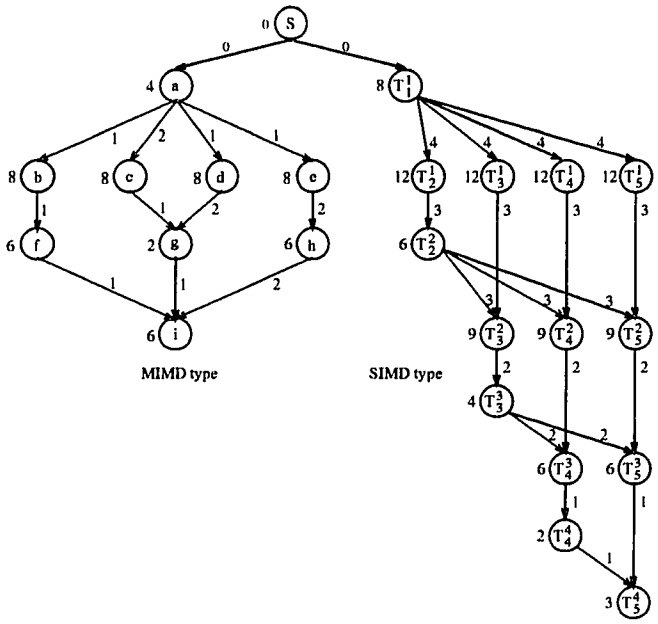


Figure 3.2 A heterogeneous subtask consists of MIMD and SIMD code segments.

3.3.2 System Clustering

An HC system contains a number of autonomous and heterogeneous parallel machines. Each one of these parallel machines can be modeled as an undirected graph in which nodes depict processors and edges represent the interconnection topology of the machine. These graphs further constitute an undirected graph that can represent the HC system. Therefore, two levels of undirected graphs are used to model the HC system: a machine-level graph and a system-level graph. The CNUG algorithm, presented in Chapter 2, clusters a system graph without distinguishing between machine and system level. Therefore, it may cluster a node from one machine to another before all the nodes in one machine are clustered first. In this section, the augmented system clustering (ASC) algorithm is presented to cluster an HC system graph having two levels. The system level graph is clustered after the clustering of all machine level graphs are done. The algorithm utilizes the CNUG algorithm [17] as a subroutine to cluster both levels of undirected graphs. The subroutine takes a system graph as its input and outputs a Rep graph.

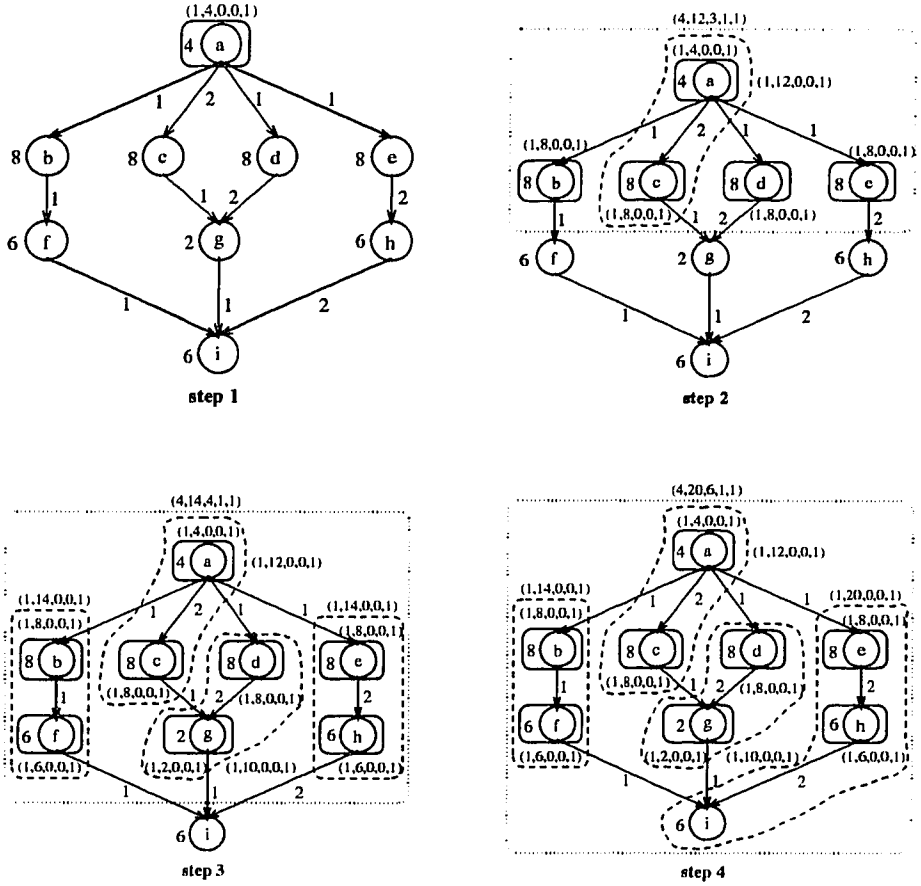


Figure 3.3 Clustering the MIMD code segment.

Algorithm ASC(G)
Input: System graph G consists of machine level graphs $G_i, 1 \leq i$ *leqn*
Output: Rep graph R
begin
  for each $G_i, 1 \leq i \leq n$ **do in parallel**
  begin
    $G'_i = \text{CNUG}(G_i)$
  end
  $G' = \cup_{i=1}^{n} G'_i$
  $R = \text{CNUG}(G')$
end

Figure 3.4 The Augmented System Clustering (ASC) algorithm.

The ASC algorithm is shown in Figure 3.4. The analysis of the time complexity is similar to that in the previous section. The time complexity of the algorithm is equal to the running time of the CNUG subroutine, which is $\mathcal{O}(N(E \log E + N^2))$, where E is the number of edges and N is the number of processors in the system graph. In the worst case, the time complexity of this algorithm will be $\mathcal{O}(N^3 \log N^2)$, where the system graph is completely connected so that $E = \mathcal{O}(N^2)$.

Consider the heterogeneous computing system shown in Figure 3.5, which consists of one MIMD machine and one SIMD machine. The MIMD machine has three processors, P1, P2, and P3. The SIMD machine has two processors, P4 and P5. Each node denotes a processor and is associated with a computation speed; each edge is associated with a communication bandwidth. The clustering of the Rep graph is also illustrated in Figure 3.5.

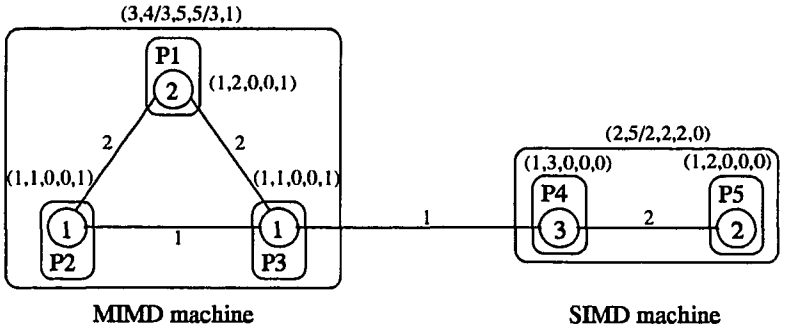


Figure 3.5 The system graph and its clustering of a heterogeneous suite.

3.3.3 Augmented Cluster-M Mapping

This section presents an augmented suboptimal Cluster-M-based mapping algorithm for mapping the Spec graph onto the Rep graph, generated using the ATC and ASC algorithms, respectively. The mapping algorithm presented here is a modified version of the Cluster-M nonuniform mapping algorithm presented in Chapter 2. The Cluster-M nonuniform mapping algorithm is proposed to map arbitrary clustered task graphs with nonuniform nodes and edges onto arbitrary clustered system graphs

with nonuniform nodes and edges. In the mapping algorithm, the mapping process is performed in a recursive fashion by a greedy algorithm matching the Spec clusters to the Rep clusters. In contrast to this technique, the algorithm presented here first maps code segments onto machines with the same computation type. It then proceeds with an enhanced recursive fine-grain mapping so that at every level an optimal assignment of Spec clusters to Rep clusters is found. We formulate and solve each step of the fine-grain cluster mapping using an ILP model. ILP solvers with polynomial time complexity are now available in software packages such as *Mathematica*, so we will treat these tools as a 'black box' and not go into the details of how ILP programs are solved.

We assume that the expected number of clusters at every level of mapping is a constant. This is based on the observation that most parallel architectures have bounded-degree nodes (every processor is connected to a constant number of other processors). Examples of such systems are mesh, binary tree, ring, and torus. Similarly, a large set of computational tasks can be expressed in the form of bounded-degree task graphs. Examples of such tasks are algorithms using a divide-and-conquer technique, which are very common in image processing.

In the original Cluster-M nonuniform mapping, five parameters are used to evaluate an optimization function at every level of clustering. Therefore it allows, for example, an SIMD node in the task graph to be mapped onto an MIMD node in the system graph (if the tradeoffs are substantial) by evaluating the execution time estimation function for various options. The solution obtained at every level of mapping is suboptimal since it does not evaluate the function for all the possibilities. The augmented Cluster-M mapping algorithm is different in two ways. First, in this algorithm we restrict mapping so that, for example, an MIMD node in the task graph can only be mapped onto an MIMD node in the system graph. Second, for every level of mapping we obtain an optimal solution by considering *all* possible task

graph-system graph-node pairs, with the restriction that they are of the same type. To map a code segment onto a machine of the same type, the following is done to obtain the fine-grain mapping.

We first begin by calculating the reduction factor $f_{(u,v)}$ and the estimated execution time of each Spec-Rep cluster pair. Then starting from the Spec clusters at the top level, assignment of these Spec clusters onto a set of suitable Rep clusters must be obtained. To do this, we model the assignment process using an ILP model, described as follows. A binary variable $\mu(S_i^u, R_j^v)$ is defined to indicate whether a Spec cluster S_i^u is mapped onto a Rep cluster R_j^v, that is, when $\mu(S_i^u, R_j^v) = 1$, S_i^u is mapped to R_j^v, otherwise, $\mu(S_i^u, R_j^v) = 0$. Each Spec cluster can be mapped to only one Rep cluster; this is represented by $\sum_{i,j} \mu(S_i^u, R_j^v) = 1$. The accumulated estimated execution time on Rep cluster R_j^v is denoted by $\Gamma(R_j^v)$, and we have $\Gamma(R_j^v) = \sum_{i,j} \mu(S_i^u, R_j^v)\tau(S_i^u, R_j^v)$. We denote the overall estimated execution time by T_m such that for all $j, T_m \geq \Gamma(R_j^v)$. Our objective is to minimize the overall estimated execution time; therefore, the objective function of our ILP model can be expressed as follows:

$$\text{minimize } T_m, \text{while } T_m \geq \Gamma(R_j^v) \text{ for all } j$$

Once the minimum T_m is found, matching Spec clusters and Rep clusters can be determined by using binary variables $\mu(S_i^u, R_j^v)$. After the Spec clusters are mapped onto the Rep clusters, the procedure is repeated, mapping the subclusters of every Spec-Rep cluster pair.

A detailed description of our mapping algorithm is presented in Figure 3.6. The time complexity of this algorithm can be analyzed as follows. We assume that the degrees of the given task graph and system graph are bounded by two constants, c and k, respectively. Furthermore, it is assumed that at a certain level of mapping the hierarchical Spec graph has c Spec clusters and the hierarchical Rep graph has k Rep clusters. Then the total numbers of iterations for the second outer **for** loop and

the most inner **for** loop are c and k. Therefore, the total number of iterations for these **for** loops is bound by $\mathcal{O}(c \times k)$. Consider the portion of ILP; it examines all instances of (S_i^u, R_j^v) pairs for all i and all j. Therefore, the running time of the ILP portion is equal to $\mathcal{O}(k^c)$. Therefore, the overall time complexity of the mapping algorithm is $\mathcal{O}(c \times k) + \mathcal{O}(k^c) = \mathcal{O}(k^c)$.

Augmented Cluster-M Mapping Algorithm(S, R)
Input: A Spec graph S and a Rep graph R
begin
  for each computational type
  begin
    calculate reduction factor $f_{(u,v)} = \frac{\sum_j \sigma_{R_j^v}}{\sum_i \sigma_{S_i^u}}$
    for each Spec cluster S_i^u
    begin
      for each Rep cluster R_j^v
        calculate the estimated execution time $\tau(S_i^u, R_j^v)$
    end
    Start Integer Linear Programming
    Set the following constraints
      $\sum_{i,j} \mu(S_i^u, R_j^v) = 1$
      $\Gamma(R_j^v) = \sum_{i,j} \mu(S_i^u, R_j^v) \tau(S_i^u, R_j^v)$
      $T_m \geq \Gamma(R_j^v)$
    Specify the following objective function
      Minimize T_m
  end
end

Figure 3.6 Augmented Cluster-M mapping algorithm.

Consider mapping the task graph illustrated in Figure 3.2 to the system graph shown in Figure 3.5. The mapping is done for each type of Spec and Rep cluster, respectively. The mapping of the MIMD Spec subgraph onto the MIMD Rep subgraph is done below. At the top level, the mapping is trivial since there is only one Spec cluster $S_0(4, 20, 6, 1, 1)$ and one Rep cluster $R_0(3, \frac{4}{3}, 5, \frac{5}{3}, 1)$. At the next level, four Spec clusters $\{S_1(1, 12, 0, 0, 1)$, $S_2(1, 14, 0, 0, 1)$, $S_3(1, 10, 0, 0, 1)$,

$S_4(1, 20, 0, 0, 1)\}$ are mapped to three Rep clusters $\{R_1(1, 2, 0, 0, 1)$, $R_2(1, 1, 0, 0, 1)$, $R_3(1, 1, 0, 0, 1)\}$. Using our mapping algorithm, S_3 and S_4 are mapped onto R_1; S_2 and S_1 are mapped to R_2 and $R3$, respectively. This implies that task modules {d, e, g, h, i} are mapped to processor P1, {b, f} are mapped to P2, and {a, c} are assigned to P3. The mapping of the SIMD Spec subgraph onto the SIMD Rep subgraph can be done in a similar way. The overall mapping result is shown in Figure 3.7.

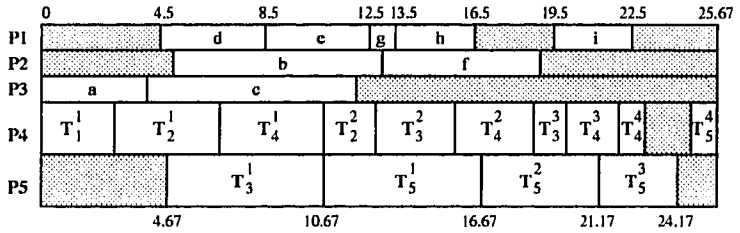


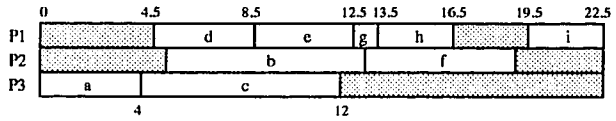
Figure 3.7 The Gantt chart of obtained schedule.

3.3.4 Comparison Study

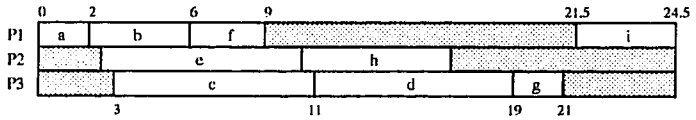
In the following, we compare our algorithm with three other graph-based mapping algorithms, including the original Cluster-M mapping algorithm, Lo's Algorithm A, as well as Shen and Tsai's A* algorithm. For the rest of the chapter, we will use Max Flow/Min Cut to refer to the Algorithm A of Lo's algorithm. Since all of these algorithms do not incorporate heterogeneity in computation and machine types in their mapping, it is only possible to compare the results of mapping each type of task module onto the same type of processor.

Consider the example we discussed in the previous section for mapping the task flow graph of Figure 3.2 to the system of Figure 3.5. The scheduling Gantt chart, using our algorithm on the assignment of the MIMD code segment onto the MIMD subsystem, is shown in Figure 3.7. The SIMD code segment shown in Figure 3.2 represents the forward elimination part of a Gaussian elimination kernel. Suppose that, using a baseline computer, it takes one clock cycle to perform an addition or

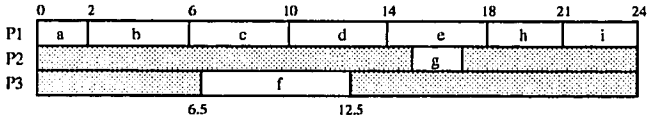
subtraction and that it takes two clock cycles to do a multiplication or division of two real numbers. Also, assume the communication amount on an edge to be the number of real numbers that need to be sent. The mapping results of the MIMD-type task modules onto the MIMD-type processors by using our suboptimal mapping, the original Cluster-M mapping algorithm, Lo's Max-Flow/Min-Cut, as well as Shen and Tsai's A* are shown in Figure 3.8. Their total execution times are 22.5, 24.5, 24, and 28, respectively. Our suboptimal mapping algorithm produces the best result.



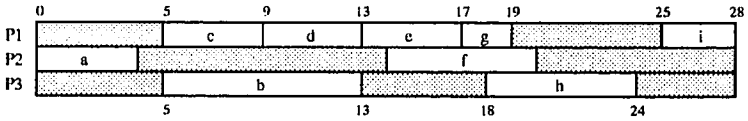
(a) The Augmented Cluster-M mapping algorithm



(b) The Cluster-M nonuniform mapping algorithm



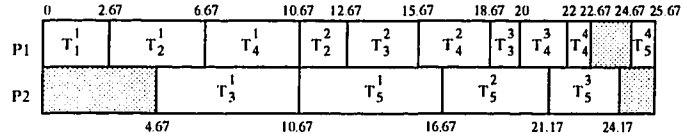
(c) Lo's Max-Flow/Min-Cut algorithm



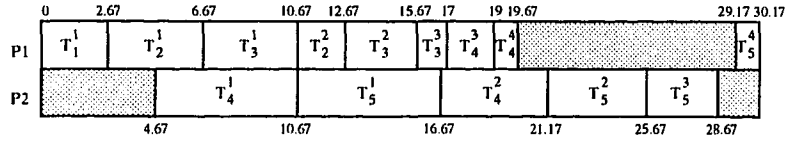
(d) Shen and Tsai's A* searching algorithm

Figure 3.8 The mapping results by using different algorithms.

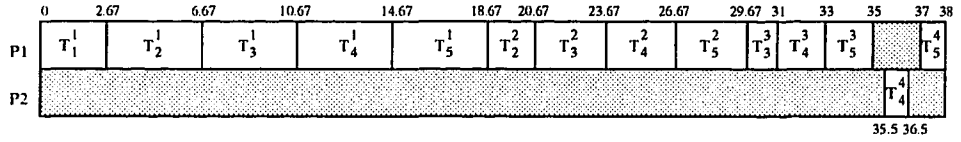
The mapping results of the SIMD-type task modules onto the SIMD-type processors are shown in Figure 3.9. The total execution time by the four different mapping algorithms are 25.67, 30.17, 38, and 33.83, respectively. Evidently the augmented Cluster-M mapping algorithm produces the best mapping, yet the original nonuniform Cluster-M algorithm (from Chapter 2) also produces a very good results.



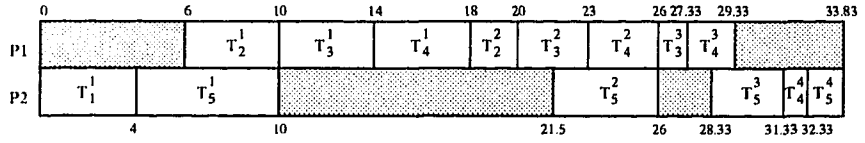
(a) The Augmented Cluster-M mapping algorithm



(b) The Cluster-M nonuniform mapping algorithm



(c) Lo's Max-Flow/Min-Cut algorithm



(d) Shen and Tsai's A* searching algorithm

Figure 3.9 The mapping results of Gaussian elimination by using different algorithms.

3.4 Conclusion

This chapter presents a brief overview of a number of existing heterogeneous mapping techniques. It also contains a study of the problem of assigning and mapping a given task onto a heterogeneous suite of computers. An optimal solution to this problem is one that leads to the minimum execution time subject to certain constraints. Finding the optimal solution is known to be computationally difficult. Therefore, this chapter presented a suboptimal solution. Two algorithms for clustering task flow graphs and system graphs were studied. A suboptimal heterogeneous mapping algorithm using the ILP model was presented. Both the clustering and mapping algorithms are extensions to the original Cluster-M mapping methodology [15, 25, 17] so that they are more suitable for heterogeneous computing. The scheduling results obtained for the presented examples, compared with other heterogeneous mapping techniques,

are better in terms of total execution time and the running time for obtaining such solutions.

CHAPTER 4

HARDWARE ESTIMATION OF HETEROGENEOUS COMPUTING

In HC, code profiling is the process of determining what types of codes are found in a given heterogeneous task. Once this information is available, it is desirable to know how many processors are needed for each of the code types. In this chapter, we propose two methods for estimating the minimum number of processors needed for each of the code types identified in a given heterogeneous task. The first method involves making use of task compatibility graphs. We show that a task compatibility graph can be generated by analyzing certain compatible relations between task module pairs of a given task flow graph. We define the resource (processor) minimization problem to be equivalent to finding the minimal number of cliques that cover the task compatibility graph or to finding the minimal number of colors that color the vertices of its complement graph, called the task conflict graph. We solve this problem using a greedy approach in $O(|V| \log |V| + |E|)$ time, where $|V|$ and $|E|$ are the number of vertices and edges of the task compatibility graph. We show that for certain types of task compatibility graphs optimal solutions can be obtained in polynomial time. The second method studied in this chapter utilizes the Cluster-M clustering methodology presented in Chapter 2 for estimating the minimum number of processors. Examples are shown to compare the estimated results obtained using different techniques.

The rest of this chapter is organized as follows. In Section 4.1, we show how to generate a task compatibility graph and a task conflict graph from the task flow graph of a given task. Our greedy algorithm for finding a minimal set of cliques for a task compatibility graph is presented in Section 4.2. We include a discussion on the special structures of task compatibility graphs in Section 4.3. We discuss the Cluster-M estimating technique in Section 4.4. Examples are presented in Section

4.5 to illustrate and compare the efficiency of the estimates obtained. The concluding remarks of this chapter are presented in Section 4.6.

4.1 Task Compatibility and Task Conflict Graphs

An application task can be represented by a task flow graph in which a set of vertices denote task modules and a set of directed edges indicate the dependency relations between the task modules. We assume that the code type of each task module is identified by a code-type profiler and is incorporated into the vertex set of the task flow graph. If the code types of two task modules are identical and these two task modules cannot be executed in parallel, then they are said to be compatible and should be assigned to the same processor. Then the processor (resource) is said to be shared by the two task modules. By analyzing a task flow graph, the number of groups of compatible task modules determines the number of processors. The idea of resource sharing is not new, it has been extensively studied in high-level synthesis of digital systems [49, 50, 55]. The use of clique partitioning for resource minimization in high-level synthesis was first discovered by Tseng [68]. The two primary advantages of sharing resources are (1) improving the productivity of the whole heterogeneous suite, and (2) decreasing the size of system graphs so that it simplifies the mapping process and reduces the communication overhead.

A task flow graph $G(V, E)$ consists of a set of vertices, $V = \{v_i | 1 \leq i \leq n\}$, which denotes the task modules to be executed, and a set of directed edges, $E = \{(v_i, v_j) | 1 \leq i \leq n, 1 \leq j \leq n\}$, which denotes a data communication existing from module v_i to v_j and that v_i must be completed before v_j starts. Each task module v_i is associated with an amount of computation A_i, i.e. the number of clock cycles required to execute all the instructions of v_i on a baseline machine. Each edge (v_i, v_j) is associated with D_{ij}, the amount of data required to be transmitted from module v_i to module v_j, where $D_{ij} \geq 1$. A task flow graph is called nonuniform if the weights

of all the nodes are not the same or if the weights of the edges differ. The code type of a task module v_i is represented by $T(v_i)$. Task modules v_i and v_j are said to be compatible if there exists any precedence relation between them (*i.e.* there is a path from v_i to v_j, or vice versa) and $T(v_i) = T(v_j)$. A task compatibility graph $G_p(V_p, E_p)$ can be derived from the task flow graph $G(V, E)$. The vertex set V_p is in one-to-one correspondence with the vertex set V, and the undirected edge set E_p denotes the compatible task module pairs. A group of compatible task modules corresponds to a subset of vertices that are all connected by edges with each other. Such a subset of vertices forms a clique in the task compatibility graph. A maximal set of compatible task modules is identical to a maximal clique in the task compatibility graph. Minimizing the number of processors is therefore equivalent to finding the minimum number of cliques that cover the task compatibility graph. An example of a task flow graph is shown in Figure 4.1. Within each vertex, the label and the computation time of its corresponding task module are indicated by the top and bottom half portions, respectively. A computation time is assumed to be the time units consumed to execute a task module on a processor which matches its code type. Different code types are represented by different node shapes, as shown in the figure. The corresponding task compatibility graph of Figure 4.1 is shown in Figure 4.2.

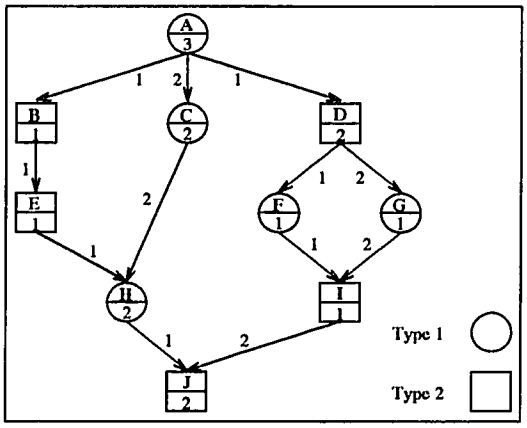


Figure 4.1 A task flow graph G.

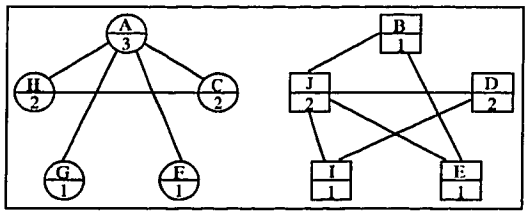


Figure 4.2 The task compatibility graph of G.

The resource sharing problem can be examined alternatively by considering the conflict between task module pairs. Two task modules are said to be conflicting if they are not compatible. A task conflict graph $G_f(V_f, E_f)$ consists of a vertex set V_f, which denotes task modules, and an edge set E_f, which denotes conflicting task module pairs. Note that the task conflict graph is the complement of the task compatibility graph as shown in Figure 4.3. Coloring the vertices of a task conflict graph provides a solution to the resource minimizing problem by assigning each color to a resource instance (processor type). Therefore, finding the minimum number of processors is equivalent to finding the minimum number of colors for coloring the vertices of a task conflict graph. Both the clique cover and vertex coloring problems have been proven to be NP-complete. In the next section, we show a greedy algorithm with polynomial time complexity for finding a suboptimal solution to the clique covering problem. For some special types of graphs, shown in Section 4.3, both the clique covering and the vertex coloring problems can be solved optimally in polynomial time. We will examine these special graphs and their utilization in minimizing the resource estimations.

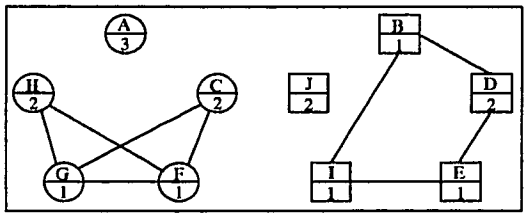


Figure 4.3 The task conflict graph of G.

4.2 The Greedy Algorithm

In this section, we present a suboptimal solution for finding cliques in polynomial time. The input to the algorithm is the task compatibility graph derived from a task flow graph as described in the previous section. We assume that the input graph G is represented by adjacency lists. The task compatibility graph may consist of one or more components. Vertices belonging to the same component have an identical code type. For example, the task compatibility graph shown in Figure 4.2 has two components which have code type one and two, respectively. S is a set of nodes forming a clique found in each iteration of the **while** loop, and C denotes a set of all cliques found by the entire algorithm. A priority queue Q is maintained to contain all the vertices in $V - S$, keyed by their degree values. The EXTRACT-MAX(Q) procedure in the algorithm is used to extract the element with the maximal key value from the priority queue Q. The algorithm first sorts vertices according to the decreasing order of their degrees. Starting from the vertex with maximal degree, which is a clique by itself, the algorithm tends to expand the size of the clique as large as possible. It then searches among the adjacent nodes of the vertex to include one of them at a time to the clique, if the clique plus the adjacent node with their edges still form a clique. This is repeated until it is not possible to include any more new adjacent nodes to the clique. The algorithm stops when all vertices of the input graph G are covered by a set of cliques. It is a greedy algorithm because it always tries to find a clique starting from the vertex with the largest value of degree. A high level description of our greedy algorithm is depicted in Figure 4.4.

To analyze the complexity of this algorithm, we denote the size of the vertex and edge sets of an input graph $G(V, E)$ by $|V|$ and $|E|$, respectively. If the graph is sparse, it is practical to implement the priority queue Q with a binary heap. For line 2, it takes $\mathcal{O}(|V| \log |V|)$ time to sort $|V|$ vertices. For each EXTRACT-MAX operation at line 7, it takes $\mathcal{O}(\log |V|)$ time, and the total worst case time complexity

```
Greedy-Clique-Cover-Algorithm(G)
 1. begin
 2.       sort the vertices of V[G] by descending order of degree
 3.       C ⟵ ∅
 4.       Q ⟵ V[G]
 5.       while Q ≠ ∅ do
 6.       begin
 7.            u ⟵ EXTRACT-MAX(Q)
 8.            S ⟵ {u}
 9.            for each vertex v ∈ Adjacent[u] do
10.            begin
11.                 if S ∪ {v} forms a clique then
12.                 begin
13.                      S ⟵ S ∪ {v}
14.                      Q ⟵ Q − {v}
15.                 end
16.            end
17.            C ⟵ C ∪ S
18.       end
19. end
```

Figure 4.4 Greedy-Clique-Cover-Algorithm

is $\mathcal{O}(|V|\log|V|)$. Note that edges (u, v) are examined exactly once at line 9, and edges (v, w), where $w \in S$, are also examined exactly once at line 11. Because u is extracted from Q at line 7 and all elements of S are removed from Q at line 14, no matter if $S \cup \{v\}$ forms a clique or not, edges (u, v) and (v, w) will not be examined again in next loop. Therefore, the running time in the **for** loop of lines 9-16 is $\mathcal{O}(|E|)$. Thus the total running time of the entire algorithm is $\mathcal{O}(|V|\log|V| + |E|)$.

4.3 Special Task Conflict and Compatibility Graphs

The algorithm presented in the last section gives a suboptimal solution for finding a minimal set of cliques in polynomial time. In this section, we show that if the input task graphs have special structures, then the clique covering or the vertex

coloring problems can be solved optimally in polynomial time. The applications of special conflict and compatibility graphs in high-level synthesis have been thorough investigated in [62]. In this section we discuss how these special graphs applying to minimizing the resource estimation for HC tasks. The special types of task conflict graphs may be interval or chordal graphs, while the task compatibility graphs may be comparability graphs. These specific graphs are detailed in the following subsections.

4.3.1 Interval Graphs

We define the lifetime of a task module to be the duration from the beginning to the end of its estimated execution based on the task flow graph. Two task modules whose lifetimes overlap and whose computational types are the same can not be assigned to the same processor. Overlapping lifetimes can be represented by the intersection between a set of continuous intervals. An intersection graph is obtained by representing each interval by a vertex and connecting two vertices by an edge if and only if their corresponding intervals overlap [46]. The intersection graph of a set of intervals along the real line is called an interval graph. Figure 4.5 illustrates an example of interval graph and its interval representation. Interval graphs can be recognized in $\mathcal{O}(|V|+|E|)$ time and colored in $\mathcal{O}(|V|\log|V|)$ time, where $|V|$ is the number of vertices and $|E|$ is the number of edges in $G(V,E)$ [62].

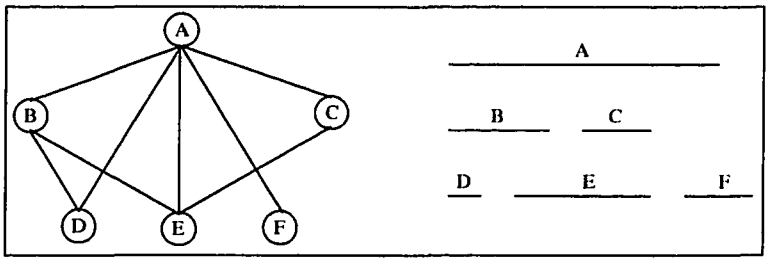


Figure 4.5 An interval graph and its interval representation [62].

4.3.2 Chordal Graphs

We explained the structure of interval graphs which are a special type of task conflict graphs. The other special type of conflict graph called chordal graph is discussed here. A graph is a chordal graph if and only if it is the intersection graph of a family of subtrees of a tree [46]. Each vertex of chordal graphs corresponds to a subtree and two vertices are connected by an edge if and only if their corresponding subtrees are intersected. Figure 4.6 [46] shows an example of chordal graph and its corresponding subtree representation. Chordal graphs are useful because they can be recognized and colored both in polynomial time. Rose et al. [57] used a lexicographic breadth-first search to recognize chordal graphs in $\mathcal{O}(|V| + |E|)$ time and Golumbic [46] presented a fast algorithm for coloring chordal graphs also in $\mathcal{O}(|V| + |E|)$ time.

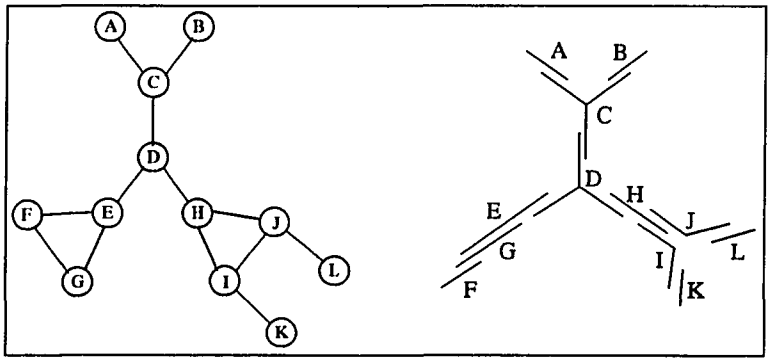


Figure 4.6 A chordal graph and its subtree representation [62].

4.3.3 Comparability Graphs

We have discussed two special types of task conflict graphs and now we turn our attention to the special type of task compatibility graphs called comparability graphs. Before presenting what a comparability graph is, we need to introduce the transitive orientation property. The transitive orientation property states that each edge of an undirected graph G can be assigned a one-way direction in such a way that the resulting oriented graph G' is closed under transitivity [46]. A comparability graph is an undirected graph which is transitively orientable. Transitive orientation of a

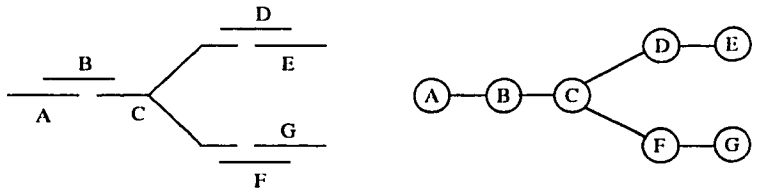
comparability graph and recognition of comparability graphs can be performed in $\mathcal{O}(|V||E|)$ time and $\mathcal{O}(|V| + |E|)$ space [46]. If the vertices of the graph, however, are linearly ordered in advance, a transitive orientation can be constructed in $\mathcal{O}(|E|)$ time. The following propositions hold for comparability graphs.

Theorem 1 *(Gilmore and Hoffman [46]) An undirected graph G is an interval graph if and only if G is a chordal graph and its complement G^{-1} is a comparability graph.*

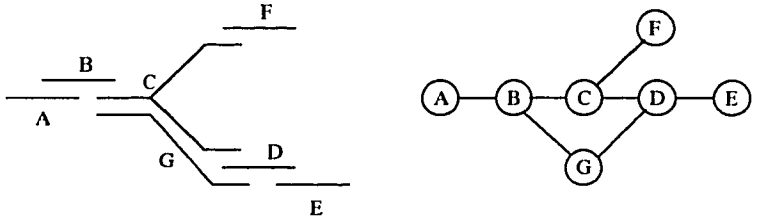
Theorem 2 *(Lekkerkerker and Boland [46]) An undirected graph G is an interval graph if and only if the following two conditions are satisfied:*

1. G is a chordal graph, and
2. any three vertices of G can be ordered in such a way that every path from the first vertex to the third vertex passes through a neighbor of the second vertex.

The set of three vertices which fail to satisfy the second condition of Theorem 2, is called an astroidal triple. Springer and Thomas [62] identifies two kinds of astroidal triples, branch and skip astroidal triples, which may result from branching of overlapping lifetime intervals. A set of intervals are called branch intervals if they consist of a branch. Branch astroidal triple arises when more than two branches include an interval that does not overlap the branch interval. A branch is called short branch if it overlaps all intervals in the branch, otherwise it is called long branch. Skip astroidal triples can be generated if a branch interval connects two long branches but does not overlap one of the short branches. Figure 4.7(a) and 4.7(b) [62] depicts examples of branch and skip astroidal triples, respectively. In Figure 4.7(a), interval C is a branch interval and vertices (A,E,G) is a branch astroidal triple; vertices (A,F,E) in Figure 4.7(b) is a skip astroidal triple. Therefore conflict graphs that are interval graphs, and the compatibility graphs which are comparability graphs, can



(a) Branch astroidal triples.



(b) Skip astroidal triples.

Figure 4.7 Two kinds of astroidal triples [62].

be obtained if the lifetimes of task modules generate no branch and skip astroidal triples.

We have studied the special cases of compatibility graph and conflict graph, called interval graph, chordal graph and comparability graph. The clique covering problem or vertex coloring problem can be solved optimally in polynomial time in these graphs. Therefore, we can estimate the resource requirements in polynomial time for heterogeneous tasks that can be represented by these special graphs.

4.4 Estimating Using Clustering Technique

Our proposed technique is based on the Cluster-M clustering methodology presented in Chapter 2. The algorithm presented in this section is an extension of the CNDG algorithm and can be used for estimating the resource requirements of a given heterogeneous task. We use clustering here to find out what are the number of processors needed if some of the subtasks are to be mapped onto the same processor. The clustering algorithm will identify the minimum number of processors required for exploring the maximum parallelism in the given task graph. The clustering of join-

node and fork-node is similar to the CNDG algorithm except that the parent (child) node chosen to be embedded must have the same computation type as the join-node (fork-node).

The proposed clustering algorithm is shown in detail in Figure 4.8. The time complexity of this algorithm is the same with the CNDG algorith, which is $O(|E_t|)$. In practice, the time complexity of this algorithm is $O(M)$ if the number of edges is proportional to the number of nodes. To illustrate this algorithm, consider the task graph of seven modules and its Spec graph as shown in Figure 4.9. Each module is labeled with its computation amount and each edge is labeled with the amount of data communication. Since the nodes embedded together are to be assigned to the same processor, we can estimate the number of processors to be the number of clusters that consists of no subclusters. For example, in Figure 4.9, three type-one processors are estimated since clusters (A, C, H), (F) and (G) are three nonseparable type-one clusters. Similarly, there are two nonseparable type-two clusters, (B, E) and (I, J), therefore two type-two processors are required to execute the two clusters.

4.5 Comparison Results

In this section we present a number of examples comparing our estimated results to the optimal minimum number of processors needed. For every example we show the number of processors of each type estimated by our algorithm, followed by the efficiency obtained in using these many processors using an optimal schedule. We then compare this with the efficiency obtained using optimal number of processors required, with the optimal schedule. To concentrate only on the goodness of our resource estimation technique, we assume the architecture is a virtual system in which processors are completedly connected, and that an unlimited number of each type of processor is available. We further assume that the bandwidth of communication links in the architecture is large enough, such that data transportation between two

```
Clustering Algorithm
begin
    divide the directed graph into a number of layers
    for each node at layer 1 do
        make it into a cluster and calculate its parameters
    for each of the other layers do
    begin
        for all edges (v_i, v_j) do
        begin
            if v_i is a fork-node then
            begin
                select a child node which has the largest edge weight and
                the same computation type as v_i
                embed the child node to v_i
                if the child nodes of v_i are not in a cluster then
                begin
                    merge them with v_i into a cluster
                    calculate the parameters of the new cluster
                end
            end
            if v_j is a join-node then
            begin
                select a parent node which has the largest edge weight and
                the same computation type as v_j
                embed v_j to the parent node
                if the parent nodes of v_j are not in a cluster then
                begin
                    merge them with v_j into a cluster
                    calculate the parameters of the new cluster
                end
            end
        end
    end
end
```

Figure 4.8 Clustering algorithm.

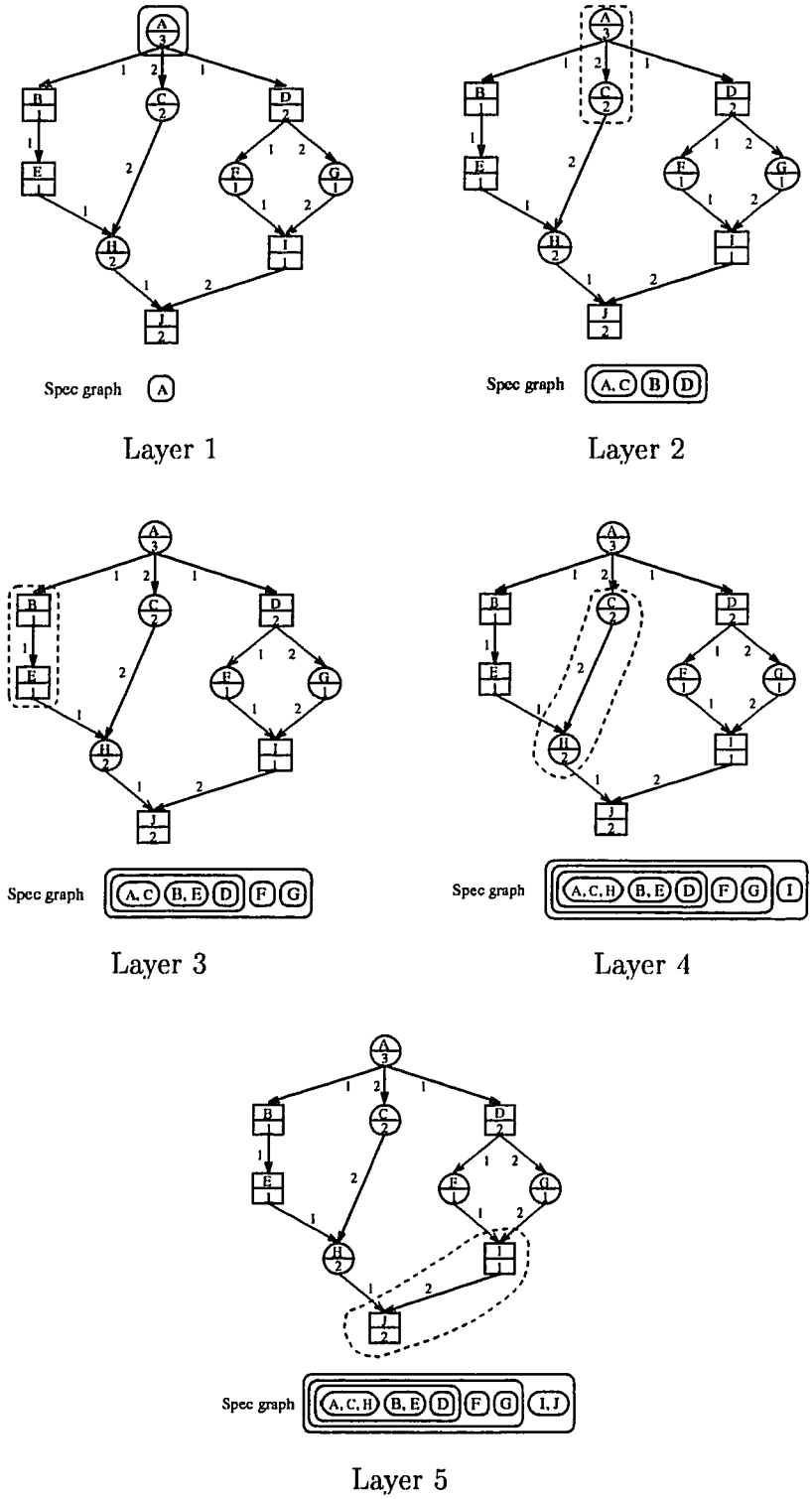


Figure 4.9 A task graph and steps for obtaining the Spec graph.

processors can be completed in one unit of time. Also, data conversion overheads between two different type of processors are ignored here. We denote the number of processor used by N and the efficiency by η.

Consider Example 1, its task flow graph and task compatibility graph are demonstrated in Figure 4.10 and 4.11 respectively. By analyzing the task compatibility graph and using our algorithm, we identified a number of cliques as shown in Figure 4.12. For the code type one, four cliques are found: {A, D, P}, {G}, {J}, {M}. There are three cliques for code type two: {N, E, K}, {B}, {H}, and also three cliques for code type three: {O, F, L}, {C}, {I}. This implied that we estimate four processors of type one, three processors of type two and three processors of type three are to be necessary for executing the task graph. In the optimal case, only two processors of each type are required to complete this task. The Gantt charts and efficiencies of the optimal schedule for using both the estimated number of processors and the optimal number of processors, are depicted in Figure 4.13. Both of them take 12 units of time to complete, therefore in (a) $\eta = \frac{11}{60}$ and (b) $\eta = \frac{11}{36}$.

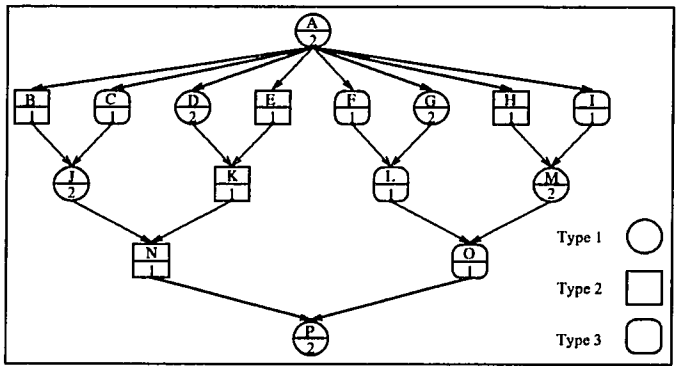


Figure 4.10 Task flow graph of Example 1.

Consider Example 2 which is more complicated than Example 1. Its task flow graph, task compatibility graph, and identified cliques are illustrated in Figure 4.14, 4.15, and 4.16, respectively. The estimated number of processor are also four type-one, three type-two and three type-three by our algorithm. Compared to the optimal

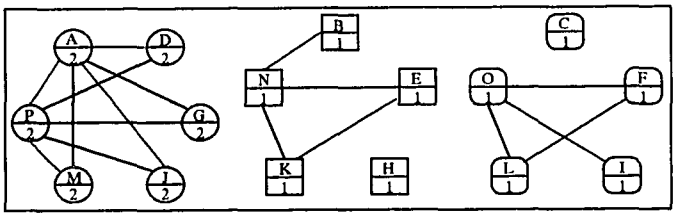


Figure 4.11 Task compatibility graph of Example 1.

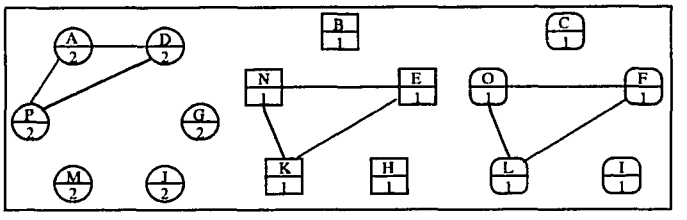
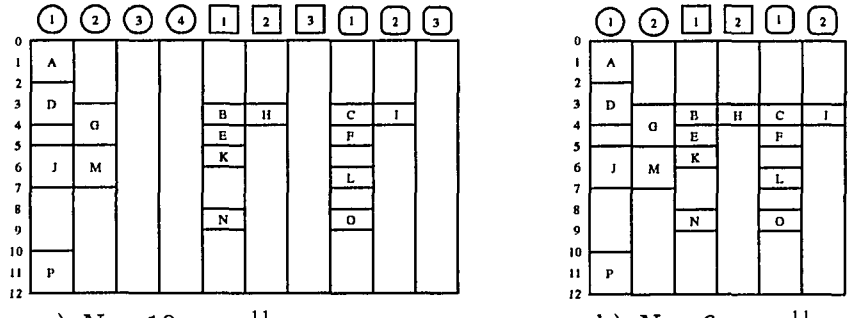


Figure 4.12 Identified cliques of Figure 4.11.

case, one type-one and one type-two processors are redundant. The Gantt charts of them are depicted in Figure 4.17 where total running time is 13 time units, therefore in (a) $\eta = \frac{27}{130}$ and (b) $\eta = \frac{27}{104}$.

Table 4.1 shows the time complexities of the different techniques and the estimated number of processors required for the heterogeneous task given in Figure 4.18. The first technique is the greedy algorithm presented for estimating minimal number of cliques in general compatibility graphs, as shown in Figure 4.19. Methods two and three are efficient solutions by exploiting a number of special



a) $N = 10, \eta = \frac{11}{60}$ b) $N = 6, \eta = \frac{11}{36}$

Figure 4.13 Gantt charts of Example 1, using a) estimated number of processors obtained by the task compatibility graph approach and b) optimal minimum number of processors.

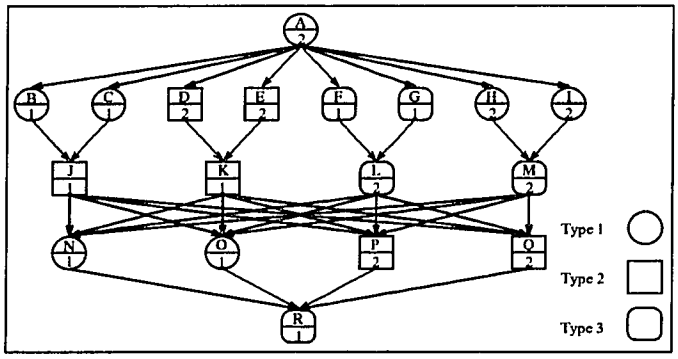


Figure 4.14 Task flow graph of Example 2.

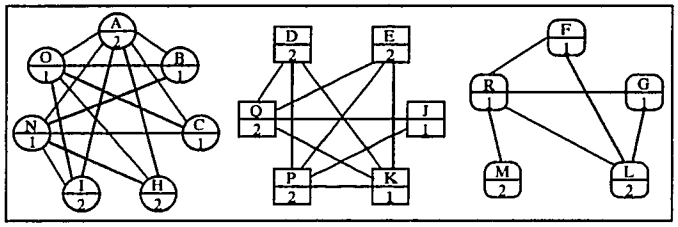


Figure 4.15 Task compatibility graph of Example 2.

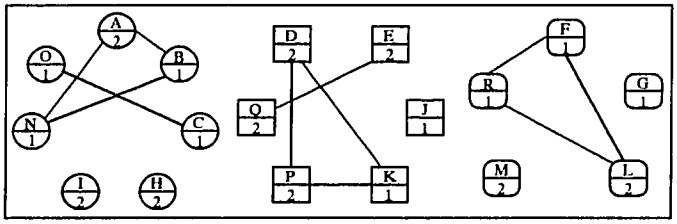
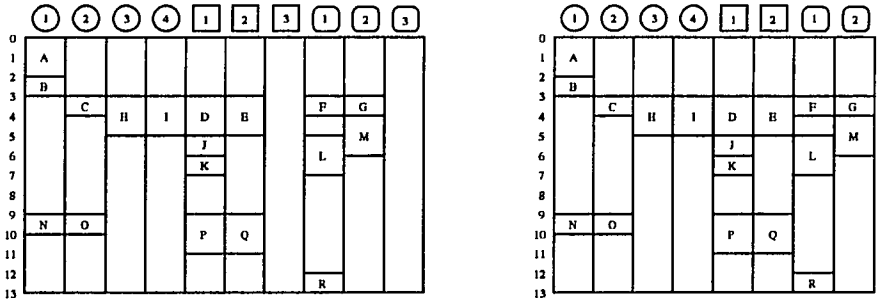


Figure 4.16 Identified cliques of Figure 10.



a) $N = 10, \eta = \frac{27}{130}$ b) $N = 8, \eta = \frac{27}{104}$

Figure 4.17 Gantt charts of Example 2, using a) estimated number of processors obtained by the task compatibility graph approach and b) optimal minimum number of processors.

structures of compatibility and conflict graphs in polynomial time. This is shown in Figure 4.20. The last one is the Cluster-M clustering technique shown in Figure 4.21.

Method	Technique	Time Complexity	Processors estimated	
			Type 1	Type 2
1	Greedy Algorithm	$\mathcal{O}(\|V\| \log \|V\| + \|E\|)$	3	3
2	Interval Graph	$\mathcal{O}(\|V\| \log \|V\|)$	3	2
3	Chordal Graph	$\mathcal{O}(\|V\| + \|E\|)$	3	2
4	Cluster-M	$\mathcal{O}(\|V\|^2)$	3	2

Table 4.1 Comparison of different resource estimating techniques.

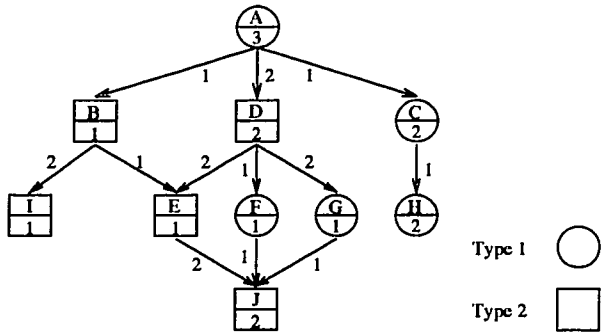


Figure 4.18 The task flow graph used for Table 4.1.

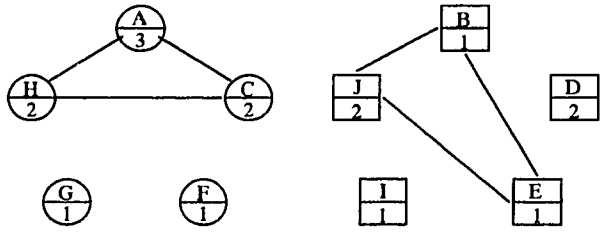


Figure 4.19 The estimated result obtaining from method 1.

4.6 Concluding Remarks

In this chapter, we presented two techniques for estimating the resource requirements for heterogeneous tasks. Using the first method, we showed that the resource

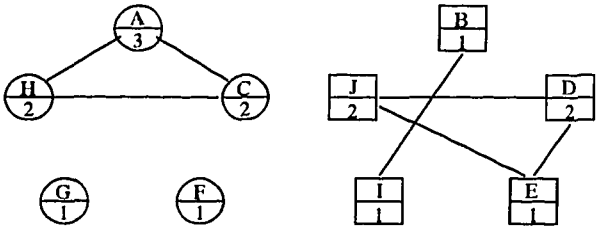


Figure 4.20 The estimated result obtaining from method 2 and method 3.

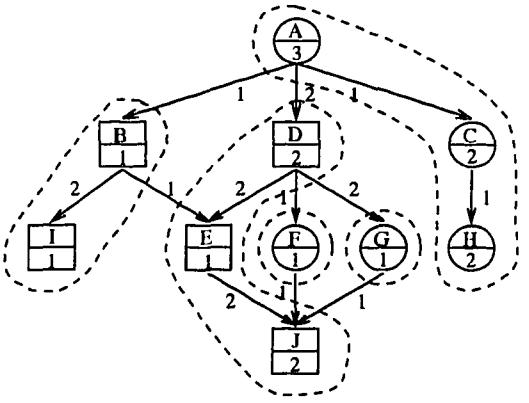


Figure 4.21 The estimated result obtaining from method 4.

minimization problem for a given task flow graph is equivalent to the minimal clique cover problem for its corresponding task compatibility graph, or the minimal coloring problem for its corresponding task conflict graph. We presented a greedy algorithm for estimating the minimum number of processors needed for each of the code types identified in a given heterogeneous task. We showed that for certain structures of task compatibility graphs, the optimal solution can be obtained in polynomial time. The second method involved using the Cluster-M nonuniform clustering methodology. A number of examples were illustrated to compare our estimations to the optimal number of processors.

CHAPTER 5

SOFTWARE REQUIREMENTS OF HETEROGENEOUS COMPUTING

A programming paradigm suitable for HC must allow the design and efficient execution of portable software so that it may be shared and/or distributed among various computers in a heterogeneous suite. Furthermore, it must support machine independent programming which does not include any architecture specific details. To meet these requirements, a programming paradigm must be both portable and scalable. Cluster-M, presented in Chapter 2, is such a paradigm which provides an environment for porting various tasks onto the machines in a heterogeneous suite such that resource utilization is maximized and the overall execution time is minimized. As described in Chapter 2, Cluster-M Specifications are high-level machine-independent programs represented in the form of Spec graph. Given a task graph, how to obtain the Spec graph was also presented. However, the Cluster-M Specification module does not have to receive a task graph as an input, rather a high-level parallel specification can be written using the Cluster-M constructs presented in this chapter. In this chapter, we first formally define scalability of heterogeneous programming paradigms. We then present a set of Cluster-M constructs which is essential for writing portable Cluster-M Specifications. Also, presented in this chapter is another portable and scalable programming paradigm, called Heterogeneous Associative Computing (HAsC)[54]. HAsC models a heterogeneous network as a coarse-grained associative computer and is designed to optimize the execution of problems where the size of the program is small compared to the amount of data processed. It uses broadcasting to avoid the mapping problem. Ease of programming and execution speed, not the utilization of idle resources are the primary goals of HAsC. We show that both paradigms are scalable. We then illustrate how these two

paradigms can be used together to provide an efficient medium for heterogeneous programming.

The rest of this chapter is organized as follows. The definitions of scalability for hardware, tasks and software in HC are presented in Section 5.1. We define the Cluster-M constructs and present an implementation of them in Section 5.2. HAsC is introduced in Section 5.3. The concurrent use of HAsC and Cluster-M is presented in Section 5.4.

5.1 Scalability

Scalability is one of the basic issues related to and addressed by both HAsC and Cluster-M, as well as many HPC (High Performance Computing) and MPP (Massively Parallel Processing) schemes. Scalability is often understood differently by different authors. For our purposes we will consider scalability to refer to hardware, tasks and software in roughly analogous fashion. In addition, scalability may refer to both homogeneous or heterogeneous architectures.

5.1.1 Homogeneous Scalability

The homogeneous case refers to multiple machines which are of the same basic architectural type, typically various-sized versions of the same vendor product. For example, an eight processor CRAY is a hardware example of a "scaled-up" version of a two-processor CRAY.

Definition 1 *We define the hardware scalability function, $\chi(a, b)$, between two homogeneous architectures, a (the larger) and b (the smaller), to be the rational-valued function giving the size multiple of a over b. In the example above, the eight-processor Cray has a scalability factor of 4 ($\chi = 4$) over the two-processor.*

Task scalability is more complex. Typically implied is the ability to take a task (algorithm plus data) executing on a small machine and execute the "same"

task on a "scaled-up" machine, utilizing additional resources of the larger machine, with performance "scaled-up" reasonably close to χ. One ambiguity in this concept is what we mean by the "same" task. If it means only executing the same program, but with possibly different (i.e. larger) data, then tasks in a homogeneous environment often "scale," particularly if the scaling factor of the data size is equal to χ.

Definition 2 *We define a type 1 task scalability function, T(a,b) for a given program applied to two different data set sizes, a (the larger) and b (the smaller) to be the rational valued function giving the size multiple of a over b. For example, if the size of a is 16K items and b is 2K items, then T=8. This means that a program is type 1 scalable if it processes data set b eight times faster than data set a, using the same hardware configuration.*

However, if we apply the above definitions to the case where both the data and the algorithm are fixed, then tasks often do not scale, not even on scaled up homogeneous hardware. To give a simple example, suppose we are computing a pixel-based imagery problem on a SIMD machine in which both the number of pixels and the number of processors is 1K. If we scaled-up to a 16K processor ($\chi = 16$), typically this task would not scale, i.e., it would not be able to exploit the additional 15K processors and we would get no increased performance. However if our original task had started with a 16K pixel problem, we would typically be able to achieve a scale up in performance, on the 16K machine over the 1K machine.

Definition 3 *We define type 2 task scalability, between two homogeneous architectures, a (the larger) and b (the smaller), to be the potential to exploit the inherent hardware scalability between them on some task of a size that fills a.*

Software scalability refers to the ability to exploit task and hardware scalability with little or no changes, other than parameters.

Definition 4 *We define the software scalability function, $\sigma(a, b)$, for the case of two homogeneous architectures, a (the larger) and b (the smaller), to be the real-valued function giving the increase in performance of a over b. Typically we expect some increase in performance but we do not generally (at least in the homogeneous case) expect "super-linear" performance, i.e., $1 \leq \sigma(a, b) \leq \chi(a, b)$. In most cases we expect σ to be a simple multiple of χ, i.e., $\sigma(a, b) = \lambda \times \chi(a, b)$, where $1/\chi(a, b) \leq \lambda \leq 1.0$. If λ is close to 1.0, i.e., $\lambda = 1 - \varepsilon$, we usually feel we have scaled up well.*

Many examples exist of scaling up in this homogeneous sense though, since it depends on a problem data size large enough to "fill" the large machine, it thus sometimes depends on an unrealistically large data set size. In particular it appears to us that some of the most recent HPC machines are "scalable" only in the sense that they could run matrix or other similar scientific problems of a size that, thus far, is not performed.

5.1.2 Heterogeneous Scalability

Heterogeneous scalability is clearly more complicated than homogeneous scalability, though it is also the case in which we can aspire the ultimate in heterogeneous computing potential, i.e, to achieve σ's significantly greater than χ. This is what we mean by super-linear performance. In the heterogeneous case, there may be no commonality between two different architectures, so "scaling" is based on the performance potential. This means, we will have two different scalability standards, namely peak MFLOPS (in either fixed 64 or 32 bit mode) or GBS ("gibbs"), billions of bits per second (processed). Using this, we can extend the χ function to the heterogeneous case. For example if we had a large vector machine, a, capable of processing 8.7 billion bits per second or 8.7 GBS, and a small SIMD machine, b, of 1.3 GBS, then $\chi(a, b) = 8.7/1.3 = 6.69$. Having extended the hardware concept of scalability to the heterogeneous case, the task and software scalability follow immediately.

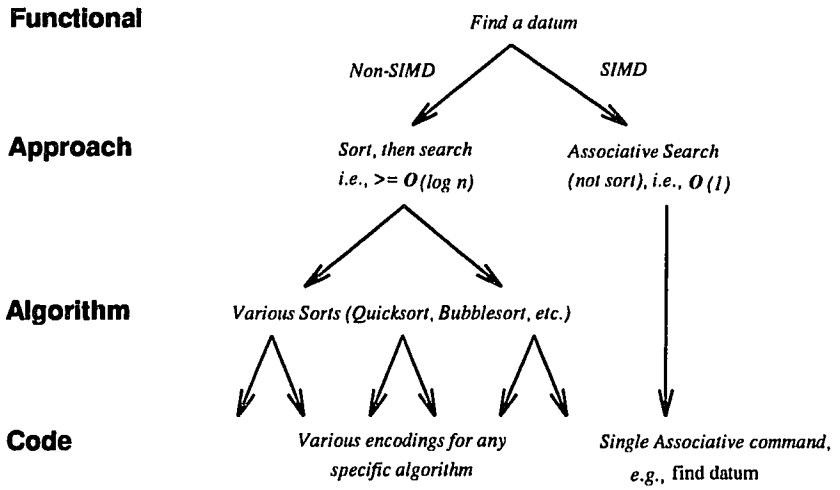


Figure 5.1 Hierarchical breakdown of a task

To understand this theorem, we need to look at Figure 5.1. We consider there to be at least four levels by which a task is defined. One is at the overall functional level, here considered to be the problem "Find a datum." Approach is the next level. By "approach" we mean something at a higher level than algorithm, perhaps a meta-algorithm. In any case, for this problem, there is a radical difference in the approach for a SIMD machine used associatively (see [54]) and non-SIMD machines. In the former case, we can use simple associative search, which is $\mathcal{O}(1)$; in the latter case we would typically use a sort, then search operation, i.e., the asymptotic performance is bounded by $\Omega(\log n)$. For the associative search on a suitable SIMD machine, there is really only one instruction "find datum", so that there is no room for differing algorithmic or code variations. However in the non-SIMD case, there are many possible variations. For example, depending on the data, parameters, architecture, etc., we could use a number of different search techniques. Similarly we could use a number of different coding schemes for each algorithm.

In this context, most researchers, when describing "scalability", do not mean that the specific code is heterogeneously scalable and generally do not mean that

the algorithm is heterogeneously scalable. For example, a matrix times a vector operation might best be done with a SAXPY style algorithm on one machine and an SDOT on another. At the same time, the term "scalability" almost never applies to the functional level in a homogeneous environment since this is far too general to have any real meaning (in the usual context of scalability). What is almost always intended is that the term "scalability" apply to the approach level. However the example above shows that this is inadequate to support efficient MPP/HPC performance. That is, a "scalable" approach to finding data would almost certainly be based on the non-SIMD, non-associative approach of "sort, then search". This might get maximal performance on non-SIMD machines and might also work on SIMDs, but never optimally. That is, the scalable approach is $\Omega(\log n)$, whereas the non-scalable SIMD version is $\mathcal{O}(1)$. This example illustrates two things:

a. It is possible to have a case where a non-scalable (at the approach level) implementation is inherently more effective than a scalable approach implemented on the same machine, and

b. It is possible to have hardware scalability one way and task/software scalability the other. Suppose the non-SIMD machine has a hardware scalability factor of κ over the SIMD, i.e., χ(non-SIMD,SIMD) $= \kappa$. However if n (the data size) is large enough, i.e., $n \geq 2^{\kappa}$, then the SIMD machine would have a task scalability OVER the non-SIMD, i.e, σ(SIMD,non-SIMD) $\geq \mathcal{O}(\log n/\kappa)$. In other words the scalable metric is inherently defective in this case. Thus we conclude:

Theorem 3 *Issues of hardware, algorithmic and software scalability at the approach, algorithm and code levels are inherently incapable of measuring the potential of HPC in heterogeneous parallel environments.*

The only kind of scalability applicable to a heterogeneous network is type 1 task scalability at the functional level. In essence **heterogeneous scalability** refers

to the property that a given software scalable program will execute efficiently on any size data set on any heterogeneous network configuration without any modification. While functional level scalability may be trivial on a homogeneous network, it is fundamental to establish a common unified programming environment for heterogeneous networks.

5.2 Cluster-M Constructs

The basic operations on the Spec clusters and their contained elements are performed by a set of constructs which form an integral part of the Cluster-M model. The following is a list and description of the constructs essential for writing Cluster-M Specifications.

- *CMAKE(LVL, ELEMENTS, x)*

  This construct creates a cluster x at level LVL which contains *ELEMENTS* as its initial elements. *ELEMENTS* is an ordered tuple of the form *ELEMENTS* $= [e_1, e_2, \cdots, e_n]$ where n is the total number of components of *ELEMENTS*. The components of *ELEMENTS* could be scalar, vector, mixed-type, or any type of data structure required by the problem.

- *CELEMENT(x, j, e)*

  This construct yields the j-th element of cluster x, and returns this element as e. If j is replaced by '-', then *CELEMENT* yields all the elements of cluster x. If x is replaced by '-', then *CELEMENT* yields all the elements of all clusters.

- *CSIZE(x, e)*

  Returns e as the number of elements of cluster x.

- *CMERGE(x, y, ELEMENTS, z)*

  This construct merges clusters x, y of level *LVL* into cluster z, $\min x, y$ of

level *LVL+1*. The elements of the new cluster are given by *ELEMENTS*. If *ELEMENTS* in *CMERGE* is replaced by '-', the elements of the new cluster are the elements of x concatenated to the elements of y.

- *CUN(op, n, x, i, e)*

  This construct applies unary operation *op* to the i-th element of cluster x, and returns the result by e. If *op* is left or right shift operation, the number of shifts is specified by n.

- *CBI(op, x, i, y, j, e)*

  This construct applies binary operation *op* to the i-th element of cluster x and the j-th element of cluster y, and returns the result by e. If i, j are replaced by '-', then the binary operation is applied to all elements of x, y.

- *CSPLIT(E, k, E1, E2)*

  This construct splits cluster E of level *LVL* at k-th element into two clusters $E1$ and $E2$.

5.2.1 Implementation of the Cluster-M Constructs

In this section, we first give a brief introduction to Program Composition Notation (PCN), a parallel programming language selected as the implementation medium for the various components of Cluster-M. We then discuss the Cluster-M constructs implemented in PCN.

5.2.1.1 Program Composition Notation (PCN) Program Composition Notation is a system for developing and executing parallel programs [14, 34]. It comprises of a high-level programming language with C-like syntax, tools for developing and debugging programs in this language, and interfaces to Fortran and C allowing the reuse of existing code in multilingual parallel programs. Programs

developed using PCN are portable across many different workstations, networks, parallel computers. The code portability aspect of PCN makes it suitable as an implementation medium for Cluster-M.

PCN focuses on the notion of program composition and emphasizes the techniques of using combining forms to put individual components (blocks, procedures, modules) together. This encourages the reuse of parallel code since a single combining form can be used to develop many different parallel programs. In addition, this facilitates the reuse of sequential code and simplifies development, debugging and optimization by exposing the basic structure of parallel programs. PCN provides a set of three core primitive composition operators: parallel, sequential, and choice composition, represented by "||", ";" and "?" respectively. More sophisticated combining forms can be implemented as user-defined extensions to this core notation. Such extensions are referred to as templates or user-defined composition operators. Program development, both with the core notation and the templates is supported by a portable toolkit. The three main components of the PCN system are illustrated in Figure 5.2.

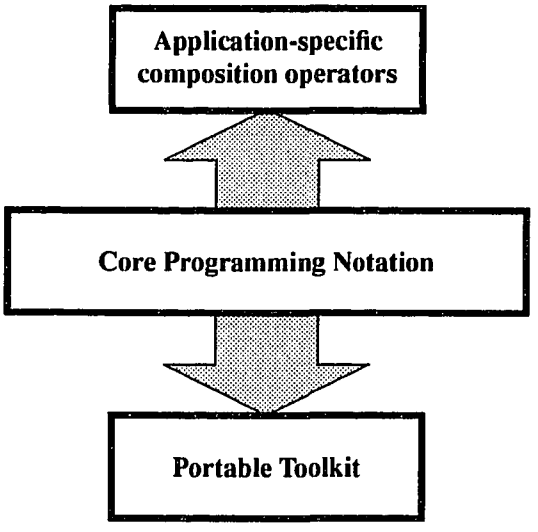


Figure 5.2 *PCN System Structure*

5.2.1.2 PCN Cluster-M Constructs The seven Cluster-M constructs are implemented in PCN as follows:

```
/* 1. Makes given elements into one cluster */
CMAKE(LVL, ELEMENTS, x)
{ || MIN_ELEMENT(ELEMENTS, n),
/* n is the smallest number in ELEMENTS */
    x = [LVL, n, ELEMENTS]
}

MIN_ELEMENT(E, n)
{; sys : list_length(E, len),
   {? len == 1 -> n = E[0],
      default -> { ? E? = [m | E1] ->
                      { ; MIN_ELEMENT1(E1, m, min),
                          n = min
                      }
   }
}

MIN_ELEMENT1(E1, m, min)
{ ? E1? = [h | E2] ->
    { ;
       { ? h < m -> m1 = h,
           default -> m1 = m
       } ,
       MIN_ELEMENT1(E2, m1, min)
    },
```

$default -> min = m$

}

/* 2. Yields an element of the cluster */

$CELEMENT(x, j, e)$

{ ; $CSIZE(x, s)$,

{ ? $j ==$ " $-$ ", $x ? = [\_, \_, x1] -> e = x1$,

$j <= s, x? = [\_, \_, x1] -> CELEMENT1(x1, j, e)$

}

}

$CELEMENT1(x, j, e)$

{ ? $j > 1 ->$

{ ? $x? = [\_|x1] ->$

$CELEMENT1(x1, j - 1, e)$,

},

$default -> e = x[0]$

}

/* 3. Yields the size of the cluster */

$CSIZE(x, s)$

{ ? $x? = [\_, \_, x2] -> CSIZE1(x2, 0, s)$,

$default -> s = 0$

}

$CSIZE1(x, acc, s)$

{ ? $x? = [\_|x1] -> CSIZE1(x1, acc + 1, s)$,

$default -> s = acc$

}

```
/* 4. Merges cluster x and y */
CMERGE(x, y, ELEMENTS, z)
{? x? = [LVL_x, _, x1], y? = [LVL_y, _, y1] - >
   {; MIN_ELEMENT(ELEMENTS, min),
      make_tuple(3, T),
      T[0] = LVL_x + 1,
      T[1] = min,
      {? ELEments == " -" - >
         {; sys : list_concat(x1, y1, xy),
            T[2] = xy
         },
         default- > T[2] = ELEMENTS
      },
      sys : tuple_to_list(T, Z, [])
   }
}

/* 5. Does the Unary operation */
CUN(op, n, x, i, e)
{; CELEMENT(x, i, e1),
   {? op == " << " - > left_shift(e1, n, e),
      op == " >> " - > right_shift(e1, n, e),
      op == "!" - > ones_complement(e1, e),
      op == "sqr" - > e = e1 * e1,
      op == " - " - > e = 0 - e1
   }
}
```

```
/* 6. Does the Binary operation */
CBI(op, x, i, y, j, e)
{; CELEMENT(x, i, e1),
   CELEMENT(y, j, e2),
   { ? op == "+" -> e = e1 + e2,
        op == "-" -> e = e1 - e2,
        op == "*" -> e = e1 * e2,
        op == "/" -> e = e1/e2,
        op == "%" -> e = e1%e2,
        op == "&" -> bitwise_and(e, e1, e2),
        op == "|" -> bitwise_or(e, e1, e2),
        op == "#" -> bitwise_xor(e, e1, e2)
   }
}

/* 7. Does the Split operation */
CSPLIT(x, k, p, q)
{ || CSIZE(x, s),
    { ? x? = [LVL, n, E] ->
        { ? k == s ->
            { || p = [LVL + 1, n, E],
                 q = [LVL + 1, 0, []],
            },
            k < s ->
            { || CSPLIT1(E, k, E1, E2),
                 MIN_ELEMENT(E1, n1),
                 MIN_ELEMENT(E2, n2),
                 p = [LVL + 1, n1, E1],
```

$q = [LVL + 1, n2, E2],$

}

}

}

}

$CSPLIT1(E, k, E1, E2)$

{ ? $k > 0 - >$

{ ? $E? = [h|t] - >$

{ || $CSPLIT1(t, k - 1, E3, E2),$

$E1 = [h|E3]$

}

},

$default - >$

{ || $E1 = [],$

$E2 = E$

}

}

5.2.2 Cluster-M Problem Specification Macros

Several operations are frequently encountered in designing parallel algorithms. Macros can be defined using basic Cluster-M constructs to represent such common operations. We next present several macros, their coding in terms of Cluster-M constructs and their PCN implementations:

5.2.2.1 Associative Binary Operation Performing an associative binary operation on N elements is a common operation in parallel applications. The Cluster-M Specification graph for input size = 8 is given in Figure 5.3. The resulting

Specification graph is an inverted tree with input values each in a leaf cluster at level 1 and the result at the root cluster at level $\log n + 1$. Using Cluster-M constructs, the macro ASSOC-BIN, written in PCN, applies associative binary operation $*$ to the N elements of input A and returns the resulting value as follows:

```
ASSOC_BIN(*, N, A)
int N, A[ ];
{ ; lvl = 0,
    make_tuple(N, cluster),
    {; i over 0 .. N − 1 ::
        { ; CMAKE(lvl, [A[i]], c),
        cluster[i] = c
      }
    },
    Binary_Op(cluster, N, op, Z)
}

Binary_Op(X, N, op, B)
int N, n;
{? N > 1− > { ; n := N/2,
                make_tuple(n, Y),
                { ; i over 0 .. n − 1 ::
                    { ; BI_MERGE(op, X[2 * i], X[2 * i + 1], Z),
                      Y[i] = Z
                    }
                } ,
                Binary_Op(Y, n, op, B)
```

$\}$,

$default- > B = X$

$\}$

$BI\_MERGE(op, X1, X2, M)$

$int\ e;$

$\{$; $CBI(op, X1, 1, X2, 1, e)$,

$CMERGE(X1, X2, [e], M)$

$\}$

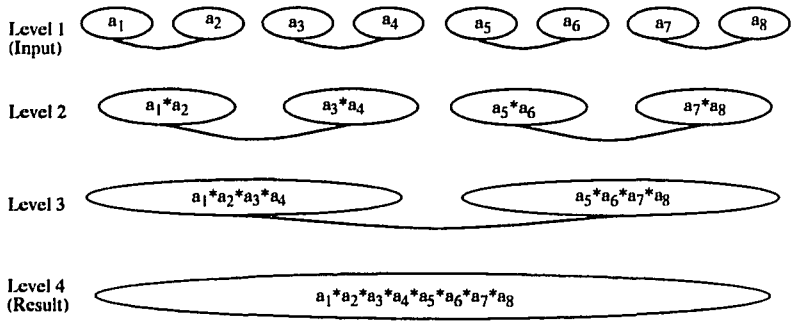


Figure 5.3 *Cluster-M Specification of associative binary macro.*

5.2.2.2 Vector Dot Product As a representative example of vector operations (Vecops), we consider here the dot product of two vectors. The vector dot product of two n-element vectors A and B is defined as $d = \sum_{i=1}^{n}(a_i \cdot b_i)$. The cluster-M Specification graph of this operation is similar to that shown in Figure 5.3. This macro can be written in terms of Cluster-M constructs and the above ASSOC-BIN macro as follows:

```
/* VECTOR DOT PRODUCT*/
DOT_PRODUCT(N, op, A, B, Z)
int N, A[ ], B[ ], C[N], e;
{; lvl = 0,
   make_tuple(N, A1),
   make_tuple(N, B1),
   {|| i over 0 .. N − 1 ::
        { ; CMAKE(lvl, [A[i]], a),
          CMAKE(lvl, [B[i]], b),
          A1[i] = a,
          B1[i] = b
        }
   } ,
   {; j over 0 .. N − 1 ::
        { ; CBI(op, A1[j], 1, B1[j], 1, e),
          C[j] := e
        }
   },
   ASSOC_BIN(" + ", N, C, Z))
}
```

5.2.2.3 SIMD Data Parallel Operations In this class of operations each operation is applied to all the input elements without any communication. In this case each operand is assigned one cluster in the problem Specification. The desired operation is applied to all clusters. The macro DATA-PAR applies operation $*$ to all N elements of input A, as follows:

$DATA\_PAR(op, n, N, A, Z)$

$int\ A[\,];$

$\{;\ lvl = 1,$

$\quad make\_tuple(N, cluster),$

$\quad \{\ ;\ i\ over\ 0\ ..\ N-1\ ::$

$\quad\quad \{\ ;\ CMAKE(lvl, [A[i]], c),$

$\quad\quad\quad cluster[i] = c$

$\quad\quad \}$

$\quad \}\ ,$

$\quad make\_tuple(N, Z),$

$\quad \{; j\ over\ 0\ ..\ N-1\ ::$

$\quad\quad \{\ ;\ CUN(op, n, cluster[j], 1, e),$

$\quad\quad\quad Z[j] = e$

$\quad\quad \}$

$\quad \}$

$\}$

5.2.2.4 Broadcast Operation This is a frequently encountered operation in parallel programs. One value is to be broadcast to all processors in the system. The problem Specification for a macro that broadcasts one value 'a' from processor x to N recipient clusters or processors, can be written in terms of Cluster-M constructs as follows:

$BROADCAST(N, e, Z)$

$\{;\ lvl = 0,$

$\quad make\_tuple(N, Z),$

$\{||\ i\ over\ 0\ to\ N-1 ::$

$\{\ ; CMAKE(lvl, [e], c),$

$Z[i] = c$

$\}$

$\}$

$\}$

The Specification graph for the broadcast operation when $N = 8$ is shown in Figure 5.4.

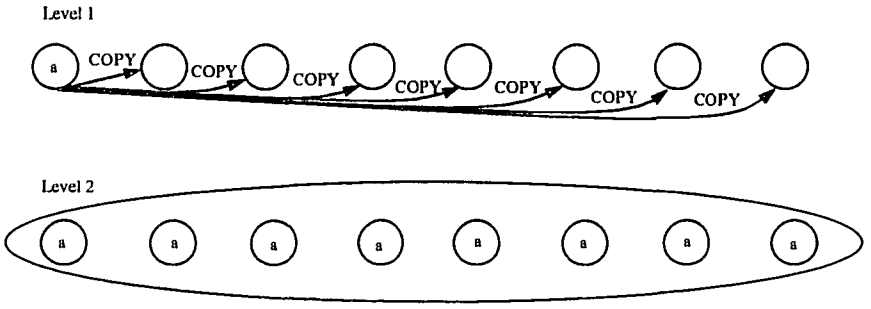


Figure 5.4 *Cluster-M Specification of broadcast macro.*

5.3 Heterogeneous Associative Computing(HAsC)

Heterogeneous Associative Computing (HAsC) models a heterogeneous network as a coarse-grained associative computer. It assumes that the network is organized into a relatively small number of very powerful nodes. Basically, each node is a supercomputer architecture (vector, SIMD, MIMD, etc). Thus each node of the network provides a unique computational capability. There may be more than one node of a specific type in the case that special properties are present. For example,

one SIMD node may be specialized for associative processing, a second SIMD node may contain a very powerful internal network configuration.

Figure 5.5 illustrates the logical similarity of an associative machine and a heterogeneous network. In particular, a disk-computer node on a network can be compared to an associative memory-PE cell. That is, as in an associative cell, the node's computer is dedicated to processing the data on the node's disk(s). The disk-to-machine data transfer rate is much more efficient than the node-to-node transfer rate. Similarly, memory-to-PE transfers are much faster than PE-to-PE transfers. Note that the associative computer and network diagrams are quite different from shared memory MIMD models. Shared memory configurations emphasize the concept that all data is equally accessible from all processors. This is not the case in a heterogeneous network.

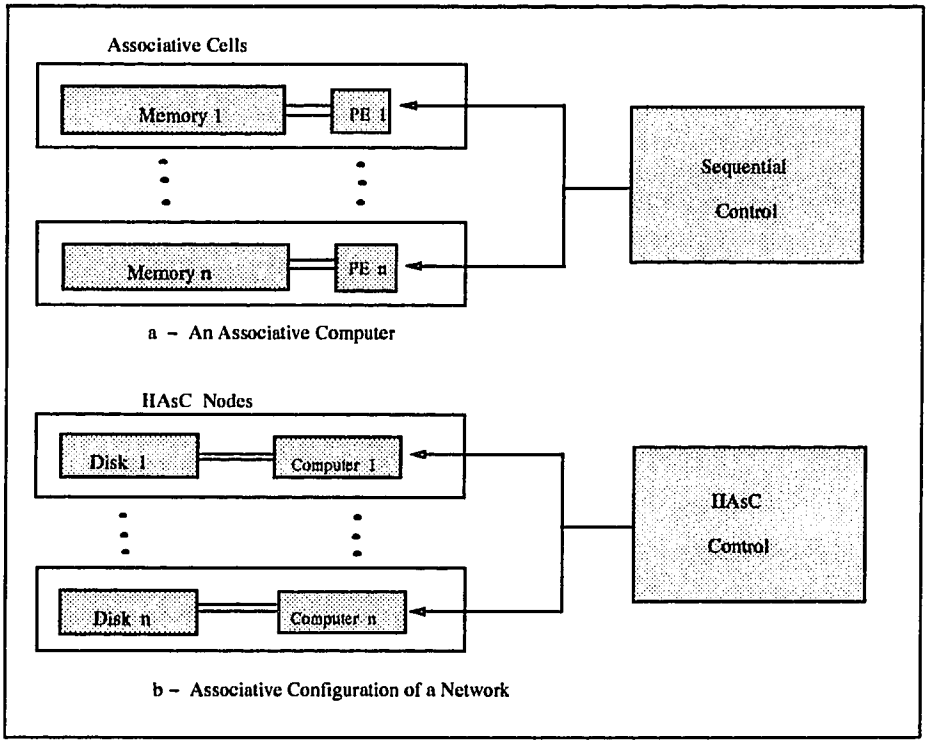


Figure 5.5 Associative Configuration of a Network.

HAsC is "layered" in that any node in the HAsC network may again be another network. Thus a HAsC node may be a HAsC cell containing more than one computer or it may be a port to another level of computing in the HAsC network. For example, most nodes may contain a general purpose computer in addition to a supercomputer to function as the node's port to the rest of the HAsC network and for file management and other support roles. Figure 5.6 shows a typical HAsC network organization. Each HAsC node has access to a number of instruction stream channels. Each channel broadcasts a different sequence of code. The HAsC node selects the appropriate channel based on its local data and previous state. The selected channel is saved in a channel register. A port, or transponder node, will accept a high level command and "translate it" into the commands appropriate for the subnetwork.

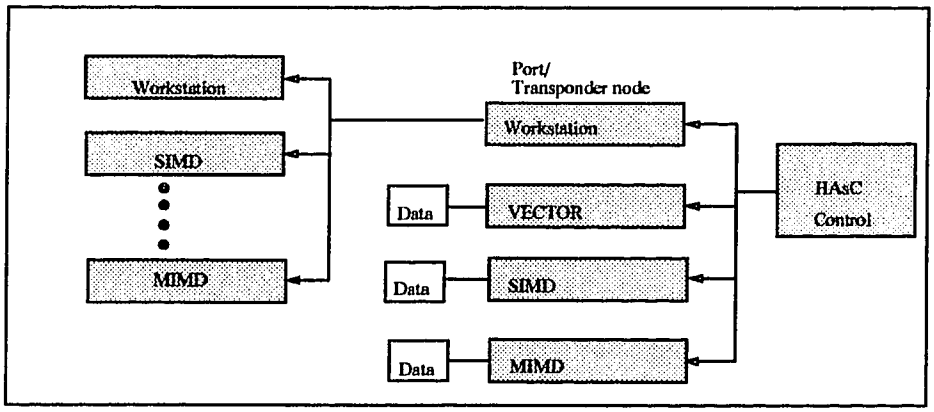


Figure 5.6 A Layered Heterogeneous Network

Some properties of the associative computing paradigm which make it well suited for heterogeneous computing are: i) efficient programming and execution with large data sets and small programs, ii) optimal data placement, iii) software scalability (see Section 5.1), iv) cellular memory allocation and v) search-process-retrieve synchronism [54].

5.3.1 Instruction Execution

In conventional machines, instructions are delivered to a CPU and are then executed without question. In HAsC, instructions are broadcast to all of the cells listening to a channel but each individual cell must determine whether or not to execute the instruction. This determination is performed as follows: Upon receipt of an instruction, a node "unifies" it with its local instruction set *and* data files.

The unification process is borrowed from Artificial Intelligence. Several languages such as Prolog and STRAND [33] incorporate the process. HAsC is different in that it uses unification only at the top level. Thus there is only one unification operation per data file, as opposed to one per record or field. This difference is critical in a heterogeneous network where communication of individual data items would be prohibitively expensive.

If there is a match, the appropriate instruction is initiated. The "instruction" may in turn issue more instructions. Thus control is distributed throughout HAsC. That is, a "program" starts by issuing a command from a control node. If a receiving node receives a command that is in effect a subroutine call, it may become a transponder control node. It may first perform some local computations and then start issuing (broadcasting) commands of its own. If the node happens to be a port node the commands are issued to its subnet as well as to its own network. Thus it is possible for multiple instruction streams to be broadcast simultaneously at several different logical network levels in a HAsC network.

In general, HAsC assumes that data is resident in a cell. As a result, data movement is minimal. However, it is common for one cell to compute a value and broadcast it to other cells. Thus, in general, there is a need to synchronize the arrival of commands and data. There are basically two cases which are handled automatically by the HAsC administrator as a part of the search-process-retrieve protocol.

The normal case is for data to be resident at a cell when the HAsC command arrives. Instruction unification and execution proceeds as described above. HAsC allows data transfers but protocol insists that the data transfer be complete before any associated commands are broadcast.

The second case involves command parameters. When a command arrives and is unified with resident data at a node, but some parameter data is missing. The unified command is then stored in a table to wait for the parameter in a synchronism process called a data rendezvous. When parameter data arrives, the rendezvous table is searched for a match. If found, the associated command is executed.

5.3.2 HAsC Administration

HAsC uses network administrators and execution engines to effect the paradigm. Each HAsC network level has a system administrator and each node in a network has its own local administrator. The local administrator monitors network traffic capturing incoming instructions and checking for illegal commands. It is also responsible for maintaining the local HAsC instruction set.

The administrator receives all incoming HAsC instructions from the local network. It then verifies if each instruction is a legal HAsC instruction. If it is, the administrator puts it in the Execution Engine queue. Otherwise, it attempts to identify the source and makes a report to the system administrator. Repeat offenses cause escalating diagnostic actions as determined by the network administrator.

If a Meta HAsC instruction such as (un)install, (un)extend or (un)augment is received, it is processed immediately. The Meta instructions will create, modify and delete HAsC instructions from the local HAsC instruction set respectively. Meta instructions can also modify local data structure definitions.

Since the instruction set can be dynamically expanded by the users, it is possible for two users to install the same instructions. The node administrator

distinguishes between the two instructions by a user *id* and a program *id* which are broadcast with every HAsC instruction.

Instructions can be added at several different logical levels: i) system, ii) project, iii) user. Typical systems level instructions would be data move and formatting commands. Project commands would be project oriented. For example, a numerical analysis project would have a matrix multiply and vector-matrix multiply instructions, while a logic programming project might have specialized logic instructions, such as unification. At the user level, one user might specify a SAXPY operation while another might want a dot product. Scalable libraries may exist at any level but most commonly at the project level.

Each node/cell has an execution engine which controls instruction execution at that node. The execution engine selects the next instruction, makes the bindings specified by instruction unification and causes the instruction to be executed. The execution engine performs the following tasks:

Save Environment
Get Next Unified Instruction
Bind Unified Variables
Establish Environment
Execute Unified Instruction
Restore Old Environment

Instruction execution may take two basic forms. First the instruction may be a HAsC program which is executed in the transponder mode. Second, the instruction may be a library call written in FORTRAN, C, LISP, etc. In this case, the established environment restrictions produces the proper interface for the appropriate language.

5.3.3 HAsC Instruction Set

This section defines the nature of the operations, the instruction format and the instruction synchronization classes of the HAsC instruction set.

HAsC is dynamic. As such, it must allow for a dynamic instruction set and data structure modifications. Thus the HAsC *install* meta instruction consist of an associative pattern and a body of code. When it is broadcast to the system, all nodes which successfully unify with the instruction gather the body of code and install it on the local node. The *extend* instruction consists of a pattern and a data definition. Responding nodes add the data definition to the local associations. *Extend* may add a named row or column to an existing association. *Augment* can be used to add an entire new association.

The patterns in these instructions contain administrative data, such as job id, project id, etc. If the node is not participating in the project or job then it does not unify and the instruction is not installed or the data definition not extended. *Uninstall*, *unextend* and *unaugment* perform the inverse operations.

Basic to the HAsC philosophy is the concept that data when initially loaded into the system is sent to the appropriate node and never moved. While this would be ideal, there will always be a need to move data from one node to another. Accordingly there are a number of HAsC move commands. Move commands can be divided into intra-association and inter-association instructions. Intra-association instructions are very much like expressions in conventional languages and are not discussed here due to lack of space. Inter-association instructions include file I/O as a special case. Inter-association *moves* must have node identifiers and for I/O, a file server, a disk or other peripheral is a legal node.

5.3.4 Associative Instruction Levels

The essence of HAsC is to model a distributed heterogeneous network as an associative data parallel computer where processor synchronization is on an instruction by instruction basis. Accordingly, in HAsC, the associative instructions are synchronized. An efficient implementation of the synchronization requires an under-

standing of how the various associative statements are mapped onto sequences of virtual machine commands and most importantly the degree of network communication complexity of the sequences.

Accordingly, this section describes a hierarchy of instructions - from the highest, most global (easiest to synchronize) to the lowest, most local (hardest to synchronize). HAsC will perform most efficiently if the programs are written using high level commands. The lower the level of the command, the more inter-node communication is required. Five different levels of instruction coupling are required to implement all of the HAsC statements on a heterogeneous network.

The communication and synchronization are built into the HAsC instruction. There is no need for the programmer to be aware of the degree of instruction communication. The five levels of instructions are presented here to more clearly delineate the relationship between associative and heterogeneous computing.

The highest level of instruction synchronization is pure associative data parallelism and involves the use of the channel registers only - i.e. there is no global coupling. There are two types of top level instructions: i) those which execute based on the channel register value only, such as logical and arithmetic expressions and ii) those which set the channel register. Data parallel logical expressions (associative searchers) can be used to set the channel registers and are "automatically" incorporated into many HAsC statements. Thus a data parallel IF or WHERE consists of only an associative search, followed by a sequence of data parallel expressions. It is a top level instruction. Top level instructions execute in real time and require no global response or communication. Most computation is done at the top level.

Figure 5.7 gives some examples of instruction synchronization. In Figure 5.7, $ is the parallel marker and is read as a plural. That is, A$ is read as As. Result$ is a data parallel pronoun referring to the last performed data parallel computation. The

top level synchronization box shows the programming style for algebraic expressions supported by HAsC.

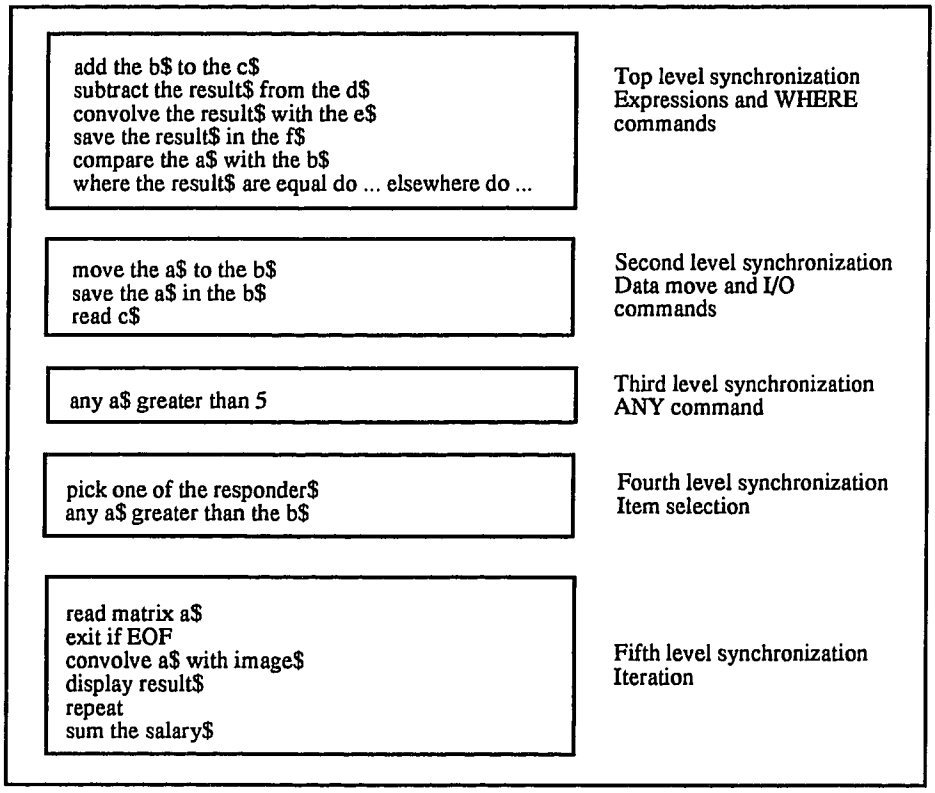


Figure 5.7 Instruction Synchronization

The second level of instruction coupling requires only global synchronism. Prime examples are the data transfer and I/O commands. I/O is always local to a cell's processor, but in general the processors may be quite different physically. Therefore I/O times may vary dramatically requiring synchronization before the next HAsC command is issued. Again, the programmer need not be aware of the synchronization requirements of this class of instructions. The synchronization is automatic. The programmer only recognizes the need for I/O or data movement.

The third level of complexity consists of simple responder commands. These commands require the ORing of the responder results of all processors (i.e. an OR

reduction). On a SIMD this is a single instruction. In HAsC, it is the simplest form of a HAsC reduction communication. The instructions at this level, such as ANY, are used to check for error conditions or determine whether special case computing needs to be done.

The fourth level is random selection. The HAsC commands in Figure 5.7 at this level consist of an associative search, followed by the selection of a responder by the "first reduction" operation. The data object of the selected responder is broadcast to the entire HAsC network for further processing.

The fifth level is iteration. The only use for iteration at the top level of HAsC is for user interaction. For example, a typical program might be one which allows the user to interactively specify kernels to be convolved with an image and to review the results, as shown in Figure 5.7. Data iteration does not exist.

HAsC is a programming paradigm designed to facilitate the utilization of heterogeneous networks. The parallel associative programming techniques are well suited for this purpose.

5.4 Cluster-M and HAsC

As described in the previous sections, HAsC is most suitable for coarse-grained heterogeneous parallel computing. It is intended to ease the programming effort and maximize execution speed, at the expense of resource balancing. Cluster-M, on the other hand, provides both coarse-grained and fine-grained mapping in a clustered fashion. It aims at maximizing both execution speed as well as resource utilization. Therefore, both paradigms can be combined to achieve a better overall performance featuring ease of programming, increase execution speed and optimal resource utilization.

5.4.1 Concurrent use of Cluster-M and HAsC

Cluster-M mapping can be applied to HAsC in several ways. First, Cluster-M can be used to determine the initial data mapping before HAsC computation begins so that the overall execution time is minimized. Secondly, Cluster-M mapping can be used to decide the fine-grained mapping of HAsC nodes as shown in Figure 5.8. Thirdly, Cluster-M can be alternated with HAsC at run time. In this approach, a Cluster-M Specification for the task is generated first. The Cluster-M Specification preserves computation and communication information in a multi-level cluster organization. Clusters at the same level represent computations at a given step which can be executed concurrently. This cluster organizational information can be sent to the HAsC network controller which then broadcasts the clusters of HAsC instructions (Figure 5.9). As described in section 5.3, the local HAsC nodes determine which of the clusters to execute based on their local configuration and data. Global results, if any, are returned to the initiating HAsC controller which may use them to select the next level of clusters to be broadcast. The process repeats until all cluster levels have been processed. This approach is a network implementation of the multiple-SIMD architecture originally described in [54].

5.4.2 Scalability of Cluster-M and HAsC

Both programming paradigms presented in this paper are machine-independent as explained in detail and are therefore heterogeneously scalable. In HAsC, a program is broadcast to the entire network and the individual nodes determine locally which instructions to execute. The global broadcasting approach means that there is no need to know how nodes are interconnected in the network or how data is distributed across the nodes. This allows data files to be analyzed dynamically at run time as they enter the HAsC system and to be directed to the node(s) (i.e. computers) best suited to process them. Broadcasting allows scalability. That is, the hardware

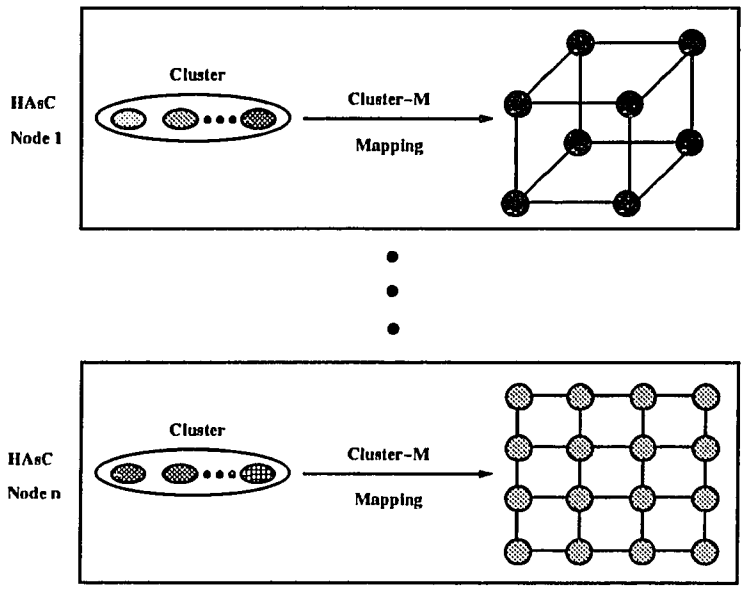


Figure 5.8 Cluster-M aided HAsC computation within HAsC nodes

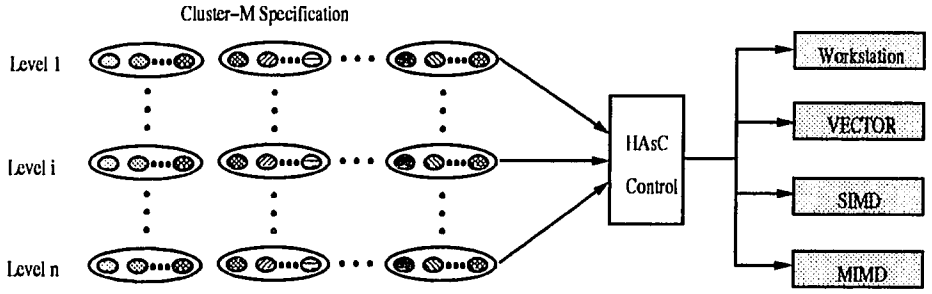


Figure 5.9 Switching between Cluster-M and HAsC

can be expanded or modified and the problem size can be changed without having to reprogram or recompile the basic HAsC program. New nodes consisting of new machines with installed HAsC software can be added to a network at any time and at any location. HAsC is not dependent on any physical machine or network configuration. This is because the instruction broadcast, cell memory organization and associative searching allows the removal of any reference to data set size and type from the program.

Cluster-M is also scalable. When a new machine is added to the heterogeneous network a new Cluster-M representation of the suite can be generated and a Cluster-

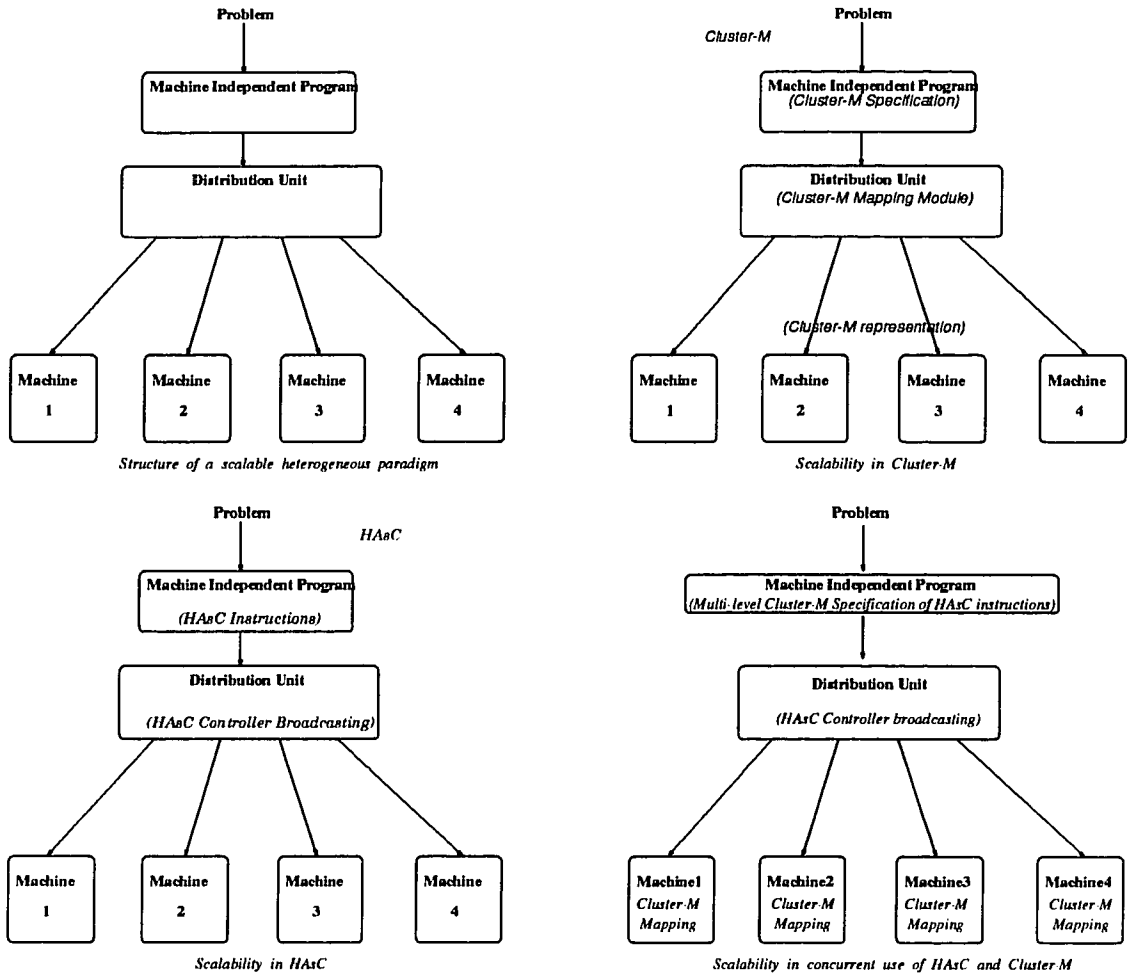


Figure 5.10 Scalability of HAsC and Cluster-M

M specification can be efficiently executed without any change. Also, an appropriate new mapping function can be computed to map the Cluster-M specification to the new Cluster-M representation. Furthermore, the two paradigms can be used concurrently as a hybrid scalable programming paradigm. See Figure 5.10 for an illustration of above.

CHAPTER 6
CONCLUDING REMARKS

In this thesis, we have discussed some theory and design issues of heterogeneous computing. We have presented a heterogeneous model of computation which can efficiently bridge the software/hardware gap in a heterogeneous environment. This model allows software portability without imposing any restrictions on the hardware. Furthermore, it allows a mechanism for predicting the performance of a given parallel program on any heterogeneous computer or suite of computers. Our Cluster-M model consists of two sets of parameters, one for representing a portable parallel program and the other for specifying the organization of the underlying heterogeneous architecture/suite. In addition, the Cluster-M model consists of an evaluation function for predicting the time performance of any two sets of parameters being considered. A tool implementing the proposed heterogeneous model of computation called Cluster-M was presented to support portable parallel algorithm design and programming. The Cluster-M tool provides a mechanism such that both sets of parameters can be extracted from any given problem and any underlying heterogeneous organization. Furthermore, it provides an efficient technique for mapping these portable programs onto heterogeneous systems using these two sets of parameters. The Cluster-M mapping algorithm, presented in Chapter 2, is the first generic algorithm for mapping nonuniform arbitrary task graphs onto nonuniform arbitrary system graphs. Given a task graph and a system graph, we have shown efficient techniques for producing the Spec and Rep graphs. These two graphs are then input to the mapping algorithm. The clustering is done only once for a given task (system) graph independent of any system (task) graphs. It is a machine-independent (application-independent) clustering and is not distinct for different mappings.

The process of the mapping algorithm presented in Chapter 2 is performed recursively by a greedy fashion matching the clusters of the task graphs (Spec

clusters) to the clusters of the system graphs (Rep clusters). In Chapter 3, we have used an extended version of the algorithm to incorporate the "type heterogeneity" (i.e., SIMD and MIMD) of tasks and systems in HC. The augmented mapping algorithm presented first maps Spec clusters to Rep clusters of similar computation type and then proceeds with an enhanced fine-grain mapping technique. Since the expected number of clusters at every level of the fine-grain mapping is constant, we have used an optimal matching strategy to enhance the algorithm. Therefore, we have formulated and solved each step of the fine-grain cluster mapping by using an integer linear programming model. We have compared the mapping results of our algorithm with those of some other heterogeneous mapping techniques.

In Chapter 4, we have proposed two methods for estimating the minimum number of processors needed for each of the code types identified in a given heterogeneous task. The input to the first method is a task compatibility graph. We have shown that a task compatibility graph can be generated by analyzing certain compatible relations between task module pairs of a given task flow graph. We have defined the resource (processor) minimization problem to be equivalent to finding the minimal number of cliques that cover the task compatibility graph, or to finding the minimal number of colors that color the vertices of its complement graph, called the task conflict graph. We estimated this using a greedy approach in $\mathcal{O}(|V| \log |V| + |E|)$ time, where $|V|$ and $|E|$ are the number of vertices and edges of the task compatibility graph. We have shown that for certain types of task compatibility graphs optimal solutions can be obtained in polynomial time. The second method proposed was using the Cluster-M methodology [25, 15]. We have presented examples comparing our estimated results to the optimal number of processors needed.

In Chapter 5, we have presented the collaboration of two heterogeneous programming paradigms, Cluster-M and HAsC. HAsC models a heterogeneous network as a coarse-grained associative computer. In HAsC a program is broadcast

to the entire network, the individual node then determines which instruction to execute. Cluster-M also allows scalability since programs written using Cluster-M are machine-independent and can be efficiently mapped and ported among different systems. A definition of scalability suitable for heterogeneous networks has been developed. HAsC and Cluster-M have been shown to be both heterogeneously scalable.

REFERENCES

1. G. Agha. *Actors: A Model of Concurrent Computations in Distributed Systems.* The MIT Press, Cambridge, MA, 1986.

2. G. Agha, C. Houck, and R. Panwar. Distributed execution of Actor systems. In *Proceedings of Fourth Workshop on Languages and Compilers for Parallel Computing*, Santa Clara, CA, 1991.

3. G. Agha and R. Panwar. An Actor-based framework for heterogeneous computing systems. In *Proc. Workshop on Heterogeneous Processing*, pages 35–42, March 1992.

4. A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, Reading, MA, 1974.

5. E. Arnould, F. Bitz, E. Cooper, H. T. Kung, R. Sansom, and P. Steenkiste. The design of Nectar: A network backplane for heterogeneous multicomputers. *ACM*, pages 205–216, 1989.

6. F. Berman. Experience with an automatic solution to the mapping problem. In L. H. Jamieson, D. B. Gannon, and R. J. Douglass, editors, *The Characteristics of Parallel Algorithms*, pages 307–334. MIT Press, Cambridge, MA, 1987.

7. F. Berman and L. Snyder. On mapping parallel algorithms into parallel architectures. *Journal of Parallel and Distributed Computing*, pages 439–458, April 1987.

8. F. Berman and B. Stramm. Prep-P: Evolution and overview. Technical report cs89-158, Dept. of Computer Science, University of California at San Diego, CA, 1987.

9. S. H. Bokhari. On the mapping problem. *IEEE Transaction on Computers*, c-30(3):207–214, March 1981.

10. S. H. Bokhari. Partitioning problem in parallel, pipelined, and distributed computing. *IEEE Trans. on Computers*, 37(1):48–57, January 1988.

11. L. Borrman, M. Herdieckerhoff, and A. Klein. Tuple space integrated into Modula-2, implementation of the Linda concept on a hierarchical multiprocessor. In *Proc. CONPAR*, 1988.

12. B. Buckle and D. Hardin. Partitioning and allocation of logical resources in a distributed computing environment. In *Tutorial : Distributed System Design*, pages 151–1. IEEE Compu. Soc. EHO, 1979.

13. N. Carriero, D. Gelernter, and J. Leichter. Distributed data structures in Linda. In *Proc. Thirteenth ACM Symposium on Principles of Programming Languages*, January 1986.

14. K. M. Chandy and S. Taylor. *An Introduction to Parallel Programming.* Jones and Bartlett Publishers, Boston, MA, 1992.

15. S. Chen and M. Eshaghian. A fast recursive mapping algorithm. *Concurrency: Practice and Experience*, 7(5):391–409, August 1995.

16. S. Chen, M. Eshaghian, A. Khokhar, and M. Shaaban. A selection theory and methodology for heterogeneous supercomputing. In *Proc. Workshop on Heterogeneous Processing*, pages 15–22, April 1993.

17. S. Chen, M. M. Eshaghian, and Y. Wu. Mapping arbitrary non-uniform task graphs onto arbitrary non-uniform system graphs. In *1995 International Conference on Parallel Processing*, volume II, pages 191–195, August 1995.

18. W. Chu, L. Holloway, M. Lan, and K. Efe. Task allocation in distributed systems. *Computer*, pages 57–59, November 1980.

19. D. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. von Eicken. LogP: Towards a realistic model of parallel computation. In *Proc. 4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, May 1993.

20. J. C. DeSouza-Batista, M. Eshaghian, A. C. Parker, S. Prakash, and Y. Wu. A sub-optimal assignment of application tasks onto heterogeneous systems. In *Proc. Heterogeneous Computing Workshop*, April 1994.

21. K. Efe. Heuristic models of task assignment scheduling in distributed systems. *IEEE Computer*, 15(6):50–56, 1982.

22. H. El-Rewini and T. G. Lewis. Scheduling parallel program tasks onto arbitrary target machines. *Journal of Parallel and Distributed Computing*, pages 138–153, September 1990.

23. H. El-Rewini, T. G. Lewis, and H. H. Ali. *Task Scheduling in Parallel and Distributed Systems.* Prentice Hall, Englewood Cliffs, NJ, 1994.

24. F. Ercal, J. Ramanujam, and P. Sadayappan. Task allocation onto a hypercube by recursive mincut bipartitioning. *Journal of Parallel and Distributed Computing*, pages 33–44, October 1990.

25. M. Eshaghian and M. Shaaban. Cluster-M parallel programming paradigm. *International Journal of High Speed Computing*, 6(2):287–309, June 1994.

26. M. M. Eshaghian. *Parallel Computing with Optical Interconnects.* PhD thesis, Dept. of Electrical Engineering-Systems, University of Southern California, Los Angeles, CA, 1988.

27. M. M. Eshaghian and R. F. Freund, editors. *Proc. Workshop on Heterogeneous Processing.* IEEE Computer Society Press, Los Alamitos, CA, April 1993.

28. M. M. Eshaghian and R. F. Freund, editors. *Proc. Workshop on Heterogeneous Computing.* IEEE Computer Society Press, Los Alamitos, CA, April 1994.

29. Mary M. Eshaghian. Parallel algorithms for image processing on OMC. *IEEE Transactions on Computers*, 40(7):827–833, July 1991.

30. D. Fernandez-Baca. Allocating modules to processors in a distributed systems. *IEEE Transactions on Software Engineering*, 15(11):1427-1436, November 1989.

31. M. J. Flynn. Very high-speed computing systems. In *Proc. IEEE*, volume 54, pages 1901–1909, 1966.

32. S. Fortune and J. Wyllie. Parallelism in random access machines. In *Proc. 10th Annual Symposium on Theory of Computing*, pages 114–118, 1978.

33. I. Foster and S. Taylor. *STRAND, New Concepts in Parallel Programming.* Prentice Hall, Englewood Cliffs, NJ, 1975.

34. I. Foster and S. Tuecke. Parallel programming with PCN. Technical report, Argonne National Laboratory, University of Chicago, IL, January 1993.

35. R. Freund. Superconcurrent processing a dynamic approach to heterogeneous parallelism. In *Proceedings of the Parallel/Distributed Computing Networks Seminar*, February 1990.

36. V. Gylys and J. Edwards. Optimal partitioning of workload for distributed systems. In *Tutorial : Distributed System Design*, pages 151-1. IEEE Compu. Soc. EHO, 1979.

37. T. C. Hu. Parallel sequencing and assembly line problems. *Operations Research*, 9(6):841–848, 1961.

38. B. Narahari L. Tao and Y. C. Zhao. Heuristics for mapping parallel computations to heterogeneous parallel architectures. In *Proc. Workshop on Heterogeneous Processing*, pages 36–41, April 1993.

39. J. Lawson and M. Mariani. Distributed data processing system design - A look at the partitioning problem. In *Tutorial : Distributed System Design*, pages 151-1. IEEE Compu. Soc. EHO, 1979.

40. C. Leangsuksun and J. Potter. Problem representation for an automatic mapping algorithm on heterogeneous processing environment. In *Proc. Workshop on Heterogeneous Processing*, pages 48–56, April 1993.

41. C. Leangsuksun and J. Potter. Designs and experiments on heterogeneous mapping heuristics. In *Proc. Workshop on Heterogeneous Computing*, pages 17–22, April 1994.

42. C. Leangsuksun, J. Potter, and S. Scott. Dynamic task mapping algorithms for a distributed heterogeneous computing environment. In *Proc. Workshop on Heterogeneous Computing*, pages 30–34, April 1995.

43. S. Lee and J. K. Aggarwal. A mapping strategy for parallel processing. *IEEE Transactions on Computers*, 36:433–442, April 1987.

44. V. M. Lo. Heuristic algorithms for task assignment in distributed systems. *IEEE Transactions on Computers*, C-37(11):1384–1397, November 1988.

45. V. M. Lo, S. Rajopadhye, S. Gupta, D. Keldsen, M. A. Mohamed, and J. A. Telle. Oregami: Software tools for mapping parallel computations to parallel architectures. In *Proc. International Conference on Parallel Processing*, 1990.

46. M. C. Golumbic. *Algorithmic Graph Theory and Perfect Graphs.* Academic Press, New York, NY, 1980.

47. J. Mahdavi, G. L. Huntoon, and M. B. Mathis. Deployment of a HIPPI-based distributed supercomputing environment at the Pittsburgh Supercomputing Center. In *Proc. Workshop on Heterogeneous Processing*, pages 93–96, March 1992.

48. C. McCreary and H. Gill. Automatic determination of grain size for efficient parallel processing. *Communications of ACM*, 32(9):1073–1078, September 1989.

49. M. C. McFarland, A. C. Parker, and R. Camposano. The high-level synthesis of digital systems. *Proceedings of the IEEE*, 78(2):301–318, February 1990.

50. G. De Micheli. High level synthesis of digital circuits. Technical Report CSL-TR-92-551, Computer Systems Laboratory, Department of Electrical Engineering and Computer Science, Stanford University, Stanford, CA 94305-4055, November 1992.

51. B. Narahari, A. Youssef, and H. Choi. Matching and scheduling in a generalized optimal selection theory. In *Proc. Heterogeneous Computing Workshop*, pages 3–8, April 1994.

52. D. Notkin, A. Black, E. Lazowska, H. Levy, J. Sanislo, and J. Zahorjan. Interconnecting heterogeneous computer systems. *Communications of the ACM*, 31(3):258–273, 1988.

53. R. Ponnusamy, J. Saltz, R. Das, C. Koelbel, and A. Choudhary. A runtime data mapping scheme for irregular problems. In *Proc. Scalable High Performance Computing Conference*, pages 216–219, May 1992.

54. J. L. Potter. *Associative Computing.* Plenum Press, New York, NY, 1992.

55. S. Prakash. *Synthesis of Application-Specific Multiprocessor Systems.* PhD thesis, Electrical Engineering-Systems Department, University of Southern California, Los Angeles, CA 90089-2562, January 1993.

56. S. Prakash and A. C. Parker. A design method for optimal selection of application-specific heterogeneous multiprocessor systems. In *Proc. Workshop on Heterogeneous Processing*, pages 75–80, April 1992.

57. D. J. Rose, R. E. Tarjan, and G. S. Leuker. Algorithmic aspects of vertex elimination on graphs. *SIAM Journal of Computing*, 5:266–283, 1976.

58. A. L. Rosenberg. Needed: A theoretical basis for heterogeneous parallel computing. Unpublished manuscript, 1994.

59. V. Sarkar. *Partitioning and Scheduling Parallel Programs for Execution on Multiprocessors.* The MIT Press, Cambridge, MA, 1989.

60. C. Shen and W. Tsai. A graph matching approach to optimal task assignment in distributed computing systems using a minmax criterion. *IEEE Transactions on Computers*, c-34(3):197–203, March 1985.

61. J. Sinclair. Effectient computation of optimal assignments for distributed tasks. *Journal of Parallel Distributed Compu.*, 4(4):342–362, 1987.

62. D. L. Springer and D. E. Thomas. Exploiting the special structure of conflict and compatibility graphs in high-level synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 13(7):843 856, July 1994.

63. H. S. Stone. Multiprocessor scheduling with the aid of network flow algorithms. *IEEE Transactions on Software Engineering*, SE-3(1):85–93, January 1977.

64. H. S. Stone. Critical load factors in distributed systems. *IEEE Transactions on Software Engineering*, SE-4:254–258, May 1978.

65. V. Sunderam and R. F. Freund, editors. *Proc. Workshop on Heterogeneous Computing.* IEEE Computer Society Press, Los Alamitos, CA, April 1995.

66. V. S. Sunderam. PVM: A framework for parallel distributed computing. *Concurrency: Practice and Experience*, 2(4):315–339, December 1990.

67. 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. In *Proc. Workshop on Heterogeneous Computing*, pages 109–120, April 1995.

68. C. Tseng and D. P. Siewiorek. Automated synthesis of data paths in digital systems. *IEEE Transactions on CAD*, CAD-5(3):379–395, July 1986.

69. L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, August 1990.

70. R. J. Vetter, D. H. C. Du, and A. E. Klietz. Network supercomputing: Experiment with a Cray-2 to CM-2 HIPPI connection. In *Proc. Workshop on Heterogeneous Processing*, pages 87–92, March 1992.

71. M. Wang, S. Kim, M. Nichols, R. Freund, and H. J. Siegel. Augmenting the optimal selection theory for superconcurrency. In *Proc. Workshop on Heterogeneous Processing*, pages 13–21, March 1992.

72. U. Warrier and C. Sunshine. A platform for heterogeneous interconnection network management. *IEEE Journal on Selected Areas in Communications*, 8(1):119–126, January 1990.

73. P. H. Winston. *Artificial intelligence*. Addison-Wesley, Reading, MA, 2nd edition, 1984.

74. M. Y. Wu and D. Gajski. Hypertool: A programming aid for message-passing systems. *IEEE Transactions on Parallel and Distributed Systems*, 1(3):101–119, 1990.

75. T. Yang and A. Gerasoulis. A parallel programming tool for scheduling on distributed memory multiprocessors. In *Proc. IEEE Scalable High Performance Computing Conference*, April 1992.

76. T. Yang and A. Gerasoulis. DSC: Scheduling parallel tasks on an unbounded number of processors. *IEEE Transactions on Parallel and Distributed Systems*, 5(9):951–967, September 1994.