Spring 1995

Mapping of portable parallel programs

Song Chen
New Jersey Institute of Technology

UMI Number: 9539580

UMI

300 North Zeeb Road
Ann Arbor, MI 48103

ABSTRACT

MAPPING OF PORTABLE PARALLEL PROGRAMS

by
Song Chen

An efficient parallel program designed for a parallel architecture includes a detailed outline of accurate assignments of concurrent computations onto processors, and data transfers onto communication links, such that the overall execution time is minimized. This process may be complex depending on the application task and the target multiprocessor architecture. Furthermore, this process is to be repeated for every different architecture even though the application task may be the same. Consequently, this has a major impact on the ever increasing cost of software development for multiprocessor systems. A remedy for this problem would be to design portable parallel programs which can be mapped efficiently onto any computer system. In this dissertation, we present a portable programming tool called Cluster-M. The three components of Cluster-M are the Specification Module, the Representation Module, and the Mapping Module. In the Specification Module, for a given problem, a machine-independent program is generated and represented in the form of a clustered task graph called Spec graph. Similarly, in the Representation Module, for a given architecture or heterogeneous suite of computers, a clustered system graph called Rep graph is generated. The Mapping Module is responsible for efficient mapping of Spec graphs onto Rep graphs. As part of this module, we present the first algorithm which produces a near-optimal mapping of an arbitrary non-uniform machine-independent task graph with M modules, onto an arbitrary non-uniform task-independent system graph having N processors, in $O(MP)$ time, where $P = \max(M, N)$. Our experimental results indicate that Cluster-M produces

better or similar mapping results compared to other leading techniques which work only for restricted task or system graphs.

MAPPING OF PORTABLE PARALLEL PROGRAMS

by
Song Chen

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 1995

APPROVAL PAGE

MAPPING OF PORTABLE PARALLEL PROGRAMS

Song Chen

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

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

Dr. James McHugh, Committee Member — Date
Director of Ph.D. Program in Computer Science
Professor of Computer and Information Science, NJIT

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

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

BIOGRAPHICAL SKETCH

Author: Song Chen

Degree: Doctor of Philosophy

Date: May 1995

Undergraduate and Graduate Education:

- Doctor of Philosophy in Computer Science,
  New Jersey Institute of Technology, Newark, New Jersey, 1995
- Master of Science in Systems Engineering,
  Shanghai Jiao Tong University, Shanghai, P. R. China, 1990
- Bachelor of Science in Computer Science,
  East China Normal University, Shanghai, P. R. China, 1987

Major: Computer Science

Presentations and Publications:

S. Chen and M. M. Eshaghian, "A Fast Recursive Mapping Algorithm," to appear in Concurrency: Practice and Experience, August 1995.

S. Chen, M. M. Eshaghian, R. F. Freund, J. L. Potter, and Y. Wu, "Evaluation of Two Programming Paradigms for Heterogeneous Computing," to appear in Journal of Parallel and Distributed Computing, 1995/1996.

S. Chen, M. M. Eshaghian, and Y. Wu, "Mapping Arbitrary Non-Uniform Task Graphs onto Arbitrary Non-Uniform System Graphs," submitted to IEEE Transactions on Parallel and Distributed Computing.

S. Chen and M. M. Eshaghian, "Tools for Design and Mapping of Portable Parallel Programs," to appear in Proceedings of Workshop on Challenges for Parallel Processing, International Conference on Parallel Processing, August 1995.

S. Chen, M. M. Eshaghian, and Y. Wu, "Mapping Arbitrary Non-Uniform Task Graphs onto Arbitrary Non-Uniform System Graphs," to appear in Proceedings of International Conference on Parallel Processing, August 1995.

S. Chen, M. M. Eshaghian, R. F. Freund, J. L. Potter, and Y. Wu, "Scalable Heterogeneous Programming Tools," Proceedings of Heterogeneous Computing Workshop, pp 89-96, April 1994.

S. Chen, M. M. Eshaghian, A. Khokhar, and M. E. Shaaban, "A Selection Theory and Methodology for Heterogeneous Supercomputing," Proceedings of Workshop on Heterogeneous Processing, pp 15-22, April 1993.

L. R. Welch, A. D. Stoyenko, and S. Chen, "Assigning ADT Modules with Random Neural Networks," Proceedings of Hawaii International Conference on Systems Science, pp 546-555, January 1993.

This work is dedicated to
my grandparents, my parents, and my lovely wife.

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 James McHugh, Professor Peter A. Ng, and Professor Sotirios G. Ziavras for serving as members of the committee and offering invaluable suggestions to this dissertation.

The author is grateful to the Department of Computer and Information Science and the National Science Foundation for funding for this project.

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

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

TABLE OF CONTENTS

Chapter **Page**

Chapter **Page**

LIST OF TABLES

LIST OF FIGURES

Figure **Page**

Figure **Page**

CHAPTER 1

INTRODUCTION

In this chapter, we first present an overview of existing parallel programming tools, and will specifically focus on tools for design and mapping of portable parallel programs. An essential component of these tools is the mapping techniques employed. For this reason, we present a detailed overview of various mapping techniques, classified in four categories. Finally, in this chapter, we introduce Cluster-M portable parallel programming tool which will be studied in detail throughout this dissertation.

1.1 Existing Parallel Programming Tools

Many parallel programming tools have been developed. They can be classified as debugger, high-level language, library of specialized routines, mapping tool, network tool, performance tool, parallelization tool, etc. [18]. For example, PVM [76] is a network tool. It consists of library routines embedded in C or FORTRAN that permit a network of heterogeneous computers to appear as one large virtual machine. PVM is simple and easy to use, therefore it is widely used. However, using PVM, the user must specify data allocation and task partitioning. PVM does not provide automatic and intelligent load balancing or mapping. Therefore, programs written in PVM may not be portable.

In this dissertation, we are only interested in portable mapping tools that port parallel programs onto different parallel systems. Using these programming tools, the user can write a parallel program without knowing all the details of the target computer where the program is to be executed. Examples of these tools include Linda, Prep-P, Oregami, Hypertool, and PYRROS [11, 6, 60, 82, 85]. Linda

[11, 1, 46] is a language extension to C and FORTRAN for parallel programming. It is a coordination language for creating parallel or distributed applications via a virtual shared memory paradigm. Linda defines a logically shared data structuring memory mechanism called tuple space. Tuple space holds two kinds of tuples: process tuples which are under active evaluation and data tuples that are passive. Ordinarily, building a Linda program involves dropping a process tuple into tuple space spawning off 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 indistinguishable from other data tuples. Linda requires large volumes of data to be exchanged to and from the shared memory. This may cause heavy congestion over available communication channels of a typical multiprocessor system. For this reason, Linda has been mostly used for coarse grain computations. Furthermore, it is very difficult to implement Linda on architectures not supporting the shared memory structure.

On the other hand, Prep-P, Oregami, Hypertool and PYRROS all include a mapping component which can map a given parallel program onto either a special or arbitrary system. However, the mapping components of Prep-P [6] and Oregami [60] are basically libraries of specialized mapping algorithms which only map regularly structured programs onto regularly structured systems. Their mappings for irregularly structured programs or systems can be very slow and not effective. Hypertool [82] and PYRROS [85] generate fast and near optimal mappings by clustering the task graphs. However, they only map the clusters of task modules onto a fully connected system.

Cluster-M [29, 30, 31, 32], the parallel programming tool to be studied in this dissertation, also includes a mapping component called Cluster-M Mapping Module. The other components of Cluster-M are Cluster-M Specification Module and Cluster-M Representation Module. Portable parallel programs can be written in Cluster-

M Specification without any information of the target computer, while the target computer system is represented by Cluster-M Representation. Cluster-M Mapping Module uses a new mapping technique which maps Cluster-M Specification onto Cluster-M Representation. In the next section, we give an overview of existing mapping techniques.

1.2 Mapping Techniques

The mapping problem has been described in a number of different ways in literature [12, 27]. In general, the mapping problem can be viewed as determining an assignment of a given program which consists of a collection of task modules that can be run serially or in parallel (representable in the the form of a task graph) onto the processing elements of the underlying architecture (representable in form of a system graph), so that some performance measure, such as total execution time, is optimized.

The mapping problem is one of the most challenging problems in parallel and distributed computing. It is known to be NP-complete in its general form as well as several restricted forms [47, 78, 79, 8, 33, 27]. Basically, the techniques used in mapping can be classified into three groups: graph theoretic, mathematical programming, and heuristics [71, 14, 2]. The graph theoretic and mathematical programming techniques are only suitable for some special mapping problems, e.g., for tasks without communication requirement, for systems with special topology, etc. In an attempt to solve the problem in the general case, a number of heuristics have been introduced. These heuristics do not guarantee an optimal solution to the problem but they try to find near-optimal solutions most of the time.

Mapping can be either static or dynamic. 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 further divided into static mapping with task duplication and static mapping without task duplication. In static mapping with task duplication, a node (task module) of task graph can be assigned to more than one node (processor) in the system graph [53, 63, 21, 19, 22, 62]. Task duplication is not permissible for tasks which perform destructive operations such as data output or modification. In static mapping without task duplication, a node (task module) of task graph is assigned to only one node (processor) in the system graph [8, 71, 5, 4, 56, 52, 59, 61, 70, 33, 82, 26, 28, 2, 40, 72, 41, 64, 57, 50, 86, 15, 17]. In this dissertation, we concentrate on static mapping without task duplication.

A static task graph or system graph can be either uniform or non-uniform. A graph is called non-uniform if the weights of nodes are different, and the weights of edges also differ. Otherwise, it is uniform. Mapping of directed task graphs (if there is precedence relation among the task modules), is called task scheduling, as studied in [78, 79, 56, 52, 61, 70, 43, 82, 3, 26, 2, 40, 72, 41, 50, 86, 15, 17]. If the task graphs to be mapped are undirected, then it is called task allocation, as studied in [8, 71, 5, 4, 59, 33, 28, 64, 57, 15]. Whether the graphs are directed or undirected, uniform or non-uniform, there are basically four types of static mappings based on the topological structures of the task and system graphs [15, 17]: (1) mapping of specialized tasks onto specialized systems [20, 73, 10]; (2) mapping of specialized tasks onto arbitrary systems [9, 33]; (3) mapping of arbitrary tasks onto specialized systems [28, 61, 2, 40, 50]; and (4) mapping of arbitrary tasks onto arbitrary systems [8, 25, 56, 5, 6, 59, 26, 60, 65, 15, 17].

1.2.1 Mapping of Specialized Task onto Specialized Systems

The most distinguished examples of research on mapping specialized tasks onto specialized systems include Coffman and Graham's early work and later Stone and Bokhari's work on a two processor system [20, 73, 74, 7, 75]. Coffman and Graham

[20] did not consider inter-processor communication cost, while Stone and Bokhari assumed a serial program with multiple modules was to be mapped onto the two processors. In the latter case, an optimal solution can be obtained using max flow min cut algorithm in polynomial time [45, 73]. An extension of the min cut to three and N processor system was also discussed. However, it was noted that this extension was not trivial and the mapping results can not guarantee to be optimal. Other examples of this group of mappings is Bokhari's partitioning and mapping of chain-like tasks on chain-linked processors or host-satellite system, and a tree-structured single-host multiple-satellite system (actually a star) [10].

1.2.2 Mapping of Specialized Task onto Arbitrary Systems

Some techniques have been developed for mapping specialized tasks onto arbitrary systems. Bokhari in [9] used the shortest tree algorithm to obtain the optimal mapping of a tree-structured task graph having M task modules onto arbitrary N processors in $O(MN^2)$ time. Again, it was also assumed that the execution of all the modules of the task was serial. Towsley in [77] gave an algorithm for mapping a series-parallel task graph onto an arbitrary system graph in $O(MN^3)$ time. Fernandez-Baca [33] observed that tree graph and series-parallel graph are actually two special cases of a k-tree, and developed an efficient algorithm for mapping any k-tree or partial k-tree task graph onto an arbitrary system graph in $O(MN^{k+1})$ time. This matches the time complexity of [9] and [77] as its special cases. Also, Fernandez-Baca developed an algorithm for mapping an almost tree with parameter k in $O(MN^{\lceil k/2 \rceil+2})$ time.

1.2.3 Mapping of Arbitrary Task onto Specialized Systems

Most mapping techniques developed fall into the third category. When the system has N processors but of a specialized structure, some specialized techniques can be used for the mapping. These techniques are especially suited for a set of specialized systems, therefore the mappings can be very efficient and effective. For example,

Ercal, et. al.'s [28] mapping algorithm on a hypercube using Kernighan-Lin's mincut bipartitioning technique [49], Sadayappan and Ercal's work on mesh [68], Lo's algorithm for bus network [58], etc. Indurkya et al. also analyzed the optimal mapping of an arbitrary task graph onto a specialized system graph with additive communication cost, e.g., a bus network [44]. Mappings on more general regular-structured task graphs were studied by Berman and Snyder using edge grammar [5]. Many mapping algorithms did not consider the system in detail, thus assuming that all the processors were fully connected [61, 2]. Ali and El-Rewini in [2] proposed an interesting graph theoretic approach for mapping M modules onto N processors by constructing a split graph containing M module nodes and N processor nodes. The mapping problem was then reduced to finding cliques of the split graph. Since a fully connected system provides the strongest communication capacity, many heuristics have been developed for mapping on such a system [70, 82, 40, 86]. Various clustering techniques can be used (especially for mapping on fully connected system) to reduce the time complexity of mappings [70, 82, 40, 86].

1.2.4 Mapping of Arbitrary Tasks onto Arbitrary Systems

General mappings from an arbitrary task onto arbitrary systems have proven to be the most difficult, especially when the task graph and system graph become large and complex. Bokhari in [8] defined the mapping problem to be matching the edges of the task graph and the system graph. The order of the task graph was assumed to be no greater than that of the system graph. The edges were uniformly weighted, and the mapping was assumed to be one-to-one onto mapping, which may not be the optimal mapping. A heuristic mapping algorithm based on local search using pair-wise exchange was presented in [8]. The time complexity of this algorithm is $O(N^3)$. Lee and Aggarwal in [56] extended Bokhari's general mapping algorithm to take into account a directed task graph with a set of communication phases.

Therefore, communication edges of different phases can be mapped independently. However, the order of the task graph was still assumed to be no greater than that of the system graph. This restriction was relaxed in [14]. Stone's max flow min cut algorithm, by which an optimal mapping can be obtained on a two processor system, can also be used for sub-optimal mappings of arbitrarily connected M modules on N processors. Lo used the max flow min cut algorithm as the first step in her heuristic algorithm [59]. The time complexity of Lo's heuristic is $O(M^2N|E_p|\log M)$, where $|E_p|$ is the number of communication links between processors. El-Rewini and Lewis presented their mapping heuristic (MH) algorithm in [26]. MH is a list scheduling heuristic which maps an arbitrary task graph onto an arbitrary system graph. In list scheduling, each task module is assigned a priority. Whenever a processor is available, a task module with the highest priority is selected from the list and assigned to this processor. MH has a time complexity of $O(M^2N^3)$. A searching algorithm can also be used to match an arbitrary task graph to an arbitrary system graph [71, 48]. Graph contraction and or clustering is often used to reduce the task graph before mapping, thus reducing the time complexity. For some regularly structured tasks, a specialized graph contraction technique, such as edge grammar, can be used to reduce the order of the task graph to a desired value [5, 60]. When task graph is arbitrary shaped, heuristic approaches such as simulated annealing, simply greedy, or critical path can be used for graph contraction or clustering [25, 4, 6, 61, 39, 28, 69, 60, 83, 85, 65, 81, 32].

An alternative approach to map an arbitrary task graph onto an arbitrary system graph is to first map the task graph onto a completely connected graph with a certain order, and second, map this completely connected graph onto the target system graph. Sarkar's edge-zeroing algorithm [70], Wu and Gajski's Modified Critical-Path (MCP) and Mobility-Directed (MD) scheduling algorithm [82], and Yang and Gerasoulis's Dominant Sequence Clustering (DSC) algorithm [86] fall into

this group. They all produce fast and good mappings from an arbitrary directed task graph onto a completely connected system graph. However, to finally map onto an arbitrary target system graph, other mapping (allocation) algorithms such as Bokhari's $O(N^3)$ mapping algorithm have to be used which may increase the overall time complexity. Also, the final mapping results on an arbitrary system may not be as good as on a completely connected system.

Most mapping techniques for mapping arbitrary tasks onto arbitrary systems only consider uniform systems [8, 56, 5, 70, 82, 60, 28, 14, 86, 62, 15]. Therefore, information about systems such as computation speed of each processor and communication bandwidth on each link is implicitly known before mapping. Even in those techniques which consider non-uniform systems [71, 59, 72], the information about the system graphs is incorporated in their task graphs. Only in [26, 15, 17], the information about the speed of the processors and the communication links is kept independent of the task graphs. Therefore, these results can be directly used towards designing portable programs which are representable in form of machine-independent task graphs.

1.3 Cluster-M

Cluster-M facilitates the design and mapping of portable parallel programs onto various multiprocessor systems by clustering not only machine-independent task graphs but also system graphs. Cluster-M has three components: the Cluster-M Specification of the given task, the Cluster-M Representation of the underlying system, and the Cluster-M Mapping Module. Portable programs are specified in Cluster-M Specifications in a way which represents concurrent computations and communications at every step of the overall execution. On the other hand, the processors of the underlying system are clustered in a hierarchical fashion so that all of those in the same cluster have an efficient communication medium. The Cluster-

M Specification and Representation are actually multi-level clustered graphs of the directed (or undirected) task graph, called the Spec graph, and the undirected system graph, called the Rep graph. The user can directly specify the Cluster-M Specification of a given task which is representable in the form of a Spec graph. On the other hand, for any given task graph, a Spec graph can be generated by using one of the clustering algorithms presented in this dissertation. Similarly, a Rep graph can be generated given the topology of the target system as input.

Both Spec and Rep graphs contain a certain number of clustering levels. In each level, there are a number of clusters which are called Spec clusters and Rep clusters respectively. A Spec cluster represents a set of concurrent computations which have inter-communication between each other. A Rep cluster represents a set of processors with a certain level of connection. The mapping is carried out from a Spec graph onto a Rep graph by matching Spec clusters to Rep clusters. As the number of clusters at each clustering level is much less than the number of original task modules or processors, the mapping process becomes very fast, yet produces sub-optimal results.

1.4 Contributions and Outline

The mapping technique presented in this dissertation is the first which produces a near-optimal mapping of an arbitrary non-uniform machine-independent task graph onto an arbitrary non-uniform task-independent system graph. The clustering is done only once for a given task graph (system graph) independent of any system graphs (task graphs). This is a machine-independent (application-independent) clustering and is not repeated for different mappings. The presented recursive mapping algorithm maps any task graph with M modules onto any system graph having N processors in $O(MP)$ time, where $P = \max(M, N)$. This time complexity guarantees faster mappings compared to other leading mapping techniques. Our

experimental results also indicate that Cluster-M produces better or similar mapping results compared to other techniques which work only for restricted task or system graphs.

The rest of the dissertation is organized as follows. In Chapter 2, we present Cluster-M programming in Cluster-M Specification Module. The clustering algorithms for uniform/non-uniform directed/undirected graphs are given in Chapter 3. Chapter 4 presents mapping algorithms for both uniform and non-uniform graphs, and experimental results and comparisons with other known techniques. Related work of mapping of specialized heterogeneous tasks, and the combined use of Cluster-M with another tool, called HAsC, are discussed in Chapter 5 and 6 respectively. Finally, conclusions are given in Chapter 7.

CHAPTER 2

CLUSTER-M PROGRAMMING

In this chapter, we show how to write portable parallel programs in form of Cluster-M Specifications. A set of Cluster-M constructs, which are essential for writing Cluster-M Specifications, is described. To illustrate programming in Cluster-M Specifications, several frequently used operations are coded using these constructs.

2.1 Cluster-M Specifications

A Cluster-M Specification of a task is a high level machine-independent program that specifies the computation and communication requirements of a given problem. A Cluster-M Specification can be translated into a Spec graph which contains multiple levels of clustering. In each level, there are a number of Spec clusters representing concurrent computations at a certain step. Clusters are merged when there is a need for communication among concurrent task modules. For example, if all n elements of an array are to be squared, each element is placed in a cluster, then the Cluster-M specification would state:

For all n clusters, square the contents.

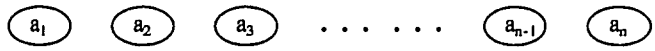


Figure 2.1 Spec graph of a unary operation on an array of size n.

Note, that since no communication is necessary, there is only one level in the Spec graph as shown in Figure 2.1. The mapping of this Specification to any architecture having n processors would be identical. Figure 2.2 shows the Spec graph of a binary associative operation on 8 elements. Initially, each element is in a

cluster. Then clusters are merged at each level when they have inter-communication. The result of this binary associative operation is obtained in the cluster at the last clustering level.

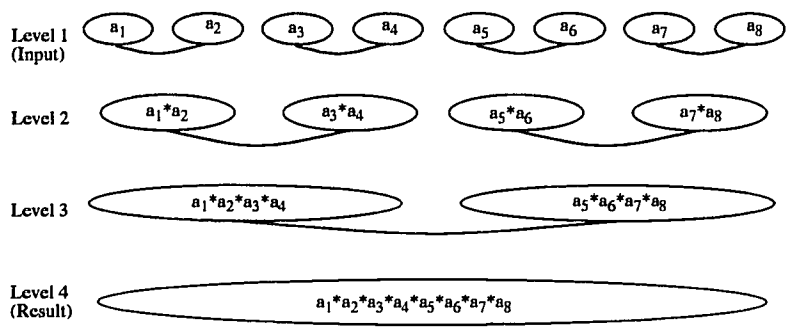


Figure 2.2 Spec graph of a binary associative operation on 8 elements.

2.2 Cluster-M Constructs

The basic operations on the clusters and their contained elements are performed by a set of constructs which form an integral part of the Cluster-M Specification Module. 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 $[e_1, e_2, \cdots, e_n]$ 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 into cluster z. 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 in 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 at the k-th element into two clusters $E1$ and $E2$.

2.3 Implementation of the Cluster-M Constructs

The Cluster-M Specification constructs have been implemented by Program Composition Notation (PCN), a system for developing and executing parallel programs

[13, 35]. PCN consists 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 and 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 such as blocks, procedures and 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 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 2.3. The implementation of the seven Cluster-M constructs is listed in Appendix A.

2.4 Cluster-M Specification Macros

Several operations are frequently encountered in writing parallel programs. 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:

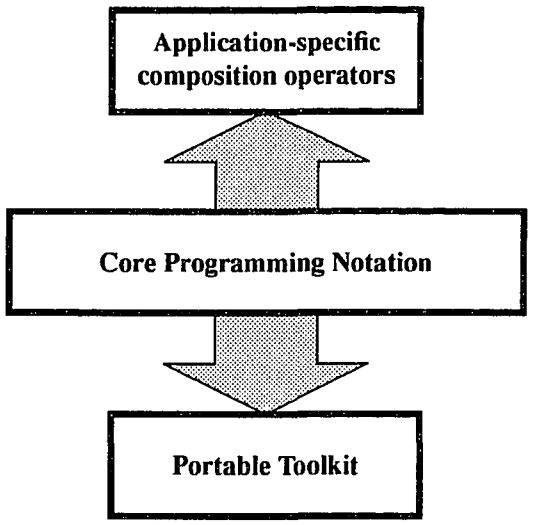


Figure 2.3 PCN system structure.

2.4.1 Associative Binary Operation

Performing an associative binary operation on N elements is a common operation in parallel applications. The Spec graph for input size = 8 is given in Figure 2.2. The resulting Spec 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:

$ASSOC\_BIN(*, N, A)$

$int\ N, A[\,];$

$\{\ ;\ lvl = 0,$

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

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

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

$\qquad cluster[i] = c$

$\quad \}$

```
    },
    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)
}
```

2.4.2 Vector Dot Product

As a representative example of vector operations, the dot product of two vectors is considered here. 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 Spec graph of this operation is similar to that shown in Figure 2.2. 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)
}
```

2.4.3 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 Cluster-M 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,
   make_tuple(N, cluster),
   { ; i over 0 .. N − 1 ::
      { ; CMAKE(lvl, [A[i]], c),
         cluster[i] = c
      }
   } ,
   make_tuple(N, Z),
   {; j over 0 .. N − 1 ::
      { ; CUN(op, n, cluster[j], 1, e),
         Z[j] = e
      }
   }
}
```

2.4.4 Broadcast Operation

A frequently encountered operation in parallel programs is broadcast operation. One value is to be broadcast to all processors in the system. The Cluster-M Specification

for a macro that broadcasts one value a from cluster x to N recipient clusters, can be written in terms of Cluster-M constructs as follows:

$BROADCAST(N, e, Z)$

$\{;\ lvl = 0,$

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

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

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

$\qquad\quad Z[i] = c$

$\qquad \}$

$\quad \}$

$\}$

The Spec graph for the broadcast operation when $N = 8$ is shown in Figure 2.4.

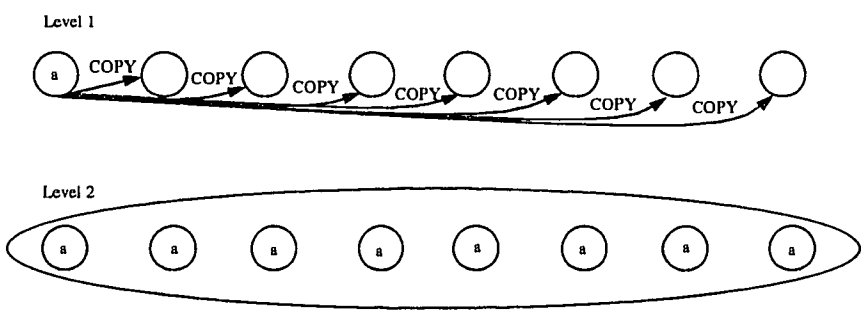


Figure 2.4 Cluster-M Specification of broadcast macro.

CHAPTER 3

CLUSTERING GRAPHS

In this chapter a set of clustering algorithms, which can be used to generate Spec and Rep graphs from arbitrary uniform/non-uniform directed/undirected task and system graphs, are presented. Clustering algorithms for uniform graphs are presented in Section 3.1, and clustering algorithms for non-uniform graphs are presented in Section 3.2. The obtained Spec and Rep graphs will be input to the Mapping Module as presented in the next chapter.

3.1 Clustering Arbitrary Uniform Graphs

This section addresses clustering algorithms for uniform graphs. If the task graph is directed, then the algorithm presented in Section 3.1.1 can be used to obtain the Spec graph. If the task graph is undirected, then the algorithm presented in Section 3.1.2 can be used to generate the Spec graph. Since it is assumed that the connections between adjacent processors of the parallel systems studied here are bi-directional, the system graphs are always undirected. A Rep graph can be obtained by the clustering algorithm for undirected graphs in Section 3.1.2. For every architecture, at least one corresponding Cluster-M Rep graph can be constructed. A Rep graph with k nested clustering levels represents a connected network of processors with diameter $\Omega(k)$.

3.1.1 Clustering Directed Graphs

Many clustering techniques have been developed to reduce the order and size of task graphs [25, 4, 61, 28, 69, 60, 40, 65]. For example, a cluster can be a clan [61] which is a set of nodes with common outside ancestors and descendants on the task graph.

Our Cluster-M based mapping requires clustering of both the task graph as well as the system graph to obtain even better and faster solutions. For clustering either the task graph or the system graph, the following algorithm is used if the input graph is directed, otherwise the algorithm presented in the next section (3.1.2) is utilized. In the scheduling problem, task graphs are directed, while in task allocation problem they are not. The system graphs, on the other hand, are always assumed to be undirected (todays computers have bi-directional links). Therefore, the algorithm presented below is to be used only for directed task graphs. In the following, a formal definition of directed task graphs, which is also applicable to undirected task graphs (with the exception that for every i,j, $(t_i, t_j) = (t_j, t_i)$), is given.

A task can be represented by a task 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 start. Furthermore, each task module t_i is associated with its amount of computation A_i, and 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. Note $A_i > 0$ and $D_{ij} \geq 0$, for $1 \leq i, j \leq M$. If an directed edge (t_i, t_j) exists, then 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 Broadcast-node. If a node has more than one parent, it is called a Merge-node. Task modules are divided into different execution steps and communications between modules are divided into different execution phases according to the data and operational precedence. Computations in the same step and communications in the same phase can be carried out in parallel, but can not start before the parent modules of those in the previous step have finished computations. In this section, it is assumed that the amount of computation within each task module and the amount of data communication between any two task modules are uniform, i.e., $A_i = 1$ and

$D_{ij} = 1$, for $1 \leq i, j \leq M$ and $(t_i, t_j) \in E_t$. This assumption leads to the simple greedy clustering in the clustering algorithm.

The algorithm for clustering directed graphs is presented in Figure 3.1. The basic idea is to merge all the nodes in each execution step if they have a common parent node or a common child node. Each cluster has a size which is defined to be the number of concurrent nodes contained in this cluster. If a Spec cluster has a size σ_{S_i} and the sizes of its sub-clusters at the lower level are $\sigma_{S_{i1}}, \cdots, \sigma_{S_{ik}}$, it is obvious that $\sigma_{S_i} = \sigma_{S_{i1}} + \cdots + \sigma_{S_{ik}}$. Also, some task modules which can not be run in parallel will be embedded into a supernode, so that they will be finally assigned to the same processor. The size of a supernode is still 1. If a parent node t_i has one or more children, one of its children is to be embedded to t_i. If a child node has one or more parents, it will be embedded to one of its parents.

The complexity of the clustering-directed-graph algorithm is on the order of the number of edges of the task graph, which is $O(M^2)$, where M is the number of nodes of the task graph. To illustrate this algorithm, the following example is presented.

A task graph of 15 modules is shown in Figure 3.2. Each module has a computation amount of 1, and each edge carries this amount of data communication. This task graph contains two subgraphs that are not connected, which means the two subtasks can be executed in parallel. The Spec graph is constructed by merging the clusters when they have communication needs as illustrated in Figure 3.2. The input task graph has nodes a to o (15 nodes). The final Spec graph is a multi-layered graph containing 9 nodes. For example, j, k and l are embedded to d, since j, k and l are in different execution steps and can not be executed concurrently. This will not only save the processor resources and communication cost, but also reduce the mapping cost since the Spec graph now only contains 9 nodes versus the original 15.

```
Clustering-directed-graphs Algorithm
group nodes of given task graph into corresponding steps
group edges of given task graph into corresponding phases
for all nodes at step 1, do
    make each node into a cluster
for all phases, do
    for all edges (t_i, t_j), do
    begin if t_j is a Merge-node, then
        begin embed t_j to t_i
            if the parent nodes of t_j are not in a cluster, then
            begin merge them into a cluster
                increase cluster size
            end
        end
        if t_i is a Broadcast-node, then
        begin k = number of nodes in cluster t_i belongs to
            if t_i has more than k children, then
            begin embed first k children to the above k nodes
                merge the rest into the above cluster
                increase cluster size
            end
            else  embed all children
        end
    end
```

Figure 3.1 Clustering-directed-graphs algorithm.

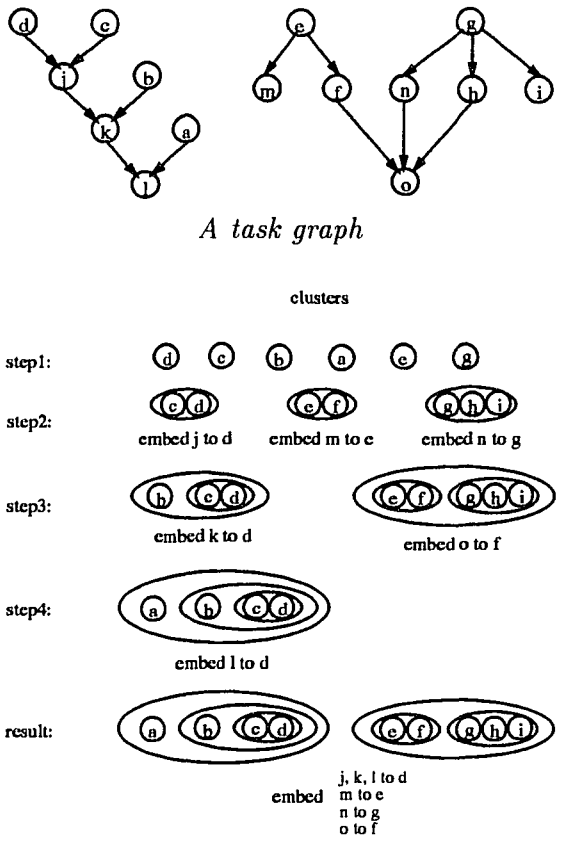


Figure 3.2 A task graph and the obtained Spec graph.

3.1.2 Clustering Undirected Graphs

The algorithm presented in this section can be used to generate the Spec graph of an undirected task graph (for allocation problem), as well as the Rep graph of a system graph (undirected). Since the definition of a directed task graph was presented in the last section, it is also applicable to an undirected task graph (with the exception of $(t_i, t_j) = (t_j, t_i)$, for all i, j). This section only presents the definition of system graphs (undirected). Then the algorithm for generating a clustered graph (Spec graph for task graph or Rep graph for system graph) out of such an undirected input graph is presented.

A parallel system can be modeled as an undirected system graph $G_p(V_p, E_p)$. $V_p = \{p_1, ..., p_N\}$ is a set of processors forming the underlying architecture, while E_p is a set of edges representing the interconnection topology of the parallel system. It is assumed that the connections between adjacent processors of the parallel systems studied here are bi-directional. 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 S_i, and the communication bandwidth/rate between two processors p_i and p_j is denoted by R_{ij}. In this section, we assume that there is a uniform speed at each processor and a uniform transmission rate over any direct communication link in the system, i.e., $S_i = 1$ and $R_{ij} = 1$, for $1 \leq i, j \leq M$ and $(p_i, p_j) \in E_p$. This assumption leads to the simple greedy clustering.

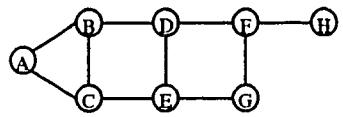
To construct a clustered graph (Rep graph or Spec graph) from an undirected input graph, initially, every node forms a cluster. This node is presented by p_i in the case of the system graph and by t_i in the case of the task graph. Then clusters which are completely connected are merged to form a new cluster. This is continued until no more merging is possible. Two clusters x and y are connected if x contains a node p_x (t_x) and y contains a node p_y (t_y), such that nodes p_x (t_x) and p_y (t_y) are connected by a direct communication link. Each cluster has a size which is the

```
Clustering-undirected-graphs Algorithm
for all nodes p_i (t_i), do
        make a cluster for p_i (t_i) at clustering level 1
set cluster level to 1
while merging is possible, do
begin for all clusters c at current level, do
        begin make c into cluster c' at next level
                delete cluster c from current level
                for all clusters x in current level, do
                if x is connected to all sub-clusters of c', then
                begin merge x into c'
                        delete x from current level
                end
        end
        increment clustering level by 1
end
```

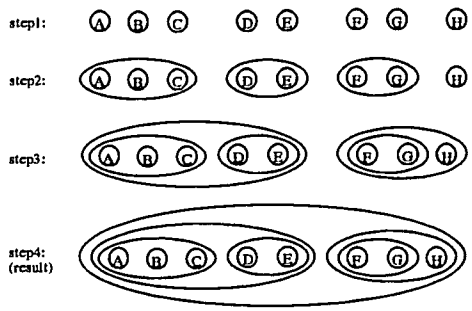
Figure 3.3 Clustering-undirected-graphs algorithm

number of nodes it contains. If a Rep (Spec) cluster has a size σ_{R_i} (σ_{S_i}) and the sizes of its sub-clusters at the lower level are $\sigma_{R_{i1}}, \cdots, \sigma_{R_{ik}}$ ($\sigma_{S_{i1}}, \cdots, \sigma_{S_{ik}}$), it is obvious $\sigma_{R_i} = \sigma_{R_{i1}} + \cdots + \sigma_{R_{ik}}$ ($\sigma_{S_i} = \sigma_{S_{i1}} + \cdots + \sigma_{S_{ik}}$). The algorithm for clustering undirected graphs is shown in Figure 3.3. An example is shown in Figure 3.4. The undirected graph shown can represent a system graph, therefore the generated output is shown as a Rep graph. However, if the same input is an undirected task graph for allocation problem, then the generated output is a Spec graph. Figures 3.5, 3.6, 3.7, and 3.8 show the clustered graphs of the hypercube, mesh, ring, and a completely connected graph respectively.

The running time of this implementation is analyzed as follows. In every level, each cluster in that level is compared with the higher numbered clusters in the same level and check if they form a clique. Suppose at a certain level of the system graph (undirected task graph), there are m clusters $c_1, \cdots, c_m$, with each cluster c_i containing P_i number of processors (T_i number of task modules). It is clear



An undirected graph.



Clustering of the undirected graph.

Figure 3.4 An undirected graph and its clustering.

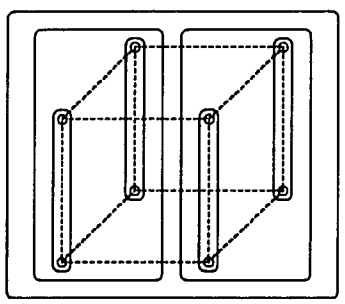


Figure 3.5 A clustered graph of a hypercube.

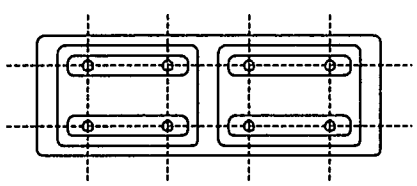


Figure 3.6 A clustered graph of a mesh.

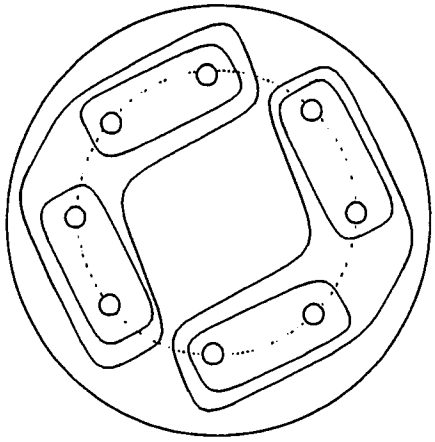


Figure 3.7 A clustered graph of a ring.

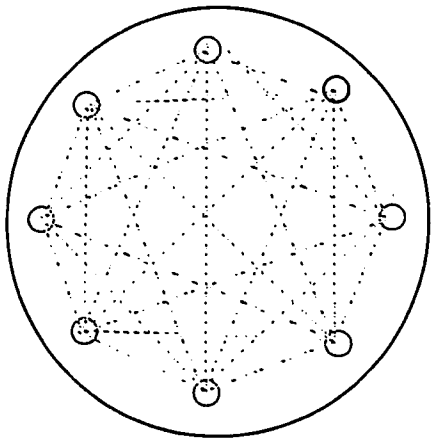


Figure 3.8 A clustered graph of a completely connected graph.

$\sum_{i=1}^{m} P_i = N$ ($\sum_{i=1}^{m} T_i = M$), where N is the number of underlying processors (M is the number of task modules). The time of clustering at this level is dominated by the total number of comparisons made to determine if each cluster is connected to all sub-clusters of another cluster at next level, which is at most $\sum_{i=1}^{m} \sum_{j=i+1}^{m} P_i P_j \leq \sum_{i=1}^{m} P_i N \leq N^2$ ($\sum_{i=1}^{m} \sum_{j=i+1}^{m} T_i T_j \leq \sum_{i=1}^{m} T_i M \leq M^2$). The number of levels can be at most $N-1$ ($M-1$). Therefore, the total time complexity of this algorithm is $O(N^3)$ ($O(M^3)$).

3.2 Clustering Arbitrary Non-Uniform Graphs

In this section, two algorithms for clustering non-uniform graphs are presented. The clustering is done only once for a given task graph (system graph) independent of any system graphs (task graphs). It is a machine-independent (application-independent) clustering and is not repeated for different mappings. Once a Spec graph and a Rep graph are obtained, a sub-optimal mapping can be generated by using a fast recursive mapping algorithm to be presented in Chapter 4.

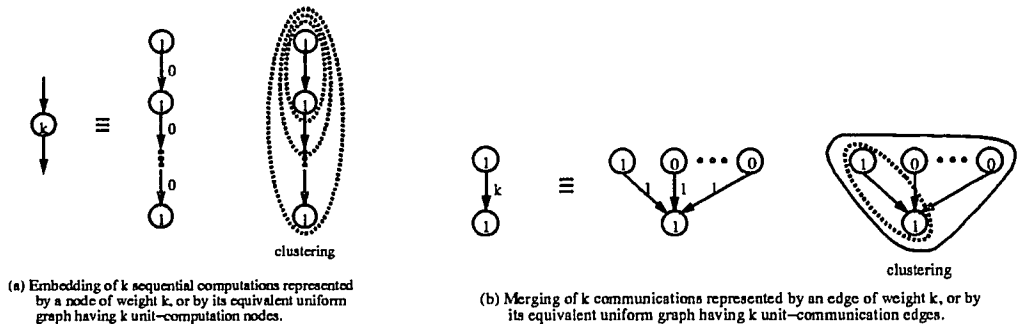
3.2.1 Clustering Non-Uniform Directed Graphs

The definition of directed task graph is the same as that presented in Section 3.1.1, except that computation amount A_i and data transmission amount D_{ij} may be non-uniform for different nodes and edges in the task graph.

A clustering algorithm called clustering-non-uniform-directed-graphs is shown in detail in Figure 3.9. The algorithm is briefly described in the following. It begins with a quadruple of parameters (σ_S, δ_S, Π_S, π_S). Each of these parameters is described as follows. The size of a cluster is denoted by σ_S, and represents the maximum number of nodes in this cluster that can be computed in parallel. The number of levels in a cluster represents the maximum sequential computation length of each node in the cluster, and is denoted by δ_S. The total amount of communi-

```
Clustering-non-uniform-directed-graphs Algorithm
group nodes of given task graph into corresponding steps
group edges of given task graph into corresponding phases
for all nodes at step 1, do
begin make it into a cluster
      calculate the parameters of each cluster
end
for all phases, do
      for all edges (t_i, t_j), do
      begin if t_j is a Merge-node, then
            begin sort all incoming edges to t_j in descending
                  order of communication amount
                  embed the first child node to t_i
                  if the parent nodes of t_j are not in a cluster, then
                  begin merge them into a cluster
                        calculate the parameters of the new cluster
                  end
            end
            if t_i is a Broadcast-node, then
            begin sort all outgoing edges from t_i in descending
                  order of communication amount
                  embed the first child node to t_i
                  if the child nodes of t_i are not in a cluster, then
                  begin merge them into a cluster
                        calculate the parameters of the new cluster
                  end
            end
      end
```

Figure 3.9 Clustering-non-uniform-directed-graphs algorithm.



(a) Embedding of k sequential computations represented by a node of weight k, or by its equivalent uniform graph having k unit–computation nodes.

(b) Merging of k communications represented by an edge of weight k, or by its equivalent uniform graph having k unit–communication edges.

Figure 3.10 Two types of clustering.

cation among all clustering levels is denoted by Π_S, and the average communication amount at current (top) level is denoted by π_S. Furthermore, there are two types of operations performed on clusters: embedding and merging. Embedding is when two or more sequential clusters are combined into one cluster as shown in Figure 3.10. This is shown in perforated line in Figure 3.10(a) and (b). Merging is when a number of concurrently executable sub-clusters are grouped to form a new cluster. This is shown by a solid line in Figure 3.10(b). The embedded cluster in Figure 3.10(a) has $\sigma_S = 1$, $\delta_S = k$, $\Pi_S = 0$ and $\pi_S = 0$. The merged cluster in Figure 3.10(b) has $\sigma_S = k$, $\delta_S = 2$, $\Pi_S = k$ and $\pi_S = k - 1$. In each of the two figures, the value of the quadruple obtained is identical for both the uniform and non-uniform equivalent representations.

The clustering is done step by step. Each clustering step corresponds to a computation step. At every step, the nodes (clusters) are clustered as follows. If a node is a Merge-node, it is first embedded onto one of its parent nodes, all the parent nodes are merged into a larger cluster, similar to what was done in Section 3.1.1. A simple case where a Merge-node has only two parent nodes is shown in Figure 3.11. Similarly, a general case is shown in Figure 3.12, where a Merge-node has n parent nodes, and the n edges are sorted in descending order of the edge weight (communication amount). If a node is a Broadcast-node, then one of its child nodes will be embedded into this Broadcast-node, and then the rest of the child nodes

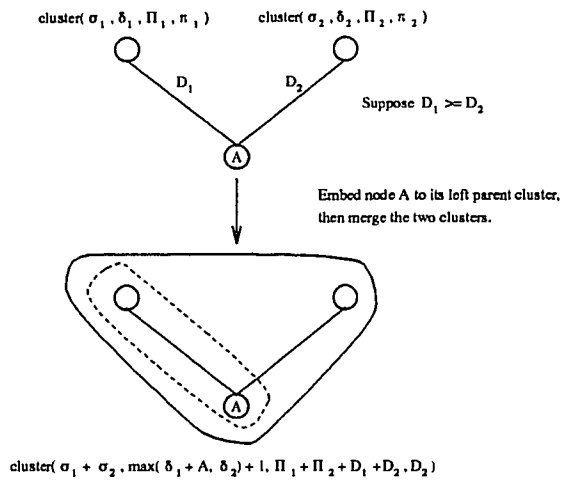


Figure 3.11 Clustering on a Merge-node.

will be merged with the Broadcast-node into a larger cluster. A simple case where a Broadcast-node has only two child nodes is shown in Figure 3.13. Similarly, a general case is shown in Figure 3.14, where a Broadcast-node has n child nodes, and the n edges are sorted in descending order of their weights. Note that since the task graphs are independent of the system graphs (unlike [70, 82, 86]), they do not contain the information about computation time and communication delay. Therefore, only one child node can be embedded to the parent node in both the merge and broadcast cases shown above. Consider an example in Figure 3.15, where $D_1 \geq D_2 \geq \cdots \geq D_n$. If A_i and D_i $(1 \leq i \leq n)$ are actual computation time and communication delay, then more than one child node should be embedded to the parent Broadcast-node if possible. Suppose $D_j \geq \sum_{i=1}^{j-1} A_i$ for $j = 2, \cdots, k$, but $D_{k+1} < \sum_{i=1}^{k} A_i$, then the first k child nodes can be embedded to the parent node. However, since A_i and D_i $(1 \leq i \leq n)$ are just computation and communication amounts, the time spent on such computations and communications is unknown before they are mapped onto a particular system. Therefore, only the first child node can be embedded to the parent node. However, the embedding of multiple child nodes can be done as part of the mapping, which is explained in Chapter 4.

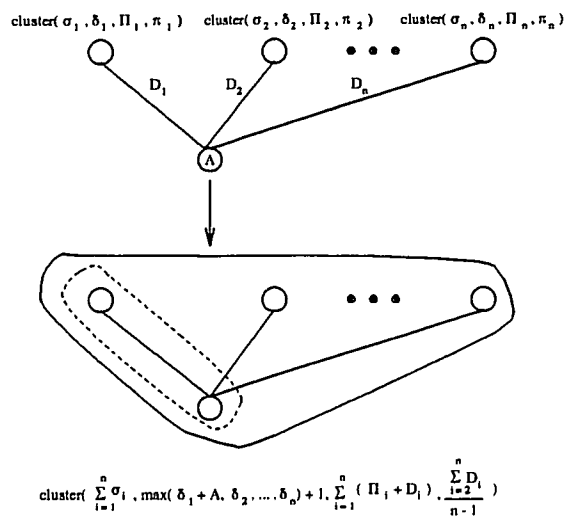


Figure 3.12 Clustering on a Merge-node: a general case.

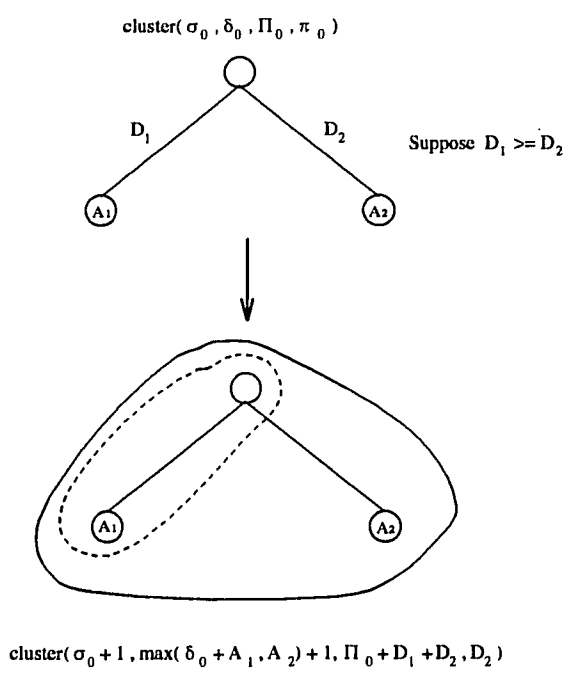


Figure 3.13 Clustering on a Broadcast-node.

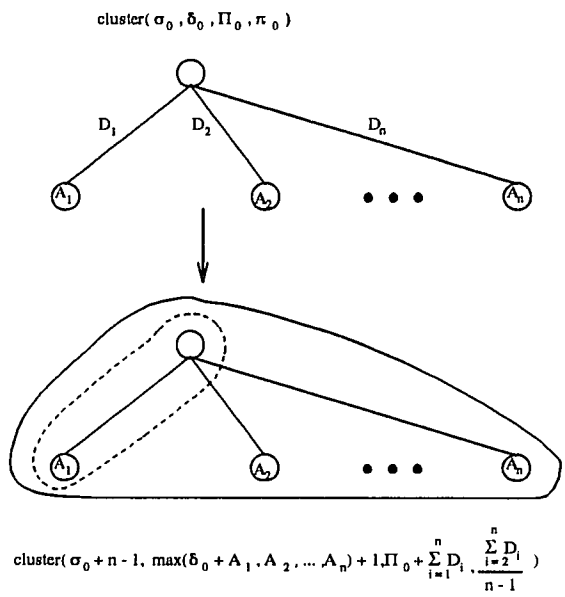


Figure 3.14 Clustering on a Broadcast-node: a general case.

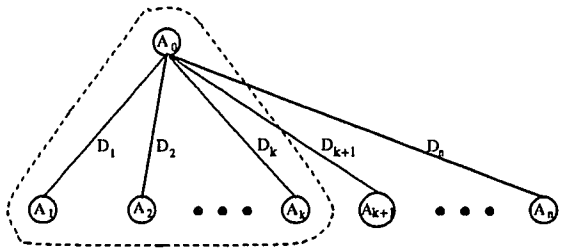
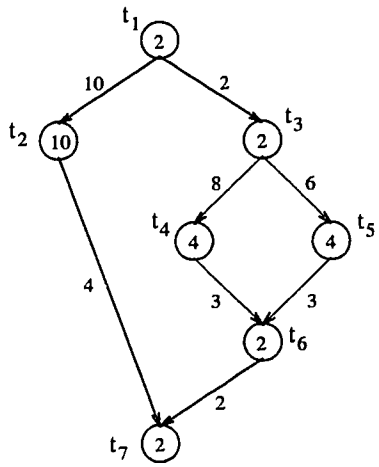
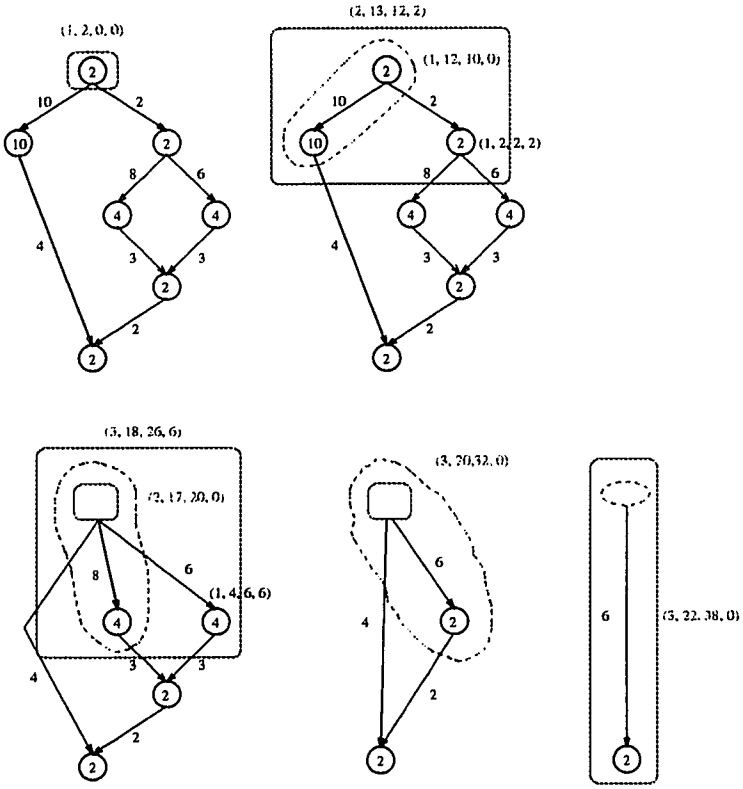


Figure 3.15 Possible embedding on a Broadcast-node.



A non-uniform task graph



Constructing the Spec graph

Figure 3.16 A task graph and the obtained Spec graph.

The time complexity of the clustering-non-uniform-directed-graphs algorithm is bound by the number of edges in the task graph, which is $O(M^2)$, where M is the number of nodes. To illustrate this algorithm, consider the task graph of 7 modules and its Spec graph as shown in Figure 3.16. Each module is labeled with its computation amount, and each edge is labeled with the amount of data communication. The Spec graph is constructed by merging the clusters when they have communication needs. The final Spec graph is a multi-level clustered graph.

3.2.2 Clustering Non-Uniform Undirected Graphs

The algorithm presented in this section can be used for generating the Spec graph of an undirected task graph (for allocation problem), as well as the Rep graph of a system graph (undirected). The definition of the directed task graph presented in the last section is also applicable to an undirected task graph (with the exception of $(t_i, t_j) = (t_j, t_i)$, for all i, j). The definition of undirected system graph is the same as that presented in Section 3.1.2, except that both the computation speeds S_i of different processors and the transmission rates R_{ij} of different communication links may be non-uniform. Therefore, the system discussed in this section can be a truly heterogeneous system. The rest of the section concentrates on the clustering of an undirected system graph to obtain Rep graphs. The same approach can be used to obtain Spec graph from a non-uniform undirected task graph.

Similar to a Spec cluster, each Rep cluster is associated with a quadruple (σ_R, δ_R, Π_R, π_R) which represents number of processors contained in the cluster, average computation speed of the processors in the cluster, total communication bandwidth, and the average communication bandwidth at the current (top) clustering level. To construct a clustered graph (Rep graph) from an undirected system graph, initially every node with computation speed S_i forms a cluster with parameters (1, S_i, 0, 0). Then clusters which are completely connected are merged to form a new cluster,

```
Clustering-non-uniform-undirected-graphs Algorithm
for all nodes p_i, do
begin make a cluster for p_i at clustering level 1
      set the parameters of the cluster to be (1, S_i, 0, 0)
end
set cluster level to 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 level
            calculate the parameters of c'
            delete clusters c_i and c_j from current level
            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 level, 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 level by 1
end
```

Figure 3.17 Clustering-non-uniform-undirected-graphs algorithm.

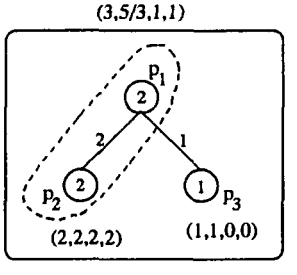


Figure 3.18 A non-uniform system graph and its clustering.

and the parameters of the new cluster are calculated accordingly. This process is continued until no more merging is possible. Two clusters x and y are connected if x contains a node p_x (t_x) and y contains a node p_y (t_y), such that node p_x (t_x) and p_y (t_y) are connected by a direct communication link. The algorithm for clustering undirected graphs is shown in Figure 3.17. An example is shown in Figure 3.18.

The running time of this implementation is analyzed as follows. For each level, first sort all the edges between clusters which takes $O(|E_p| \log |E_p|)$, where $|E_p|$ is the number of edges in the system graph. Clusters keep merging into the next levels. Suppose at a certain level, there are m clusters $c_1, \cdots, c_m$. The time for all these comparisons is at most $m * m \leq N^2$, where N is the number of processors in the system graph. The number of levels can be at most $N - 1$. 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 so that $|E_p| = O(N^2)$, the upper bound for this algorithm will be $O(N^3 \log N)$.

CHAPTER 4

CLUSTER-M MAPPING

For a given problem, a high level machine independent parallel program can be presented in form of a Cluster-M Specification which is directly representable as a Spec graph. In addition, a Spec graph can be generated directly from a given task graph, using one of the algorithms in the last chapter. On the other hand, given a system graph representing an underlying architecture or organization, a Rep graph can be generated as shown in the last chapter. In this chapter, given a Spec graph and a Rep graph as the input to the Mapping Module, efficient mapping algorithms are presented which produce sub-optimal matching of them. The mapping procedure presented in this chapter has a much lower time complexity than the traditional mappings since it contains a graph matching procedure for which both of the input graphs have been clustered. The uniform mapping algorithm presented in Section 4.1 has a time complexity of $O(MN)$, while the non-uniform mapping algorithm in Section 4.3 has a time complexity of $O(MP)$, where $P = \max(M, N)$. Extensive experimental results and comparisons with other leading mapping techniques are also presented in this chapter.

4.1 Cluster-M Uniform Mapping

This section presents a fast recursive mapping algorithm which produces sub-optimal mapping of a uniform Spec graph onto a uniform Rep graph in $O(MN)$ time. A set of preliminaries are given below, followed by a high level description of the mapping algorithm.

4.1.1 Uniform Mapping Algorithm

The mapping function is defined by $f_m : V_t \xrightarrow{onto} V_p$. Following the precedence constraints and the computation and communication requirements of the original task graph, a schedule can be obtained which places each task module t_i to processor $f_m(t_i)$ at the proper time (earliest possible starting time). Since the communication weights along edges of both uniform task graphs and system graphs are 1, the communication time of the task graph edge (t_i, t_j) is equal to $dist(f_m(t_i), f_m(t_j))$, where $dist(p_i, p_j)$ is the shortest distance between processor p_i and p_j. It is also assumed that it takes no time to communicate data within the same processor, i.e., $dist(p_i, p_i) = 0$.

A schedule can be illustrated by a Gantt chart which consists of a list of all processors and for each processor a list of all task modules allocated to that processor ordered by their execution time [27]. The total execution time of a schedule, defined by T_m, is 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 the shortest execution time of a given task on a system is considered to be the ultimate goal in scheduling, T_m is taken as the measure of quality to scale a mapping. However, since T_m can only be calculated once a schedule has been obtained, it is difficult to predict T_m in the process of mapping. Therefore, another objective function is to be presented as the mapping heuristic to guide the mapping process.

A detailed description of the uniform mapping algorithm is presented in Figure 4.1. This gives an overview of the algorithm. Before starting the mapping, it is necessary to compute a reduction factor denoted by f, which is essential for mapping of task graphs having more nodes than the system graphs. The reduction factor f is the ratio of the total sizes of the Rep clusters over the total sizes of the Spec clusters. It is used to estimate how many computation nodes need to share a processor. The

mapping is done recursively at each clustering level, where the best matching between Spec clusters and Rep clusters is found. For matching Spec clusters to Rep clusters, first the Spec and Rep clusters are sorted in descending order with respect to their sizes. Then to map each Spec cluster κ_{S_i} with size σ_{S_i}, search for a Rep cluster with the best matched size, i.e., closest to $f \times \sigma_{S_i}$. Therefore, the objective function to be minimized can be formulated as below:

$$|f_m| = \sum_i |f \times \sigma_{S_i} - \sigma_{R_{f_m(\kappa_{S_i})}}| \tag{4.1}$$

As shown in (4.1), the objective is to find a minimum $|f_m|$ by matching all Spec clusters to Rep clusters. When the mapping at top level is done, for each pair of the mapped Spec and Rep clusters, the same mapping procedure is continued recursively at a lower level until the mapping is fine grained to the processor level.

The time complexity of the mapping algorithm is dominated by the time of finding the best matches of all Spec clusters at all levels. At each level, the time complexity of finding the best matches of all K_i Spec clusters is $O(K_i N)$, as the total number of clusters in the Rep graph is $O(N)$, where N is the number of processors. Since the total number of Spec clusters is $O(M)$, i.e. $\sum_i K_i = O(M)$, where M is the number of nodes in original task graph. Therefore, the total time complexity of this mapping algorithm is $O(MN)$.

4.1.2 Uniform Mapping Examples

In Section 3.1, a Spec graph and a Rep graph have been constructed from the original uniform task graph and system graph, as shown in Figures 3.2 and 3.4. Figure 4.2 shows the mapping from the obtained Spec graph to the Rep graph following the mapping algorithm described above. First, calculate $\sigma_S = 9$, $\sigma_R = 8$ and $f = 8/9$. Then sort the Spec and Rep clusters at top level, and find the matching Rep cluster for each Spec cluster. The Spec cluster of size 5 is mapped onto the Rep cluster of

```
Cluster-M Uniform Mapping Algorithm
sort all Spec clusters at top level in descending order of sizes.
sort all Rep clusters at top level in descending order of sizes.
calculate σ_S, σ_R and f.
if f > 1, let f = 1.
calculate the required size of the Rep cluster matching κ_S1 to be f × σ_S1
for each Spec cluster at top level sorted list, do
begin if a Rep cluster of required size is found, then
        match the Spec cluster to the Rep cluster
        delete the Spec and Rep cluster from Spec and Rep list
end
for each unmatched Spec cluster, do
begin if the size of the first Rep cluster > the required size, then
        begin split the Rep cluster into two parts with one part having the required size
                match the Spec cluster to this part
                insert the other part to proper position of the sorted Rep cluster list
        end
        else
        begin merge Rep clusters until the sum of sizes ≥ the required size
                if =, then
                match the Spec cluster to the merged Rep cluster
                else
                begin split the merged Rep cluster into two parts with one having required size
                        match the Spec cluster to this part
                        insert the other part to the sorted Rep list
                end
        end
end
for each matching pair of Spec cluster and Rep cluster, do
begin if the Rep cluster contains only one processor, then
        map all the modules in the Spec cluster to the processor
        else
        begin go to the sub-clusters of the Spec and Rep cluster
                (thus they are pushed to top level)
                call the same mapping algorithm for these clusters
        end
end
```

Figure 4.1 Uniform mapping algorithm

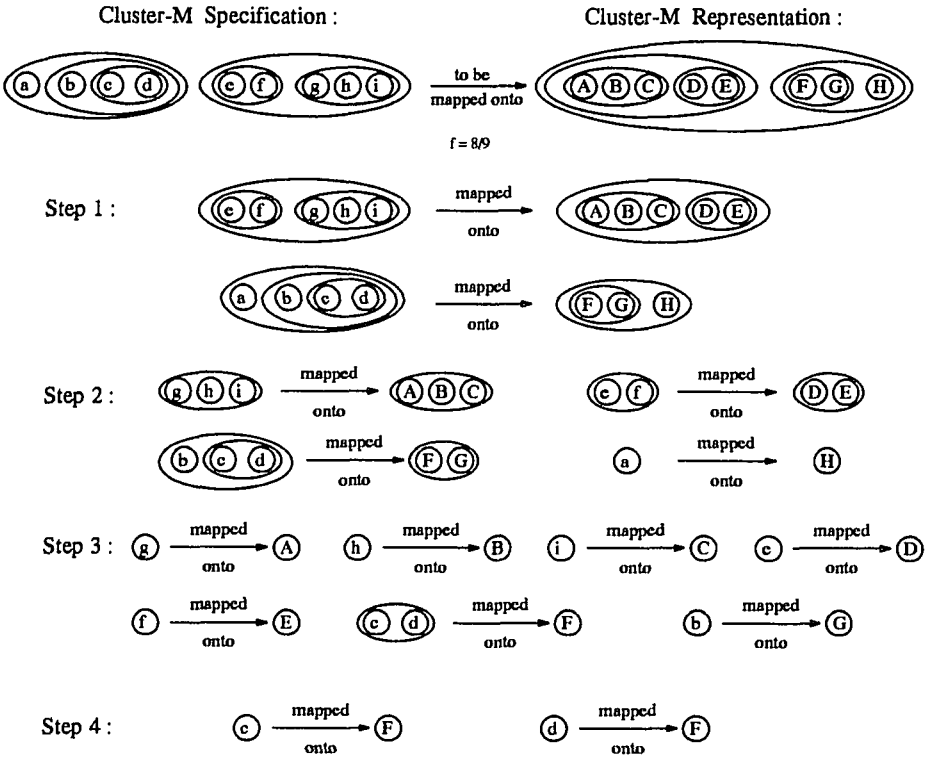


Figure 4.2 A mapping example.

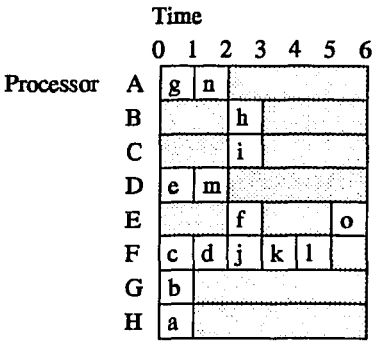


Figure 4.3 Gantt chart of the obtained schedule.

the same size, however the Spec cluster of size 4 has to be mapped onto the Rep cluster of size 3 since this is the closest matching of sizes. Then the same procedure is applied to the Spec clusters at the lower level. As shown in step 2 in Figure 4.2, task module a is mapped onto Rep cluster H, which contains a single processor. In step 3, modules b, e, f, g, h and i are mapped onto corresponding processors. Finally in step 4, modules c and d are both mapped onto processor F. Since modules j, k and l are embedded to module d (see Figure 3.2), they are also mapped onto processor F, to which d is mapped onto. Similarly, modules m, n and o are mapped onto processors D, A and E respectively. Now all the task modules in the original task graph have been mapped onto corresponding processors. Figure 4.3 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, $T_m = 6$.

4.2 Uniform Mapping Comparison Results

This section presents a set of experimental results that have been obtained in comparing Cluster-M mapping algorithm with other leading techniques for mapping uniform arbitrary task graphs onto uniform arbitrary system graphs. The examples selected here are the same as those presented and experimented by the authors of the papers reporting the leading techniques [8, 56]. The following three 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; and (3) the number of processors used, N_m. From (2) and (3), speedup $S_m = \frac{T_s}{T_m}$ and efficiency $\eta = \frac{S_m}{N_m}$ can be obtained, where T_s is the sequential execution time of the task. In the following, we present the comparison results for both the scheduling problem and the allocation problem.

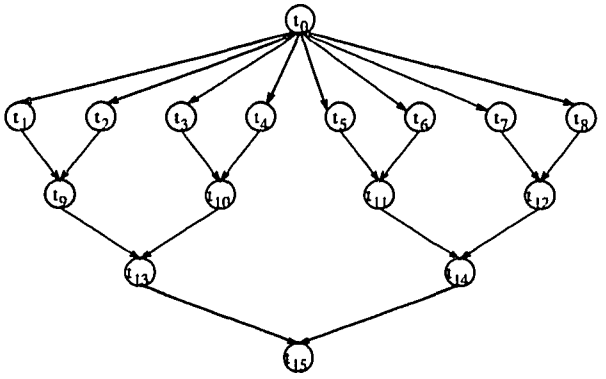
4.2.1 Task Scheduling Results

In comparison study of task scheduling of uniform graphs, we choose Lee and Aggarwal's mapping strategy [56]. Their mapping strategy considers the task graph as directed graph and differentiate nodes and edges into different computation steps and communication phases in order to accurately calculate the actual communication cost between two non-adjacent processors. However, Lee and Aggarwal's strategy maps the entire task graph onto the system graph without graph contraction or clustering. Also, it assumes that the order of the task graph is no greater than that of the system graph. The time complexity of Lee and Aggarwal's algorithm is $O(N^3)$, while ours is $O(MN)$ (i.e., if $M = N$, then ours is $O(N^2)$).

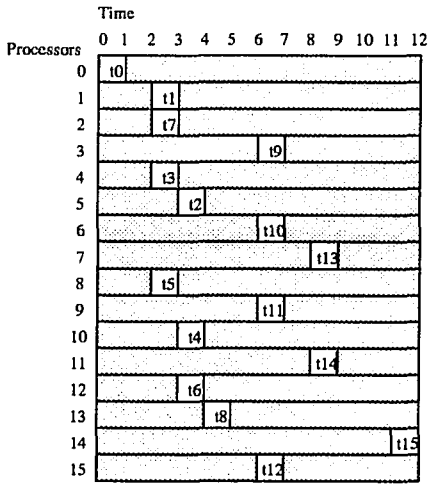
Given a task graph as shown in Figure 4.4(a), the mapping obtained by Lee and Aggarwal on a 16-processor hypercube is (t_0 t_1 t_7 t_9 t_3 t_2 t_{10} t_{13} t_5 t_{11} t_4 t_{14} t_6 t_8 t_{15} t_{12}) [56]. The final schedule following the task graph precedency is illustrated in Figure 4.4(b), while the schedule obtained from Cluster-M mapping is illustrated in Figure 4.4(c). An optimal schedule, which also uses fewer number of processors, is shown in Figure 4.4(d).

4.2.2 Task Allocation Results

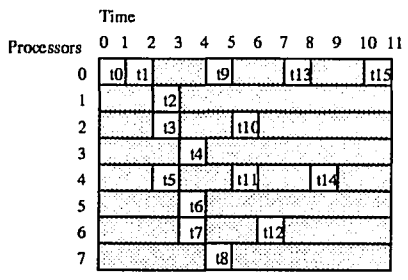
The goal of task allocation is to minimize the communication delay between processors and to balance the load among processors. The problem of task allocation arises when specifying the order of executing the task modules is not required.



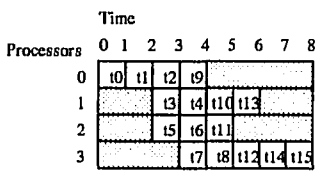
(a) Task graph



(b) Lee and Aggarwal's mapping, $T_c = O(N^3)$, $T_m = 12$, $N_m = 16$



(c) Cluster-M mapping, $T_c = O(MN)$, $T_m = 11$, $N_m = 8$



(d) An optimal mapping, $T_c = O(2^{MN})$, $T_m = 8$, $N_m = 4$

Figure 4.4 Comparison example with Lee and Aggarwal's strategy.

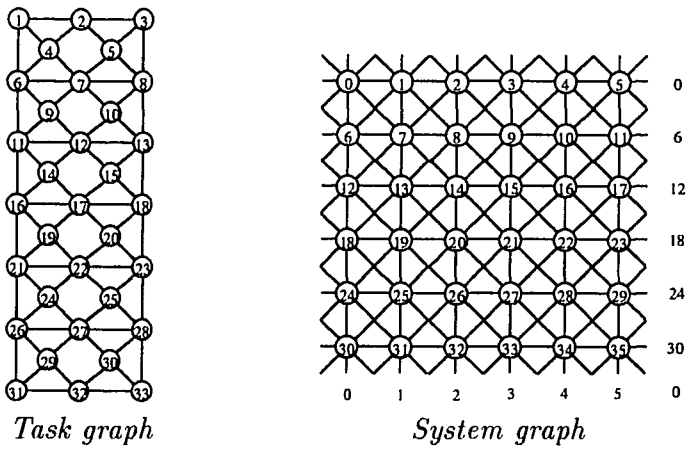


Figure 4.5 Comparison with Bokhari's mapping: task and system graph.

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. The measure of mapping quality in task allocation is still T_m.

Cluster-M mapping algorithm are compared to Bokhari's mapping (allocation) algorithm [8] using uniform undirected task graphs. Bokhari's algorithm has the running time complexity of $O(N^3)$, while Cluster-M has $O(MN)$. Bokhari's algorithm 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.

In comparing Cluster-M with Bokhari's for the example shown in Figure 4.5 having a 33-node task graph and a 6×6 finite element machine (FEM) [8], a Sun SPARC station 1 was used. The results are shown in Table 4.1. The lower bound on T_m as described before is 9, and yet both Cluster-M and Bokhari's algorithms have obtained near optimal results of $T_m = 17$ and 13, respectively. The above example uses the same structured task and system graph as tried in [8]. Other randomly generated task and system graphs have also been tested. Table 4.2 shows the mapping results and comparisons for 10 randomly generated task and system graphs of 10 nodes. Similar results were obtained for other random graphs. Bokhari's

algorithm is not applicable to mapping a larger task graph onto a smaller system graph (called cardinality variation [5]). However, Cluster-M mapping is an efficient approach to the mapping problems having topological and cardinality variation in their input graphs [5].

4.3 Cluster-M Non-Uniform Mapping

This section presents an efficient mapping algorithm which produces sub-optimal matching of a non-uniform Spec graph onto a non-uniform Rep graph in $O(MP)$ time, where $P = \max(M, N)$. A high level description of the mapping algorithm is presented in Section 4.3.1. In Section 4.3.2, a few examples are given to illustrate the non-uniform mapping algorithm.

4.3.1 Non-Uniform Mapping Algorithm

The mapping function is defined to be $f_m : V_t \xrightarrow{onto} V_p$, as in Section 4.1. However, since both the task graph and the system graph may be non-uniform, we assume that the communication time of the task graph edge (t_i, t_j) is equal to $\sum_{(x,y) \in path(f_m(t_i), f_m(t_j))} \frac{D_{t_i t_j}}{R_{xy}}$, where $path(p_i, p_j)$ is the shortest path between processor p_i and p_j.

A detailed description of the non-uniform mapping algorithm is presented in Figure 4.6. In the following, an overview of the algorithm is given. Before the mapping begins, it is necessary to compute a reduction factor denoted by f, which is essential for mapping of task graphs having more nodes than the system graphs. The reduction factor f is the ratio of the total sizes of the Rep clusters over the total sizes of the Spec clusters. This is used to estimate how many computation nodes are to share a processor. The mapping is done recursively at each clustering level, where the best matching between Spec clusters and Rep clusters are found. For matching Spec clusters to Rep clusters, first the Spec and Rep clusters are sorted

Table 4.1 Mapping of Bokhari's algorithm and Cluster-M

Task	Mapped processor	
module	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_c	$O(N^3)$	$O(MN)$
T_m	13	17
T_r (sec)	152.5	0.05

Table 4.2 Comparisons of mappings of Bokhari's algorithm and Cluster-M

Random graphs	T_m			running time (sec)	
of 10 nodes	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

in descending order with respect to the four parameters $(\sigma, \delta, \Pi, \pi)$. 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 number of levels are sorted first.

Second, each of the Spec clusters (denoted by κ_{S_i}) is mapped as follows. First search for the Rep cluster (denoted by κ_{R_j}) with the best matched size, i.e., closest to $f \times \sigma_{S_i}$. The first objective function in mapping is thus the same as formulated in Equation (4.1) in Section 4.1.1. If multiple Rep clusters with the matching size are found, one is selected with the minimum estimated execution time. The estimated execution time of mapping Spec cluster κ_{Si} onto Rep cluster κ_{Rj}, $\tau(\kappa_{Si}, \kappa_{Rj})$, is equal to the number of clustering levels of κ_{Si} times the average computation and communication time at each level, as formulated in Equation (4.2). If no Rep cluster with a matching size can be found for a Spec cluster, either merge or split (unmerge) Rep clusters until a matching Rep cluster is found.

$$\tau(\kappa_{Si}, \kappa_{Rj}) = \delta_{Si}(\frac{1}{\delta_{Rj}} \times \frac{1}{f} + \frac{\Pi_{Si} \times \delta_{Rj}}{\Pi_{Rj} \times \delta_{Si}}) \tag{4.2}$$

Thirdly, for every matched pair of the Spec and Rep clusters, the following is done to embed communication intensive nodes together. (This is similar to the

clustering process in [70, 82, 86]. However, here it is only done in the mapping step so that the clustering of the task graph is kept independent of the system graph, as described in the Chapter 3.) If a Spec cluster has multiple sub-clusters and the average communication time between these sub-clusters is greater than the possible computation time of a sub-cluster as formulated in Inequality (4.3), then embed the sub-clusters onto a sub-cluster having the largest size, and calculate the parameter quadruple for the new cluster. We then insert it in the proper position in the sorted list of Spec clusters for mapping, and repeat the matching as described above by Equations (4.1) and (4.2) 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 level, and therefore this Spec cluster is removed from the list.

$$\frac{\pi_{Si}}{\pi_{Rj}} > \frac{\min(\sigma_{sub-cluster} \times \delta_{sub-cluster})}{\delta_{Rj}} \times \frac{1}{f} \tag{4.3}$$

In the above mapping algorithm, the worst case of a mapping at a level i happens either when (case 1) for each Spec cluster, all the remaining Rep clusters have the matching size, therefore Equation (4.2) is used to select the best Rep cluster; or when (case 2) for each Spec cluster, no Rep cluster of matching size is found, therefore Rep clusters are merged/split recursively until a Rep cluster of matching size is obtained. Suppose the number of Spec clusters at level i is K_i. In both cases described above, or in any combination of the two cases, it takes $O(K_i N)$ time to find the best matches of 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 (4.3) is satisfied, the extra time taken in embedding will be $O(M)$. Since the total number of Spec clusters is $O(M)$, i.e., $\sum_i K_i = O(M)$, where M is the number of nodes in original task graph. Therefore, the total time complexity of this mapping algorithm is $\sum_i (K_i N + M) = O(MN) + O(M^2) = O(MP)$, where $P = \max(M, N)$.

```
Cluster-M Non-Uniform Mapping Algorithm
sort all Spec clusters at top level in descending order of σ_S, δ_S, Π_S, and π_S.
sort all Rep clusters at top level in descending order of σ_R δ_R, Π_R, and π_R.
calculate f.
calculate the required size of the Rep cluster matching κ_S, to be f × σ_S,
for each Spec cluster κ_S, at top level sorted list, do
begin if the cluster has only one sub-cluster
      then go to a lower level where there are multiple or no sub-clusters
      if at least a Rep cluster of required size is found
      then begin select the Rep cluster κ_Rj with minimum τ(kappa_S,, kappa_Rj)
                 match the Spec cluster to the Rep cluster
                 delete the Spec and Rep cluster from Spec and Rep list
           end
end
for each unmatched Spec cluster, do
begin if the size of the first Rep cluster > the required size
      then begin split the Rep cluster into two parts with one part having required size
                 match the Spec cluster to this part
                 insert the other part to proper position of the sorted Rep cluster list
           end
      else  begin merge Rep clusters until the sum of sizes ≥ the required size
                 if = then match the Spec cluster to the merged Rep cluster
                 else     begin split the merged Rep cluster into two parts
                                with one having required size
                                match the Spec cluster to this part
                                insert the other part to the sorted Rep list
                          end
           end
end
for each matching pair of Spec cluster and Rep cluster, do
begin if the Rep cluster contains only one processor
      then map all the modules in the Spec cluster to the processor
      else if Inequality (4.3) is satisfied
           then  begin  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
                        embed these sub-clusters onto the selected one
                        calculate the parameters for the new cluster
                        insert it into the sorted Spec cluster list
                 end
           else  begin  delete the Spec and Rep cluster from Spec and Rep list
                        go to the sub-clusters of the Spec and Rep cluster
                        call the same mapping algorithm for these clusters
                 end
end
```

Figure 4.6 Non-uniform mapping algorithm.

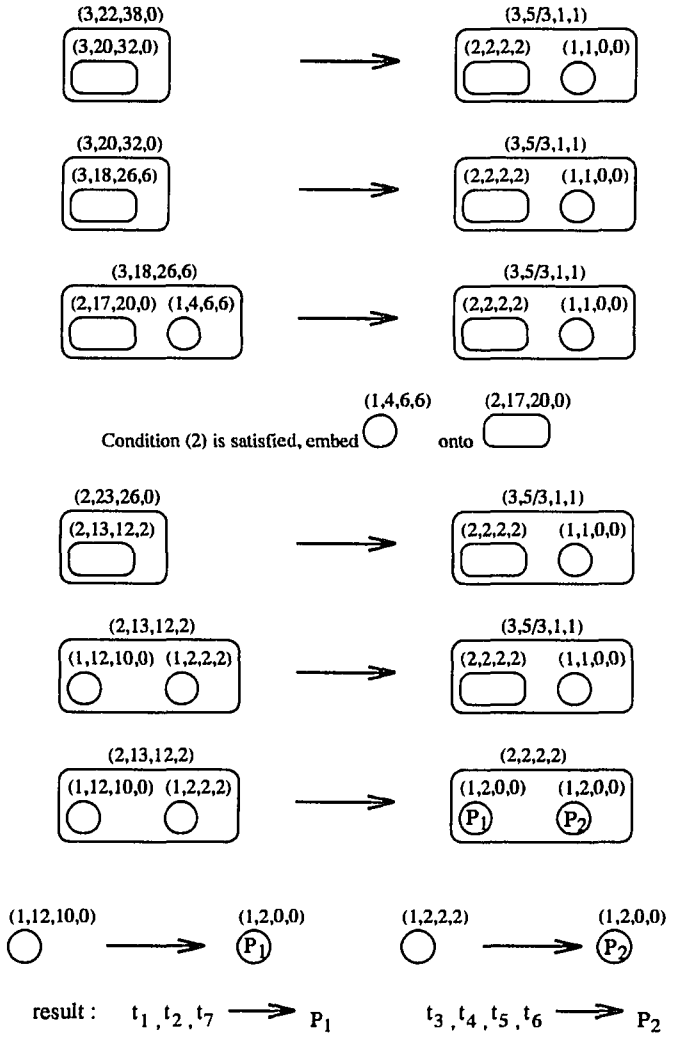


Figure 4.7 A mapping example.

4.3.2 Non-Uniform Mapping Examples

In Section 3.2, a Spec graph and a Rep graph were constructed from the original non-uniform task graph and system graph, as shown in Figure 3.16 and 3.18. Figure 4.7 shows the snapshot of the mapping process from the obtained Spec graph to the Rep graph following the mapping algorithm described above. Figure 4.8 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, $T_m = 10$.

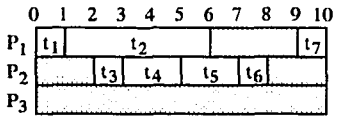


Figure 4.8 Gantt chart of the obtained schedule.

To show that the same task graph can be mapped onto various system graphs, three different system graphs are chosen and shown in Figure 4.9. Figure 4.9(a) is the same task graph as shown in Figure 3.16. Figure 4.9(b) shows a uniform fully connected system graph and its clustering. The computation speed of each processor and the communication bandwidth of each communication link is equal to 2. The result of Cluster-M mapping onto this graph is shown in Figure 4.9(c). In Figure 4.9(d), the system is fully connected with unit computation speed at each processor, but having higher communication bandwidths at the edges. In this case, the Cluster-M mapping algorithm distributes the task modules as shown in Figure 4.9(e), to all three processors to utilize the relatively high communication bandwidth available. On the other hand, if the system is fully connected with unit communication bandwidth but having higher computation speeds at the processors as shown in Figure 4.9(f), 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 4.9(g).

Finally, an example of mapping a real application task onto a heterogeneous system is given. The Gaussian elimination algorithm used in LINPACK [23, 24] is chosen. The FORTRAN code is given in Figure 4.10. Suppose it takes 1 unit of time to do an addition or subtraction, and it takes 2 units of time to do a multiplication or division of two real numbers. It is also assumed that the communication amount of sending/receiving each real number to be 1. A task graph for computing the Gaussian elimination of a 5×5 matrix is shown in Figure 4.11(a). In each task module T_j^k, column j is modified by using column k. Suppose that the system running this task

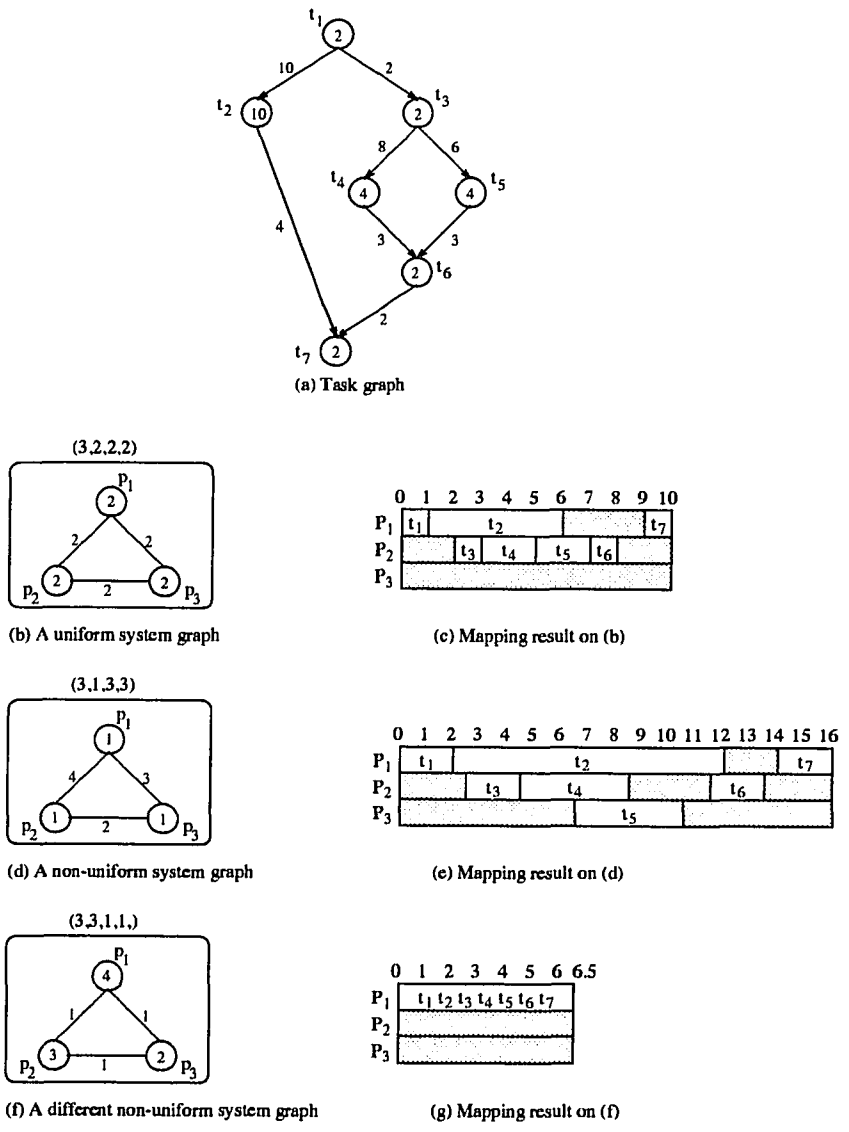


Figure 4.9 Mappings on different system graphs.

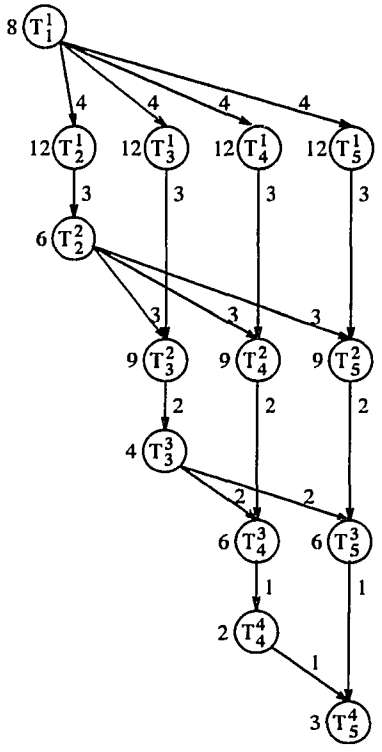
```
      SUBROUTINE KJI(A,LDA,N)
C
C     SAXPY
C     FORM KJI-SAXPY
C
      REAL A(LDA,N)
      DO 40 K=1,N-1
         DO 10 I=K+1,N
            A(I,K)=-A(I,K)/A(K,K)
 10      CONTINUE
         DO 30 J=K+1,N
            DO 20 I=K+1,N
               A(I,J)=A(I,J)+A(I,K)*A(K,J)
 20         CONTINUE
 30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

Figure 4.10 Gaussian elimination algorithm.

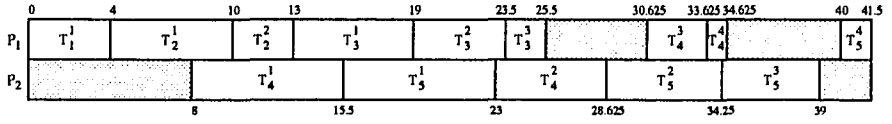
contains only two workstations p_1 and p_2. Also, p_1 and p_2 have speed of 2 and 1.6 respectively, and are connected with a link of bandwidth 1. The mapping result using Cluster-M technique is illustrated in Figure 4.11(b).

4.4 Non-Uniform Mapping Comparison Results

Presented in this section is a set of experimental results that have been obtained in comparing Cluster-M mapping algorithm with other leading techniques. 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. Again, the following three 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; and (3) the number of processors used, N_m.



(a) Task graph



(b) Mapping result

Figure 4.11 The mapping example of a 5×5 matrix Gaussian elimination.

Since there is no existing mapping technique which maps a machine-independent arbitrary non-uniform task onto an arbitrary non-uniform system, it is not easy to choose candidates for the comparison study. Therefore, the comparison study is focused on the leading mapping techniques designed for arbitrary non-uniform tasks, but for specialized systems only. The mapping techniques in this category include McCreary and Gill's Clan [61], El-Rewini and Lewis's MH [26], Sarkar's Edge-Zeroing clustering [70], Wu and Gajski's MCP [82], and Yang and Gerasoulis' DSC [86]. These algorithms have proven to be very effective and efficient in mapping arbitrary and non-uniform directed tasks. Similar to Cluster-M mapping algorithm, these algorithms also cluster the task graphs before the mapping. Except for MH, which is a list scheduling algorithm, they all assume that the target systems are fully connected with an 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. For a fully connected system, it does not matter to which processor a cluster is mapped. If the system graph is arbitrary but uniform, some allocation algorithms such as Bokhari's pairwise exchange mapping [8] can be used for one-to-one mapping of clusters of task modules onto processors [85]. The following comparison results show that Cluster-M produces better or similar mapping results with less time complexity compared to the other mapping techniques studied here.

4.4.1 Comparison with McCreary and Gill's Clan Algorithm

McCreary and Gill's Clan algorithm finds suitable sized grain (cluster) of task modules to be assigned to the same processor before scheduling the tasks [61]. A clan is a set of nodes X of the directed task graph G_t iff for all $t_x, t_y \in X$ and all $t_z \in G_t - X$ such that (1) t_z is a parent node of t_x iff t_z is a parent node of

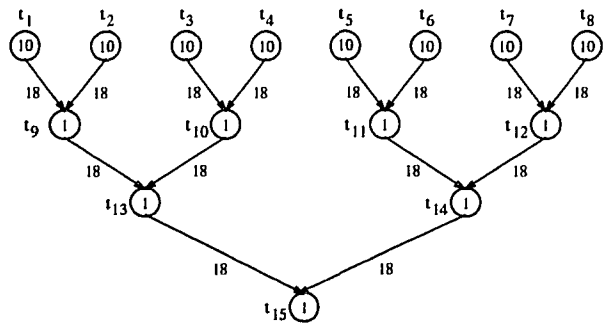
t_y; or (2) t_z is a child node of t_x iff t_z is a child node of t_y. Informally, a clan is a subset of nodes where every element outside the set is related in the same way to each member in the set. An $O(M^3)$ parsing algorithm has been proposed that decomposes a task graph into clans. In McCreary and 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 and Gill's algorithm, the following task modules of the task graph shown in Figure 4.12(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

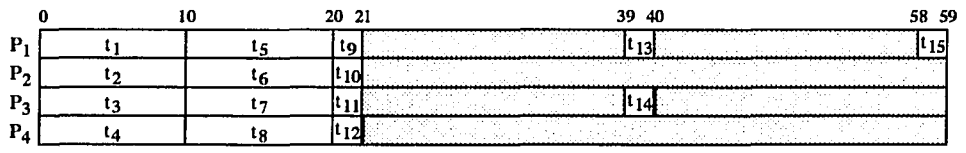
As task module 13 receives data from 9 and 10, it is assigned to P_1. Similarly, 14 is also assigned to P_2 and 15 is assigned to P_1. The schedule resulting from this assignment appears in Figure 4.12(b). Even though Cluster-M clustering and mapping algorithms are different and more generic than Clan, similar results have been obtained as shown in Figure 4.12(c).

4.4.2 Comparison with El-Rewini and Lewis's Mapping Heuristic

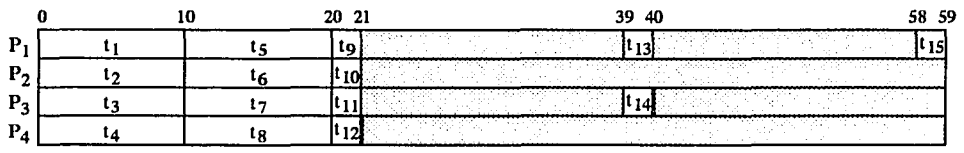
Next Cluster-M mapping algorithm is compared with El-Rewini and Lewis's mapping heuristic (MH) algorithm [26]. The time complexity of MH is $O(M^2N^3)$, while Cluster-M has an $O(MN)$ time complexity. Given a task graph as shown in Figure 4.13(a) and a uniform 8-processor hypercube ($D_{ij} = 1$, if edge (t_i, t_j) exists, $1 \leq i, j \leq 18$), the schedule obtained from MH is illustrated by a Gantt chart in Figure 4.13(b) [26]. Similarly, the Gantt chart of the schedule obtained by Cluster-M mapping is shown in Figure 4.13(c). An optimal schedule is also shown in Figure 4.13(d). Both



(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 4.12 Comparison example with Clan.

MH and Cluster-M mappings have produced close to optimal T_m for this example, yet Cluster-M is faster by a factor of $O(MN^2)$.

4.4.3 Comparison with Wu-Gajski's MCP Algorithm

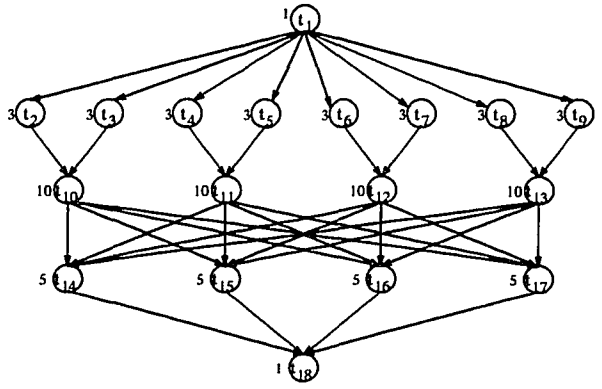
The Modified Critical Path (MCP) algorithm [82] is based on critical path introduced by Hu [42]. A critical path in a DAG is a path of greatest weight from a source node to a sink node, including the weights of all the nodes and edges along this path. The critical paths can be shortened by removing communication 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 3.16 and the system graph as shown in Figure 4.9(b), a transformed task graph incorporating the information about the system graph has to be generated first as shown in Figure 4.14(b). The mapping results by Cluster-M and MCP are shown in Figure 4.14(c) and (d) respectively. Cluster-M has produced a mapping with $T_m = 10$ while MCP has $T_m = 10.5$. The time complexity of MCP is $O(M^2 \log M)$.

4.4.4 Comparison with Sarkar's Edge-Zeroing Algorithm

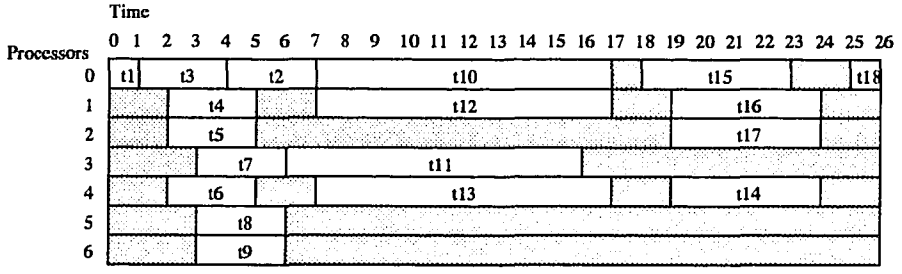
The basic idea of Sarkar's Edge-Zeroing algorithm is to repetitively zero the highest weighted edge if it does not increase the estimated T_m, until all the edges have been examined. Its time complexity is $O(|E_t|(M + |E_t|))$, where $|E_t|$ is the number of edges in the task graph. Figure 4.14(e) shows the mapping result obtained by the Edge-Zeroing algorithm on the same example used for MCP in Figure 4.14. This result matchs that of Cluster-M.

4.4.5 Comparison with Yang and Gerasoulis' DSC Algorithm

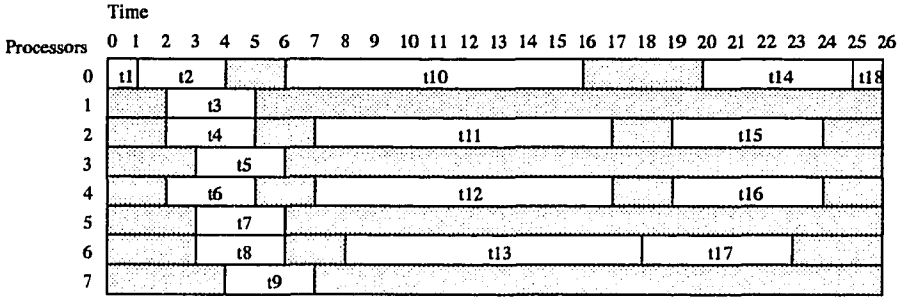
Yang and Gerasoulis' Dominant Sequence Clustering (DSC) algorithm [86] is also based on critical path and edge zeroing, and it incorporates several other heuristics



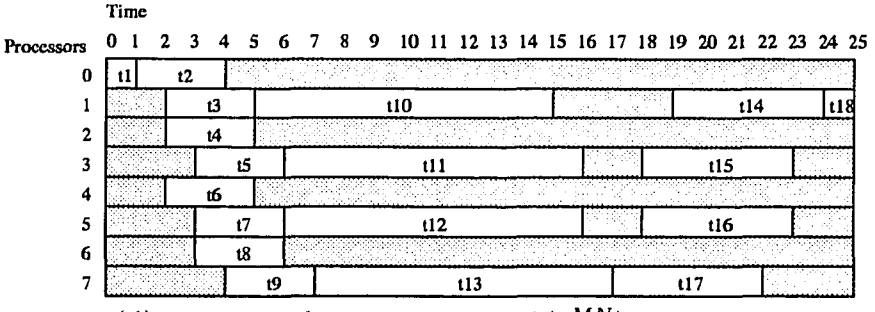
(a) Task graph



(b) MH mapping, $T_c = O(M^2N^3)$, $T_m = 26$, $N_m = 7$

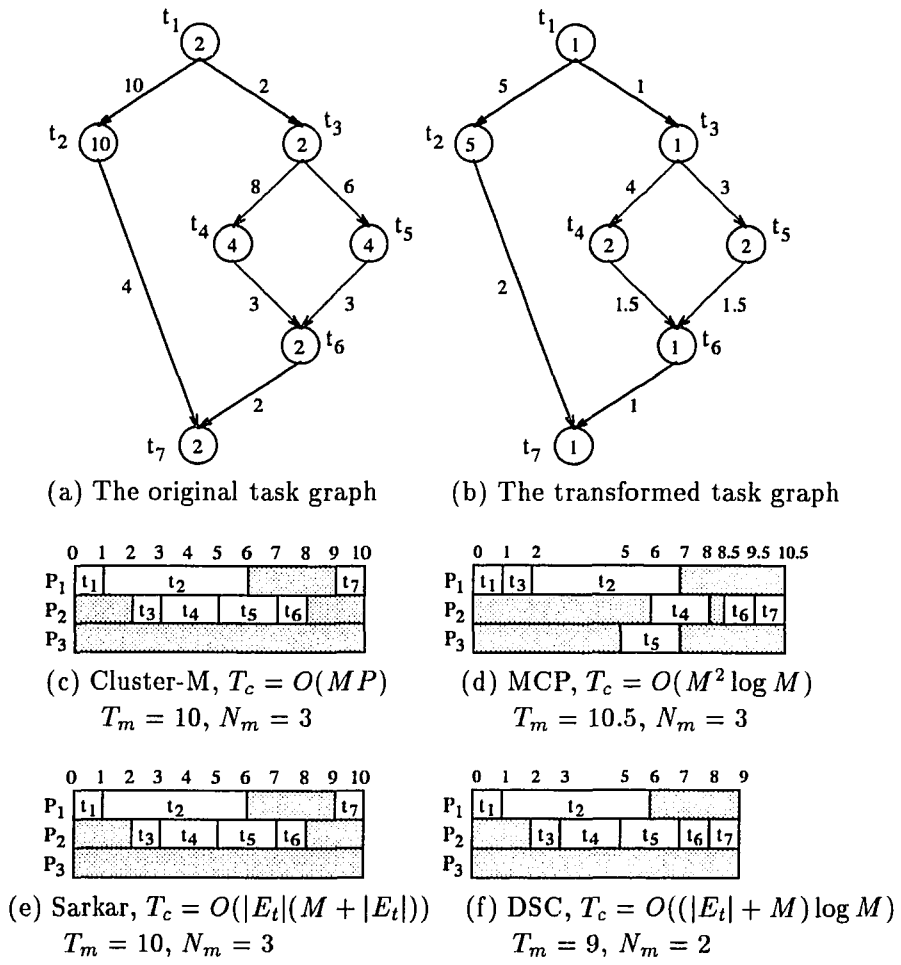


(c) Cluster-M mapping, $T_c = O(MP)$, $T_m = 26$, $N_m = 8$



(d) An optimal mapping, $T_c = O(2^{MN})$, $T_m = 25$, $N_m = 8$

Figure 4.13 Comparison example with MH.



(a) The original task graph

(b) The transformed task graph

(c) Cluster-M, $T_c = O(MP)$
$T_m = 10$, $N_m = 3$

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

(e) Sarkar, $T_c = O(|E_t|(M + |E_t|))$
$T_m = 10$, $N_m = 3$

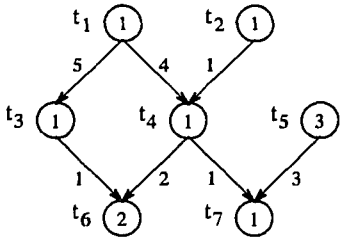
(f) DSC, $T_c = O((|E_t| + M) \log M)$
$T_m = 9$, $N_m = 2$

Figure 4.14 Comparison example with MCP, Sarkar and DSC.

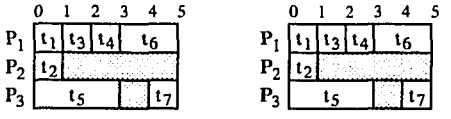
for better clustering. DSC can find optimal schedules for some special DAG's such as fork and join. However, the task graphs considered in DSC are not machine-independent, and similar to the above three techniques, it cannot map to non-uniform systems such as those shown in Figure 4.9(d) and (f). The time complexity of DSC is $O((|E_t| + M)\log M)$, where $|E_t|$ is the number of edges in the task graph.

Figure 4.14(f) shows the mapping result obtained by DSC for the same example which was studied in comparison with MCP and Sarkar's algorithms. Among these results for this example, DSC's is the best and actually the optimal, yet the result by Cluster-M is very close to optimal. In the following, we show several more comparison examples with DSC. These examples are taken from [84, 86]. Figure 4.15 and 4.16 show the mapping of two task graphs onto unbounded number of identical processors fully connected by identical communication links. The computation speed and communication bandwidth of the system in Figure 4.15 are both 1, while the computation speed and communication bandwidth of the system in Figure 4.16 are both 2.

Finally, the same example is taken that was used for comparison with MH in Figure 4.13. As shown in Figure 4.13, the mapping by MH has $T_m = 26$ and $N_m = 7$, and an optimal mapping uses 8 processors and has $T_m = 25$. The mapping results by Cluster-M and DSC are illustrated in Figure 4.17(b) and (c). If a 4-processor hypercube is used, the mappings of the same task graph by Cluster-M and DSC are shown in Figure 4.17(d) and (e).

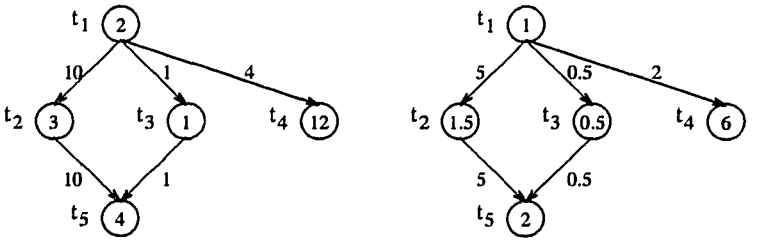


(a) Task graph

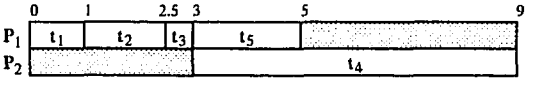


(b) Cluster-M, $T_m = 5$, $N_m = 3$ (c) DSC, $T_m = 5$, $N_m = 3$

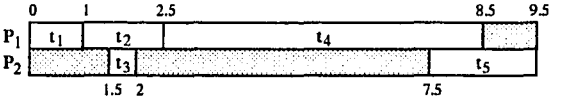
Figure 4.15 Comparison example 2 with DSC.



(a) Task graph (b) Transformed task graph

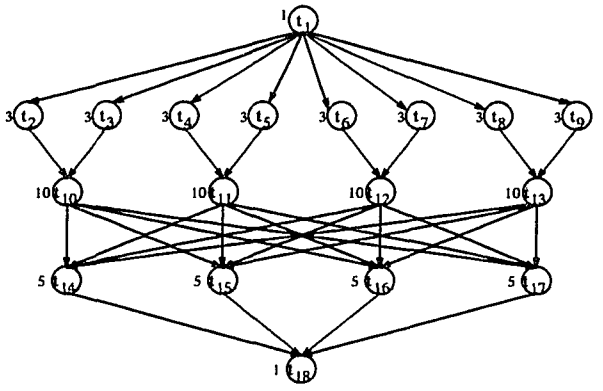


(c) Cluster-M mapping result, $T_m = 9$, $N_m = 2$

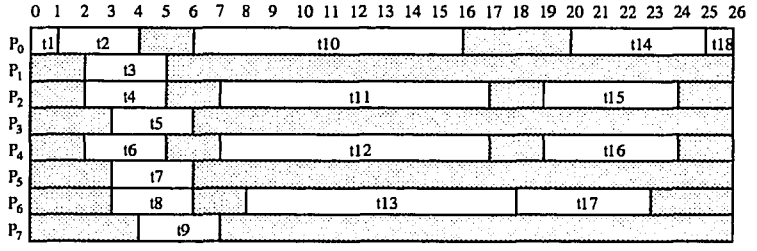


(d) DSC-I's mapping result, $T_m = 9.5$, $N_m = 2$

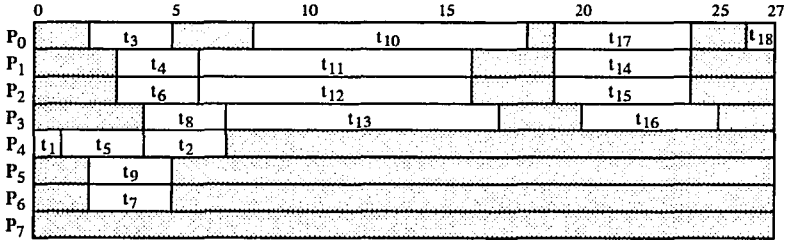
Figure 4.16 Comparison example 3 with DSC.



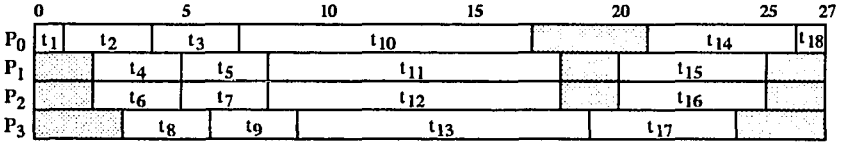
(a) Task graph



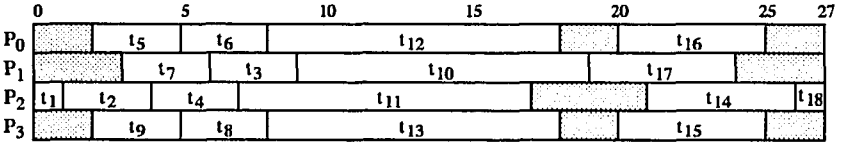
(b)Cluster-M mapping on 8 processors, $T_m = 26$, $N_m = 8$



(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 4.17 Comparison example 4 with DSC.

CHAPTER 5

HIERARCHICAL CLUSTER-M MAPPING FOR HETEROGENEOUS COMPUTING

Heterogeneous Computing (HC) has been proposed as a novel approach towards solving computationally demanding application tasks due to exploiting the heterogeneity of a variety of high-performance computers [38, 51]. There are two types of HC systems: a mixed-mode machine and a mixed-machine system. A mixed-mode machine is a single machine which can operate in different modes of parallelism [34], while a mixed-machine system is a suite of diverse machines which are interconnected by a high-speed network. These diverse machines are high-performance computers with different parallelism modes (such as vector processors, SIMD, MIMD, and mixed-mode machines). Khokhar et. al. [51] have addressed the challenges and issues posed by HC systems. The trend of HC has also brought researchers' attention to the problem of mapping tasks onto a suite of heterogeneous computers [80, 16, 54, 55, 67]. However, many existing mapping techniques which were designed for uniform homogeneous systems are not suitable for mappings on heterogeneous systems. This chapter studies the problem of mapping specialized application tasks onto heterogeneous systems by applying the mapping algorithms presented in the last chapter.

In [37] Freund proposed the Optimal Selection Theory (OST) which is a proof of existence of an optimal configuration of heterogeneous machines for executing an application task such that the total execution time is minimized. OST was then augmented by Wang et. al. [80], called the Augmented Optimal Selection Theory (AOST), to incorporate non-optimal machine choices and non-uniform decompositions of code segments. This chapter first presents the Heterogeneous Optimal

Selection Theory (HOST), which is an extension to OST and AOST. HOST includes various additional assumptions to be more suitable for a HC environment. Based on HOST, a modified Cluster-M mapping algorithm is presented for mapping bound-degree heterogeneous task graphs onto bound-degree heterogeneous system graphs. At each step of mapping, instead of using a greedy algorithm for matching the Spec clusters to the Rep clusters as we did in last chapter, the optimal matching is found by using Integer Linear Programming (ILP). The comparison results show that the modified mapping algorithm produces better mappings than the original algorithm as well as other heterogeneous mapping techniques.

5.1 Heterogeneous Optimal Selection Theory (HOST)

In Freund's Optimal Selection Theory (OST) [37], it is assumed that the number of machines available is unlimited and an application task comprises several uniform and non-overlapping code segments. Each segment has homogeneous parallelism embedded in its computations. Also, code segments are considered to be executed serially. A code segment is further decomposed into different code blocks. All code blocks within a code segment have the same type of parallelism and can be executed concurrently. The goal of OST is to assign the code blocks within each code segment to the available matching machine types such that the code segment can be optimally executed. Augmented Optimal Selection Theory (AOST) [80] 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 which is most suitable for one type of machine may have to be assigned to another type.

HOST, presented in this section, is an extension to AOST in two ways: it incorporates the effects of various fine grain mapping techniques available on individual machines, and the task is assumed to have heterogeneous embedded parallelism.

The input format to HOST, as shown in Figure 5.1, allows concurrent execution of mutually independent code segments. An application task is decomposed into several subtasks. Subtasks are executed serially. Each subtask may contain a collection of code segments which can be executed in parallel. A code segment consists of homogeneous parallel instructions. Each code segment is further divided into several code blocks which can be executed concurrently. These code blocks are to be assigned to the machines of the same type. A machine type is identified according to the underlying architectures, such as SIMD, MIMD, vector, scalar, etc. Each machine type may have more than one model, for example, a hypercube and a mesh may be two models of SIMD machine type. In HOST, heterogeneous code blocks of different code segments can be executed concurrently on different types of machines, exploiting heterogeneous parallel computations embedded in a given application.

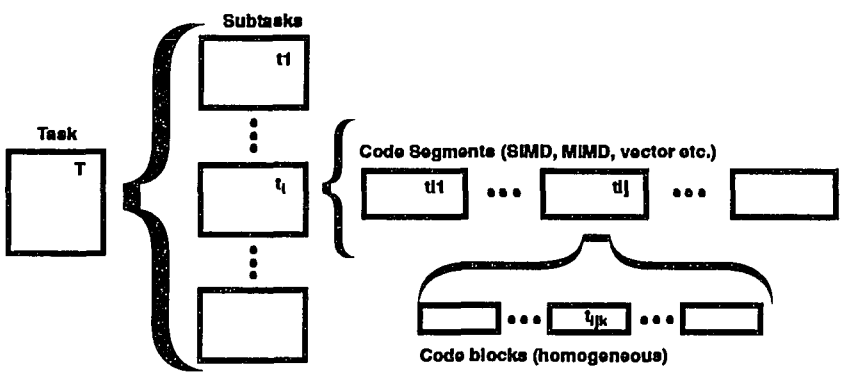


Figure 5.1 Input format to HOST.

To express the formulation of HOST, some parameters need to be defined. Let S be the number of code segments of a given subtask, and M be the number of different machine types to be considered. Let $\eta[t]$ be the number of machine models of type t, $\alpha[t]$ be the number of mappings available on machine type t, and $\beta[t,l]$ be the number of machines of model l of type t available. Assume $v[t,j]$ is the maximum number of code blocks code segment j can be decomposed into. Define

$\gamma[t,j]$ to be the number of machines of type t that are actually used to execute code segment j. Therefore, $\gamma[t,j] = \min(\sum_{l=1}^{\eta[t]} \beta[t,l], v[t,j])$. A parameter $m[t,k]$ is defined to specify the effect of the mapping technique available for a code block k on machine type t. Assume that for a particular mapping m on machine type t, the best matched code segment can obtain the optimal speedup $\theta[t,m]$ in comparison to a baseline system. A real number $\pi[t,j]$ indicates how well a code segment j can be matched with machine type t. $\Lambda[t,k]$ is a utilization factor when running code block k on a machine of type t. Thus $0 \leq \pi[t,j] \leq 1$ and $0 \leq \Lambda[t,k] \leq 1$. Let $p[j]$ be the percentage of time spent executing code segment j within overall execution of a given subtask on baseline machine. $\sum_{j=1}^{S} p[j] = 1$. Similar to the definition of $p[j]$, let $p[j,k]$ be the percentage of time spent executing code block k within overall execution of code segment j on baseline machine. $\sum_{k=1}^{\gamma[t,j]} p[j,k] = 1$.

Suppose code segment j is assigned to machine type t. For each code block k within code segment j, there is a mapping $m[t,k]$. Let $\mu[t,j]$ be mapping vector for code segment j on machine type t.

$$\mu[t,j] = (m[t,1], m[t,2], \cdots, m[t,\gamma[t,j]]), 1 \leq m[t,k] \leq \alpha[t].$$

With this mapping vector μ on machine type t, the relative execution time of segment j will be:

$$\delta[t,j,\mu] = \max_{1 \leq k \leq \gamma[t,j]} \frac{p[j] \times p[j,k]}{\theta[t,m[t,k]] \times \pi[t,j] \times \Lambda[t,k]}.$$

Therefore, different mappings, μ, available on machine type t result in different execution times of segment j. Let $\lambda[t,j]$ be the minimum execution time of segment j among all the possible mappings on type t.

$$\lambda[t,j] = \min_{\mu[t,j]} \delta[t,j,\mu[t,j]]$$

Let the machine type selection vector τ indicate the selection of machine types for code segment 1 to S, such that $\tau = (t[1], t[2], \cdots, t[S])$. Let $c[t[j]]$ denote the cost

Table 5.1 Notations used in HOST formulation

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

of machine selected to execute code segment j, and C be the total cost constraint. Define $\chi[\tau]$ to be the execution time of the given subtask with heterogeneous machine type selection τ on all the code segments, such that $\chi[\tau] = \max_{1\leq j\leq S} \lambda[t[j],j]$, then HOST is formulated as follows:

For any subtask, there exists a τ with

$$\min_{\tau} \chi[\tau] \textit{ subject to } \sum_{j=1}^{S}(\gamma[t[j],j] \times c[t[j]]) \leq C$$

For an easy reference, all the notations used in HOST formulation are listed in Table 5.1.

5.2 Modeling the Input to HOST

HOST, as described in the previous section, is an existence proof for an optimal selection of processors for a given subtask in HC. The input formulation in HOST

assumes that a parallel task T is divided into subtasks t_i, $1 \leq i \leq N$. Each subtask t_i is further divided into code segments t_{ij}, $1 \leq j \leq S$, which can be executed concurrently. Each code segment within a subtask can belong to a different type of parallelism (i.e. SIMD, MIMD, vector, etc.), and thus should ideally be mapped onto a machine with a matching type of parallelism. Each code segment may further be decomposed into several concurrent code blocks with the same type of parallelism. These code blocks t_{ijk}, $1 \leq k \leq B$, are suited for parallel execution on machines having the same type of parallelism. This decomposition of the task into subtasks, code segments, and code blocks is shown in Figure 5.1.

A good model of this input format is needed to facilitate the mapping of tasks onto a heterogeneous architecture. In addition to modeling the input format, the architecture being considered for the execution of the task should also be modeled. Several requirements for this model are identified as follows:

- The modeling of the input format should handle the decomposition of the task into subtasks, code segments, and code blocks, while preserving the information regarding the type of parallelism present in each portion of the task. This is essential to match the type of each code block with a suitable machine type in the system.
- The model should handle parallelism at fine grain and coarse grain levels.
- Modeling of the input code should emphasize the communication requirements of the various code segments.
- The modeling of the input code should be independent of the underlying architecture.
- The modeling of the system should provide the mode of computation of each machine in the system.

- The interconnection topology of individual architectures should be systematically represented in the model at both system and machine levels.

Cluster-M meets most of the above requirements. However, Cluster-M has no provision to model the heterogeneity present in the task and the system. In the next section, an extension to Cluster-M, called Hierarchical Cluster-M (HCM), is presented to incorporate the heterogeneous types both in tasks and systems. Then, a HOST based HCM mapping algorithm is presented in Section 5.4 which finds sub-optimal selection and mapping of each subtask. The HCM mapping algorithm is compared with other techniques in Section 5.5.

5.3 Hierarchical Cluster-M (HCM)

Hierarchical Cluster-M (HCM) is an extension to Cluster-M to exploit parallelism at the subtask, code segment, code block, and instruction levels. This is accomplished by modifying both the Cluster-M Specification and Representation. The extended Cluster-M Specification takes into account the type of parallelism present in each portion of the task. The modification to the Cluster-M Representation takes into account the presence of several interconnected machines in the system, providing a spectrum of computational modes.

5.3.1 HCM Specification

The HOST formulation can be applied to a non-uniform task graph $G_t = (V_t, E_t)$ as defined in Chapter 3. In a non-uniform task graph, each task module t_i is a code block. Task modules of the same type of computation requirements compose a code segment. A subtask consists of several sequential or concurrent code segments of different types. Thus, the Hierarchical Cluster-M Specification can be constructed in the same way as the original Cluster-M Specification. Besides, in HCM Specification, each Spec cluster is also labeled by a computation type. Only clusters of the same

type can be embedded and merged. For example, given a heterogeneous subtask as shown in Figure 5.2, the HCM Spec graph can be obtained by clustering the MIMD and vector type task modules (code blocks) respectively. Therefore, the obtained HCM Spec graph will consist of two subgraphs: one contains MIMD type clusters and the other contains vector type clusters. The MIMD type Spec subgraph is illustrated in Figure 5.3.

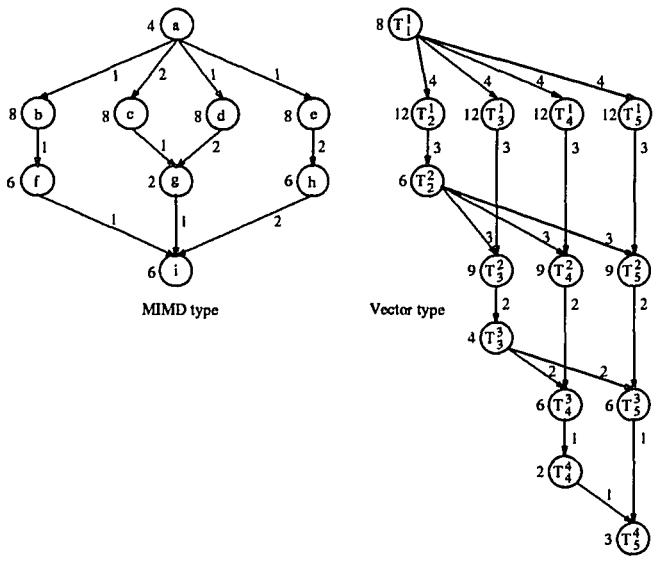


Figure 5.2 A heterogeneous subtask consists of MIMD and vector code segments.

5.3.2 HCM Representation

The Hierarchical Cluster-M Representation of a system consists of two layers of clustering: *system layer* and *machine layer*. System layer clustering consists of several levels of nested clusters. At the lowest level of clustering each machine in the system is assigned a cluster by itself. Completely connected clusters are merged to form the next level of clustering. This process is continued until no more merging is possible. Machine layer clustering is obtained in the same way as described in

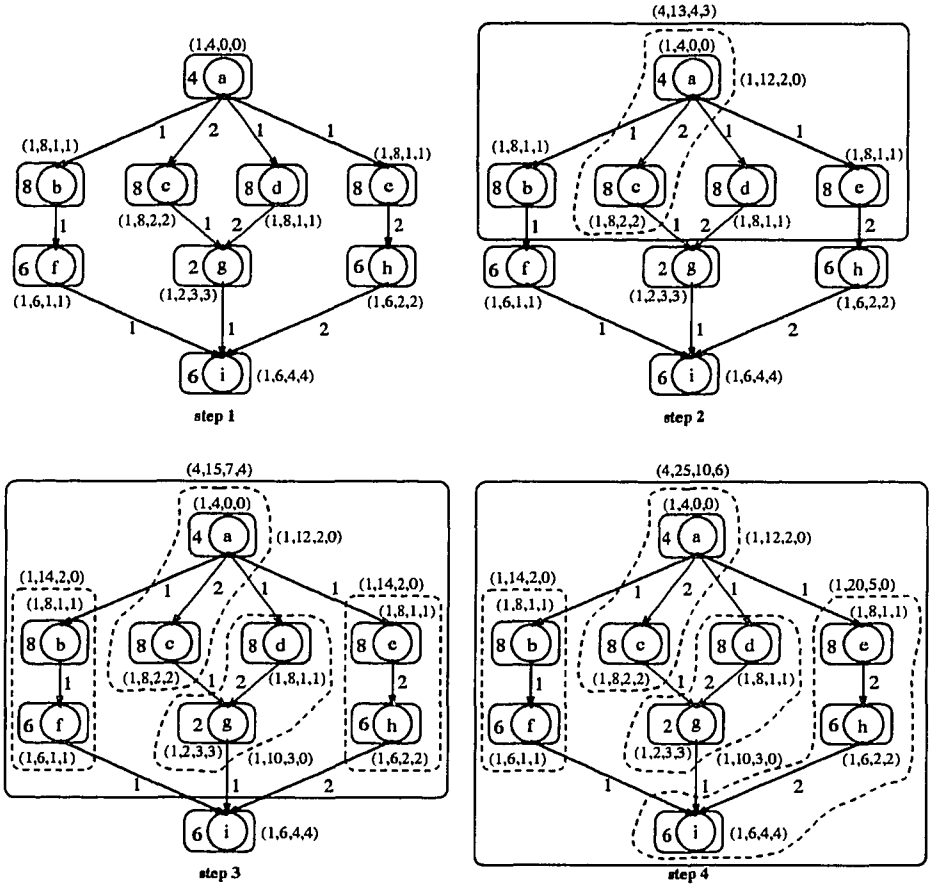


Figure 5.3 Construction of the Spec subgraph of the MIMD code segment.

Chapter 3. For a heterogeneous suite of interconnected computers, the HCM Representation is obtained as follows:

1. For each computer in the system, apply the Clustering-non-uniform-undirected-graphs algorithm as in Chapter 3 to obtain Cluster-M Representation of all the processors in the computer. The resulting clusters are called machine level clusters.

2. Each resulting cluster is labeled according to the type of parallelism present in the cluster (i.e. SIMD, MIMD, vector, etc.).

3. Treat each computer as a processor and apply the Clustering-non-uniform-undirected-graphs algorithm. At the first level of clustering, each computer in the system is in a cluster by itself. Each clustering level is constructed by merging clusters from the lower level that are completely connected. This is continued until no more clustering is possible. The resulting clusters are called system level clusters.

A heterogeneous parallel computing system is shown in Figure 5.4, which consists of one MIMD machine and one vector machine. The MIMD machine has three processors, P1, P2 and P3, and the vector machine has two processors P4 and P5. The clustering of the HCM Rep graph is also shown in this figure.

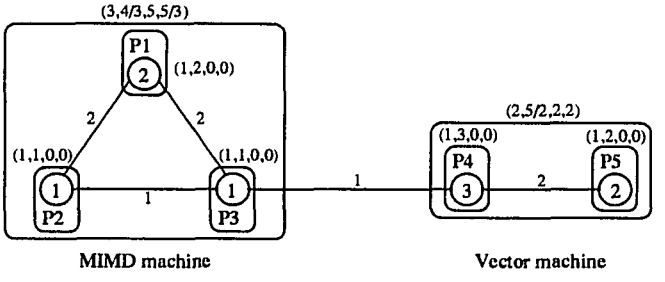


Figure 5.4 The system graph and its clustering of a heterogeneous suite.

5.4 HCM Bound-Degree Mapping Algorithm

Given the HCM Spec graph and HCM Rep graph, the mapping can be done for each type Spec subgraph and Rep subgraph respectively, using the original Cluster-M mapping algorithm as presented in Chapter 4. However, the original Cluster-M mapping, whether for uniform or non-uniform graphs, does not find optimal matching of Spec clusters with Rep clusters at each level. In this section, we present an HCM mapping algorithm for bound-degree task and system graphs. This mapping algorithm is a modified version of the original Cluster-M mapping algorithm. Instead of greedily matching Spec clusters to Rep clusters in the original mapping, the new algorithm finds an optimal matching using integer linear programming yet still maintains a polynomial time complexity.

Many parallel systems such as ring, binary tree, and mesh have constant degree. Many applications can also be expressed in bound-degree graphs such as in image processing, most divide-and-conquer applications, etc. If the degree of a graph is bound by a constant k, the number of sub-clusters within each Spec or Rep cluster at any clustering level will be at most k. Therefore, the function 4.2 in Chapter 4 can be used to find optimal matching of each Spec cluster without increasing the time complexity of mapping. In the following, a modified mapping algorithm is presented which uses an Integer Linear Programming (ILP) approach to find the optimal matching between Spec clusters and Rep clusters at each mapping step. *Mathematica 2.2 for SPARC*, a product by Wolfram Research, Inc., is used to solve this ILP problem.

Assume that the degrees of the given task graph and system graph are bound by two constants k_S and k_R, respectively. To formulate each mapping step into an ILP model, a binary variable $\mu(\kappa_{S_i}, \kappa_{R_j})$ is defined to indicate whether a Spec cluster κ_{S_i} is mapped onto a Rep cluster κ_{R_j}. $\mu(\kappa_{S_i}, \kappa_{R_j}) = 1$ if κ_{S_i} is mapped to κ_{R_j}. Otherwise, $\mu(\kappa_{S_i}, \kappa_{R_j}) = 0$. This transfers the mapping problem into an ILP

model in which each Spec cluster can be mapped to only one Rep cluster, which can be represented by $\sum_j \mu(\kappa_{S_i}, \kappa_{R_j}) = 1, 1 \leq i \leq k_S$. The estimated execution time on Rep cluster κ_{R_j} is denoted by $\Gamma(\kappa_{R_j})$, and $\Gamma(\kappa_{R_j}) = \sum_i \mu(\kappa_{S_i}, \kappa_{R_j})\tau(\kappa_{S_i}, \kappa_{R_j})$. Since the overall execution time is denoted by T_m, there are constraints that for all j, $T_m \geq \Gamma(\kappa_{R_j})$. The objective is to minimize the overall estimated execution time. Therefore, the objective function of our ILP model can be expressed as follows.

$$Minimize\ T_m, while\ T_m \geq \Gamma(\kappa_{R_j})\ for\ all\ j$$

Once the minimal T_m is found, the matching of Spec clusters and Rep clusters can be determined by using binary variables $\mu(\kappa_{S_i}, \kappa_{R_j})$.

HCM Bound-Degree Mapping Algorithm
begin
  for each machine type
    calculate reduction factor $f = \frac{\sum_j \sigma_{R_j}}{\sum_i \sigma_{S_i}}$
    for each Spec cluster $S_i, 1 \leq i < k_S$
    begin
      for each Rep cluster $R_j, 1 \leq j < k_R$
        $\tau(S_i, R_j) = \delta_{S_i}(\frac{1}{\delta_{R_j}} \times \frac{1}{f} + \frac{\Pi_{S_i} \times \delta_{R_j}}{\Pi_{R_j} \times \delta_{S_i}})$
    end
    /* *Starting Integer Linear Programming* */
    /* *Set Constraints* */
    $\sum_j \mu(S_i, R_j) = 1, 1 \leq i < k_S$
    $\Gamma(R_j) = \sum_i \mu(S_i, R_j)\tau(S_i, R_j), 1 \leq j < k_R$
    $T_m \geq \Gamma(R_j), 1 \leq j < k_R$
    /* *Objective Function* */
    Minimize T_m
  end
end

Figure 5.5 HCM bound-degree mapping algorithm.

A detailed description of the HCM bound-degree mapping algorithm is presented in Figure 5.5. The time complexity of this algorithm can be analyzed as follows. The numbers of iteration for the outer **for** loop and the inner **for** loop are at most k_S and k_R, respectively. Therefore, the total number of iteration for these **for** loops is bound by $O(k_S \times k_R)$. Consider the portion of ILP in this algorithm, it examines all instances of $(\kappa_{S_i}, \kappa_{R_j})$ pairs for all i and j. The running time of this portion, hence, is equal to $O((k_R)^{k_S})$. The overall time complexity of the mapping algorithm is therefore $O(k_S \times k_R) + O((k_R)^{k_S}) = O((k_R)^{k_S})$. However, since both k_S and k_R are constants, it is still a polynomial time complexity.

Consider mapping the task graph illustrated in Figure 5.2 to the system graph of Figure 5.4. The mapping is done for each type of Spec and Rep clusters respectively. The mapping of the MIMD Spec subgraph onto the MIMD Rep subgraph is done as below. At the top level, the mapping is trivial since there is only one Spec cluster $\kappa_{S_0}(4, 25, 10, 6)$ and one Rep cluster $\kappa_{R_0}(3, \frac{4}{3}, 5, \frac{5}{3})$. At the next level, four Spec clusters $\kappa_{S_1}(1, 12, 2, 0)$, $\kappa_{S_2}(1, 14, 2, 0)$, $\kappa_{S_3}(1, 10, 3, 0)$, and $\kappa_{S_4}(1, 20, 5, 0)$ are to be mapped to three Rep clusters $\kappa_{R_1}(1, 2, 0, 0)$, $\kappa_{R_2}(1, 1, 0, 0)$, and $\kappa_{R_3}(1, 1, 0, 0)$. Using the modified mapping algorithm, κ_{S_3} and κ_{S_4} are mapped onto κ_{R_1}, and κ_{S_2} and κ_{S_1} are mapped to κ_{R_2} and κ_{R3} respectively. It 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 vector Spec subgraph onto the vector Rep subgraph can be done in a similar way. The overall mapping result is shown in Figure 5.6.

5.5 Comparison Study

In this section, the HCM bound-degree mapping algorithm is compared with the original Cluster-M non-uniform mapping algorithm as well as other two techniques which are capable of mapping tasks onto distributed heterogeneous systems. Lo's

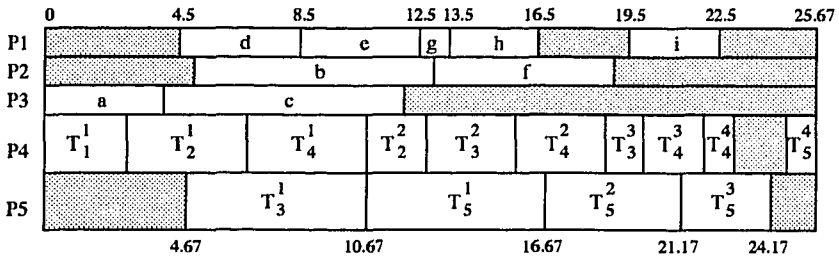


Figure 5.6 The obtained mapping result.

mapping algorithm in [59] is a heuristic which combines recursive invocation of max flow min cut algorithm to find suboptimal assignments of tasks to heterogeneous processors. In [71], Shen and Tsai considered a cost function and a minmax criterion for minimization of the cost function, then solve the mapping problem by the well-known A* algorithm. Since both algorithms do not incorporate the heterogeneous computation and machine types in their mapping, it is only possible to compare the result of mapping each type of task modules (code blocks) onto the same type of processors respectively.

Considering the example discussed in the previous section for mapping the task graph of Figure 5.2 to the system graph of Figure 5.4. The mapping results of MIMD type task modules onto MIMD type processors by HCM bound-degree mapping, the original Cluster-M non-uniform mapping, Lo's heuristic, and Shen and Tsai's A* heuristic are shown in Figure 5.7. Their total execution times are 22.5, 24.5, 24, and 28 respectively. HCM bound-degree mapping produces the best result.

The mapping results of the vector type task modules onto the vector type processors are shown in Figure 5.8. The T_m by the four different mapping algorithms are 25.67, 30.17, 38, and 33.83, respectively. Again the HCM bound-degree algorithm produces the best mapping, yet the mapping of the original Cluster-M algorithm is also very good.

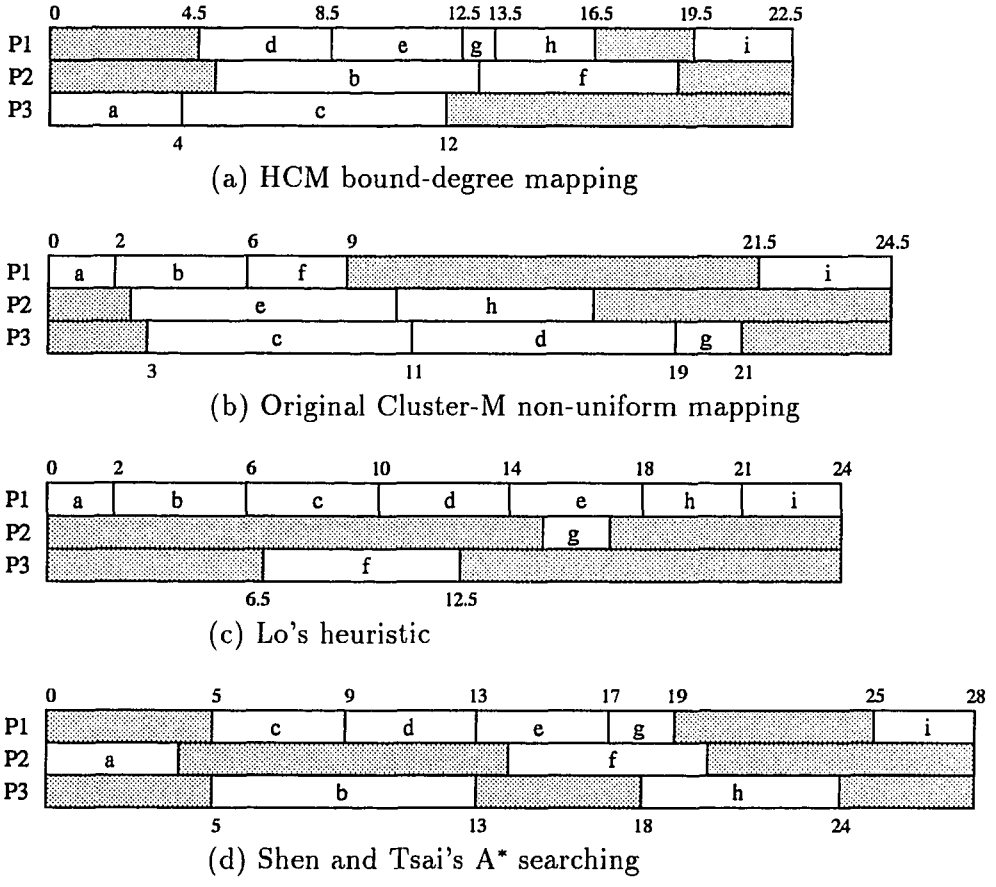
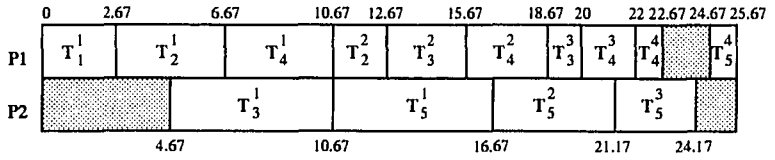
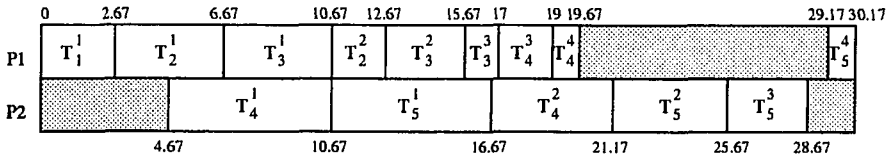


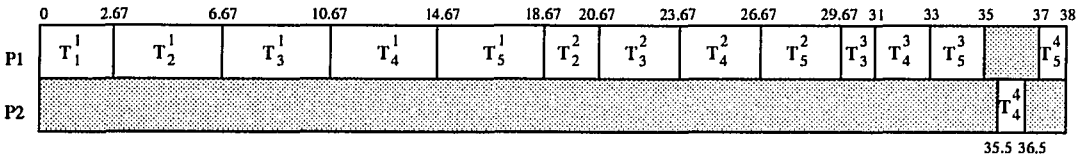
Figure 5.7 The mapping results of MIMD code blocks onto MIMD machine.



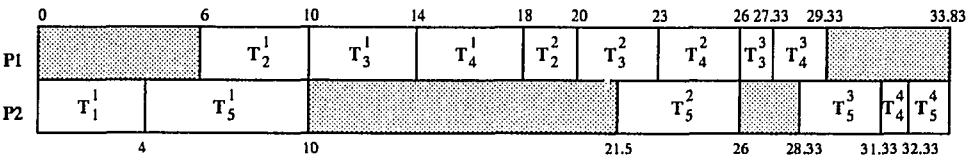
(a) HCM bound-degree mapping



(b) Original Cluster-M non-uniform mapping



(c) Lo's heuristic



(d) Shen and Tsai's A* searching

Figure 5.8 The mapping results of Gaussian elimination on the vector machine.

CHAPTER 6

COMBINED USE OF CLUSTER-M WITH HASC

In this chapter, we study how Cluster-M can be used together with a different programming paradigm, called Heterogeneous Associative Computing (HAsC), to provide an efficient scheme for heterogeneous programming. Unlike other existing heterogeneous orchestration tools which are MIMD based, HAsC is for data-parallel SIMD associative computing. HAsC models a heterogeneous network as a coarse-grained associative computer. It is designed to optimize the execution of tasks where the program size is small compared with the amount of data processed. Ease of programming and execution speed are the primary goals of HAsC. On the other hand, Cluster-M can be applied to both coarse-grained and fine-grained networks. Cluster-M provides an environment for porting heterogeneous tasks onto the machines to maximize the resource utilization and to minimize the execution time. Both Cluster-M and HAsC can efficiently support heterogeneous networks by preserving a level of abstraction without containing any architecture details. They are both machine-independent and scalable for various network and task sizes.

6.1 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 (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 a case where special properties are present. For example, one SIMD node

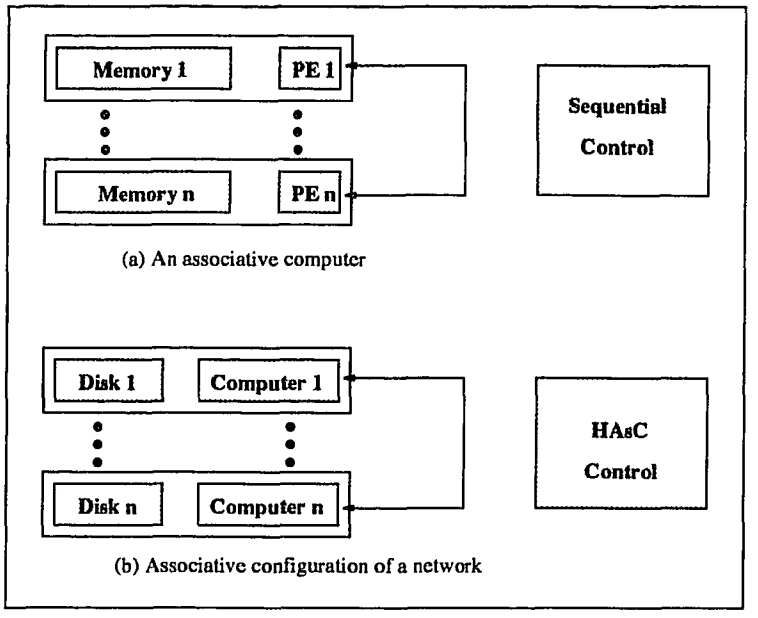


Figure 6.1 Analogy between an associative computer and an associative configuration of a network.

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

Figure 6.1 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. 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, just as 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.

HAsC is "layered" so that any node in the HAsC network may 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,

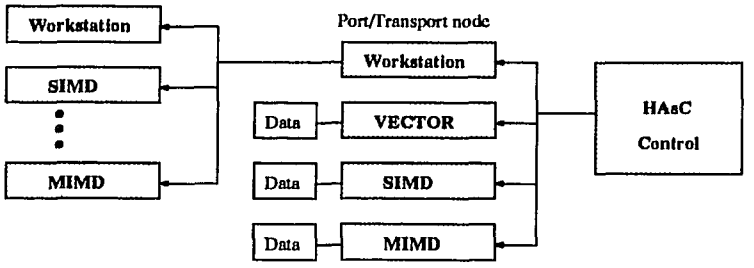


Figure 6.2 A layered heterogeneous network.

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. Figure 6.2 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 command(s) appropriate for the subnetwork.

Some of the properties of the associative computing paradigm well suited for heterogeneous computing include: (1) efficient programming and execution with large data sets and small programs; (2) optimal data placement; (3) software scalability (see Section 6.3); (4) cellular memory allocation; and (5) search-process-retrieve synchronism [66].

In HAsC, instructions are broadcast to all of the cells listening to a channel, but each individual cell must determine whether 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. Several languages such as Prolog and STRAND [36] incorporate this 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, 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 in a cell when the HAsC command arrives. Instruction unification and execution proceed as described above. HAsC allows data transfers, but protocol insists that the data transfer be completed 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 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.

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 HAsC administrator receives all incoming HAsC instructions from the local network. It then verifies if each instruction is legal. 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 program *id* which is broadcast with every HAsC instruction.

Instructions can be added at several different logical levels: (1) system, (2) project, and (3) user. Typical system level instructions would be data move and formatting commands. Project commands would be project oriented. For example, a numerical analysis project would have matrix multiplication and vector-matrix multiplication 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, and 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 produce the proper interface for the appropriate language.

HAsC must allow for a dynamic instruction set and data structure modifications. Thus the HAsC *install* Meta instruction consists 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, are sent to the appropriate node and are 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.

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. A hierarchy of instructions is briefly described here - 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.

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: (1) those which execute based on the channel register value only, such as logical and arithmetic expressions; and (2) 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 6.3 gives an example of instruction synchronization, where $ is the parallel marker. Result$ is a data parallel pronoun referring to the results of the last performed data parallel computation. The top level synchronization box shows the programming style for algebraic expressions supported by HAsC.

add the b$ to the c$ subtract the result$ from the d$ convolve the result$ with the e$ save the result$ in the f$ compare the a$ with the b$ where the result$ are equal do ... elsewhere do ...	Top level synchronization Expressions and WHERE commands
move the a$ to the b$ save the a$ in the b$ read c$	Second level synchronization Data move and I/O commands
any a$ greater than 5	Third level synchronization ANY command
pick one of the responder$ any a$ greater than the b$	Fourth level synchronization Item selection
read matrix a$ exit if EOF convolve a$ with image$ display result$ repeat sum the salary$	Fifth level synchronization Iteration

Figure 6.3 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 and 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 consist of simple responder commands. These commands require the ORing of the responder results of all processors (i.e. an OR reduction). On a SIMD machine 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 6.3 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. Data iteration does not exist in Figure 6.3.

6.2 Combined Use of Cluster-M and HAsC

HAsC is most suitable for coarse-grained heterogeneous parallel computing. It is intended to ease the programming effort and to maximize execution speed. 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, increased execution speed and optimal resource utilization.

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 within HAsC nodes as shown in Figure 6.4. 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 6.5). As described in Section 6.1, 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 described in [66].

The following illustrates the combined use of Cluster-M and HAsC by an example. Given two 7×7 real matrices A and B, suppose we want to calculate $U_A \times U_B^T$, where $L_A \times U_A = A$ and $L_B \times U_B = B$. The matrices L_A (or L_B) and U_A (or U_B) have the same dimensions as A; L_A (or L_B) is unit lower triangular (i.e., zeros above the diagonal and the value one on the diagonal), and U_A (or U_B) is upper triangular (i.e., zero below the diagonal). To transform the original square matrix A (or B) into the product of the two matrices L_A (or L_B) and U_A (or U_B),

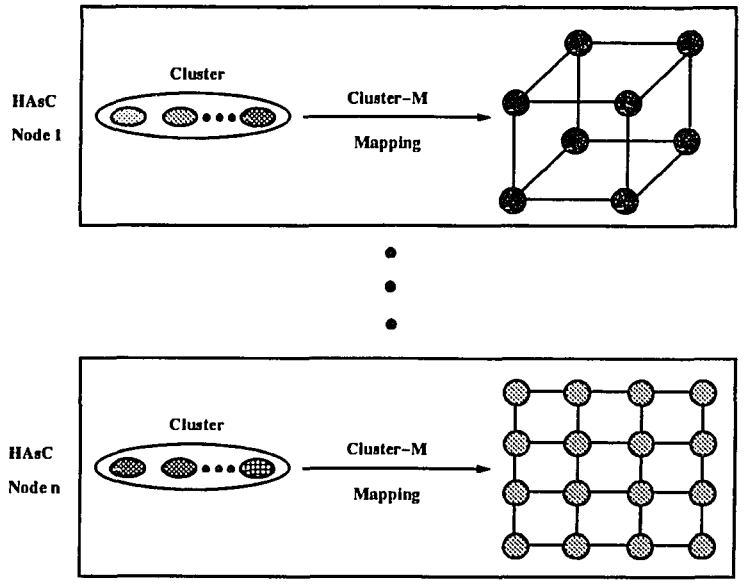


Figure 6.4 Cluster-M aided HAsC computation within HAsC nodes.

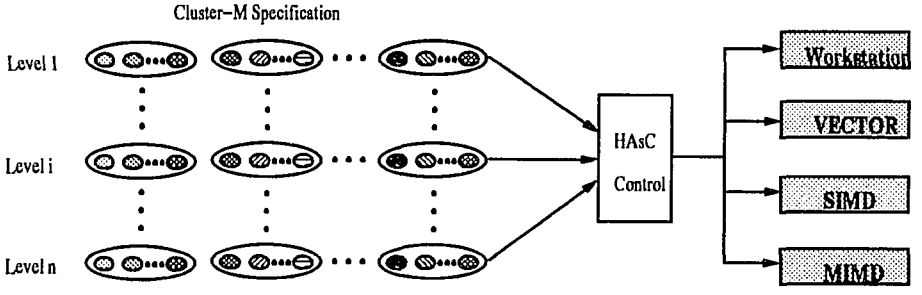
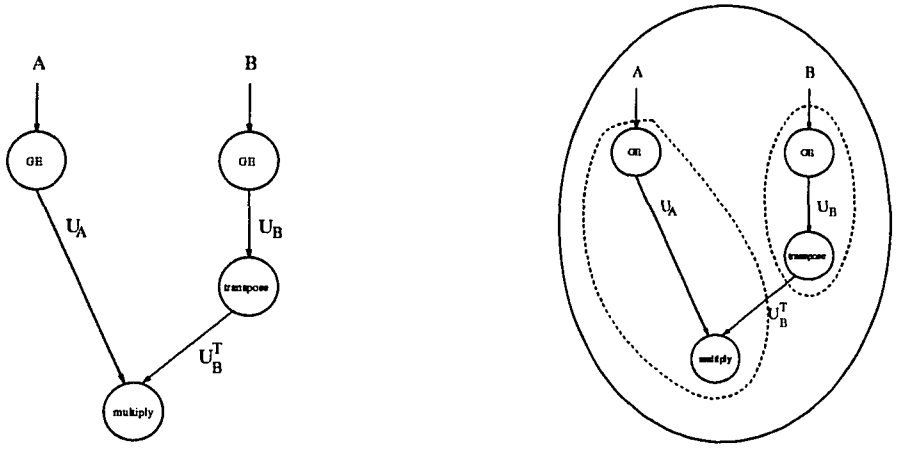


Figure 6.5 Switching between Cluster-M and HAsC.

a Gaussian Elimination (GE) algorithm can be used. Therefore, the solution to the above problem can be written at HAsC user level as below:

```
do GE on A$
save result$ in UA$
do GE on B$
transpose result$
save result$ in UBT$
multiply UA$ with UBT$
```

The task graph of this coarse-grain solution is shown in Figure 6.6(a). Using one of Cluster-M's clustering algorithms, a Spec graph can be obtained as shown in



(a) Task graph (b) Spec graph

Figure 6.6 The task graph and Spec graph of the HAsC user level instructions.

Figure 6.6(b). Suppose there is more than one HAsC node available in the system. Using Cluster-M mapping, the matrices A and B will be allocated to two different HAsC nodes, say $Node1$ and $Node2$, respectively.

Next, for each level of clustering in the Spec graph (which represents each computation step in the original task graph), the concurrent clusters at that level (which represnt concurrent computation modules) can be sent to the HAsC network controller to be broadcast to all the HAsC nodes. For example, at step 1, two clusters of HAsC user level instructions (function calls) "`do GE on` A\$" and "`do GE on` B\$" are broadcast to all HAsC nodes at the same time. The HAsC node $Node1$ will select to execute the first instruction, while the HAsC node $Node2$ will select to execute the second instruction.

Finally, Cluster-M mapping is used to decide the fine-grain mapping within each HAsC node. The GE operation, which is a function in the user level library, actually consists of many system level instructions which may look similar to the SAXPY code in LINPACK [23, 24]. The task graph of a GE on a 7×7 matrix A or B is illustrated in Figure 6.7. In each task module T_j^k, column j is modified by using column k. Suppose $Node1$ is a 2×3 torus, and $Node2$ is a 4-processor completely

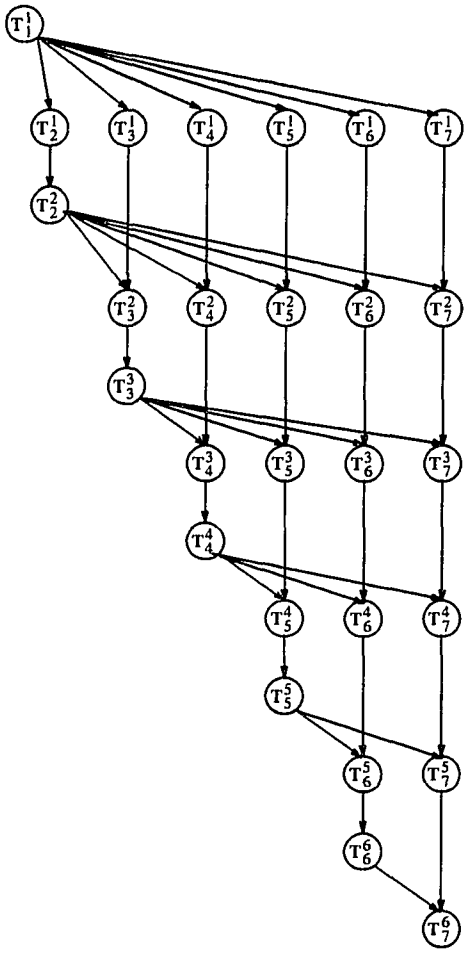


Figure 6.7 The task graph of a GE on a 7×7 matrix.

connected machine, as shown in Figure 6.8. Also, suppose for both $Node1$ and $Node2$, it takes 1 unit of time to compute each T_j^k and 1 unit of time to transmit each column between two connected processors. Using the Cluster-M clustering and mapping algorithms, the fine-grain mappings of system level HAsC instructions onto the processors within each HAsC node can be obtained, as shown in Figure 6.9.

6.3 Scalability Issues

Scalability is often understood differently by different authors. We will consider scalability to refer to hardware, tasks and software in roughly analogous fashion. In

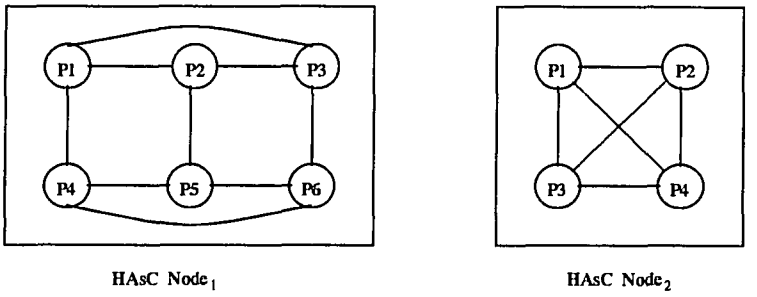


Figure 6.8 The architectures of HAsC $Node1$ and $Node2$.

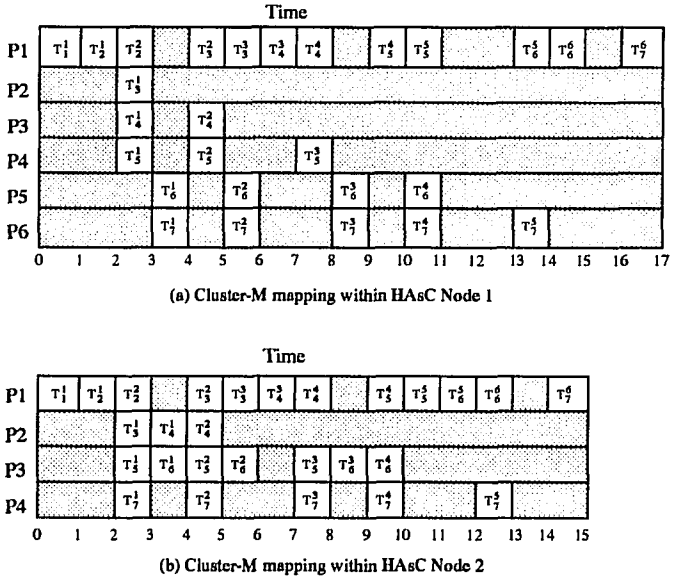


Figure 6.9 The Cluster-M mappings within the HAsC nodes.

addition, scalability may refer to both homogeneous or heterogeneous architectures. In the following, first homogeneous scalability is defined and extended to heterogeneous scalability. Then the scalability of HAsC and Cluster-M is discussed.

6.3.1 Homogeneous Scalability

Homogeneous hardware scalability refers to multiple machines which are of the same basic architectural type, typically various-sized versions of the same vendor product. The *hardware scalability* function, $\chi(\alpha, \beta)$, between two homogeneous architectures α (the larger) and β (the smaller), is defined to be the ratio of the size of α over the size of β. For example, an eight processor CRAY YMP is a hardware example of a scaled-up version of a two-processor CRAY YMP. In this example, the eight-processor CRAY YMP has a scalability factor of 4 ($\chi = 4$) over the two-processor.

Task scalability is more complex. What is 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. Thus, using the additional resources of the larger machine allows scaled-up performance reasonably close to χ. One ambiguity in this concept is what is meant 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. The *type 1 task scalability* function, $T(\alpha, \beta)$ for a given program applied to two different sized data set α (the larger) and β (the smaller), is defined to be the ratio of the size of α over the size of β. For example, if the size of α is 16K items and the size of β is 2K items, then $T = 8$. This means that a program is *type 1 scalable* if it processes data set β eight times faster than data set α, using the same hardware configuration.

However, if applying the above definitions to the case where both the data and the algorithm are fixed, then tasks often do not scale. *Type 2 task scalability*, between two homogeneous architectures α (the larger) and β (the smaller), is defined to be

the potential to exploit the inherent hardware scalability between them on some task of a size that fills α.

The *software scalability* refers to the ability to exploit task and hardware scalability, with little or no changes other than parameters. *Software scalability* function, $\sigma(\alpha, \beta)$, for the case of two homogeneous architectures α (the larger) and β (the smaller), is defined to be the real-valued function giving the increase in performance of α over β. Typically some increase in performance is expected but generally, at least in the homogeneous case, not super-linear performance, i.e., $1 \leq \sigma(\alpha, \beta) \leq \chi(\alpha, \beta)$. In most cases σ is a simple multiple of χ, i.e., $\sigma(\alpha, \beta) = \lambda \times \chi(\alpha, \beta)$, where $1/\chi(\alpha, \beta) \leq \lambda \leq 1.0$.

6.3.2 Heterogeneous Scalability

Heterogeneous scalability is clearly more complicated than *homogeneous scalability*, though it is also the case in which one can aspire to the ultimate in heterogeneous computing potential, i.e, to achieve σ significantly greater than χ. This is what is meant by super-linear performance. In the heterogeneous case, there may be no commonality between two different architectures, therefore, *hardware scalability* does not apply to the heterogeneous case.

Consider the breakdown of a task into four levels, as shown in Figure 6.10. The top level is the functional level. In this level, the function "find a datum" is specified. Next is the approach level. For this problem, there is a radical difference between the approach for a SIMD machine used associatively and non-SIMD machines. In the former case, we can use simple associative search, which is $O(1)$. In the latter case we would typically use a sort, then search operation, the asymptotic performance of which 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

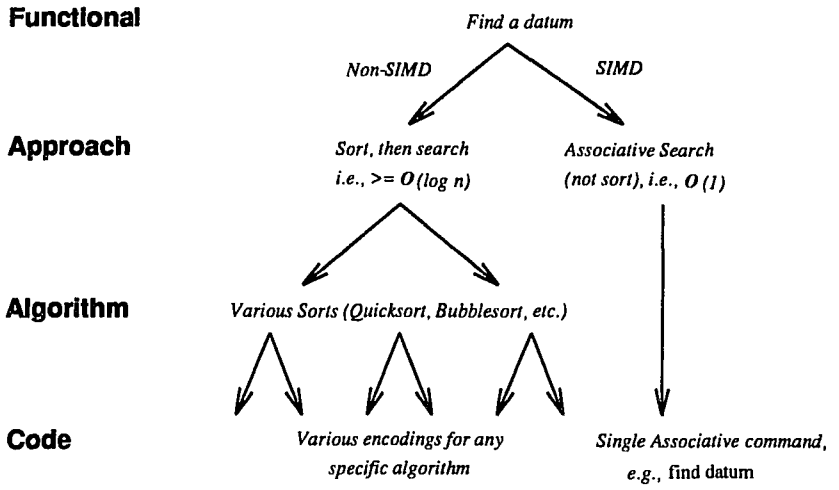


Figure 6.10 Hierarchical breakdown of a task

are many variations possible. For example, depending on the data, parameters, architecture, etc., a number of different search techniques can be used and similarly a number of different coding schemes for each algorithm could also be utilized.

In this context, the term *scalability* only applies to either functional level or approach level. In the above example, the scalable approach is the non-SIMD approach. However, this will bring the following dilemmas: (1) it is possible to have a non-scalable implementation (at the approach level) inherently more effective than a scalable approach implemented on the same machine; and (2) it is possible to have high hardware scalability but low task/software scalability, or vice versa. In other words, the scalable metric is inherently defective in this case if scalability is applied to the approach level.

In conclusion, 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 establishing a common unifying programming environment for heterogeneous networks.

6.3.3 Scalability of HAsC and Cluster-M

Both HAsC and Cluster-M are machine-independent as explained in detail and therefore support heterogeneous scalability. 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 connected 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 nodes best suited to process them. Broadcasting allows scalability. The hardware 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 new suite can be generated. However, the Cluster-M Specification, which is machine-independent, can be efficiently executed without any change. An appropriate new mapping 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. Figure 6.11 illustrates the above claims.

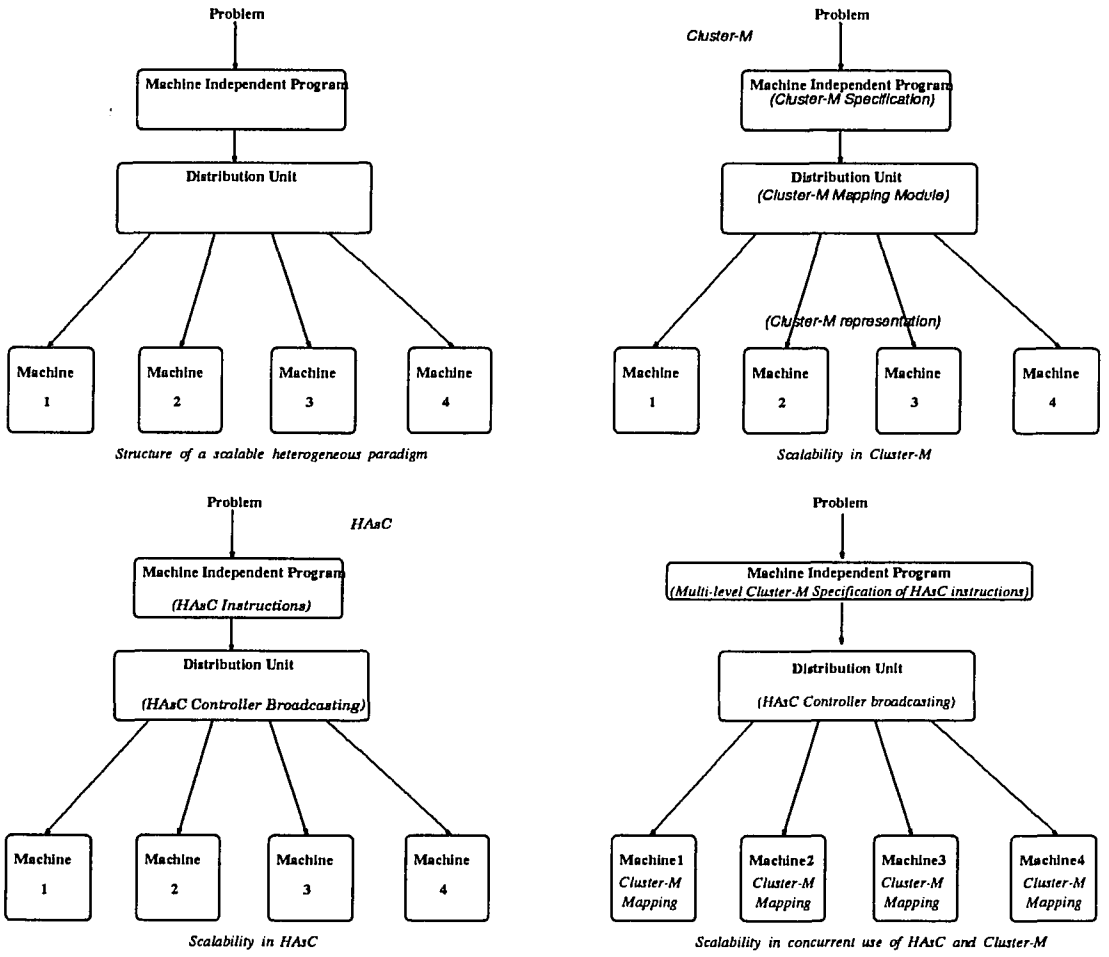


Figure 6.11 Scalability of HAsC and Cluster-M

CHAPTER 7

CONCLUSIONS

In this dissertation, we presented a generic and efficient technique for mapping portable parallel programs onto various multiprocessor systems. The presented mapping technique is based on Cluster-M, a parallel programming tool. We presented the three main components of Cluster-M: Cluster-M Specifications, Cluster-M Representations and Cluster-M Mapping Module. The Cluster-M Specifications are high level machine-independent descriptions of parallel tasks, while the Cluster-M Representations represent the computation/communication capacity and pattern of the underlying parallel computer systems. Both Cluster-M Specifications and Representations can be viewed as two special types of clustered graphs, called Spec graphs and Rep graphs, respectively. The clustering is done only once for a given task graph (system graph) independent of any system graphs (task graphs). This is a machine-independent (application-independent) clustering and is not repeated for different mappings. The Cluster-M Mapping Module maps a Spec graph onto a Rep graph. The mapping algorithms presented in this dissertation can map arbitrary tasks onto arbitrary systems, for both uniform and non-uniform graphs, in $O(MN)$ and $O(MP)$ time, respectively, where M is the number of task modules, N is the number of processors and $P = \max(M, N)$. Our experimental results indicate that Cluster-M produces better or similar mapping results compared to other leading techniques which work only for restricted task or system graphs. Lastly, several applications of the presented mapping technique to the area of heterogeneous computing were presented.

APPENDIX A

Cluster-M Constructs in PCN

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),
```

$op == \text{“}sqr\text{”} -> e = e1 * e1,$

$op == \text{“} - \text{”} -> e = 0 - e1$

}

}

/* 6. Does the Binary operation */

$CBI(op, x, i, y, j, e)$

{; $CELEMENT(x, i, e1)$,

$CELEMENT(y, j, e2)$,

{ ? $op == \text{“} + \text{”} -> e = e1 + e2,$

$op == \text{“} - \text{”} -> e = e1 - e2,$

$op == \text{“} * \text{”} -> e = e1 * e2,$

$op == \text{“}/\text{”} -> e = e1/e2,$

$op == \text{“}\%\text{”} -> e = e1\%e2,$

$op == \text{“}\&\text{”} -> bitwise\_and(e, e1, e2),$

$op == \text{“}|\text{”} -> bitwise\_or(e, e1, e2),$

$op == \text{“}\#\text{”} -> bitwise\_xor(e, e1, e2)$

}

}

/* 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
    }
}
```

REFERENCES

1. *C-Linda Reference Manual.* Scientific Computing Associates, Inc., New Haven, CT, 1990.

2. H. H. Ali and H. El-Rewini. "A graph theoretic approach for task allocation." In *Proc. Hawaii International Conference on Systems Science*, pages 577–584, 1992.

3. F. D. Anger, J. Hwang, and Y. Chow. "Scheduling with sufficient loosely coupled processors." *Journal of Parallel and Distributed Computing*, 9:87–92, 1990.

4. F. Berman. "Experience with an automatic solution to the mapping problem." *The Characteristics of Parallel Algorithms*, pages 307–334, 1987.

5. F. Berman and L. Snyder. "On mapping parallel algorithms into parallel architectures." *Journal of Parallel and Distributed Computing*, 4:439–458, 1987.

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

7. S. H. Bokhari. "Dual processor scheduling with dynamic reassignment." *IEEE Trans. on Software Engineering*, SE-5:341–349, July 1979.

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

9. S. H. Bokhari. "A shortest tree algorithm for optimal assignments across space and time in a distributed processor system." *IEEE Trans. on Software Engineering*, SE-7(6):583–589, November 1981.

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

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

12. T. L. Casavant and J. G. Kuhl. "A taxonomy of scheduling in general-purpose distributed computing systems." *IEEE Trans. on Software Engineering*, 14(2):42–45, February 1988.

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

14. V. Chaudhary and J. K. Aggarwal. "A generalized scheme for mapping parallel algorithms." *IEEE Trans. on Parallel and Distributed Systems*, 4(3):328–346, March 1993.

15. S. Chen and M. M. Eshaghian. "A fast recursive mapping algorithm." To appear at *Concurrency: Practice and Experience*, August 1995.

16. S. Chen, M. M. Eshaghian, A. Khokhar, and M. E. Shaaban. "A selection theory and methodology for heterogeneous supercomputing." In *Proc. Second Heterogeneous Processing Workshop*, 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." To appear at *Proc. International Conference on Parallel Processing*, August 1995.

18. D. Y. Cheng. "A survey of parallel programming languages and tools." Report RND-93-005, NASA Ames Research Center, Moffett Field, CA, 1993.

19. Y. Chung and S. Ranka. "Applications and performance analysis of a compile-time optimization approach for list scheduling algorithms on distributed memory multiprocessors." In *Proc. Supercomputing '92*, pages 512–521, 1992.

20. E. G. Coffman and R. L. Graham. "Optimal scheduling for two processor systems." *Acta Informatica*, 1:200–213, 1972.

21. J. Y. Colin and P. Chritienne. "CPM scheduling with small communication delays and task duplication." *Operations Research*, 39(4):680–684, 1991.

22. S. Darbha and D. P. Agrawal. "SDBS: A task duplication based optimal scheduling algorithm." In *Proc. Scalable High Performance Computing Conference*, pages 756–763, 1994.

23. J. J. Dongarra, J. Bunch, C. Moler, and G. Stewart. *LINPACK User's Guide.* SIAM, Philadelphia, PA, 1979.

24. J. J. Dongarra, F. Gustavson, and A. Karp. "Implementing linear algebra algorithms for dense matrices on a vector pipeline machine." *SIAM Review*, 26(1):91–112, 1984.

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

26. H. El-Rewini and T. G. Lewis. "Scheduling parallel program tasks onto arbitrary target machines." *Journal of Parallel and Distributed Computing*, 9:138–153, 1990.

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

28. F. Ercal, J. Ramanujam, and P. Sadayappan. "Task allocation onto a hypercube by recursive mincut bipartitioning." *Journal of Parallel and Distributed Computing*, 10:35–44, 1990.

29. M. M. Eshaghian. "Cluster-M parallel programming model." In *Proc. International Parallel Processing Symposium*, pages 462–465, March 1992.

30. M. M. Eshaghian and R. F. Freund. "Cluster-M paradigms for high-order heterogeneous procedural specification computing." In *Proc. Workshop on Heterogeneous Processing*, pages 47–49, March 1992.

31. M. M. Eshaghian and M. E. Shaaban. "A Cluster-M based mapping methodology." In *Proc. International Parallel Processing Symposium*, pages 213–221, April 1993.

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

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

34. S. A. Fineberg, T. L. Casavant, and H. J. Siegel. "Experimental analysis of a mixed-mode parallel architecture using bitonic sequence sorting." *Journal of Parallel and Distributed Computing*, 11(3):239–251, March 1991.

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

36. I. Foster and T. Stephen. *STRAND, New Concepts in Parallel Programming.* Prentice Hall, 1975.

37. R. F. Freund. "Optimal selection theory for superconcurrency." In *Proc. Supercomputing '89*, pages 699–703, November 1989.

38. R. F. Freund and D.S. Conwell. "Superconcurrency: A form of distributed heterogeneous supercomputing." *Supercomputing Review*, 3:47–50, October 1990.

39. A. Gerasoulis, S. Venugopal, and T. Yang. "Clustering task graphs for message passing architectures." In *Proc. ACM International Conference of Supercomputing*, June 1990.

40. A. Gerasoulis and T. Yang. "A comparison of clustering heuristics for scheduling directed acyclic graphs on multiprocessors." *Journal of Parallel and Distributed Computing*, 16:276–291, 1992.

41. D. H. Gill, T. J. Smith, T. E. Gerasch, J. V. Warren, C. L. McCreary, and R. E. K. Stirewalt. "Spatial-temporal analysis of program dependence graphs for useful parallelism." *Journal of Parallel and Distributed Computing*, 19:103–118, October 1993.

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

43. J. Hwang, Y. Chow, F. D. Anger, and C. Lee. "Scheduling precedence graphs in systems with interprocessor communication times." *SIAM Journal on Computing*, 18:244-257, 1989.

44. B. Indurkya, H. S. Stone, and X. Lu. "Optimal partitioning of randomly generated distributed programs." *IEEE Trans. on Software Engineering*, SE-12(3):483–495, March 1986.

45. L. R. Ford Jr. and D. R. Fulkerson. *Flows in Networks.* Princeton University Press, Princeton, NJ, 1962.

46. S. Kambhatla, J. Inouye, and J. Walpole. "Experiences with BeLinda: A synthetic Linda benchmark for parallel computing platforms." In *Proc. International Conference on Parallel Processing*, 1990.

47. R. M. Karp. "Reducibility among combinatorial problems." *Complexity of Computer Computations*, 1972.

48. H. Kasahara and S. Narita. "Practical multiprocessor scheduling algorithms for efficient parallel processing." *IEEE Trans. on Computers*, c-33(11):1023–1029, November 1984.

49. B. W. Kernighan and S. Lin. "An efficient heuristic procedure for partitioning graphs." *Bell System Technical Journal*, February 1970.

50. A. A. Khan, C. L. McCreary, and M. S. Jones. "A comparison of multiprocessor scheduling heuristics." In *Proc. International Conference on Parallel Processing*, pages II 243–250, 1994.

51. A. Khokhar, V. K. Prasanna, M. E. Shaaban, and C. Wang. "Heterogeneous computing: Challenges and opportunities." *IEEE Computer*, 26(6):18–27, June 1993.

52. S. J. Kim and J. C. Browne. "A general approach to mapping of parallel computation upon multiprocessor architectures." In *Proc. International Conference on Parallel Processing*, volume 3, pages 1–8, 1988.

53. B. Kruatrachue and T. Lewis. "Grain size determination for parallel processing." *IEEE Trans. on Software Engineering*, January 1988.

54. 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.

55. 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.

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

57. R. Leland and B. Hendrickson. "An empirical study of static load balancing algorithms." In *Proc. Scalable High-Performance Computing Conference*, pages 682–685, 1994.

58. V. M. Lo. "Algorithms for static task assignment and symmetric contraction in distributed computing systems." In *Proc. International Conference on Parallel Processing*, pages 239–244, August 1988.

59. V. M. Lo. "Heuristic algorithms for task assignment in distributed systems." *IEEE Trans. on Computers*, 37(11):1384–1397, November 1988.

60. 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.

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

62. M. A. Palis, J. Liu, and D. S. L. Wei. "Task clustering and scheduling for distributed memory parallel architectures." Technical report, Department of Electrical and Computer Engineering, New Jersey Institute of Technology, 1995.

63. C. H. Papadimitriou and M. Yannakakis. "Towards an architecture-independent analysis of parallel algorithms." *SIAM Journal on Computing*, 19(2):322–328, April 1990.

64. F. Pellegrini. "Static mapping by dual recursive bipartitioning of process and architecture graphs." In *Proc. Scalable High-Performance Computing Conference*, pages 486–493, 1994.

65. R. Ponnusamy, N. Mansour, A. Choudhary, and G. C. Fox. "Mapping realistic data sets on parallel computers." In *Proc. 7th International Parallel Processing Symposium*, pages 123–128, April 1993.

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

67. 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.

68. P. Sadayappan and F. Ercal. "Nearest-neighbor mapping of finite element graphs onto processor meshes." *IEEE Trans. on Computers*, C-36(12):1408–1424, December 1987.

69. P. Sadayappan, F. Ercal, and J. Ramanujam. "Cluster partitioning approaches to mapping parallel programs onto a hypercube." *Parallel Computing*, 13:1–16, 1990.

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

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

72. G. C. Sih and E. A. Lee. "A compile-time scheduling heuristic for interconnection-constrained heterogeneous processor architectures." *IEEE Trans. on Parallel and Distributed Systems*, 4(2):75–87, February 1993.

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

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

75. H. S. Stone and S. H. Bokhari. "Control of distributed processes." *IEEE Computer*, July 1978.

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

77. D. Towsley. "Allocating programs containing branches and loops within a multiple processor system." *IEEE Trans. on Software Engineering*, SE-12(10):1018–1024, 1986.

78. J. D. Ullman. "NP-complete scheduling problems." *Journal of Computer Systems and Science*, June 1975.

79. K. Vairavan and R. DeMillo. "On the computational complexity of a generalized scheduling problem." *IEEE Trans. on Computers*, c-25(11):1067–1073, November 1976.

80. 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.

81. L. R. Welch, S. Chen, A. D. Stoyenko, and A. K. Ganesh. "Applying random neural networks to exploit parallelism and conserve processors in ADT module assignments." Technical report, Department of Computer and Information Science, New Jersey Institute of Technology, 1993.

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

83. J. Yang, L. Bic, and A. Nicolan. "A mapping strategy for MIMD computers." In *Proc. International Conference on Parallel Processing*, 1991.

84. T. Yang and A. Gerasoulis. "List scheduling with or without communication delays." Technical report, Department of Computer Science, Rutgers University, 1992.

85. 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.

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