Order Number 9311514

The assignment problem in distributed computing

Medepalli, Anand, Ph.D.

Iowa State University, 1992

U·M·I
300 N. Zeeb Rd.
Ann Arbor, MI 48106

The assignment problem in distributed computing

by

Anand Medepalli

A Dissertation Submitted to the

Graduate Faculty in Partial Fulfillment of the

Requirements for the Degree of

DOCTOR OF PHILOSOPHY

Department: Mathematics
Major: Mathematics

Approved:

Signature was redacted for privacy.

In Charge of Major Work

Signature was redacted for privacy.

For the Major Department

Signature was redacted for privacy.

For the Graduate College

Members of the Committee:

Signature was redacted for privacy.

Iowa State University
Ames, Iowa
1992

DEDICATION

Dedicated to my family, especially to my mother, whose sacrifice and love made this day possible. This achievement is mine in name and hers in spirit.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACKNOWLEDGEMENTS

I would be forever thankful for the encouragement, help, and support I received from my thesis advisor Dr. David Fernández-Baca of the Department of Computer Science in the completion of this degree. I could not have chosen a better person to help me define and achieve, what is easily the most significant goal of my life. David introduced me to research and to the world of algorithms and was ever so patient in doing so. He took me on without knowing anything about me and trusted me to be able to conduct research. He was a friend who stood by me during moments of frustration and was a patient teacher when I needed guidance in my work. I learned a lot from you David, and I can never thank you enough for all that you have done for me.

I would also like to record my appreciation for Dr. Peter Colwell of the Department of Mathematics for agreeing to be my co-major professor. Though he did not have any technical input in my thesis work, I am certain that I could not have completed this degree without his advice, kindness, patience, and the words of encouragement he always had for me. Without his help, I could not have met the language requirement in Russian. Thank you Dr. Colwell for everything — you made my life in the department a very pleasant one.

I would also like to thank Dr. Wolfgang Kliemann and Dr. Stephen Willson,

both of the Department of Mathematics, Dr. Howard Meeks of the Department of Industrial and Manufacturing Systems Engineering, and Dr. Gurpur Prabhu of the Department of Computer Science for serving on my program of study committee. In addition, I would like to thank two professors who have had a profound influence on my student life. One is Dr. Arlington Fink of the Department of Mathematics at Iowa State, who introduced me to the field of optimization, and the other is Dr. Tarun Kumar Mukherjee of the Department of Mathematics in Jadavpur University, Calcutta, who showed me the beauty of mathematics for the first time, and changed the course of my life forever. I know I would not be where I am today without their teachings.

My family, both immediate and extended, especially my late grandfather and uncles, have a great deal to do with this day. Over the years, they stood by me and my dreams, and helped me get to this stage. I remain eternally grateful to all of them.

No person is an island, and I am no exception. My friends at Iowa State were god-sent and they made my life as a graduate student just that much more easier and fun. Thank you Anindya for standing by me when I needed your support. Thank you Vimal and Nitin for helping me laugh at a time when I desperately needed to. Living with you two as roommates was a high point in my life at Ames, and I will always cherish the times we had. Thank you John, Cindy and Ellen for providing me with the family I needed when I was thousands of miles away from home. We will always be family. Especially you Cindy, who knew exactly what to say to make me feel better whenever I was down. And yes John, I owe you a few meals!! Thank you J.D., Lisa, Kathy, Tim, Jay, Kirk and all my friends in the Department of

Mathematics, who made studies a challenge and at the same time, the department a fun and friendly place to visit. A special thanks to Lisa, who was always there for me to help put things in perspective. Thank you Kurt and Paul, for patiently answering all my questions about Canvas, Adobe and Mathematica in the math lab. Thank you Piyush, for helping me get started on the *C* language, and for the company during summer, when this thesis was on its last leg. Thank you Joe, for helping with shell programming and for the numerous tips on *C*. Thank you Tim-san for the help in Russian. Thank you Madonna, Ruth, and Jan, for all the help and cheer in the math office.

Two very special people in my life are Mithu and Titu. Both have been my friends for a long time, and both have been there for me whenever I needed them. Thank you both, for helping me start and finish this degree.

Last, but not least, I would like to acknowledge the Department of Mathematics at Iowa State University, for the financial support during my tenure as a graduate student.

CHAPTER 1. INTRODUCTION

A recent trend in computer systems has been to distribute computation among several physical processors. Distributed processing applications range from large database installations, where processing load is distributed for organizational efficiency, to high-speed signal processing and image analysis, where extremely fast processing must take place in real-time environments. The modularity, flexibility and reliability of distributed processing makes it attractive to users. One of the driving forces behind this interest in distributed systems is the affordability and wide availability of large heterogeneous networks of workstations. These systems have several advantages over traditional systems. These include the capacity for incremental growth, increased reliability and availability, since parts of the system can be down without disturbing the users on other parts. Programs can be written so as to exploit the different capabilities of the processors in the network in the sense that, program modules can be assigned to different processors, depending on their particular computational requirements. The drawback, however, is the overhead of communication protocol, which can be a major source of inefficiency. In this dissertation, we shall study some of the combinatorial optimization problems that arise when trying to make the best use possible of the processing power of distributed systems.

The Distributed System Environment

A *distributed system* is a collection of two or more *processors* each with its own private memory. These processors are interconnected by a communication network and a system-wide operating system provides a message-passing mechanism among the processors. The processors may vary in size and function. They may include personal computers, workstations, and large general-purpose computer systems.

One of the motivations behind distributed systems is *computation speedup.* Suppose a particular computation, which we will refer to as a *job*, can be partitioned into a number of communicating tasks, which we will refer to as *modules.* A module could be a collection of procedures or subroutines, or could be one or more data files. The availability of a distributed system allows us to distribute the modules among various processors. The motivation in doing so is to take advantage of the specific efficiencies of certain processors in executing certain types of computation. Thus if our job does floating point computation in one procedure and extensive symbol manipulation in another, we would like to execute the first procedure on a processor with a powerful floating point unit, and the second on a processor with an instruction set designed for symbol manipulation. The program activity then moves among processors as execution proceeds. The program may be serial, in which case only one module is active on one processor at a time, or parallel, in which case several modules are concurrently active on several processors. In addition, if a particular processor is overloaded with modules, some of them may be moved to other, lightly loaded, processors. This in effect, expedites the job and enhances system performance. A representation of a generic distributed system is shown in Figure 1.1. The key elements in this system are the set of n modules $\{M_1, \cdots, M_n\}$, which make up a job, and a module allo-

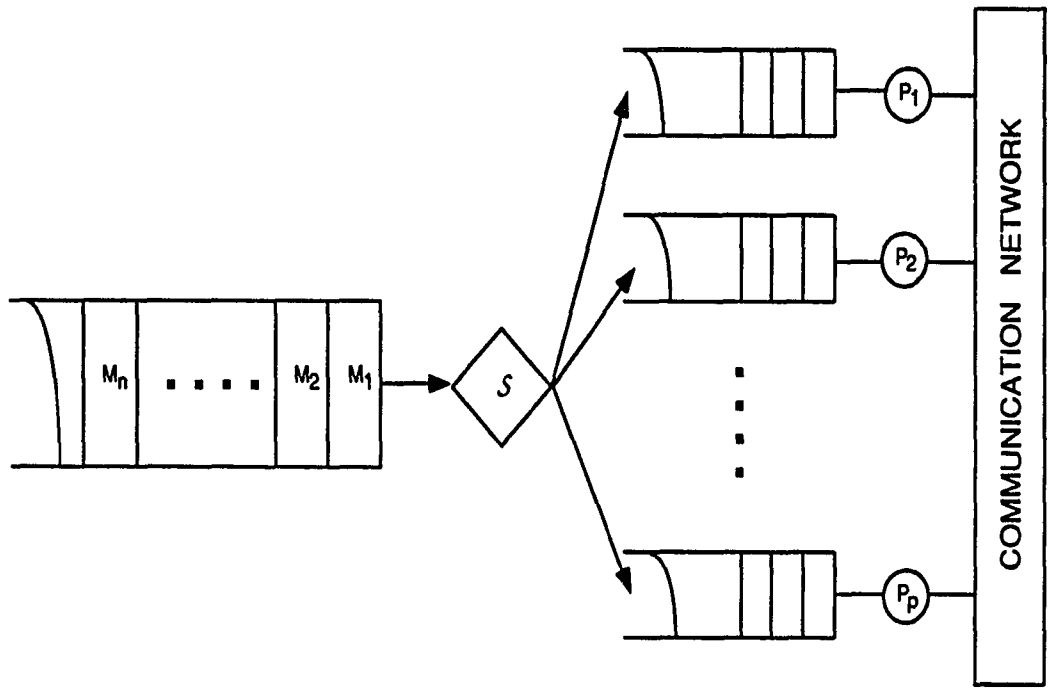


Figure 1.1: A distributed processing system

cation mechanism, S, which assigns each of the modules to one of the p processors, $\{P_1, \cdots, P_p\}$.

The Module Allocation Problem (MA)

In this dissertation, we focus on the allocation or the assignment problem in distributed computing, which involves the initial assignment of the modules of a job to the processors in the distributed system. We assume that the job partitioning process has been performed and that jobs arrive in the system already partitioned. We further assume that a module is an indivisible entity, the smallest viable computational unit. Executing a module on a processor involves the so called *execution cost.* In general, any module can be assigned to any processor, but because of differ-

ent processor capabilities, speeds, or resources, the cost of executing a module may differ from processor to processor. The cost may be the running time of the module, a financial cost, or some other measure of resource usage. When modules have data to communicate to one another, the processors to which they are assigned to, must communicate with each other. When this happens, a *communication cost* between the modules is incurred because of the overhead due to the communication protocols and transmission delays in the communication network. We assume that these costs are all available to us. Our goal is to assign these modules to the processors so as to minimize a certain measure of the *total cost* of the job.

Factors Involved in the Assignment Problem

On the surface, the assignment problem may seem simple. An intuitive strategy would be to assign the modules so that all processors in the system are evenly loaded. See Figure 1.2 for an example system where this strategy is implemented. We assume that each module has identical processing requirements and each processor has identical processing abilities. For simplicity, we further assume that each processor can process one module per unit of time. For the illustrated case, the system is able to process the entire job in two time units. This would be an acceptable solution if all interprocessor communication (IPC) costs are zero; i.e., there is no overhead of passing control and parameters from a module resident on one processor to one resident on another. However, on most real systems this overhead is significant. If, on the other hand, we attempt to minimize the IPC without considering load balancing, communicating modules tend to be assigned to relatively few processors, thus overloading them. An example of this is shown in Figure 1.3. The allocation used

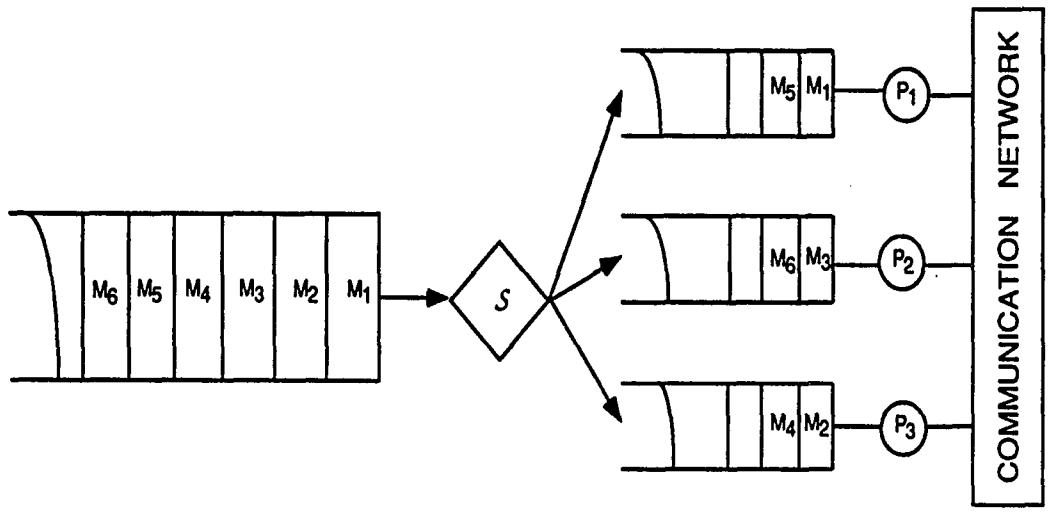


Figure 1.2: A balanced load allocation strategy

there generates a minimum IPC, but the processing time for the entire job increases by a factor of three. Thus, it is clear that the two conflicting factors, load balancing and IPC, influence the allocation strategy for optimal system performance.

Applications of Module Allocation

The module allocation problem is an important aspect of all phases of the development of a distributed system.

- *Design phase.* Here, it is necessary to evaluate competing design configurations including network topology, channel bandwidth, number of processors, etc. of the distributed system. Module allocation allows determination of the value of these parameters to achieve a desired level of performance.

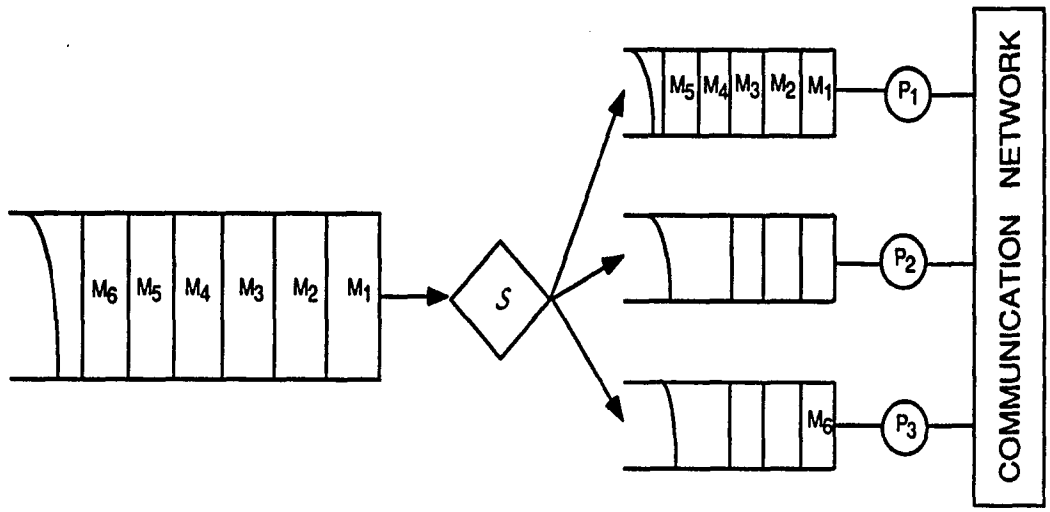


Figure 1.3: A minimum IPC allocation strategy

- *Scheduling phase.* Here, local CPU scheduling of the individual modules is done, with due consideration to the overall progress of the job. In certain real-time systems, it is important to assign incoming modules to processors to meet critical timing constraints. An additional complication arises in this phase due to the presence of unfinished modules still resident in processor queues, and in the processors themselves, at the time of a new module allocation. It cannot be assumed that all processors are ready and available at allocation time. Further, there may be *precedence constraints* involved, wherein a certain module has to wait for some other module to finish execution before it can start executing. Module allocation can provide a strategy for optimum use of available resources.

- *Migration phase.* In this phase, dynamic reassignment of modules to processors in response to changing loads on the processors and communication networks is done. Module allocation can provide a strategy to reallocate modules dynamically. This process should be transparent to the user.

In practical situations, system resources are limited. That is, the number of available processors, processor speed, memory capacity, and the number and types of peripheral devices are fixed and limited by available system resources. In real-time applications, allowed elapsed time is also a limited resource. In this case, the module allocation strategy must provide simple and fast methods to meet system performance requirements.

Model Formulation: The Graph Theoretic Approach

Graph theoretic techniques have been successful in modeling many problems of assignment in distributed systems. The reason is that the notions of vertex, edge and graph partitioning from graph theory are very similar to the concepts of module, communication and program partitioning respectively in distributed programs. There is usually a very clear relationship between a problem and its graph theoretic model. This gives a great insight into the structure and properties of the problem. However, the creation of the model is not enough, a solution to the required problem must be found. There exist many problems that can be stated very simply in graph theoretic terms but are as yet unsolved.

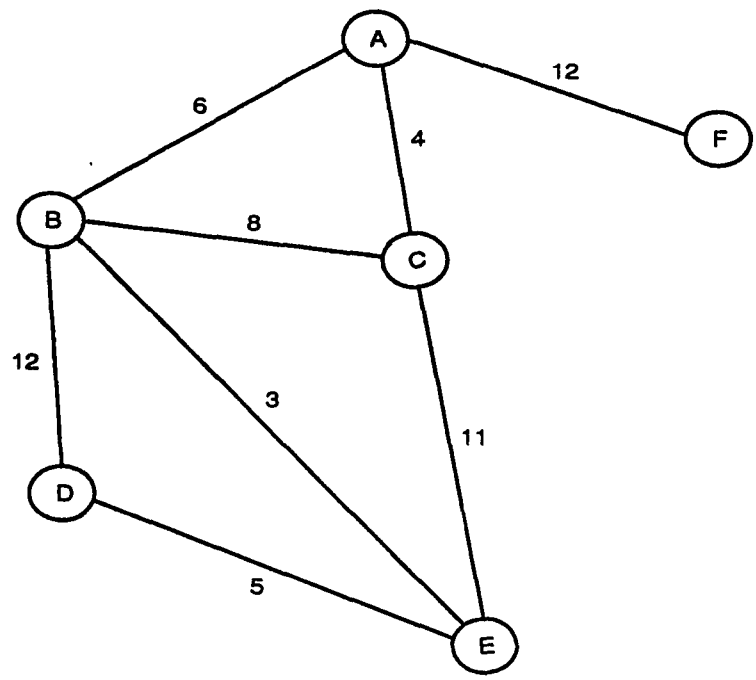


Figure 1.4: An example communication graph

Problem Definition

Suppose we have a distributed program of n modules each of which must be assigned to one of p processors and an undirected graph G[1], called the *communication graph* of the system, whose vertices are the modules of the program and whose edges indicate that the corresponding modules communicate. We assume that the processors are completely connected, so that any processor can communicate with any other processor. Without loss of generality, assume that the modules are numbered from 1 to n and processors from 1 to p. P_i denotes the processor to which module i has been assigned. The cost of executing module i on processor v is denoted by

[1]Throughout this dissertation, $V(G)$ and $E(G)$ denote vertex and edge sets respectively of the graph G. We also assume that $\mid V(G) \mid = n$ and $\mid E(G) \mid = m$.

$e_i(v)$. When a module i must be assigned to a particular processor v, $e_i(u) = \infty$, for all processors $u \neq v$. If modules i and j communicate, then $c_{ij}(u,v)$ denotes the communication cost between modules i and j when they are assigned to processor u and v respectively. If $u = v$ then $c_{ij}(u,v)$ can be interpreted as an *interference cost*, which could indicate the degree of incompatibility between modules i and j. For instance, a pair of modules that are both highly CPU-bound would have greater interference costs than a pair in which one module is CPU-bound and the other is I/O-bound. If $c_{ij}(u,v) = \infty$ for all processors u and v, then modules i and j must be assigned to the same processor. Communication costs are said to be *uniform* if the communication cost between co-resident modules is zero and every pair of modules assigned to different processors incur the same communication cost, (i.e., for any two modules i and j, $c_{ij}(u,v) = r_{ij}$ is independent of u and v if $i \neq j$). We assume that all the costs are available a priori in tables. An example communication graph with six modules A — F, and uniform communication costs is shown in Figure 1.4. Note that in this example, the costs are represented as weights on the edges.

An *assignment* of the job is a complete specification of the processors on which modules of the job are executed. It can be represented by a vector $\mathcal{A} = (P_1, \cdots, P_n) \in \{1, \cdots, p\}^n$. In the most common version of the module allocation problem, which we will refer to as MA, the *cost* $C(\mathcal{A})$ of $\mathcal{A}$ is the sum of all the module execution costs and the intermodule communication costs given the assignment $\mathcal{A}$; i.e.,

$$C(\mathcal{A}) = \sum_{i \in V(G)} e_i(P_i) + \sum_{(i,j) \in E(G)} c_{ij}(P_i, P_j).$$

The problem is to find an assignment of minimum cost; i.e., to find the assignment $\mathcal{A}^*$ such that $C(\mathcal{A}^*) \leq C(\mathcal{A})$ for all possible assignments $\mathcal{A}$. Throughout this dis-

sertation, we denote the value of the optimum solution by C^*.

Variants of the Assignment Problem

In this section we introduce some variations of MA. Subsequent chapters consider these problems in detail.

Parametric Module Allocation (PMA)

In multiple computer systems, the optimal assignment of a distributed system is sensitive to load conditions on the processors and the traffic on the IPC link. In other words, costs vary over time. Often one or more of the processors on which a distributed program is running is time-shared with other applications. The optimal assignment then changes each time the load on one of the processors changes. This is because, as more and more load is put on the processor, the time (i.e., the cost) for executing modules on it increases. The optimal assignment at a new value of load may warrant a relocation of some modules between the processors. On the other hand, there could be transmission delays on the communication network, which might slow the links in the network, leading to higher communication costs. The problem thus is to find a sequence of optimal assignments that are found as the loads vary. We refer to this problem as PMA and investigate it in Chapter 4.

Constrained Module Allocation (CMA)

The solution to MA assumes that there are no resource constraints on any of the processors, so that if need be, we may assign any number of modules, requiring any amount of any resource, to any processor. This creates no problem if all processors

have enough resources to cater to the entire program. This, however, may not be possible in real situations and hence we need to take into account resource restrictions on processors. We assume that one of the processors has a limited resource, referred to as "memory" and investigate the solution of the assignment problem subject to this constraint in Chapter 5. We refer to this problem as CMA.

Balanced Module Allocation (BMA)

The optimum solution to MA may be very unbalanced, in that several modules may be placed on a single processor in order to minimize IPC. This may lead to the overloading of the processor. As such, it would be desirable to obtain assignments which distribute the modules among the processors evenly, thus balancing the loads on them. As it turns out, this problem is closely related to CMA and is called the balanced module allocation problem. This problem is referred to as BMA and is also investigated in Chapter 5.

Other Constraints

Other constraints that arise in real situations include *precedence relationships* among modules, which specify the execution sequence of the modules; *real-time constraints* which indicate the maximum amount of time that a processor is allowed to finish processing the modules assigned to it; *queuing delays* which arise due to a module waiting to begin execution on a processor which is busy executing some other module; etc. We do not consider these constraints in our work. Needless to say, however, that these are important considerations and make solving MA that much harder.

Outline of the Thesis

We conclude this chapter with a plan of this dissertation. Chapter 2 presents some complexity results and the computationally intractable nature of MA, CMA and BMA, thus highlighting the theoretical limitations in solving them. We show that MA is NP-complete and that both CMA and BMA are strongly NP-complete. Chapter 3 presents a survey of past and related work on this problem. Chapter 4 discusses PMA. In that chapter, we consider communication graphs which are k–trees[2]. We develop efficient algorithms to solve this restricted version of PMA. As an auxiliary result, we present an algorithm to find a $(k+1)$–vertex separator in a k–tree. The results of this work will appear in *IEEE Transactions on Computers.* Chapter 5 focuses on CMA and BMA. We present exact dynamic programming algorithms to solve these problems and present approximate algorithms for k–trees. Faster algorithms for trees with uniform costs are also developed. Chapter 6 discusses the implementation and experimental results of the dynamic programming algorithms developed in Chapter 5. Finally, Chapter 7 concludes the thesis and presents some future directions and open problems.

[2]See Chapter 4 for a definition

CHAPTER 2. COMPUTATIONAL COMPLEXITY OF THE ASSIGNMENT PROBLEM

This chapter deals with the computational complexity of MA, CMA and BMA. We begin with a review of algorithm analysis techniques and the theory of NP-completeness. See [27, 39, 58] for more details on these subjects.

Analyzing Algorithms

By *analyzing* an algorithm, we mean predicting the resources that the algorithm requires. Occasionally, resources such as memory or communication bandwidth are of primary concern, but most often it is the computational time that we want to measure. We would like to do that without actually implementing it on a specific computer. The advantages in doing so are clear. It is much more convenient to have simple measures for the efficiency of an algorithm than to implement it and test the efficiency every time a certain parameter in the underlying computer system changes.

Unfortunately, it is usually impossible to predict the exact behavior of an algorithm as there are too many influencing factors. Instead, we try to extract the main characteristics of the algorithms by defining certain parameters and measures that are most important for the analysis. Many implementation details are ignored. The analysis is thus only an approximation; however, even this approximate analysis can

yield significant information about the algorithm.

The usual methodology used to predict an approximate run time of an algorithm ignores constant factors and concentrates on the behavior of the algorithm as the input size increases. The number of different possibilities for inputs is enormous and most algorithms behave differently for different inputs. In general, the running time of an algorithm increases with the size of the input, so the running time of the algorithm is defined as a function of the input size. We next formalize the concepts of "input size" and "running time" of an algorithm.

INPUT SIZE. The input size depends on the problem being studied. For many problems, such as sorting, the input size is the number of items in the input. For other problems, like multiplying two integers, the input size is the total number of bits needed to represent the input in ordinary binary notation. Sometimes, more than one number describes the input size. For example, if the input to the algorithm is a graph, then the input size can be described by the numbers of vertices and edges in the graph. In this dissertation, we let the number of modules and processors be the input size to our algorithms. For the rest of this chapter, unless otherwise specified, n will denote the input size.

RUNNING TIME. The running time, also known as the *time complexity*, of an algorithm on a particular input is the number of primitive operations or "steps" executed. We identify one or more major steps in the algorithm; for instance, in sorting, comparisons constitute a major step. We assume that these major steps dominate the computation. Since we will ignore constant factors, it will suffice to estimate the total number of the major steps executed by the algorithm and report that as its running time. Throughout this dissertation, we shall be interested in the

worst-case running time, which is the longest running time for any input of size n. Even though this may be overly pessimistic for some algorithms, for others the worst case occurs fairly often. We need the following definition.

Definition 2.1 (THE O NOTATION) A function $g(n)$ is $O(f(n))$ for another function $f(n)$ (pronounced "Big Oh" of $f(n)$), if there exist constants c and N, such that, for all $n \geq N$, we have $g(n) \leq cf(n)$.

Note that, by this definition, the function $g(n)$ may be substantially less than $cf(n)$. The O notation bounds it only from above. For example, $5n^2 + 15 = O(n^2)$ since $5n^2 + 15 \leq 6n^2$ for $n \geq 4$. This notation allows us to ignore the constants conveniently. We always write $O(n)$ instead of, say, $O(5n + 4)$. Similarly, we write $O(\log n)$ without specifying the base of the logarithm, because changing the base would change the logarithm only by a constant. Also, note that $O(1)$ denotes a constant.

An Overview of NP-completeness

The Complexity Class P

Definition 2.2 (POLYNOMIAL-TIME ALGORITHM) An algorithm is said to be a polynomial-time algorithm if, for inputs of size n, its running time is $O(n^c)$ for some constant c.

Many problems, like sorting an array of numbers, have polynomial-time algorithms. Such algorithms are said to be *efficient* and the corresponding problems are said to be *tractable*. The *complexity class P* (for polynomial time) is the class of all tractable problems. The terminology can be misleading, since, after all, algorithms

that run in $O(n^{10})$ are not efficient by any standard. Nevertheless, this definition is valid from the practical viewpoint — the vast majority of tractable problems have practical solutions and conversely, algorithms whose running times are larger than any polynomial are not usually practical for large inputs.

Decision Problems

Definition 2.3 (ABSTRACT PROBLEM) An abstract problem is a binary relation on a set I of problem *instances* and a set S of problem *solutions*.

As an example, consider the problem of finding the shortest path between two given vertices a and b in a graph G. An instance of this problem is a triple $\langle G, a, b \rangle$. A solution is a sequence of vertices in the graph, with an empty sequence denoting the empty path. The abstract problem is a relation that associates the triple with a solution.

Definition 2.4 (DECISION PROBLEM) A *decision problem* is an abstract problem having a yes/no solution. In this case, the problem is a function that maps the instance set I to the solution set $\{0, 1\}$.

In the above shortest-path example, the decision version would be to answer the following question. Given G, vertices a and b in G and an integer $k \geq 0$, does there exist a path whose length is at most k?

The theory of NP-completeness compels us to cast optimization problems, such as our assignment problem, as decision problems. This can be done by imposing a bound on the function to be optimized. If we can provide evidence that the decision

version is hard (i.e., not tractable), we also provide evidence that the related optimization problem is hard. It is usually easier to go this route than to try and directly deal with the optimization problem.

The Complexity Class NP

A *verification algorithm* is a two-argument algorithm A, where one argument is an ordinary input string x and the other is a binary string y called the *certificate*. A two-argument algorithm A *verifies* an input string x if there exists a certificate y such that $A(x, y) = 1$. The *language* $\mathcal{L}$ verified by a verification algorithm A is the set of binary strings x for which there exists a certificate y, such that $A(x, y) = 1$. In other words, A uses y to prove that $x \in \mathcal{L}$. Further, for any string $x \notin \mathcal{L}$, there must be no certificate proving that $x \in \mathcal{L}$. Note that an input $x \in \mathcal{L}$ may have many certificates that do not verify x; all we need is one certificate which will verify its membership in $\mathcal{L}$. The running time of a verification algorithm refers to the worst-case running time for inputs $x \in \mathcal{L}$ (inputs not in $\mathcal{L}$ are ignored). The *complexity class NP* (for nondeterministic polynomial time) is the class of languages that can be verified by a polynomial-time algorithm. While it is known that P $\subset$ NP, it is still an open problem if P = NP.

Polynomial-time Reductions

A decision problem $\mathcal{D}$ can be viewed as a *language-verification* problem. Let $\mathcal{L}$ be the subset of all possible inputs $\mathcal{I}$ for which the answer to $\mathcal{D}$ is "yes". $\mathcal{L}$ is the *language* corresponding to $\mathcal{D}$. $\mathcal{D}$ is thus to verify whether or not a given input $x \in \mathcal{L}$. In what follows, we use the terms problem and language interchangeably.

Definition 2.5 (POLYNOMIAL REDUCTION) Let $\mathcal{L}_1$ and $\mathcal{L}_2$ be two languages from the input spaces $\mathcal{I}_1$ and $\mathcal{I}_2$ respectively. We say that $\mathcal{L}_1$ is polynomially reducible to $\mathcal{L}_2$ if there exists a polynomial-time algorithm that converts each input $i_1 \in \mathcal{I}_1$ to another input $i_2 \in \mathcal{I}_2$ such that $i_1 \in \mathcal{L}_1$ if and only if $i_2 \in \mathcal{L}_1$.

Intuitively, a problem $\mathcal{D}_1$ can be reduced to another problem $\mathcal{D}_2$, if any instance of $\mathcal{D}_1$ can be easily rephrased as an instance of $\mathcal{D}_2$. Thus, if indeed $\mathcal{D}_1$ reduces to $\mathcal{D}_2$, then $\mathcal{D}_1$ is, in a sense, "no harder to solve" than $\mathcal{D}_2$.
We now define NP-completeness.

Definition 2.6 (NP-HARD PROBLEM) A problem X is said to be an NP-hard problem if every problem in NP is polynomially reducible to X.
(NP-COMPLETE PROBLEM) A problem X is said to be an NP-complete problem if (1) $X \in$ NP, and (2) X is NP-hard.

Thus, the NP-complete problems are intractable. [27, 39, 58] provide several examples of NP-complete problems.

Approximation Algorithms

Many problems of practical significance are NP-complete, but are too important to abandon merely because obtaining an optimal solution is hard. If a problem is NP-complete, we are unlikely to find a polynomial-time algorithm for solving it exactly. However, there are two approaches to getting around NP-completeness. First, if the actual inputs are small, then an exponential-time algorithm may be satisfactory. Second, it may still be possible to obtain near-optimal solutions in

polynomial time. In practice, this second approach is often good enough. We have the following definitions.

Definition 2.7 (ϵ-APPROXIMATE ALGORITHM) An ϵ-approximate algorithm for a minimization problem Π is an algorithm that, for any instance I of Π, produces a solution of cost C such that $C \leq (1+\epsilon)C^*$, where C^* is the cost of the optimum solution for I.

Definition 2.8 (APPROXIMATION SCHEME) An approximation scheme for Π is a family of algorithms $\{A_\epsilon\}$ such that for each $\epsilon > 0$, A_ϵ is an ϵ- approximate algorithm for Π.

Definition 2.9 (FULLY POLYNOMIAL-TIME APPROXIMATION SCHEME (FPTAS))
(1) An approximation scheme is a polynomial-time approximation scheme if, for any fixed $\epsilon > 0$, A_ϵ runs in time polynomial in the size of its input.
(2) The scheme is a fully polynomial-time approximation scheme if its running time is polynomial both in $1/\epsilon$ and the input size.

Definition 2.10 (STRONG NP-COMPLETENESS) A decision problem on graphs is strongly NP-complete if there exists a polynomial q such that the problem remains NP-complete even when restricted to the case where no cost exceeds $q(n)$, where n is the number of vertices.

We need the following lemma in our subsequent discussions. It is taken from [39], pp. 140–141.

Lemma 2.1 *Suppose that* Π *is an optimization problem on graphs such that (a) the decision version of* Π *is strongly NP-complete, and (b) for any instance* I *of* Π, *the*

optimal cost is polynomially bounded in n and in the size of the largest cost appearing in I. Then there exists no FPTAS for Π, *unless* $P = NP$.

Complexity of MA, CMA and BMA

We now investigate the complexity of module allocation in order to show the difficulties and limitations encountered in solving the assignment problem. Recall that p represents the number of processors and P_i denotes the processor that module i is assigned to.

We begin with the following theorem, the idea for which is due to Tamir [82]. To our knowledge, the proof of this theorem has not appeared in print.

Theorem 2.1 *MA with uniform costs is NP-complete even for* $p = 3$.

Proof. The *3-way Cut problem* is defined as follows: Given a graph G, a set of 3 specified nodes A, B, and C , find a minimum cardinality subset $S \subseteq E(G)$, such that the removal of S from $E(G)$ disconnects each of the above three nodes from the other two. Dalhaus, et al. [28] have shown that the 3-way cut problem is NP-complete. We reduce an instance $\mathcal{C}$ of the 3-way Cut problem to an instance $\mathcal{M}$ of MA with uniform communication costs and three processors as follows.

Let the nodes A, B and C correspond to the 3 processors and let the communication graph be $H = G - \{A, B, C\}$. Define execution and communication costs as follows. For each module $i \in V(H)$, define

$$e_i(A) = \begin{cases} 0 & \text{if neither } (i,B) \text{ nor } (i,C) \in E(G) \\ 1 & \text{if exactly one of } (i,B) \text{ and } (i,C) \in E(G) \\ 2 & \text{otherwise.} \end{cases}$$

$e_i(B)$ and $e_i(C)$ are similarly defined.

For each edge $(i,j) \in E(H)$ and for any two processors u and v, define

$$c_{ij}(u,v) = \begin{cases} 1 & \text{if } u \neq v \\ 0 & \text{otherwise.} \end{cases}$$

The theorem follows immediately from the following claim.

Claim: There is a solution S to $\mathcal{C}$ if and only if there is a solution $\mathcal{A}$ to $\mathcal{M}$ with cost $C(\mathcal{A}) = |\ S\ |$.

Proof. First, let S be a solution to $\mathcal{C}$. Let V_x be the connected component in G containing the node x, where $x \in \{A, B, C\}$. Note that there could be other components in G as well. Define an assignment $\mathcal{A}$ to solve $\mathcal{M}$ as follows. For each $i \in V(H)$, define

$$P_i = \begin{cases} x & \text{if } i \in V_x, \text{ where } x \in \{A, B, C\}. \\ A & \text{otherwise.} \end{cases}$$

The following two cases are possible.

Case 1: $i \in V_A$. Now, if i is not connected to either of B and C in G, then those edges need not be cut. This is reflected in the execution costs, since $e_i(A) = 0$ in this case. If i is connected to exactly one of B and C, then that edge has to be cut and this also is reflected in the execution costs, since in that case $e_i(A) = 1$. The case when i is connected to both B and C is taken care of by defining $e_i(A) = 2$. Next, consider the edges. Suppose $(i,j) \in E(G)$ where $j \notin \{A, B, C\}$. If $i, j \in V_A$, then (i,j) need not be cut; consequently in $\mathcal{M}$, the cost on this edge is defined to

be zero. On the other hand, if $j \notin V_A$, then (i,j) has to be cut, for which, in $\mathcal{M}$, we define the cost on such edges to be 1. Similar arguments can be made about the nodes in V_B and V_C.

Case 2: $i \notin V_x \ \forall\ x \in \{A,B,C\}$. In this case, i is not connected to any of A,B,C in G and hence it does not belong to any of V_A, V_B, V_C after $\mathcal{C}$ is solved. As such, edges involving i do not contribute to $\mathcal{S}$. By definition, the corresponding module node i in H is assigned to A and $e_i(A) = 0$. Also, all the modules i is connected to are assigned to A as well, so there is no communication cost incurred between these modules.

Thus we see that the assignment $\mathcal{A}$ we define, indeed solves $\mathcal{M}$, and furthermore, $C(\mathcal{A}) = |\ \mathcal{S}\ |$.

Conversely, suppose that $\mathcal{A}$ is a solution to $\mathcal{M}$. Define, for each $x \in \{A,B,C\}$, the set

$$V_x = \{x\} \cup \{i \in V(H)\ :\ P_i = x\}.$$

Then, define,

$$S_1 = \{(u,v) \in E(G)\ :\ u \in V_x \text{ and } v \in V_y, \text{ where } x,y \in \{A,B,C\} \text{ with } x \neq y\}.$$

Again, for $x \in \{A,B,C\}$, define

$$S_x = \{(u,z)\ :\ u \in V_x \text{ and } z \neq x\}.$$

Finally, define

$$\mathcal{S} = S_1 \cup \left(\bigcup_{x \in \{A,B,C\}} S_x \right).$$

Then, clearly, $\mathcal{S}$ is a solution to $\mathcal{C}$, and furthermore, arguments similar to the one in the previous part can be used to show that $| \mathcal{S} | = C(\mathcal{A})$. Hence the claim. □

Regarding the complexity of finding an approximation algorithm to solve MA, the following result was proved in [32].

Theorem 2.2 *Unless $P = NP$, there exists no polynomial-time ϵ-approximate algorithm for MA, even if $p = 3$ and the underlying graph is planar and bipartite.*

Given the difficulty in solving MA, it is not surprising that CMA and BMA are even harder. In fact, they are strongly NP-complete, as will be shown later. We shall consider their decision versions, where, along with an instance of each problem, we are given an integer U and are asked to determine if there exists a solution of cost at most U. CMA was proved NP-hard in [68] by a reduction from the knapsack problem. Indeed, the argument in [68] implies that the problem remains NP-hard regardless of the structure of the communication graph. For broader classes of graphs, we have the following result.

Theorem 2.3 *CMA is strongly NP-complete for $p \geq 2$ even if G is planar. Therefore, unless $P = NP$, there exists no FPTAS for CMA.*

Proof. By Lemma 2.1, it suffices to prove the strong NP-completeness of CMA. We use reduction from the vertex cover problem, which is defined as follows: Given a graph G and an integer $K < |V(G)|$, determine if there exists a $A \subseteq V(G)$ such that for all $(u, v) \in E(G)$, $\{u, v\} \cap A \neq \emptyset$, and $|A| \leq K$. This problem is strongly NP-complete even for planar graphs [39]. An instance of vertex cover can be reduced to an instance of CMA with communication graph G, $p = 2$, $U = M = K$ and where

costs and memory requirements are defined as follows. For all $i \in V(G)$, $m_i = 1$, $e_i(1) = 1$, and $e_i(2) = 0$. For all $(i,j) \in E(G)$, $c_{ij}(a,b) = n+1$ if $a = b = 2$, and $c_{ij}(a,b) = 0$ otherwise. Intuitively, $x_i = 1$ will mean that $i \in A$, and $x_i = 2$ will mean that $i \notin A$. Obviously, all costs are polynomially bounded in n. □

Theorem 2.4 *BMA is strongly NP-complete, regardless of the structure of the communication graph. Therefore, unless $P = NP$, there is no FPTAS for BMA.*

Proof. We use a reduction from the minimum makespan schedule problem (MMS), which is known to be strongly NP-complete [39] — the rest follows from Lemma 2.1. The input to MMS is a set of n jobs, with processing times t_i, to be scheduled on p identical processors. A schedule is an assignment of jobs to processors. The makespan of a schedule is the maximum time that a processor is busy under that schedule. The problem is to find a schedule that has minimum makespan. MMS is a special case of BMA where the execution cost of a module is independent of the processor to which it is assigned and all communication costs are zero. Obviously again, all costs are polynomially bounded in n. For such instances of BMA, the actual communication graph is immaterial. □

We conjecture the following result.

Conjecture 2.1 *Unless $P = NP$ there exists no FPTAS for BMA for any fixed $p \geq 2$.*

It is thus clear that the assignment problem and its variants are all quite difficult problems to solve. Despite the negative results for CMA and BMA, we show in Chapter 5 that these problems do have FPTAS when the communication graphs are k-trees.

CHAPTER 3. SURVEY OF PAST WORK

The module assignment problem has received a lot of attention in the past decade. One approach to this problem has been through the development of *centralized* algorithms. These algorithms minimize an objective function which meets the goals mentioned in Chapter 1. This dissertation focuses on this approach. To put our work in context, we shall review the work that has been done in the field.

Network Flow Techniques

The pioneering work in module allocation was done by Harold Stone [79], who applied network flow algorithms to solve MA in a dual-processor distributed system with uniform communication costs. He showed that MA in this case, can be transformed into a network flow problem such that there is a one-to-one correspondence between assignments and cutsets. Using any one of the several available maximum flow algorithms (e.g., see [27]), one can find a minimum cut in the network, which in turn due to the above correspondence, gives the assignment of minimum cost. Since the maximum flow problem can be solved in $O(n^3)$ time[1], MA in a dual-processor system with uniform communication costs can be solved in $O(n^3)$ time. For the three processor case, Stone [81] extended his earlier result and developed an algorithm that

[1]There are faster algorithms which run in time $O(n^3/\log n)$, see [22].

finds a minimum weight three way partition in the communication graph. This algorithm works in most cases, however there are pathological graphs for which it fails to find the optimal cost three way partition. For these graphs, the algorithm does indicate that the solution found is not optimal and gives a bound on the sub-optimality. The fact that Stone's algorithm does not work in all cases is not surprising, since, as was shown in Chapter 2, the three processor problem is NP-complete.

The assignments obtained using Stone's network flow techniques are *static* in the sense that once a module is assigned to a processor, it remains there throughout program execution. In order to make the best use of resources in a distributed system, we must relocate modules during program execution whenever this leads to improved efficiency. Such an assignment is said to be *dynamic*. There is, however, the extra cost of relocation in this case and clearly the gains from relocating modules must outweigh the cost of relocation. Further, there are the residence costs — these are costs of modules residing on processors without executing. Bokhari [15] extended Stone's network flow techniques to obtain an algorithm that finds an optimum dynamic assignment (i.e., one that minimizes the sum of execution, residence, relocation, and communication costs). The complexity of Bokhari's dynamic algorithm is same as that of Stone's static algorithm.

Lo [57] has pointed out that Stone's model (and for that matter Bokhari's above model) has the deficiency that it makes no direct effort to achieve a balance in processor workloads, yielding assignments which utilize only a few of the processors. She extended Stone's model to include interference costs between modules when they are assigned to the same processor. In that work, a heuristic algorithm is developed which combines recursive invocation of maximum flow algorithms with

a greedy-type algorithm to find sub-optimal assignments. She has experimentally shown that the addition of interference costs as a factor in the model greatly improves the concurrency of the assignments.

Dynamic Programming Techniques

MA is a special case of *nonserial dynamic programming.* As such, techniques for solving nonserial dynamic programming problems can be adapted to solve MA in certain cases. A well-known technique is *variable elimination* [9]. The basic idea in this technique is to replace the given problem by another one with fewer variables such that the optima of both the problems are the same. This is done by somehow capturing all the eliminated information in the objective function of the new problem. This process continues until either all variables are removed or the number of variables is sufficiently small to solve the problem directly by exhaustive enumeration.

Arora and Rana [4] seem to be among the first to use these ideas. They considered the two-processor MA when the communication graph G is a *tree*[2] and developed a $O(n)$ algorithm to solve the problem. Recently Sagar, et al. [71] found that the algorithm fails to find the optimum assignment on some graphs and suggested a modification. They may have been unaware of the fact that soon after Arora and Rana's work was published, Bokhari [16] gave a correct $O(np^2)$ time algorithm to solve MA on p processors when G is a tree. Billionnet [10] gave a $O(np)$ algorithm for the same problem when all the communication costs are uniform. Towsley [83] has solved MA when the number of processors is p and G is a *series-parallel graph.* His algorithm has a run time of $O(np^3)$. A mistake in his paper and its correction

[2]See Chapter 4 for definitions of trees, series-parallel graphs and k-trees.

were reported recently [53] and acknowledged by Towsley [84]. Fernández-Baca [32] generalized these results and developed a $O(np^{k+1})$ algorithm when G is a *partial k-tree* and a $O(np^{\lceil k/2 \rceil + 2})$ algorithm when G is an *almost tree* with parameter k. Indeed, trees are partial 1-trees and series-parallel graphs are partial 2-trees.

Parametric Module Allocation

The optimal assignment of tasks to processors in a distributed system is sensitive to load conditions on the processors and to the traffic on the communication links. In other words, costs vary over time. The optimal assignment may change when the load on one of the processors changes, since, as more and more load is put on the processor, the time (i.e., the cost) for executing modules on it increases. The optimal assignment at a new load may warrant a relocation of some modules between the processors. Communication links may also have varying loads, which, in turn, may affect the optimal assignment. Thus, it would seem that successive instances of MA have to be solved, where each instance differs from the others by modification of some parts of the problem data. Rather than solving each instance from scratch, it is desirable to develop methods which address the problem of efficiently solving all of the instances. The goal of parametric computing is to compute the cost of the optimum solution as a function of the parameters. Given this function, the cost of an optimum solution for any values of the parameters can be retrieved rather than computed. For simplicity, we assume that the loads (i.e., costs) vary as linear functions of some parameter t, for convenience referred to as *time*. Since the cost of any assignment is a linear function of t, one can verify that, in general, the function describing the optimum assignment is a *concave piecewise linear function* (plf), see

[42]. Points at which the plf changes its slope are called its *breakpoints.* It is at these points that the optimum assignment changes. See Chapter 4 for further definitions and examples.

Parametric MA for 2-processor systems was first studied by Stone [80], who analyzed the problem of determining the sequence of optimal assignments as the load on one of the processors is held fixed while the load on the other processor is varied. He showed that in this case, there can be no more than $n + 1$ different assignments. Each assignment is a line in the two-dimensional space. The optimal assignment is the lower envelope of these lines and can be found using an algorithm developed by Eisner and Severance [31], which uses no more than $n + 1$ applications of the maximum flow algorithm. Sinclair [77] studied the case where processor loads remain constant, but where there are varying transmission delays. Fernández-Baca and Slutzki [35] showed that if all costs (both execution and communication) vary, then for a tree structured program and fixed p, the optimal assignment has polynomial number of breakpoints and can be computed in polynomial time. Gusfield [42] considered the situation where the loads on both the processors vary simultaneously. Suppose that the loads on the two processors vary with the parameters t_1 and t_2 respectively. In this case, the cost of each assignment is no longer a line, but a plane in the three dimensional space. The optimal assignment, which is the lower envelope of all these planes, is therefore a *convex polyhedral set.* Each of its sides represents an assignment and is optimal for all points (t_1, t_2) that lie within its projection on the $t_1 - t_2$ plane, called the *load plane.* Cartensen [21] showed that the number of faces on the polyhedron, or equivalently the number of regions on the load plane (and hence assignments) may be exponential in n. Algorithms to construct this polyhedron have been given in [36]

and [42].

Constrained and Balanced Module Allocation

The first results on CMA are due to Rao, et al. [68], who studied the problem of finding the optimal assignment in a two-processor system where there is a memory constraint on one processor and infinite memory on the other. They were motivated by the situation that arises when a host computer with large memory shares its load with a smaller, more specialized processor with limited memory. They showed that this problem is equivalent to the knapsack problem, which is NP-complete [39]. Rao, et al. devised a method that uses network-flow techniques to reduce the size of the communication graph by condensing certain sets of modules, according to some criteria, into a single node. One of the minimum cuts in this reduced graph happens to be a feasible minimum cut in the original graph. This minimum cut is obtained by exhaustively enumerating all the cuts in the reduced graph. Unfortunately, this set of cuts can be exponential in size in the worst case. As such, their method does not guarantee polynomial efficiency for the general case.

BMA was originally proposed by Chu and Lan [25] as a way to obtain assignments where processor loads are balanced. Those authors considered uniform communication costs along with the additional constraint of precedence relationships among the modules. A two-phase heuristic algorithm was developed. In phase I, some modules are grouped into a single set. This grouping is based on factors like reducing IPC, so that heavily communicating modules are put into one set to avoid the IPC, and on the precedence relationship of the modules. These sets form a much smaller assignment tree for the phase II, where the actual assignment of the modules takes place.

This assignment is found using exhaustive enumeration and in the worst case, may take exponential time.

In this context, we should also mention an earlier work due to Chu, et.al. [24]. There, a 0–1 quadratic programming approach is taken (see next section) for a system with p processors, all of which have memory restrictions. In addition, each processor has a real-time constraint in that, the length of time required to execute all the modules assigned to a single processor is restricted. These constraints ensure a more balanced assignment than a regular solution to the corresponding instance of MA. Once again, however, this approach is not efficient, but serves as a heuristic to generate sub-optimal assignments.

Alternative Approaches

While the graph-theoretic approach is simple and adapts naturally to the assignment problem, it has some limitations. It cannot easily incorporate such features as memory restrictions, load balancing mechanisms, precedence constraints, etc. Further, it cannot measure the impact of queuing delays on throughput. These limitations have prompted researchers to consider alternative approaches.

One approach which has been used successfully is 0–1 quadratic programming. This is a flexible technique since we can easily incorporate constraints into the model as appropriate to the application, which is difficult, if not impossible, with the graph-theoretic approach. Pioneering this effort were Chu, et al. [24], who formulated the assignment problem as a 0–1 quadratic programming problem with linear constraints. They considered memory restrictions on all processors and the real-time constraint and developed heuristics to solve this problem. They also included the possibility

of processors being not fully connected. Recently, Billionet, et al. [11, 12] also considered the same approach to the assignment problem. In [11], MA is considered, while in [12], all processors are allowed to have memory constraints. Instead of solving this problem directly, the approach taken is to relax one of the constraints and form the Lagrangian dual of the original problem. Using branch and bound techniques, they solve the dual to obtain sub-optimal solutions to the original problem. This novel approach works rather well in finding approximate solutions to the assignment problem with one or more constraints.

Another method is to use efficient search techniques to find the optimal solution in the search space. Price and Pooch [67] discuss how such techniques can be applied to solve nonlinear assignment problems. Doty et al. [30] present a set of assignment problems and propose solution techniques based on dynamic programming.

Related Problems

Partitioning Problems for Parallel and Pipelined Programs

The research reported so far dealt with serial programs, i.e., programs in which only one module is active on one processor at a given time. For completeness, we shall survey results for the case of parallel programs, where two or more modules may execute concurrently for various periods during the lifetime of a program. The objective is to reduce the total "*wall-clock*" time of the program by running different parts of the program in parallel. All the factors that influence the time to execute a serial distributed program also apply to parallel programs. In addition, there is the problem of scheduling the parallel computation, i.e., arranging the order of execution of the various modules on the processors. This is the *scheduling problem* and is

beyond the scope of this dissertation.

The problem of optimally assigning the modules of a parallel program in a distributed system is also NP-complete [19]. In the general *partitioning problem*, one is given a multicomputer system with a specific interconnection pattern as well as a parallel program composed of modules that communicate with each other in a specified manner. One is required to assign the modules to the processors in such a way that the total execution time of the program is minimized. The partitioning problem has several applications including signal processing [20], image analysis [78] and the solution of partial differential equations [74]. Iqbal, et al. [50] studied the problem of uniformly distributing the load of a parallel program over a multiprocessor system and suggested different strategies for load balancing. They discuss both static and dynamic methods to do this. Bokhari [19] proposed efficient algorithms to solve the following problems:

1. *Partition chain-structured parallel or pipelined programs over chain-connected systems.*

   A *chain-structured program* has n modules numbered $1, \cdots, n$ such that module i is connected only to modules $i+1$ and $i-1$ (excluding of course, modules 1 and n, which are connected to only modules 2 and $n-1$ respectively). We can similarly define a *chain-connected system* of p processors. Given a set of n modules connected in a chain-like fashion and a chain-connected multiprocessor system of size $p < n$, the problem is to assign subchains of modules to processors so as to minimize the load on the most heavily loaded processor. The constraint is that the partitions of the chains have to be such that adjacent modules must be assigned to the same or to adjacent processors. We call

this the *contiguity* constraint. This problem is considered for both the parallel and pipelined processing. In *pipelined* processing, each processor works on a distinct frame of data. The maximum rate of processing is determined by the processor that takes the maximum time to perform its task — the *bottleneck* processor.

2. *Partition multiple chain-structured parallel or pipelined programs over single-host, multiple-satellite systems.*

   A *host-satellite* system consists of one large host computer connected with several satellite computers, each of which receives a stream of data from a real-time environment. These data streams may have different arrival rates and the individual satellites may have different computational capabilities. To each stream corresponds a chain-structured program, the modules of which are to be executed on the corresponding satellite or possibly in part by the more powerful host. The objective is to minimize the total execution and communication time on the bottleneck processor. We assume that each chain-structured program has fewer than n modules.

3. *Partition multiple arbitrarily-structured serial programs over single-host, multiple-satellite systems.*

4. *Partition single tree-structured parallel or pipelined programs over single-host, multiple identical satellites systems.*

Bokhari solved (1) by a *minimum bottleneck path* algorithm. To solve (2), (3) and (4) he used an algorithm which solves the *minimum sum-bottleneck path* problem

Table 3.1: Runtimes of the algorithms for the partitioning problem

Problem Structure	Processor Structure	Processing	Partial Constraint	Hansen-Lih's Complexity	Bokhari's Complexity
Single chain of n modules	single chain of p processors	pipelined/parallel	contiguous p sub-chains	$O(n^2p)$	$O(n^3p)$
p chains, totally n modules	single host, p dissimilar satellites	individual pipelined/parallel chains executing in parallel	contiguous two subchains of each processor	$O(n \log n)$	$O(\frac{n^2}{p} \log n)$
arbitrary p programs, totally n modules	single host, p dissimilar satellites	individual serial programs executing in parallel	none	$O(\frac{n^3}{p^2} \log \frac{n}{p})$	$O(\frac{n^4}{p^3})$
single tree of n modules	single host with $p < n$ identical satellites	pipelined/parallel	maximal subtrees on satellites	$O(n(\log n)^2)$	$O(n^2 \log n)$

on a graph. The latter consists in finding a path in a doubly-weighted graph which minimizes the maximum of the sum of the first type of weights and of the maximum of the second type of weights associated with its edges. This algorithm has a run time complexity of $O(v^2 \log e)$ for a graph of v vertices and e edges. Hansen and Lih [44] improved Bokhari's algorithms for the above four problems. For (1) they used dynamic programming and for (2) they used sorting and bisection search for the bottleneck value. They also noted that Bokhari's algorithms for problems (3) and (4) can be improved using the recent results of Gallo, et al. [38] and by implementing Dijkstra's algorithm [29] with a heap structure. Table 3.1 summarizes and compares their work with that of Bokhari's. Iqbal [48] developed an approximation technique which optimally solves the above four problems in time no worse than $O(np \log(C/\epsilon))$ where C is the cost of assigning all modules to one processor and ϵ is the desired accuracy. Another paper by the same author [49] discusses the partitioning problem. A recent paper by Nicol and O'Hallaron [63] also addresses the same problem.

The Mapping Problem

A closely related problem to MA is the following. Besides the communication graph G_C of the modules, we are also given the graph G_i describing the interconnection of the processors. The maximal number of hops between two processors in G_i to which two adjacent modules in G_C are assigned is called the *dilation* of that assignment. clearly it would be desirable to find an assignment which minimizes this dilation. Chugthai [26] addressed an extreme form of this problem where he

considered G_C to be a complete binary tree and G_i to be an 8-nn array[3]. Now, by keeping the dilation below a specified value, we can keep the communication delay between any two adjacent tasks in G_C low. An assignment is said to be *acceptable* if its dilation is less than or equal to the specified value. Characterizations and the use of acceptable assignments for given G_i and G_C were discussed by Shin and Chen [75].

Suppose the same setup as in the above paragraph, except that we have a parallel program now. Further suppose that the number of modules in G_C is equal to the number of processors in G_i and we have to assign exactly one module per processor. The *mapping problem* is to find an assignment which maximizes the number of edges of G_C being mapped onto the edges of G_i by the assignment. This problem is equivalent to the graph isomorphism problem [18] and is hence NP-complete. Bokhari [17] found a heuristic which finds a good solution rather than an exact one. Berger and Bokhari ([7], [8]) considered a variant of the mapping problem where it is no longer required to map exactly one module onto one processor.

Summary

While this chapter is by no means an exhaustive survey, it should give the reader some insight into the state of the art of the module allocation problem and should show that the problem has received and continues to receive considerable attention. Some other techniques and related research are reported in [6, 23, 43, 59, 62, 66]. A good overview of the module assignment problem can be found in [18].

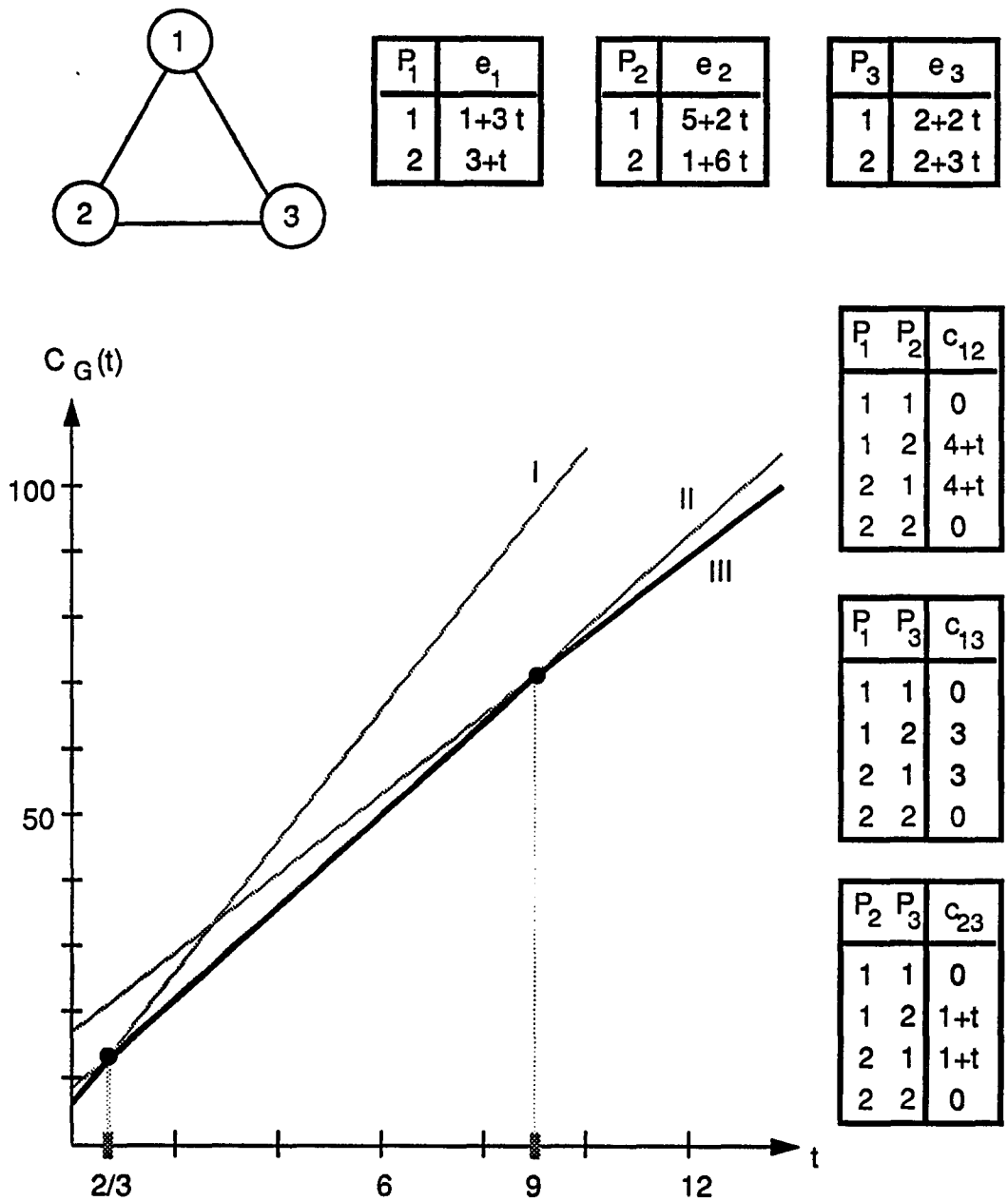
[3]An 8-nn array is one in which each processor is connected to to its 8 "nearest neighbors".

CHAPTER 4. PARAMETRIC MODULE ALLOCATION

Introduction

Parametric module allocation (PMA), is the problem of allocating modules to processors in a distributed system to minimize total costs when the costs are all functions of some parameter t, $0 \leq t < \infty$. In this chapter, we shall consider the case where the communication graph G is a *partial k-tree* (see the next section for definitions). We assume that all costs are linear functions of t; i.e., all costs are of the form $a + bt$ where $a, b \geq 0$. As mentioned in Chapter 3, we can interpret t as time and can view the changing costs as varying costs over time as the situation in the distributed system changes. We also observed there that the cost of the optimal assignment in G is a function of t under these assumptions. Let $C_G(t)$ be the plf describing the cost of the optimal assignment. Since MA is a minimization problem, $C_G(t)$ is the lower envelope of all the lines associated with assignments. Recall that $C_G(t)$ is a piecewise linear function and that the points at which the slope of $C_G(t)$ changes are called its breakpoints.

An instance of PMA is shown in Figure 4.1. There, we exhibit a communication graph along with the execution and communication cost functions of the modules. Among the several possible assignments, three contribute to the optimal solution as shown in the figure. $C_G(t)$ is the lower envelope of the associated lines. The



P_1	e_1
1	1+3 t
2	3+t

P_2	e_2
1	5+2 t
2	1+6 t

P_3	e_3
1	2+2 t
2	2+3 t

P_1	P_2	c_{12}
1	1	0
1	2	4+t
2	1	4+t
2	2	0

P_1	P_3	c_{13}
1	1	0
1	2	3
2	1	3
2	2	0

P_2	P_3	c_{23}
1	1	0
1	2	1+t
2	1	1+t
2	2	0

Figure 4.1: Communication graph, cost functions and the plf describing the optimal solution for a 3-module, 2-processor system.

breakpoints occur at $t = 2/3$ and $t = 9$. The assignment $P_1 = P_2 = P_3 = 2$, of cost $6 + 10t$ (line I) is optimal in the interval (0,2/3); $P_1 = P_2 = P_3 = 1$, of cost $8 + 7t$ (line II), is optimal in (2/3,9); $P_1 = 2, P_2 = P_3 = 1$, of cost $17 + 6t$ (line III), is optimal in $(9, \infty)$. Stone [80] showed that for the restricted version of parametric MA he was studying, $C_G(t)$ has $O(n)$ breakpoints. Fernández-Baca and Slutzki [35] showed that if G is a tree and all costs vary, then $C_G(t)$ has $O(n^{1+\log p})$ breakpoints and can be computed in $O(n^{1+\log p} \log n)$ time. Here we extend the results of [35] to show that if G is a partial k–tree, then $C_G(t)$ has $O(n^{1+(k+1)\log p})$ breakpoints and can be computed in $O(n^{1+(k+1)\log p} \log n)$ time. Thus, for reasonably tree-structured programs running on systems where the number of processors is fixed, the number of distinct optimum assignments that are encountered when processor load and transmission delays are varied, is polynomially-bounded.

Organization of the Chapter

We begin with some basic definitions, notations and results needed in this and the next chapter. Our first result is a $O(n)$ algorithm to find a $(k+1)$–vertex separator of an n–vertex k–tree. This result generalizes the $O(n)$ algorithm to find the centroid of a tree due to Kariv and Hakimi [52]. Subsequently, we present an algorithm that solves PMA on partial k–trees. We then discuss applications of PMA to parametric versions of the vertex cover, independent set, and 0–1 quadratic programming problems on partial k–trees, and conclude the chapter with a discussion of related open problems. Note that all logarithms in this chapter, and for that matter, in this dissertation, are to the base 2.

Preliminary Concepts and Results

Definition 4.1 (k–TREE) A graph G is a *k–tree* if and only if

(1) it is either a complete graph on k vertices, or

(2) it has a vertex v with exactly k neighbors forming a k–clique, such that $G - \{v\}$ is a k–tree.

In case (2) of the above definition, v is called a *k–leaf.*

Definition 4.2 (PARTIAL k-TREE) A *partial k–tree* is a subgraph of a k–tree.

An n–vertex k–tree has $k(n-k) + k(k-1)/2$ edges and $k(n-k)+1$ k–cliques. Trees and series-parallel graphs are partial k–trees with $k = 1$ and $k = 2$ respectively, *Halin* graphs are partial 3-trees, and almost trees with parameter r are partial $(r+1)$–trees. These and other results are surveyed in [13, 14]. For $k = 1, 2, 3$ it is possible to determine if an n–vertex graph G is a partial k–tree, and, if so, to construct an embedding k–tree in $O(n)$ time [2, 60]. Reed [69] has shown that, for any fixed k, it is possible to determine in $O(n \log n)$ time whether G is a partial k–tree.

Without any loss of generality, we can assume that the communication graph given to us is a k–tree for the following reason. Any instance of the assignment problem on a partial k–tree G can be converted into an equivalent k–tree problem by (1) finding an embedding k–tree H and (2) creating a new instance of the allocation problem with the same execution costs and where the communication cost between modules i and j, $(i,j) \in E(H)$, is the same as in the original problem if $(i,j) \in E(G)$, and equal to zero, regardless of the assignments for i and j, if $(i,j) \notin E(G)$. Clearly, the optima for both problems are the same. In subsequent discussions we therefore always assume that the input graph is a k–tree.

Let G be a k–tree. The definition of a k–tree gives a *reduction sequence* or *vertex elimination ordering* $Seq(G)$, by which, starting from G, we arrive at a final k–clique, called the *root clique*, by repeatedly removing k–leaves and their incident edges. Let K_v be the k–clique induced by the neighbors $\{u_1, \cdots, u_k\}$ of v at the time of its elimination. For each $u_i \in K_v$, $1 \leq i \leq k$, let ${K_v}^i$ be the k–clique induced by the vertices in $\Big(K_v - \{u_i\}\Big) \cup \{v\}$. For a k–clique K, let $I(K) = \{v\colon K_v = K\}$. Each vertex in $I(K)$ is said to be an *immediate descendant* of K. If K is a k–clique then $v \notin K$ is a *descendant* of K in a given reduction sequence if and only if when v was being removed, each vertex it was adjacent to was either a member of K or was a descendant of K. The subgraph induced by the descendants of K is called a *branch* on K and is denoted by $B(K)$. A *partial* k–tree is a subgraph of a k–tree. Given an arbitrary graph G, an *embedding* k–tree G' of G is a k–tree such that $V(G') = V(G)$ and $E(G) \subset E(G')$. The general problem of finding an embedding k–tree for an arbitrary graph is NP-hard for arbitrary k but can be solved in $O(n)$ time for $k \leq 3$ and in $O(n \log n)$ time for each fixed $k \geq 4$, see [60, 69].

Figure 4.2 illustrates the various definitions. The 2–tree G shown there has 23 2–cliques, one for each edge in the graph. Vertices of G are numbered according to a valid reduction sequence. According to this sequence we have, for example, $K_1 = \{2, 3\}$, $K_3 = \{11, 13\}$, and $K_{11} = \{12, 13\}$. The descendants of clique $K = \{9, 10\}$ are vertices 6, 7, and 8; thus, the branch on K is the subgraph induced by $\{6, 7, 8\}$. The only immediate descendant of clique $\{9, 10\}$ is vertex 8. Finally, note that $K_{11}^1 = \{11, 12\}$ and $K_{11}^2 = \{11, 13\}$.

The motivation in considering communication graphs which are partial k–trees is as follows. Modular programs whose communication graphs are tree-like include pro-

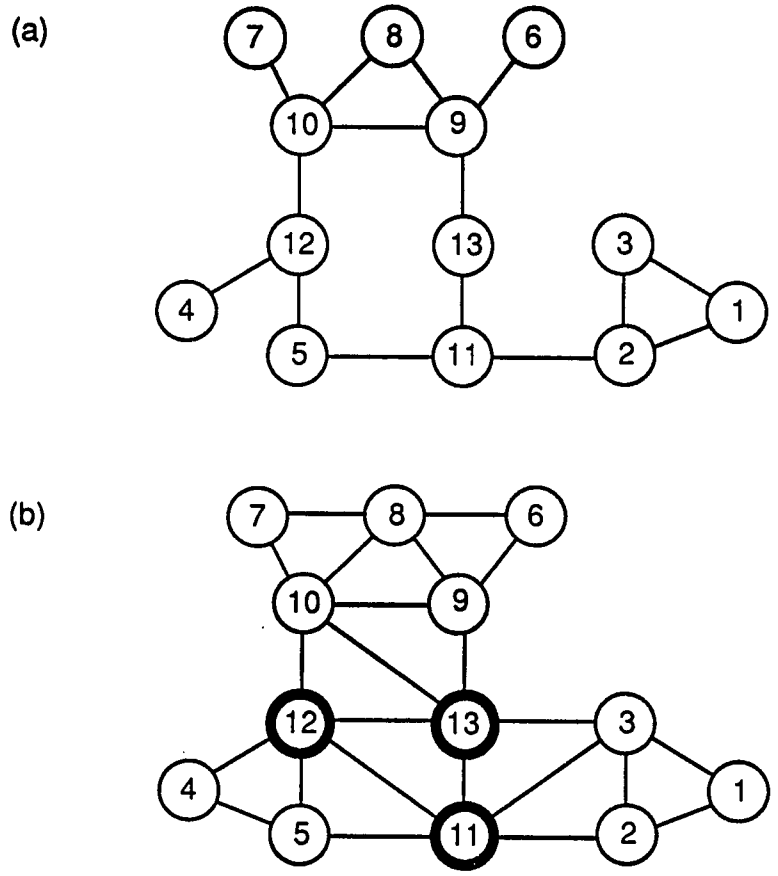


Figure 4.2: (a) A partial 2–tree and (b) An embedding 2–tree

grams written as a hierarchy of subroutines. It has been suggested (see [85]) that all large modular programs should deliberately be constructed with a tree-like structure for ease of understanding, maintenance and high reliability. Series-parallel graphs include program graphs in which modules lie in loops or in conditional branches [83]. Therefore working on partial k–trees is a natural extension of previous work to include more kinds of programs. We have the following results.

Lemma 4.1 *If K is a k–clique in G, then there are no edges (u, v) in G such that $u \in B(K) - K$ and $v \notin B(K) \cup K$.*

Proof. Suppose u is connected to one or more vertices $v \notin B(K) \cup K$. Since $u \in B(K)$ and $v \notin B(K)$, any such v must be eliminated before u. Choose v to be the *last* vertex in $Seq(G)$ such that $v \notin B(K) \cup K$ and $(u, v) \in E(G)$. Since, by assumption, v is not a descendant of K, when v is eliminated, at least one of its neighbors is not in $B(K) \cup K$. Let $w_1, \cdots, w_r$ be the neighbors of v not in $B(K) \cup K$. Since v is also a neighbor of u at the time of elimination and G is a k tree, u must be connected to each w_i. But $w_1, \cdots, w_r$ follow v in $Seq(G)$, and are not in $B(K) \cup K$, contradicting the definition of v. □

Lemma 4.2 *For all $v \in Seq(G)$ and for all $1 \leq i, j \leq k$, $i \neq j$*

(i) $B({K_v}^i)$ and $B({K_v}^j)$ have no vertices in common

(ii) there is no edge (u_i, u_j) such that $u_i \in B({K_v}^i)$ and $u_j \in B({K_v}^j)$.

Proof. Follows from Lemma 4.1. □

Lemma 4.3 $B(K) = \bigcup_{v \in I(K)} \left[\{v\} \cup \left(\bigcup_{i=1}^{k} B(K_v{}^i)\right)\right]$.

Proof. Follows from the definitions. □

In what follows, we shall assume that a plf is represented by the sequence of segments that make it up. For each segment, we store the associated optimum solution. With this representation, plf's can be easily manipulated. Let $b(g)$ denote the number of breakpoints of a plf g. The following two lemmas are taken from [35].

Lemma 4.4 *Let $f_1(t)$ and $f_2(t)$ be plf's. Then*

(i) $b(f_1 + f_2) \leq b(f_1) + b(f_2)$.

(ii) If f_1 and f_2 are both concave or both convex then

$b(min\{f_1, f_2\}) \leq b(f_1) + b(f_2) + 1$.

(iii) $f_1 + f_2$ *and* $min\{f_1, f_2\}$ *can be computed in time* $O(b(f_1) + b(f_2))$.

Lemma 4.5 *Let $n_1, \ldots, n_q$ be a partition of n; i.e. $\sum_{r=1}^{q} n_r = n$, such that $0 < n_r \leq n/2$ for $1 \leq r \leq q$. Then, for every real number $\alpha \geq 1$,*

$$\sum_{r=1}^{q} n_r^{\alpha} \leq \frac{n^{\alpha}}{2^{\alpha-1}}.$$

Finding a Separator in a k–tree

It is known that every n–vertex partial k–tree G has a set of vertices S of size $k+1$ such that no connected component of $G - S$ has more than $\lfloor (n-k)/2 \rfloor$ vertices [70]. Such an S is referred to as a *separator*. For example, in Figure 4.2(b) the set of vertices $\{11, 12, 13\}$ is a separator. We shall present an algorithm that finds a separator in a k–tree G given a reduction sequence $Seq(G)$. The algorithm uses a

copy H of the original k-tree G and associates an integer $S(K)$, called the *state* of K, which is initialized to zero, with every k-clique K of H. $S(K)$ is updated when a k-leaf v with $K = K_v$ is removed.

Algorithm 4.1 *(Separator of a k-tree)*

1. $H \leftarrow G$; $Seq(H) \leftarrow Seq(G)$; $S(K) \leftarrow 0$ for all k-cliques K in H.
2. Find the next vertex v in $Seq(H)$.
3. Let K_0 be the $(k+1)$-clique induced by K_v and v.
4. $S_0 \leftarrow S(K_v) + \sum_{i=1}^{k} S(K_v{}^i) + 1$.
5. **if** $S_0 \geq \lceil (n-k)/2 \rceil$ **then**
   Halt : K_0 is the separator.
   else $S(K_v) \leftarrow S_0$; $H \leftarrow H - \{v\}$; go to step 2.

It can be verified that the set $\{11, 12, 13\}$ is, indeed, the output returned by Algorithm 4.1 for the graph in Figure 4.2(b). We now prove the validity of this procedure.

Lemma 4.6 *The following property holds after every step of Algorithm 4.1.*
For every k-clique K in H, (i) $S(K) < \lceil (n-k)/2 \rceil$ *and (ii)* $S(K) = |\, B_e(K)\,|$, *where* $B_e(K) = \bigcup_{v \in I(K) - V(H)} \left[\{v\} \cup \left(\bigcup_{i=1}^{k} B(K_v{}^i) \right) \right]$.

(Intuitively, $B_e(K)$ is the portion of $B(K)$ that has been collapsed into K so far).

Proof. Clearly, (i) holds after every step. After step 1, $H = G$ and for all k-cliques K in H, $S(K) = 0$. Further $I(K) - V(H) = \emptyset$ since $I(K) \subset V(H) = V(G)$. Therefore $B_e(K) = \emptyset$ and hence (ii) is true after step 1. Notice that we now only need to consider the effect of step 5, since it is the only place in the algorithm where

H is altered and a state is updated. Assume (ii) holds immediately prior to step 5. To maintain (ii) after v is eliminated, step 5 must add to $S(K_v)$ the size of the portion of G containing v and the descendants of each K_v^i, $1 \leq i \leq k$; i.e., it must add

$$r = \mid \bigcup_{v \in I(K)} \left[\{v\} \cup \left(\bigcup_{i=1}^{k} B({K_v}^i) \right) \right] \mid .$$

Now, just before vertex v is eliminated from H, $I(K_v^i) \cap V(H) = \emptyset$, $1 \leq i \leq k$, since all descendants of K_v^i must be eliminated before v. Thus, $B_e(K_v^i) = B(K_v^i)$, for each i. Since (ii) holds, $S(K_v^i) = \mid B(K_v^i) \mid$ for $1 \leq i \leq k$. Thus,

$$r = 1 + \sum_{i=1}^{k} S(K_v^i),$$

which is precisely the value added to $S(K_v)$ by the algorithm. □

Note that each execution of step 5, except for the last one, reduces the number of k-cliques in H by k and increases $\sum_{K \in H} S(K)$ by one (indeed, this sum always equals the number of eliminated vertices). Thus, at some point, we must have $S_0 \geq \lceil (n-k)/2 \rceil$, which implies that Algorithm 4.1 terminates. By Lemmas 4.2 and 4.3, removal of K_0 splits G into $k+2$ subgraphs, namely the $B(K_v^i)$'s, $B_e(K_v)$, and

$$R = G - \left(\{v\} \cup K_v \cup B_e(K_v) \cup \left(\bigcup_{i=1}^{k} B({K_v}^i) \right) \right),$$

with no edges connecting any two of them. Now by Lemma 4.6,

$$\mid B(K_v^i) \mid = \; S(K_v^i) < \lceil (n-k)/2 \rceil \text{ for } 1 \leq i \leq k,$$

$$\mid B_e(K_v) \mid < \; \lceil (n-k)/2 \rceil,$$

and

$$S_0 = \mid \{v\} \cup B_e(K_0) \cup \left(\bigcup_{i=1}^{k} B({K_v}^i) \right) \mid .$$

Since $S_0 > \lceil (n-k)/2 \rceil$, the set

$$\{v\} \cup K_v \cup B_e(K_v) \cup \left(\bigcup_{i=1}^{k} B(K_v{}^i) \right)$$

has at least $k + \lceil (n-k)/2 \rceil$ vertices. Therefore, R has at most

$$n - (k + \lceil (n-k)/2 \rceil) = \lfloor (n-k)/2 \rfloor$$

vertices. Thus, no connected component of $G - K_0$ has more than $\lfloor (n-k)/2 \rfloor$ vertices. We conclude that K_0 is the desired separator. Since there are at most $(n-k-1)$ vertices to be removed, Algorithm 4.1 takes $O(n)$ time on an n–vertex k–tree. We thus have the following result.

Theorem 4.1 *Algorithm 4.1 correctly finds a $(k+1)$–vertex separator in $O(n)$ time.* □

Remark. The algorithm proves constructively that a k–tree has a separator of $k+1$ vertices. □

Parametric Module Allocation on Partial k–trees

We now present a recursive algorithm PARAM that constructs the function $C_G(t)$ which describes the cost of the optimum assignment in G as a function of t. We assume that the underlying graph G is a partial k–tree and that all costs are linear functions of a parameter t, $0 \leq t < \infty$.

Algorithm 4.2 *(Parametric Module Allocation on a Partial k–tree)*

Procedure PARAM(G, $C_G(t)$) ;
INPUT: G together with all its costs.
OUTPUT: $C_G(t)$.
begin { PARAM }
(1) Find an embedding k–tree H of G and $Seq(H)$;
(2) **for** all edges $(i,j) \in E(H) - E(G)$ **do**
for all u, v such that $1 \leq u, v \leq p$ **do**
$c_{ij}(u,v) := 0$;
(3) Use Algorithm 3.1 to find a separator $S_H = \{u_1, \cdots, u_l\}$ in H ;
(4) $Assign(S_H)$:= { all assignments to modules in S_H } ;
(5) **for** each $A = (P_{u_1}, \cdots, P_{u_l}) \in Assign(S_H)$ **do begin**
(6) $C_{S_H}(t;A) := \sum_{i=1}^{l} e_{u_i}(P_{u_i}) + \sum_{1 \leq i < j \leq l} c_{u_i u_j}(P_{u_i}, P_{u_j})$;
(7) **for** each component M of $H - S_H$ **do begin**
(8) **for** each vertex i in M **do begin**
(9) **for** $q = 1$ to p **do**
(10) $e_i(q) := e_i(q) + \sum \{c_{ji}(P_j, q) : (i,j) \in E(H)$ and $j \in S_H\}$;
end;
(11) PARAM(M, $C_M(t)$) ;
(12) Restore original weights in M
end;
(13) $C_{S_H}(t;A) := C_{S_H}(t;A) + \sum \{C_M(t) :$ $M$is a component of $H - S_H\}$
end;
(14) $C_G(t) := \min\{ C_{S_H}(t;A) : A \in Assign(S_H) \}$
end { PARAM }.

PARAM is a divide-and-conquer algorithm that is closely related to procedures presented in [35, 56]. After embedding the original problem into an equivalent prob-

lem on a k–tree (steps 1 and 2), it finds a separator S_H and then considers each possible assignment A to the variables associated with the separator. Step 6 records in $C_{S_H}(t;A)$ the cost of each such assignment when restricted to the subgraph induced by the vertices of S_H (this graph is complete, since H is a k–tree). In steps 7–10 each connected component M of $H - S_H$ is considered. The communication costs between the vertices in the separator and those of M are incorporated into the execution cost functions of vertices in M. This allows us to set up several independent subproblems, which are solved recursively in step 11. Finally, step 14 combines the various solutions to construct C_G. A full proof of correctness of Algorithm 4.2 can be obtained using the techniques developed in [35, 56]. Note that steps 6 and 10 manipulate linear functions while steps 13 and 14 manipulate plf's. We have the following result.

Theorem 4.2 *For any n–vertex partial k–tree G, $b(C_G(t))$ is $O(n^{1+(k+1)\log p})$ and $C_G(t)$ can be constructed in $O(n^{1+(k+1)\log p \log n})$ time.*

Proof. Let $b(n)$ denote $\max\{b(C_G(t))\}$ over all k–trees G with n vertices. Since S_H has at most $k+1$ vertices, there are at most p^{k+1} assignments in S_H. From step 14 of Algorithm 4.2 and from Lemma 4.4(ii), we have

$$b(C_G(t)) \leq \sum \{b(C_{S_H}(t;A)) : A \in Assign(S_H)\} + p^{k+1} - 1.$$

Thus, we shall concentrate on finding an upper bound on $b(C_{S_H}(t;A))$.

Observe that after step 6, $C_{S_H}(t;A)$ is simply a linear function of t since it is the sum of linear functions; thus, after that step, $b(C_{S_H}(t;A)) = 0$. After step 13

we have, using Lemma 4.4(i), that

$$b(C_{S_H}(t;A)) \leq \sum_{r=1}^{q} b(C_{M_r}(t)) \leq \sum_{r=1}^{q} b(n_r)$$

where $M_1, \cdots, M_q$ are the connected components of $H - S_H$ and $n_r = \mid M_r \mid$. Thus

$$b(C_G(t)) \leq p^{k+1} \sum_{r=1}^{q} b(n_r) + p^{k+1} - 1$$

and

$$b(n) \leq p^{k+1} \max\{ \sum_{r=1}^{q} b(n_r)\} + p^{k+1} - 1,$$

where the maximum is taken over all partitions $n_1, \cdots, n_q$ of $n - k$ such that $n_r \leq \lceil (n-k)/2 \rceil; r = 1, \cdots, q$. We have the following lemma.

Lemma 4.7

$$b(n) \leq a n^{1+(k+1)\log p} - 1$$

for a suitable constant a.

Proof. The lemma is certainly true for sufficiently small values of n. Assume that the lemma is true for all values less than n. For the induction step, we argue as follows. We have

$$\begin{aligned} b(n) &\leq p^{k+1} \max \left\{ \sum_{r=1}^{q} b(n_r) \right\} + p^{k+1} - 1 \\ &\leq p^{k+1} \max \left\{ \sum_{r=1}^{q} (a n_r^{1+(k+1)\log p} - 1) \right\} + p^{k+1} - 1, \\ &\qquad \text{by induction hypothesis.} \\ &= a p^{k+1} \max \left\{ \sum_{r=1}^{q} (n_r^{1+(k+1)\log p} - 1) \right\} - q p^{k+1} + p^{k+1} - 1 \end{aligned}$$

$$
\begin{aligned}
&\leq ap^{k+1} \max \left\{ \sum_{r=1}^{q} (n_r^{1+(k+1)\log p} - 1) \right\} - 1, \text{ since } q \geq 1 \\
&\leq ap^{k+1} \frac{(n-k-1)^{1+(k+1)\log p}}{2^{(k+1)\log p}} - 1, \quad \text{by Lemma 4.5} \\
&= a(n-k-1)^{1+(k+1)\log p} - 1 \\
&\leq an^{1+(k+1)\log p} - 1
\end{aligned}
$$

which proves the lemma, and, in turn, the first part of the theorem.

Next, let $T(n)$ denote the worst-case running time of the algorithm. Step 1 takes $O(n \log n)$ time [69]. Steps 2 and 3 take $O(kn)$ time each. Step 4 takes $O(p^{k+1})$ time. The *for* loop in step 5 is iterated at most p^{k+1} times. Step 6 takes $O(k^2)$ time. Step 10 takes $O(p \mid E(G) \mid)$ time over all iterations, which is $O(pkn)$, since $\mid E(G) \mid$ is $O(kn)$. The recursive calls take a total of $\sum_{r=1}^{q} T(n_r)$ time. Step 12 takes $O(kn)$ time. Step 13 is implemented using the techniques given in [35]. First, the components of $H - S_H$ are sorted according to their size. Assume, without loss of generality, that $n_1 \leq \ldots \leq n_q \leq \lfloor (n-k)/2 \rfloor$. This sorting process takes $O(q \log q)$ time, which is clearly dominated by $O(n \log n)$. The sum $\sum C_M(t)$ is computed as follows. If $M_1, \ldots, M_q$ are the components of $H - S_H$, then the sum is

$$[\ldots + [[C_{M_1}(t) + C_{M_2}(t)] + C_{M_3}(t)] + \ldots] + C_{M_q}(t).$$

By Theorem 4.2, $b(C_{M_r}) = O(n_r^{1+(k+1)\log p})$ for every r, $1 \leq r \leq q$. Hence by Lemma 4.4(*iii*), the time to compute the above sum is $O\left(\sum_{s=2}^{q} \sum_{r=1}^{s} n_r^{1+(k+1)\log p} \right)$. However,

$$O(\sum_{s=2}^{q} \sum_{r=1}^{s} n_r^{1+(k+1)\log p}) \leq \sum_{s=1}^{q} n_s \sum_{r=1}^{q} n_r^{(k+1)\log p}$$

$$
\begin{aligned}
&\leq \sum_{s=1}^{q} n_s \frac{(n-k-1)^{(k+1)\log p}}{2^{(k+1)\log p}-1}, \quad \text{by Lemma 4.5} \\
&= \frac{2}{p^{k+1}}(n-k-1)^{(k+1)\log p} \sum_{s=1}^{q} n_s \\
&= \frac{2}{p^{k+1}}(n-k-1)^{1+(k+1)\log p} \\
&\leq \frac{2}{p^{k+1}} n^{1+(k+1)\log p} \\
&\leq n^{1+(k+1)\log p}
\end{aligned}
$$

Thus, the sum in step 13 can be computed in time $O(n^{1+(k+1)\log p})$, which is proportional to the total number of breakpoints of the functions being added. Steps 6 to 13 are executed $O(p^{k+1})$ times and thus the total time spent on them is bounded by

$$p^{k+1}\Big(\sum_{r=1}^{q} \mathcal{T}(n_r) + O(n^{1+(k+1)\log p})\Big).$$

By step 13, Lemma 4.4, and the first part of the present theorem, $C_{S_H}(t; A)$ has $O(n^{1+(k+1)\log p})$ breakpoints. Thus by Lemma 4.4(*iii*) the time required to compute the minimum of the plf's in step 14 is $O(n^{1+(k+1)\log p})$.

Summarizing, the total time spent in constructing $C_G(t)$ satisfies the recurrence relation

$$\mathcal{T}(n) \leq p^{k+1} \max \left\{ \sum_{r=1}^{q} \mathcal{T}(n_r) \right\} + cn^{1+(k+1)\log p}$$

for some constant c and where the maximum is taken over all partitions $n_1, \cdots, n_q$ of $n-k$ such that $n_r \leq \lceil (n-k)/2 \rceil$; $r = 1, \cdots, q$. We have the following lemma.

Lemma 4.8

$$\mathcal{T}(n) \leq an^{1+(k+1)\log p} \log n$$

for a suitable constant $a \geq c$.

Proof. The lemma holds clearly for sufficiently small values of n. Assume that the lemma holds for all values less than n. In the inductive step, we argue as follows.

$$
\begin{aligned}
\mathcal{T}(n) &\leq p^{k+1} \max \left\{ \sum_{r=1}^{q} \mathcal{T}(n_r) \right\} + cn^{1+(k+1)\log p} \\
&\leq p^{k+1} \max \left\{ \sum_{r=1}^{q} a n_r^{1+(k+1)\log p} \log n_r \right\} + cn^{1+(k+1)\log p}, \\
&\qquad\qquad \text{by induction hypothesis.} \\
&\leq p^{k+1} \max \left\{ \sum_{r=1}^{q} a n_r^{1+(k+1)\log p} \log \lfloor (n-k)/2 \rfloor \right\} + cn^{1+(k+1)\log p} \\
&= a p^{k+1} \log \lfloor (n-k)/2 \rfloor \max \left\{ \sum_{r=1}^{q} n_r^{1+(k+1)\log p} \right\} + cn^{1+(k+1)\log p} \\
&\leq a p^{k+1} \log \left(\lfloor (n-k)/2 \rfloor \right) \frac{(n-k-1)^{1+(k+1)\log p}}{2^{(k+1)\log p}} + cn^{1+(k+1)\log p} \\
&= a(\log(n-k-1)-1)(n-k-1)^{1+(k+1)\log p} + cn^{1+(k+1)\log p} \\
&\leq a(\log n - 1) n^{1+(k+1)\log p} + cn^{1+(k+1)\log p} \\
&= a n^{1+(k+1)\log p} \log n - (a-c)\, n^{1+(k+1)\log p} \\
&\leq a n^{1+(k+1)\log p} \log n,
\end{aligned}
$$

where the last inequality holds if and only if $a \geq c$. This proves the lemma, which in turn proves the second part of the theorem. □

Further Results

In this section, we present some applications of MA. Specifically, we comment on three combinatorial optimization problems that are closely related to MA: the *minimum weight vertex cover problem* (VC), the *maximum weight independent set problem*(IS), and the *0–1 quadratic programming problem* (QP).

Let G be an undirected graph every vertex of which has a weight that is a linear function of a parameter t. The *weight* of a set S of vertices in G is the sum of the weights of the vertices in S. The weight of S is therefore a linear function of t.

The Vertex Cover Problem

A *vertex cover* of G is a set A of vertices such that for each edge (a, b) in G, at least one of a and b is in A. VC is to find a vertex cover of minimum weight. Let $W_{VC}(t)$ denote the weight of the optimum vertex cover as a function of t. VC can be formulated as MA with $p = 2$ (see [35, 56]). Thus when G is a k–tree, we conclude from Theorem 4.2 that the number of breakpoints of $W_{VC}(t)$ is $O(n^{k+2})$ and that $W_{VC}(t)$ can be computed in $O(n^{k+2} \log n)$ time.

The Independent Set Problem

An *independent set* of G is a set B of vertices such that no two vertices in B are adjacent. IS is to find an independent set of maximum weight. Let $W_{IS}(t)$ denote the weight of the optimum independent set as a function of t. It is well known that a set of vertices S is a maximum weight independent set of G if and only if $V(G) - S$ is a minimum weight vertex cover of G. Thus $W_{IS}(t) = W(t) - W_{VC}(t)$ where $W(t)$ is the weight of $V(G)$. $W(t)$ is a linear function of t, while $W_{VC}(t)$ is a concave plf

of t. Therefore, $W_{IS}(t)$ is a convex plf of t. Thus, the results obtained above for VC imply $W_{IS}(t)$ has $O(n^{k+2})$ breakpoints and can be computed in $O(n^{k+2}\log n)$ time.

The 0–1 Quadratic Programming Problem

QP is the problem of computing $\min_x f(x) = b^T x + x^T Q x$, where $x = (x_1, \cdots, x_n)$ and $Q = [q_{ij}]$ is an $n \times n$ symmetric matrix with all zeroes on its diagonal, subject to $x_i \in \{0, 1\}$ for $i = 1, \cdots, n$. Several applications of QP are presented in [40]. To every instance of QP we can associate a graph G where $V(G) = \{1, \cdots, n\}$ and $(i, j) \in E(G)$ if and only if $q_{ij} \neq 0$ [5]. QP can thus be reformulated as a problem of minimizing

$$f(x) = \sum_{i \in V(G)} b_i x_i + 2 \sum_{(i,j) \in E(G)} q_{ij} x_i x_j$$

subject to

$$x_i \in \{0, 1\}.$$

We can transform any instance I_{QP} of QP, with associated graph G, into an instance I_{MA} of MA with $p = 2$, communication graph G, and where costs are defined as follows. For each $i \in V(G)$, $e_i(1) = 0$ and $e_i(2) = b_i$; for each $(i, j) \in E(G)$, $c_{ij}(P_i, P_j) = 2q_{ij}$ if $P_i = P_j = 2$, $c_{ij}(P_i, P_j) = 0$ otherwise. It is now easy to check that the values of the optimum solutions to I_{QP} and I_{MA} are equal, and, furthermore, that $(P_1, \cdots, P_n)$ is an optimum solution to I_{MA} if and only if $(x_1, \cdots, x_n)$, with $x_i = P_i - 1$ for $i = 1, \cdots, n$, is an optimum solution to I_{QP}.

In the parametric version of QP, the b_i's and the q_{ij}'s are linear functions of t. $W_{QP}(t)$, which gives the cost of the optimum solution as a function of t, is a plf. The relationship between MA and QP, together with Theorem 4.2, imply that, when the

associated graph is a k–tree, the number of breakpoints of $W_{QP}(t)$ is $O(n^{k+2})$ and that this function can be computed in $O(n^{k+2} \log n)$ time.

Discussion

We note that the bound of Theorem 4.2 is not tight for 1–trees (i.e., trees), since in [35], a $O(n^{1+\log p})$ bound was proved. It is an open question whether Theorem 4.2 is tight for $k > 1$, or, indeed, whether the bounds in [35] are the best possible.

CHAPTER 5. CONSTRAINED AND BALANCED MODULE ALLOCATION

Introduction

In this chapter, we shall be concerned with two versions of the assignment problem. The first is *constrained module allocation* (CMA), where the objective is to minimize a certain measure of total system cost, subject to a resource constraint on one of the processors. The second is *balanced module allocation* (BMA), where the objective is to minimize the maximum processor load. We design exact dynamic programming algorithms for both problems, which lead to approximation schemes for the case where the communication graph is a partial k–tree. Faster algorithms are presented for trees with uniform communication costs.

In CMA, one of the processors, for convenience assumed to be processor 1 (Recall that the modules are numbered from 1 to n and processors from 1 to p), has a limited amount of a certain resource, which we shall refer to as "memory". The remaining processors have unlimited memory. Let $M \in Z_0^+$ (the set of non-negative integers) denote the memory capacity of processor 1, and for $i \in \{1,\ldots,n\}$, let $m_i \in Z_0^+$ denote the memory requirement of module i. An assignment X is *feasible* if the total amount of memory required by the modules assigned to processor 1 does not exceed that processor's available memory; i.e., if $\sum\{m_i : x_i = 1\} \leq M$. The objective is to

find a feasible assignment X that minimizes

$$C(X) = \sum_{i \in V(G)} e_i(x_i) + \sum_{(i,j) \in E(G)} c_{ij}(x_i, x_j). \tag{5.1}$$

Note that in the absence of the memory constraint, CMA becomes the standard module allocation problem, MA.

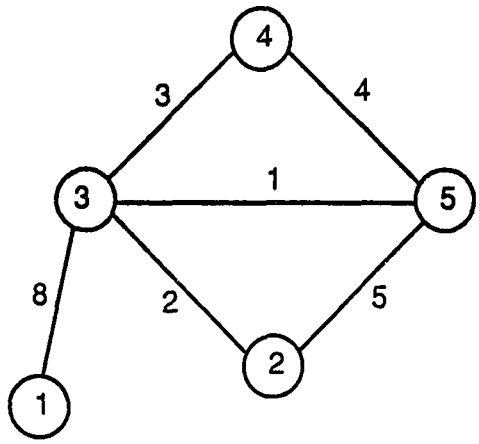
In BMA, the objective is to obtain assignments where processor load is balanced. The *total load* on processor r for a given assignment X, denoted $\mathrm{Ld}_r(X)$, is the total execution cost of the modules assigned to that processor plus the sum of the communication costs from that module to all modules; i.e.,

$$\mathrm{Ld}_r(X) = \sum_{i \in V(G)} \{e_i(x_i) : x_i = r\} + \sum_{(i,j) \in E(G)} \{c_{ij}(x_i, x_j) : x_i = r \text{ or } x_j = r\}$$

Note that if two communicating modules are assigned to different processors then their communication cost will contribute to the load on *both* processors. The *optimum balanced assignment problem* (BMA) is to find an assignment X that minimizes

$$\mathrm{Load}(X) = \max_{r \in \{1,\ldots,p\}} \mathrm{Ld}_r(X) \tag{5.2}$$

To clarify our definitions, we now illustrate CMA and BMA by means of the inter-module communication graph of Figure 5.1 (a), taken from Sinclair[76]. There are 5 modules and we consider a 3-processor system. The execution costs and memory requirements of the modules are given in Figure 5.1 (b). Communication costs are uniform and the r_{ij}'s are shown on the edges. Table 5.1, which was computed using the algorithms developed in this paper, shows the optimum solutions to instances of CMA and BMA derived from Figure 5.1. In the case of CMA, we have computed the solutions for distinct values of M. Note that for a sufficiently large M, the cost of the



(a) Intermodule communication graph with uniform costs

		Modules				
		1	2	3	4	5
Processors	1	40	25	20	30	15
	2	20	25	45	35	30
	3	25	20	25	30	40
Memory		20	30	25	15	35

(b) Module execution costs and their memory requirements

Figure 5.1: An Example Problem to illustrate CMA and BMA

optimum solution to CMA is the same as that of the corresponding instance of MA obtained by eliminating the memory constraint on processor 1. In our example, any value of $M \geq 105$ yields an optimum solution of cost 118, which matches the solution given by Sinclair in [76]. In the case of BMA, there are 2 optimum solutions, both of which are listed. Recall that the cost of the optimum assignment in this case is the maximum processor load. This explains the large difference between the values of the optimum solutions for this problem and CMA. Note also that BMA does a better job of distributing the load evenly than does CMA.

Table 5.1: CMA and BMA results for the example problem in Figure 5.1

Problem	Value of M	Assignment					Cost of the Assignment	Memory requirement on Processor 1
		x_1	x_2	x_3	x_4	x_5		
CMA	≥ 105	2	1	1	1	1	118	105
	100	2	3	1	1	1	120	75
	50	3	3	3	1	1	124	50
	≤ 25	3	3	3	3	2	140	0
BMA	N/A	3	1	3	2	1	56	N/A
	N/A	3	2	3	1	1	56	N/A

Organization of the chapter

To begin with, we develop exact, dynamic programming algorithms for both CMA and BMA with arbitrary costs. These have exponential running times in the worst case. However, these same algorithms are considerably faster when the underlying communication graph is a partial k-tree. In these cases, our algorithms run in pseudo-polynomial time and lead to FPTAS, which in polynomial time, compute solutions to within any desired level of accuracy. We develop faster exact and approximate algorithms for trees with uniform costs. We conclude the chapter by discussing further results and related open problems. The algorithmic results of this chapter are summarized in Table 5.2. Recall that C^* denotes the value of the optimum solution. Also, ϵ is a bound on the relative error of the solution obtained by the approximation schemes.

Nonserial Dynamic Programming

In [32], MA was shown to be a special case of *nonserial dynamic programming* [9]. This observation was the basis for an algorithm for MA in [32], based on the

Table 5.2: Summary of the runtimes of the algorithms for CMA and BMA

Problem	Problem Structure	Exact Algorithm	Approximation Scheme
CMA	k–trees with arbitrary costs	$O(np^{k+1}(C^*)^2)$	$O(n^3p^{k+1}(1/\epsilon^2 + \log C^*))$
	trees with uniform costs	$O(np(C^*)^2)$	$O(n^3p(1/\epsilon^2 + \log C^*))$
	almost trees with parameter r with arbitrary costs	$O(np^{\lceil r/2\rceil+2}(C^*)^2)$	$O(n^3p^{\lceil r/2\rceil+2}(1/\epsilon^2 + \log C^*))$
BMA	k–trees with arbitrary costs	$O(np^{k+1}(C^*)^{2p})$	$O(n^{2p+1}p^{k+1}(1/\epsilon^{2p} + \log C^*))$
	trees with uniform costs	$O(np(C^*)^{2p})$	$O(n^{2p+1}p(1/\epsilon^{2p} + \log C^*))$
	almost trees with parameter r with arbitrary costs	$O(np^{\lceil r/2\rceil+2}(C^*)^{2p})$	$O(n^{2p+1}p^{\lceil r/2\rceil+1}(1/\epsilon^{2p} + \log C^*))$

technique of *variable elimination.* We shall briefly describe this technique here. See [9] for more details.

In nonserial dynamic programming (NSDP), we are asked to minimize (or maximize) a function

$$f(y_1, \ldots, y_n) = \sum_{i \in T} f_i(Y^i), \tag{5.3}$$

where variable y_i, $1 \leq i \leq n$, takes on values from a finite set $\{1, \ldots, d_i\}$, T is an index set, and each *term* f_i is a function of a subset Y_i of the variables. We assume that the values of the terms for different values of their arguments are provided in tables. Associated with 5.3, there is an *interaction graph* G, whose vertices are integers i such that y_i is a variable, and such that there is an edge between vertices i and j if and only if y_i and y_j appear together as arguments of some term. In that

case, we say that y_i and y_j *interact.* Clearly thus, MA is a special case of NSDP — the communication graph of an instance of MA is simply the interaction graph of the objective function of MA.

A well-known technique to solve NSDP problems is *variable elimination.* Consider the problem of minimizing 5.3. Our goal is to replace this problem by another one with fewer variables such that the minimum for the new problem is the same as that of the original problem. Assume, without loss of generality, that $y_1, \ldots, y_l$ are the variables to be eliminated. Let

$$D = \{i \mid f_i \text{ is a function of at least one of } y_1, \ldots, y_l\}$$

and

$$Y_D = \left(\cup_{i \in D} Y_i\right) - \{y_1, \ldots, y_l\},$$

i.e., Y_D is the set of variables that appear together with one of $y_1, \ldots, y_l$ in some term. Define

$$f_a(y_1, \ldots, y_l, Y_D) = \sum_{i \in D} f_i(Y_i)$$

and let

$$f_b(Y_D) = \min_{y_1, \ldots, y_l} f_a(y_1, \ldots, y_l, Y_D).$$

Then it can be shown that

$$f'(y_{l+1}, \ldots, y_n) = f_b(Y_D) + \sum_{i \in T - D} f_i(Y_i) \tag{5.4}$$

has the same minimum as f [9]. The new term f_b captures all the information about the eliminated variables and terms that is needed for minimization. The interaction graph for the new problem can be obtained from that of the original problem by

removing the vertices corresponding to eliminated variables, along with their incident edges, and by completely connecting the vertices corresponding to variables in Y_D. Since there is such a close correspondence between variable elimination and the removal of vertices from the interaction graph, in what follows, we shall refer to variable and vertex (or module) elimination interchangeably. The elimination process is continued until either all variables are removed or the number of variables is sufficiently small enough to solve the problem directly by exhaustive enumeration. Once we have a choice of $y_{l+1}, \ldots, y_n$ that minimizes f', an assignment $(y_1, \ldots, y_n)$ that minimizes f can be obtained by standard *back pointer* techniques employed in many dynamic programming algorithms [55]. This technique calls for storing, when constructing f_b from f_a, the values of $y_1, \ldots, y_l$ that minimize f_a, together with each entry in the table for f_b.

Bokhari's algorithm for trees (1–trees) [15], Towsley's algorithm for series-parallel graphs (2–trees) [83], and Fernández-Baca's algorithm on partial k–trees [32] are applications of the variable elimination technique. In the next two sections, we shall extend the approach used in [32] to obtain enumerative algorithms for CMA and BMA. Recall from Chapter 2 that both of these problems are strongly NP-complete. As one would expect therefore, these algorithms are exponential in the worst case and, hence, are practical only for relatively small problem instances. As we will later see, their performance is significantly better on partial k–trees. Furthermore, on this same class of graphs, our algorithms lead to fully polynomial-time approximation schemes.

Variable Elimination and CMA

In what follows, the terms we shall be dealing with are *list functions*; i.e., functions that map each assignment of their variables to a list of pairs of nonnegative integers. One example is $\mathcal{F}_1(x_1)$, $x_1 \in \{1,2\}$, where $\mathcal{F}_1(1) = \langle(2,10),(4,8),(6,6)\rangle$ and $\mathcal{F}_1(2) = \langle(5,9),(10,2)\rangle$. Note that the lengths of lists need not be the same, even within a given function. Given two lists L_1, L_2 of pairs, $L_1 + L_2$ is the list of all pairs (c,b) where $c = c_1 + c_2$, $b = b_1 + b_2$, for some $(c_1,b_1) \in L_1$ and $(c_2,b_2) \in L_2$.

CMA is a special case of a problem that we shall call PAIR-NSDP. The latter involves list functions $\mathcal{F}$ that can be expressed as sums of one or more terms, which are themselves list functions; i.e.,

$$\mathcal{F}(x_1,\ldots,x_n) = \sum_{j\in\mathcal{J}} \mathcal{F}_j(X^j), \tag{5.5}$$

where variable x_i, $1 \le i \le n$, takes on values in the set $\{1,\ldots,p\}$, $\mathcal{J}$ is an index set, and, for each $j \in \mathcal{J}$, term $\mathcal{F}_j$ is a list function of $X^j \subseteq \{x_1,\ldots,x_n\}$. In addition to (5.5), we are given an integer M. The problem is to find, among all the lists associated with all possible assignments, the pair with the smallest first component among all pairs whose second component is at most M . If no such pair exists, the problem is *infeasible*. Formally, let opt $^{\text{pair}}$ be a function that, given a list L of pairs, returns $\min(\{\infty\} \cup \{c : (c,b) \in L, b \le M\})$. Then, PAIR-NSDP is the problem of computing

$$\min_{x_1,\ldots,x_n} \text{opt}^{\text{pair}} \mathcal{F}(x_1,\ldots,x_n). \tag{5.6}$$

Before proceeding, let us interpret CMA as an instance of PAIR-NSDP. For each

$i \in V(G)$, we define the list function $\mathcal{E}_i(x_i)$ as

$$\mathcal{E}_i(x_i) = \begin{cases} \langle (e_i(x_i), m_i) \rangle & \text{if } x_i = 1 \\ \langle (e_i(x_i), 0) \rangle & \text{otherwise.} \end{cases}$$

and, for each $(i,j) \in E(G)$, we define the list function $\mathcal{C}_{ij}(x_i, x_j)$, as

$$\mathcal{C}_{ij}(x_i, x_j) = \langle (c_{ij}(x_i, x_j), 0) \rangle.$$

We can now formulate CMA as an instance of PAIR-NSDP with objective function

$$\mathcal{F}(x_1, \ldots, x_n) = \sum_{i \in V(G)} \mathcal{E}_i(x_i) + \sum_{(i,j) \in E(G)} \mathcal{C}_{ij}(x_i, x_j).$$

For any assignment $X = (x_1, \ldots, x_n)$, $\mathcal{F}(X)$ is a single-element list $\langle (c, b) \rangle$, where c and b are the cost and the memory requirement of X, respectively. The interaction graph of this instance of PAIR-NSDP is the communication graph of the instance of CMA.

We will find it more convenient to work with a slight modification of PAIR-NSDP. Suppose L is a list of pairs and U is a positive integer. Let $\text{opt}_U^{\text{pair}}$ be given by

$$\text{opt}_U^{\text{pair}}(L) = \min(\{\infty\} \cup \{c : (c, b) \in L, c \leq U, b \leq M\}).$$

The problem we will actually be solving is to compute

$$\min_{x_1, \ldots, x_n} \text{opt}_U^{\text{pair}} \mathcal{F}(x_1, \ldots, x_n). \tag{5.7}$$

That is, the objective function is the same as that of (5.6), except that we only consider feasible solutions with cost at most U. The motivation for this formulation will become clear later. We note, however, that problems (5.6) and (5.7) are equivalent, provided U is sufficiently large.

To solve (5.7) we follow the variable elimination technique. We repeatedly reduce the size of the problem by removing variables one at a time until we are left with a function of a single variable. We then solve the associated problem by exhaustive enumeration. The basic step in this algorithm is procedure PAIR-VAR-ELIM, which reduces an instance of PAIR-NSDP to another instance with one fewer variable, but with the same optimum solution. We shall assume, without loss of generality, that the variable to be eliminated is x_1. PAIR-VAR-ELIM uses a function REDUCE that exploits the following simple dominance relation to bring down the size of its input list L: Given two pairs $p_1 = (c_1, b_1)$ and $p_2 = (c_2, b_2)$ where $c_1 = c_2$, p_1 *dominates* p_2 if $b_1 \leq b_2$.

REDUCE(L): Return a maximal sublist L' of L such that (i) for all $(c, b) \in L'$, $c \leq U$ and $b \leq M$, and (ii) for any two pairs $p_1, p_2 \in L'$, p_1 does not dominate p_2.

If L is sorted in lexicographic order, it is easy to implement REDUCE so that it runs in $O(|L|)$ time.

Procedure PAIR-VAR-ELIM

Input: An instance of PAIR-NSDP with objective function $\mathcal{F}(x_1, \ldots, x_n)$ and whose interaction graph is G, and a cost upper bound U.

Output: A new instance of PAIR-NSDP with the same optimum solution as the input instance, objective function $\mathcal{F}_1(x_2, \ldots, x_n)$, and a new interaction graph G_1.

Step 1. Compute the set $\tilde{X}$ of all variables that interact with x_1 and the index set $\mathcal{J}_1 = \{j \in \mathcal{J} : x_1 \in X^j\}$.

Step 2. Construct the tabular representation of the function $\mathcal{H}_1(\tilde{X})$, defined as follows.

$$\mathcal{H}_1(\tilde{X}) = \text{REDUCE}\left(\bigcup_{x_1 \in \{1,\ldots,p\}} \mathcal{H}(x_1, \tilde{X})\right) \tag{5.8}$$

where

$$\mathcal{H}(x_1, \tilde{X}) = \sum_{j \in \mathcal{J}_1} \mathcal{F}_j(X^j). \tag{5.9}$$

Step 3. Return the instance of PAIR-NSDP with objective function

$$\mathcal{F}_1(x_2, \ldots, x_n) = \mathcal{H}_1(\tilde{X}) + \sum_{j \in \mathcal{J} - \mathcal{J}_1} \mathcal{F}_j(X^j)$$

and interaction graph G_1, , which is obtained from G by removing x_1 and introducing edges between every pair of vertices that interacted with x_1 in G.

□

It is easy to check that G_1 is the interaction graph of the function $\mathcal{F}_1$ returned in Step 3. The following lemma proves the correctness of procedure PAIR-VAR-ELIM.

Lemma 5.1

$$\min_{x_1,\ldots,x_n} opt_U^{\text{pair}} \mathcal{F}(x_1, \ldots, x_n) = \min_{x_2,\ldots,x_n} opt_U^{\text{pair}} \mathcal{F}_1(x_2, \ldots, x_n).$$

Proof. Let $\tilde{X}$, $\mathcal{H}$, $\mathcal{H}_1$, and $\mathcal{F}_1$ be as defined in procedure PAIR-VAR-ELIM. Note that

$$\mathcal{F}(x_1, \ldots, x_n) = \mathcal{H}(x_1, \tilde{X}) + \sum_{j \in \mathcal{J} - \mathcal{J}_1} \mathcal{F}_j(X^j).$$

Therefore, we have

$$
\begin{aligned}
&\min_{x_1,\ldots,x_n} \mathrm{opt}_U^{\mathrm{pair}} \mathcal{F}(x_1,\ldots,x_n) \\
&\quad = \min_{x_1,\ldots,x_n} \mathrm{opt}_U^{\mathrm{pair}} \left(\mathcal{H}(x_1,\tilde{X}) + \sum_{j\in\mathcal{J}-\mathcal{J}_1} \mathcal{F}_j(X^j) \right) \\
&\quad = \min_{x_2,\ldots,x_n} \mathrm{opt}_U^{\mathrm{pair}} \left(\bigcup_{x_1\in\{1,\ldots,p\}} \left(\mathcal{H}(x_1,\tilde{X}) + \sum_{j\in\mathcal{J}-\mathcal{J}_1} \mathcal{F}_j(X^j) \right) \right) \\
&\quad = \min_{x_2,\ldots,x_n} \mathrm{opt}_U^{\mathrm{pair}} \left(\bigcup_{x_1\in\{1,\ldots,p\}} \left(\mathcal{H}(x_1,\tilde{X}) \right) + \sum_{j\in\mathcal{J}-\mathcal{J}_1} \mathcal{F}_j(X^j) \right) \\
&\quad = \min_{x_2,\ldots,x_n} \mathrm{opt}_U^{\mathrm{pair}} \left(\mathcal{H}_1(\tilde{X}) + \sum_{j\in\mathcal{J}-\mathcal{J}_1} \mathcal{F}_j(X^j) \right) \\
&\quad = \min_{x_2,\ldots,x_n} \mathrm{opt}_U^{\mathrm{pair}} \mathcal{F}_1(x_2,\ldots,x_n),
\end{aligned}
$$

as desired. In going from the second to the third line of this derivation, we are simply using the observation that, while in the third line, the minimum is over $(x_2,\ldots,x_n)$, all lists involving x_1 are explicitly considered by taking a union over all possible values of x_1. The definition of $\mathrm{opt}_U^{\mathrm{pair}}$ gives the equivalence of the two minima. In going from the third to the fourth line, we have used the fact that only $\mathcal{H}$ depends on x_1. The fifth line is obtained by observing that all numbers are nonnegative and that, therefore, REDUCE will only discard a pair in $\bigcup_{x_1\in\{1,\ldots,p\}} \mathcal{H}(x_1,\tilde{X})$ if it cannot possibly lead to a better solution than one of the pairs that remains. This proves the lemma. □

Clearly, step 2 is the crucial part of procedure PAIR-VAR-ELIM. For efficiency, we shall implement it as follows. First, we construct the tabular representation of $\mathcal{H}_0(x_1,\tilde{X}) = \mathrm{REDUCE}(\mathcal{H}(x_1,\tilde{X}))$ by carrying out the sum in equation (5.9) for each possible assignment to $x_1,\tilde{X}$. The sums in this equation are done in pairs, applying REDUCE after each step, in order to ensure that at all times all

lists will have $O(U)$ elements. Therefore, each list addition will require $O(U^2)$ time. Afterwards, we construct the tabular representation of $\mathcal{H}_1$ by noting that $\mathcal{H}_1(\tilde{X}) = \text{REDUCE}(\bigcup_{x_1} \mathcal{H}_0(x_1, \tilde{X}))$. We will not at present give an analysis of the running time of PAIR-VAR-ELIM since it depends critically on the structure of the interaction graph. In particular, this number is a function of the maximum number of variables with which an eliminated variable interacts. This number can be made very low for certain interaction graphs, such as trees and series-parallel graphs (where it equals 1 and 2, respectively), but will be high for others. In fact, for any given graph, certain elimination orderings can be far better than others. We return to this subject later again.

We can combine our solution to (5.7) with a search scheme SEARCH to obtain an algorithm for (5.6) [51]. We describe this scheme next. Assume, for simplicity, that $C^* > 0$.

Procedure SEARCH

(1) Set $U = 1$.

(2) Compute the cost C of the optimum solution to (5.7).

(3) **if** $C = \infty$ **then** $U := 2U$; **goto** step (2)
else Halt.

Clearly, at termination, $C = C^*$. Note that at all times $U < 2C^*$ and that the number of iterations is $O(\log C^*)$.

Remarks. Note that in step 3 of SEARCH, the value of U need not be doubled; in fact any factor greater than 1 (e.g., 1.25 or 1.5) will result in $O(\log C^*)$ iterations. Note also the practical utility of keeping U small. An excessive value for U has the potential of increasing the amount of time and space used by our algorithms, since, if

U is too large, the algorithms may generate pairs whose cost component is larger than C^*. Such pairs will never lead to an optimum solution and should thus be discarded. The importance of this observation is borne out in the experimental results reported in Chapter 6.

Variable Elimination and BMA

The ideas used for CMA can be modified to obtain an algorithm for BMA. BMA is a special case of a problem that we shall call TUPLE-NSDP. The objective function in TUPLE-NSDP has the same form as equation (5.5). As before, $\mathcal{F}$ and all the terms are list functions. In TUPLE-NSDP, however, list functions map each assignment of their variables to a list of p-tuples of nonnegative integers. As in PAIR-NSDP, we shall assume that list functions are implemented as tables of pointers to lists. The addition operator will represent tuple-list addition; i.e., given two lists L_1, L_2 of p-tuples, $L_1 + L_2$ is the list of all tuples $(t_1, \ldots, t_p)$ where, for $1 \leq j \leq p$, $t_j = r_j + s_j$, for some $(r_1, \ldots, r_p) \in L_1$ and $(s_1, \ldots, s_p) \in L_2$.

Let L be a list of p-tuples and let U be a positive integer. In keeping with the conventions of the previous section, let $\text{opt}_U^{\text{tuple}}(L)$ be defined as

$$\text{opt}_U^{\text{tuple}}(L) = \min(\{\infty\} \cup \{\text{weight}(T) : \text{weight}(T) \leq U, T \in L\}).$$

where, for a tuple $T = (t_1, \ldots, t_p)$, $\text{weight}(T) = \max_{1 \leq r \leq p} t_r$. TUPLE-NSDP is the problem of computing

$$\min_{x_1, \ldots, x_n} \text{opt}_U^{\text{tuple}} \mathcal{F}(x_1, \ldots, x_n). \tag{5.10}$$

BMA can be formulated as an instance of TUPLE-NSDP as follows. For each

$i \in V(G)$, we define $\mathcal{E}_i(x_i) = \langle(\mathcal{E}_{i1}(x_i), \ldots, \mathcal{E}_{ip}(x_i))\rangle$, where, for $1 \leq r \leq p$,

$$\mathcal{E}_{ir}(x_i) = \begin{cases} e_i(x_i) & \text{if } x_i = r \\ 0 & \text{otherwise.} \end{cases}$$

Note that this conforms with the fact that module i contributes only to the load of the processor it is assigned to.

For each $(i,j) \in E(G)$, let $\mathcal{C}_{ij}(x_i, x_j) = \langle(\mathcal{C}_{ij1}(x_i, x_j), \ldots, \mathcal{C}_{ijp}(x_i, x_j))\rangle$, where, for $1 \leq t \leq p$,

$$\mathcal{C}_{ijt}(x_i, x_j) = \begin{cases} 0 & \text{if } x_i \text{ and } x_j \neq t \\ c_{ij}(x_i, x_j) & \text{otherwise.} \end{cases}$$

Note that if either one of the modules i and j is assigned to t and the other is assigned to some other processor s, then the communication cost $c_{ij}(x_i, x_j)$ contributes to the load on both t and s. If both modules are assigned to t, then $c_{ij}(x_i, x_j)$ contributes only to the load on t as an interference cost, and if neither module is assigned to t, then neither contributes to the load on t.

BMA is thus an instance of TUPLE-NSDP with objective function

$$\mathcal{F}(x_1, \ldots, x_n) = \sum_{i \in V(G)} \mathcal{E}_i(x_i) + \sum_{(i,j) \in E(G)} \mathcal{C}_{ij}(x_i, x_j).$$

Note that, for any assignment $X = (x_1, \ldots, x_n)$, $\mathcal{F}(X)$ consists of a single tuple $(t_1, \ldots, t_p)$, where $t_j = \mathrm{Ld}_j(X)$.

Procedure TUPLE-VAR-ELIM is identical to PAIR-VAR-ELIM except that, instead of REDUCE, it uses operation TRIM, which is described below.

TRIM(L): Return a maximal sublist L' of L such that (i) for all tuples $T \in L'$, weight(T) $\leq U$ and (ii) for any two tuples $T_1, T_2 \in L'$, $T_1 \neq T_2$.

It is not hard to devise a $O(|L|)$-time implementation of TRIM, provided the tuples in L are in lexicographic order. The correctness of TUPLE-VAR-ELIM can be proved as in Lemma 5.1.

Module Allocation on Partial k–Trees

The algorithms of the previous sections have running times that are exponential in n for arbitrary graphs. This situation, however, is considerably better for *partial k–trees* [1], which are well-suited for the variable elimination approach. As observed in Chapter 4, we assume that the input graph G is a k–tree and that the elimination ordering $Seq(G)$ of the vertices in G is provided.

CMA on Partial k–trees

We shall first show that if the communication graph is a k–tree, procedure PAIR-VAR-ELIM can be implemented to run in $O(p^{k+1}U^2)$ time. We assume that Step 2 of this procedure is implemented as described; i.e. REDUCE is applied after each list addition. Then, for any assignment X^j, $|\mathcal{F}_j(X^j)| \leq U+1$ and each list addition takes $O(U^2)$ time. Next, we note that if G is a k–tree, when x_1 is eliminated, one vertex, at most k edges, and at most k k-cliques disappear. In other words, x_1 interacts with at most k other variables. Consequently, $|\mathcal{J}_1| \leq k+1$ and k pairwise sums need to be done to calculate $\mathcal{H}$. Further, each of the $k+1$ variables takes its value from $\{1,\ldots,p\}$. Thus the procedure takes $O(kp^{k+1}U^2)$ time, which is $O(p^{k+1}U^2)$ since k is a constant.

Since PAIR-VAR-ELIM is applied $O(n)$ times, the total running time of the algorithm is $O(np^{k+1}U^2)$ as claimed. To find C^*, we incorporate this algorithm into the

search procedure SEARCH. Recall that, at all times during the search $U \leq 2C^*$. Since the value of U is doubled in each iteration, the time to find C^* will be $O(np^{k+1}((2C^*)^2+(C^*)^2+(C^*/2)^2+(C^*/4)^2+\ldots+1))$, which is $O(np^{k+1}(C^*)^2)$.

As discussed above, for each $K \in G$ and each $X_K \in \{1,\ldots,p\}^k$, $\mathcal{H}_K(X_K)$ will have $O(C^*)$ elements. Since G has $k(n-k)+1$ k-cliques, the total memory requirement is $O(np^kC^*)$.

To construct the optimum assignment we can use standard back pointer techniques. Each pair generated in step 6 is the sum of $k+2$ pairs, k of which come from cliques $K_j(l)$, $j \in K(l)$ that are disappearing as a consequence of the elimination of vertex l. For each such newly-generated pair, we maintain a pointer to the k pairs associated with the disappearing cliques. With this structure, it is possible to reconstruct the optimum solution in $O(n)$ time, once the algorithm is done, by tracing back following the pointers.

BMA on Partial k–Trees

As before, assume variables are eliminated following a natural ordering. The behavior of TUPLE-VAR-ELIM is quite similar to that of PAIR-VAR-ELIM. For that reason we only point out the main differences between the two procedures.

The application of TRIM ensures that the size of the lists manipulated by BMA is $O(U^p)$. In analogy to CMA, we compute the sum in Step 2 of TUPLE-VAR-ELIM in pairs and applying TRIM after each list addition. This will ensure that at all times we will be manipulating lists of size $O(U^{2p})$. Using this observation we can show that the cost of the optimum assignment of cost at most U can be obtained in $O(np^{k+1}U^{2p})$ time and the cost of an optimum assignment can be found in $O(np^{k+1}(C^*)^{2p})$ time.

The total memory requirement for BMA is $O(np^k(C^*)^p)$ and an optimum solution can be constructed in $O(n)$ time using back pointers.

Module Allocation on Trees with Uniform Costs

If the communication graph is a tree, then the algorithms of the previous section imply that CMA will run in $O(np^2U^2)$ time, leading to a $O(np^2(C^*)^2)$ algorithm to determine the optimum assignment. Also, the respective time bounds for BMA can be seen to be $O(np^2U^{2p})$ and $O(np^2(C^*)^{2p})$. These bounds are true for the arbitrary communication costs case; however, as we will see these time bounds can be improved for the uniform costs case, using a modification of Billionet's approach [10], by a factor of p. Recall that in this case, co-resident modules incur a zero communication cost and communicating modules i and j, if assigned to different processors, will incur a communication cost r_{ij}.

In describing the algorithms refered to above, we depart from the convention we adopted so far in the description of the algorithms in the general cases. Here, the elimination process starts at a leaf. We eliminate a leaf variable and the information about it is stored in the $\mathcal{E}$ function of its neighbor; i.e., we store the eliminated information in the execution costs of the neighbor. At termination, all the information about the eliminated vertices is stored in the root of the tree and the optimum solution at the root is obtained by exhaustive enumeration.

CMA on Trees with Uniform Costs

The proposed algorithm, which we call UTCMA, is given below.

Algorithm UTCMA($\mathcal{M}, U$)

begin

1 **forall** $i \in G$ and $a \in \{1, \ldots, p\}$ **do**

2 $\quad \mathcal{E}_i(a) \leftarrow \langle (e_i(a), w(a)) \rangle$;

3 **while** $|V(G)| > 1$ **do begin**

4 $\quad$ Choose any leaf $i \in G$;

5 $\quad$ Let j be the neighbor of i in G;

6 $\quad \mathcal{Q} \leftarrow \bigcup_{a \in \{1,\ldots,p\}} \mathcal{E}_i(a)$;

7 $\quad \mathcal{Q} \leftarrow$ REDUCE($\mathcal{Q}$);

8 $\quad$ **forall** $b \in \{1, \ldots, p\}$ **do begin**

9 $\quad\quad \mathcal{E}_j(b) \leftarrow \mathcal{E}_j(b) \oplus (\mathcal{E}_i(b) \cup (\mathcal{Q} \oplus \langle (r_{ij}, 0) \rangle))$;

10 $\quad\quad \mathcal{E}_j(b) \leftarrow$ REDUCE($\mathcal{E}_j(b)$)

$\quad$ **end**;

11 $\quad G \leftarrow G - \{i\}$

end;

12 Let s be the remaining vertex of G;

13 **return** $\min_{a \in \{1,\ldots p\}} \mathrm{opt}_U^{\mathrm{pair}}(\mathcal{E}_s(a))$

end

The correctness of this approach follows from the correctness of Billionet's algorithm and arguments similar to those used for CMA. We analyze the runtime of UTCMA next.

Steps 1–2 of UTCMA take $O(np)$ time. The **while** loop beginning at step 3 is carried out $n-1$ times. The initialization of the $\mathcal{E}_i$'s and the application of REDUCE in step 10 ensure that, immediately before and after every execution of the **while** loop, $|\mathcal{E}_i(a)| \leq U+1$, for all $i \in V(G)$ and all $a \in \{1,\ldots,p\}$. Thus, steps 6 and 7 can be implemented in $O(pU)$ time. Since, after step 7, $|\mathcal{Q}| \leq U+1$, steps 8–10 take $O(pU^2)$ time. After the **while** loop is exited, we will have $|\mathcal{E}_S(a)| \leq U+1$ for all $a \in \{1,\ldots,p\}$ in lines 12 and 13. Thus, the cost of an optimum feasible assignment of cost not exceeding U can be computed in $O(npU^2)$ time.

We can combine this algorithm with procedure SEARCH to obtain the time bound of $O(np(C^*)^2)$ to obtain C^*.

It can also be verified that the space requirement of this algorithm is $O(nC^*)$. Back pointers can be used to reconstruct the optimum solution in $O(n)$ time.

BMA on Trees with Uniform Costs

The algorithm we propose to solve this problem is called UTBMA. It is given below.

Algorithm UTBMA($\mathcal{M}, U$)

begin

1 **forall** $i \in G$ and $a \in \{1,\ldots,p\}$ **do**

2 $\quad\mathcal{E}_i(a) \leftarrow \langle (T : T[a] = e_i(a) \text{ and } T[d] = 0,\ \forall\, d \neq a, 1 \leq d \leq p)\rangle$;

3 **while** $|V(G)| > 1$ **do begin**

4 $\quad$Choose any leaf $i \in G$;

5 $\quad$Let j be the neighbor of i in G;

6 $\mathcal{Q} \leftarrow \bigcup_{a\in\{1,\ldots,p\}} \big(\mathcal{E}_i(a)$
$\oplus\langle(T : T[a] = r_{ij} \text{ and } T[d] = 0\ \forall\ d \neq a, 1 \leq d \leq p)\rangle\big)$;

7 $\mathcal{Q} \leftarrow$ TRIM($\mathcal{Q}$);

8 **forall** $b \in \{1,\ldots,p\}$ **do begin**

9 $\mathcal{E}_j(b) \leftarrow \mathcal{E}_j(b)\oplus$
$(\mathcal{E}_i(b) \cup (\mathcal{Q} \oplus \langle T : T[b] = r_{ij} \text{ and } T[d] = 0\ \forall\ d \neq b, 1 \leq d \leq p)\rangle)$;

10 $\mathcal{E}_j(b) \leftarrow$ TRIM($\mathcal{E}_j(b)$)

end;

11 $G \leftarrow G - \{i\}$

end;

12 Let s be the remaining vertex of G;

13 **return** $\min_{a\in\{1,\ldots,p\}} \text{opt}_U^{\text{tuple}}(\mathcal{E}_s(a))$

end

Just as in the case of UTCMA, we can conclude that an optimum assignment can be computed in $O(np(C^*)^{2p})$ time where C^* is the cost of this assignment and that the space requirement of this algorithm is $O(n(C^*)^p)$. Back pointers can be used to reconstruct the optimum solution in $O(n)$ time.

Approximation Schemes

We shall now use the algorithms from the previous sections together with well-known scaling techniques [51, 55] to obtain FPTASs for CMA and for BMA with fixed p, when the communication graph is a k-tree or a tree with uniform costs. Our schemes rely on the following generic procedure.

APPROX-MA($\mathcal{M}, U_1, U_2, \epsilon$): $\mathcal{M}$ is an instance of CMA or BMA, whose optimum solution has value C^*, U_1, U_2, are positive integers, where $U_1 \leq C^*$ and $U_2 \geq U_1$, and $\epsilon > 0$ is the allowed relative error. Return the cost C' of a feasible solution to $\mathcal{M}$ such that $C' - C^* \leq \epsilon C^*$ and $C' \leq U_2$. If no such solution exists, return ∞.

We implement APPROX-MA as follows. Let $J = \max\{1, \lfloor U_1\epsilon/(2nk)\rfloor\}$. Given an instance $\mathcal{M}$, we construct another instance $\mathcal{M}'$ with the same graph, memory requirements, and memory constraint, but where the execution and communication costs are $e'_i = \lceil e_i/J \rceil$ for all $i \in V(G)$ and $c'_{ij} = \lceil c_{ij}/J \rceil$ for all $(i,j) \in E(G)$. APPROX-MA invokes a procedure XMA($\mathcal{M}, U$), which, given an instance $\mathcal{M}$ of the problem at hand (BMA or CMA on trees or k-trees), returns the best solution of cost not exceeding U. APPROX-MA returns $J \cdot C'$, where $C' = \text{XMA}(\mathcal{M}', U_2/J)$.

We need to verify that $J \cdot C'$ fulfills the necessary requirements. It is not hard to show that, for both CMA and BMA, if $C' < \infty$, $J \cdot C' - C^* \leq J(|V(G)| + |E(G)|)$. Thus, since $|V(G)| = n$ and, for a k-tree, $|E(G)| = k(n-k) + k(k-1)/2$, we have $J \cdot C' - C^* \leq \epsilon C^*$. Thus, $J \cdot C'$ is the desired ϵ-approximate solution. On the other hand, if $C' = \infty$, it must be because $U_2 < (1+\epsilon)C^*$. Therefore APPROX-MA produces the required output.

The running time of APPROX-MA depends on that of XMA. For CMA, the running time is

$$O(np^{k+1}(U_2/J)^2) = O(n^3 p^{k+1}(1/\epsilon)^2(U_2/U_1)^2).$$

For BMA, the time is

$$O(n(p(U_2/J)^p)^{k+1}) = O(n^{2p+1}p^{k+1}(1/\epsilon)^{2p}(U_2/U_1)^{2p}).$$

Now, using a technique due to Johnson and Niemi [51], we can apply APPROX-MA to obtain FPTAS's for CMA and for BMA for fixed p. The procedure consists of two phases. In the first, using the following procedure, we find values U_1, and U_2 such that $U_1 \leq C^* \leq U_2$ and $U_2/U_1 \leq 2$.

1. $U_1 \leftarrow 1$
2. $U_2 \leftarrow 4U_1$
3. $C' \leftarrow$ APPROX-MA$(\mathcal{M}, U_1, U_2, 1)$
4. **if** $C' < \infty$ **then return** $U_1 = \lceil C'/2 \rceil$ and $U_2 = C'$
5. **else** $U_1 \leftarrow 2U_1$; **goto** 2

The above algorithm iterates steps 2–5 $O(\log C^*)$ times, and when it calls APPROX-MA, it does so with $\epsilon = 1$ and $U_2 = 4U_1$. Also, at all times, $U_1 \leq C^*$, assuming $C^* > 0$. Furthermore, since $\epsilon = 1$, at termination $C' \leq 2C^*$, which implies that at this point U_1 and U_2 satisfy the desired conditions. More details can be found in [51]. The first phase takes $O(n^3 p^{k+1} \log C^*)$ time for CMA and $O(n^{2p+1} p^{k+1} \log C^*)$ time for BMA.

In the second phase, we use the above values of U_1 and U_2 and the desired error ratio ϵ to call APPROX-MA$(\mathcal{M}, U_1, U_2, \epsilon)$. By the definition of APPROX-MA, this will give us a solution of value at most $(1+\epsilon)C^*$. Since as a result of the first phase, $U_2/U_1 \leq 2$, this will take $O(n^3 p^{k+1}(1/\epsilon^2))$ time for CMA, while for BMA the bound is $O(n^{2p+1} p^{k+1}(1/\epsilon^{2p}))$.

The total running time for each algorithm is obtained by simply adding the work for the two phases. Similar analysis can be done for UTCMA and UTBMA. These running times are recorded in the last column of Table 5.2.

Discussion

Our variable elimination algorithms for CMA and BMA can be viewed as having been constructed from their counterpart MA algorithms (see [32]) by replacing integer-valued functions, integer addition, and the "min" operation, with list functions, list sum, and union, respectively. To keep list sizes small, REDUCE or TRIM were used. In essence, a variable-elimination algorithm for MA can be transformed into algorithms for CMA and BMA whose running times are slower by factors of U^2 and U^{2p}, respectively, where the factors account for the sizes of the lists that are manipulated. Table 5.2 lists all the runtimes. That same table also lists results for almost trees which are based on algorithms described in [32].

Several open problems remain. One is whether the large amount of memory required by our algorithms can be reduced by, say, using the techniques of [51]. Another question is whether there exist approximation schemes for the problem with multiple resource constraints. For instance, can the work of Frieze and Clark [37] on multi-dimensional knapsack problems be adapted to the multiple-resource constrained MA? Finally, even though our algorithms seem to perform better in practice (see Chapter 6) than the results in Table 5.2 imply, the magnitude of these time bounds limits their applicability. While the results of Chapter 2 pose significant theoretical limitations to the amount of improvement that could be expected, faster algorithms may exist for uniform cost problems on k–trees.

CHAPTER 6. IMPLEMENTATION AND EXPERIMENTAL RESULTS

We have implemented the exact variable elimination algorithms of Chapter 5. This chapter discusses the implementation details and the experimental results. We illustrate the entire working of the program using the CMA example of Chapter 1.

The Data Structure

We shall briefly discuss the main data structures used by the program. The fundamental building blocks of the data structure are *function descriptors* which represent terms. Each term $\mathcal{F}_j(X^j)$ is implemented using a dynamically-allocated data structure with two components: a list of variables in X^j and an array of size p^r, where r denotes the number of variables in X^j. Each entry of this array is a pointer to the list associated with a particular assignment to the variables of X^j. Depending on whether CMA or BMA is being considered, these lists contain either pairs or p-tuples. All lists are maintained sorted in lexicographic order, which simplifies the implementation of REDUCE and TRIM. Note that, since the pointers to lists are in an array, the beginning of any given list can be accessed in constant time.

To implement the variable elimination process, the objective function $\mathcal{F}$ is represented as follows. We maintain an array A of length n, whose i^{th} element $A[i]$

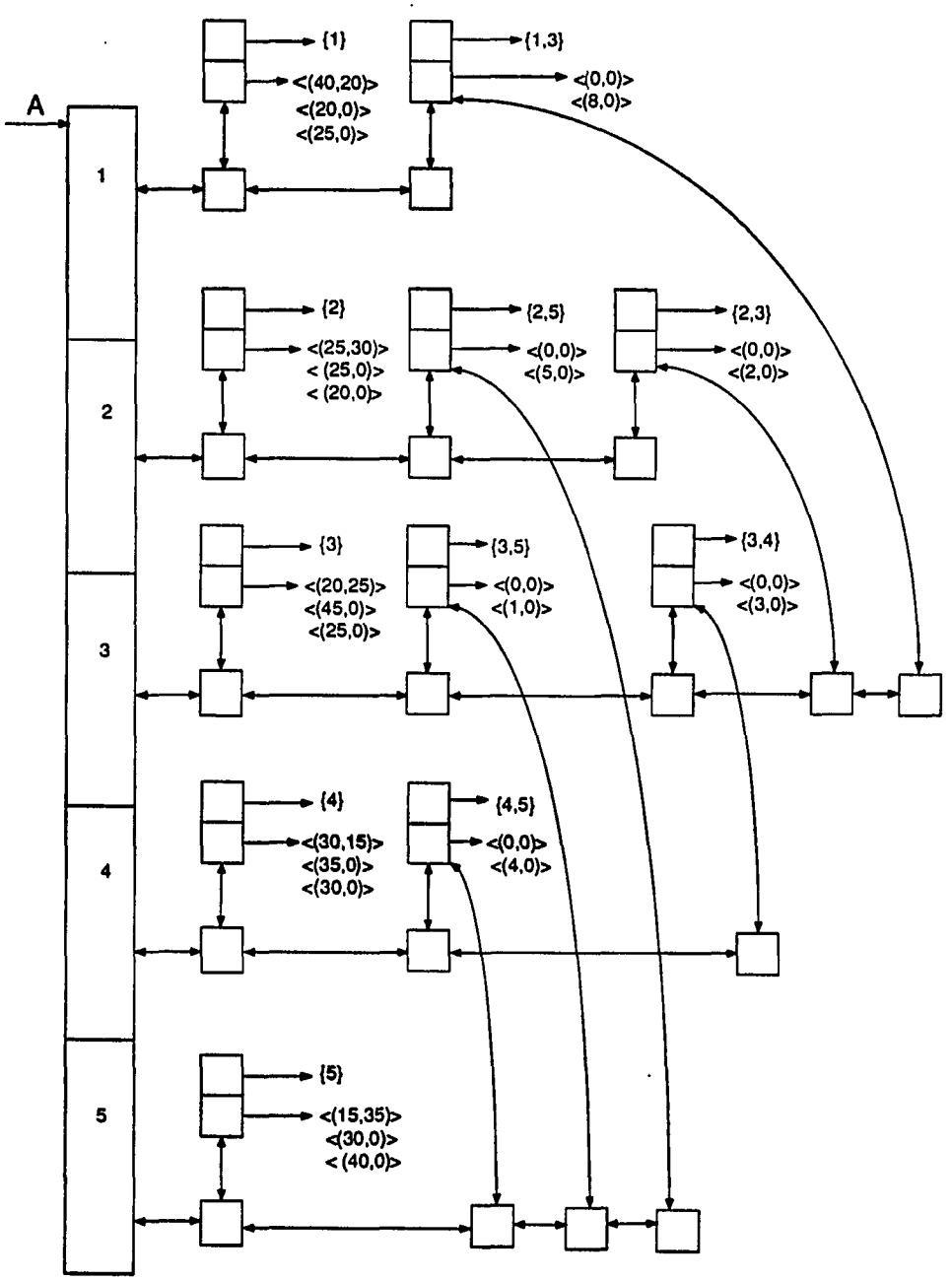


Figure 6.1: The initial CMA data structure for Figure 5.1(a)

is a pointer to a list of all terms $\mathcal{F}_j(X^j)$ such that $x_i \in X^j$. This list enables us to implement Step 1 of the elimination efficiently. The initial data structure for the CMA example of Chapter 5 is shown in Figure 6.1. Recall that $p = 3$ in this case. The first entry in the list for the variable x_i is the function descriptor for $\mathcal{E}_i(x_i)$. Each subsequent element is a pointer to the function descriptor for $C_{ij}(x_i, x_j)$ for every j such that modules i and j communicate. For every function descriptor, we show (1) its variable list and (2) the lists associated with each possible assignment. Initially, these are 1-element lists. Since $p = 3$, each $\mathcal{E}_i$ has 3 lists. For instance, for $\mathcal{E}_2(x_2)$, these lists are $\langle(25, 30)\rangle$, $\langle(25, 0)\rangle$, and $\langle(20, 0)\rangle$, indicating the execution cost and memory requirement on processors 1, 2 and 3 respectively. For the C_{ij}'s, we have adopted a different convention. Since there are 9 possible assignments to x_i, x_j, listing all of them could unnecessarily clutter the figure. Instead, we used the fact that costs are assumed to be uniform, and we show only 2 of the possible lists: one representing the case where modules i and j are co-resident (the (0,0) pair) and the other representing the case where i and j are assigned to different processors. Thus in Figure 6.1, the function $C_{35}(x_3, x_5)$ has pairs (0,0) reflecting the case when both the modules 3 and 5 are assigned to the same processor, and (1,0) indicating the case when they are put on different processors. Note that the descriptor for $C_{35}(x_3, x_5)$ is accessible via both 3 and 5.

The Variable Elimination Process

To illustrate the behavior of the variable elimination process, we shall show the changes in the objective function that result from the first two invocations of the procedure PAIR-VAR-ELIM. When x_1 is eliminated, Step 2 of PAIR-VAR-ELIM uses

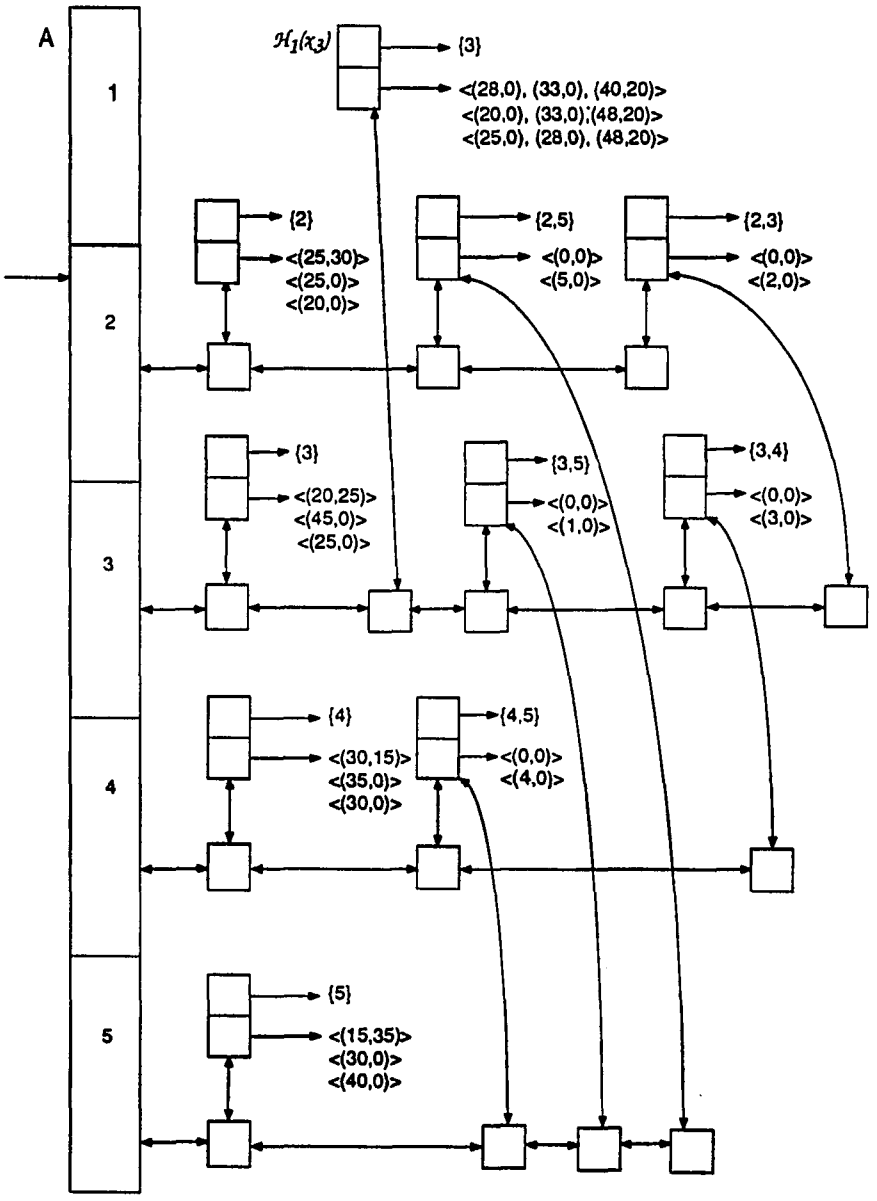


Figure 6.2: The data structure after vertex 1 is eliminated

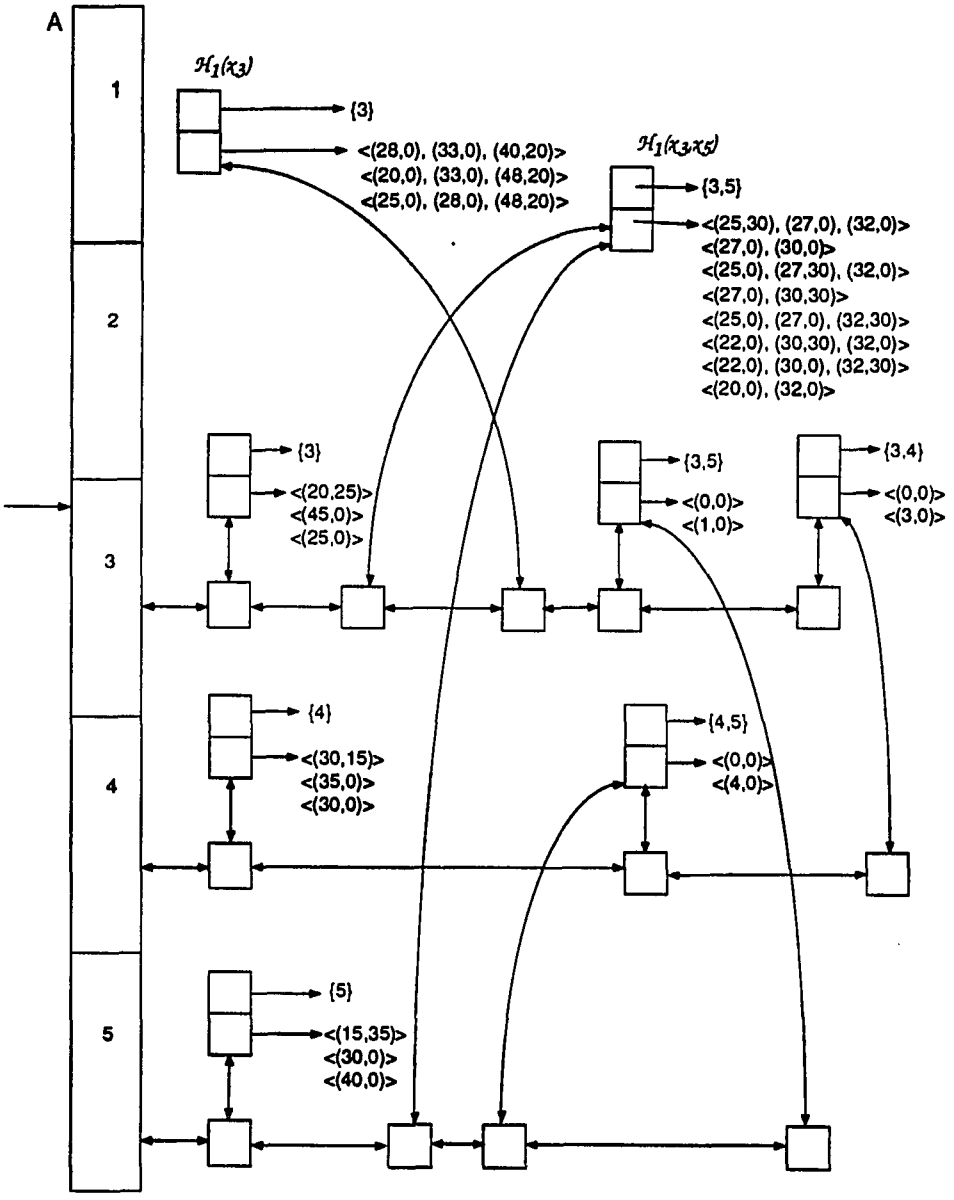


Figure 6.3: The data structure after vertex 2 is eliminated

list addition to combine $\mathcal{E}_1(x_1)$ and $\mathcal{C}_{13}(x_1, x_3)$ into a function $\mathcal{H}(x_1, x_3)$, which is then reduced to a term $\mathcal{H}_1(x_3)$. A pointer to the descriptor for this term is added to the list $A[3]$. The data structure representing the function resulting from Step 3 of PAIR-VAR-ELIM is shown in Figure 6.2. When x_2 is eliminated, Step 2 of PAIR-VAR-ELIM combines $\mathcal{E}_2(x_2), \mathcal{C}_{25}(x_2, x_5), \mathcal{C}_{23}(x_2, x_3)$ into a function $\mathcal{H}(x_2, x_3, x_5)$, which is then reduced to a function $\mathcal{H}_1(x_3, x_5)$, whose descriptor is made accessible via lists $A[3]$ and $A[5]$ (Figure 6.3). The elimination process continues until all the vertices are eliminated. At this stage, we will have exactly one list at hand, and since the list is sorted lexicographically, the cost component of the first pair in this list is the optimum cost C^*.

The Experiments

We have run our programs on several randomly-generated graphs. The performances of these programs are affected by a number of factors. Following Sinclair [76], we have attempted to examine their behavior as a function of (1) the number of processors, (2) the number of modules, (3) the density of the communication graph and (4) the ratio $e : c$, where e is an upper bound on all the processor execution costs and c is an upper bound on the communication costs. Furthermore, we have studied the effect of using different elimination orderings. For this purpose, our program has been designed so as to allow us to use any ordering we wish.

The Communication Graphs

To simplify the construction of test cases and the implementation of the algorithms, we considered uniform costs. Random connected graphs were generated using

the method suggested in [76]. First, the values of n and p were decided. Next, an edge between a pair of modules was created with a certain fixed probability. As might be expected, we encountered problems with physical memory limitations in trying to apply our algorithms to dense communication graphs. For this reason, we kept the edge probability low in order to guarantee that the number of edges, m, would be small; i.e., we were looking for graphs where $m \leq cn$ for some small constant c. As a result, most of the graphs we generated were partial 1- or 2-trees, with only a few being partial 3-trees. We also generated random trees using the algorithm given in [64]. Costs were randomly selected from uniform distributions. With the goal of considering systems that ranged from having relatively high to relatively low interprocessor communication, we tried $e : c$ ratios of 1:10, 1:2, 1:1, 5:1, and 10:1. Intuitively, one would expect that higher execution costs tend to force modules to be assigned to different processors.

Runtime Measurement

To get some idea of the running time of our algorithms, we have measured their performance in terms of the number of *list operations* that they do. A list operation is any step where an element is scanned or added to or deleted from a list. List operations are system-independent, and we believe, a fair measure of the running time. The number of list operations also gives us some idea of the amount of space used. Our experimental results are summarized in Figures 6.4 and 6.5.

In our simulations, we observed that in most cases, the time and space estimates derived in the previous sections are pessimistic, although the run time and space requirements go up drastically with increasing values of p, n or m. In most cases,

a good elimination ordering reduced both the run time and space requirements by great amounts. We observed that the run time is quite sensitive to the costs and the $e : c$ ratio, with the procedures being faster for communication-intensive systems than for those where execution costs are high. For example, when the ratio was 1:10, we were able to run our programs on instances with many more modules and processors than when the ratio was 1:2. A fact not evident from the plots is that the run time is also sensitive to the value of U. While, in theory, we can start with $U = 1$, this can result in an excessive number of useless iterations. Ideally, one would like the starting value of U to be as close to C^* as possible. An educated guess could be made about this value by studying the costs, however there is no way to know the "right" value of U to begin with. Our implementation uses the heuristic of choosing U to be the maximum of the costs. If an optimum is not reached with a given value of U, then U is increased by a factor of 1.5. The factor 1.5 was chosen instead of the value 2 suggested in Chapter 5 for several reasons. First, recall that any factor greater than 1 would guarantee $O(\log C^*)$ attempts before the true value of C^* is found. However, too large a factor may increase the run time and space unnecessarily (in fact, we found that, in most cases, the average list size was much smaller than U), while too low a factor may force the algorithm to attempt too many values of U. The value 1.5 was chosen as a good compromise, after testing factors ranging from 1.2 to 2.

Cost Scaling

We also tried to investigate the usefulness of scaling the costs as discussed in Chapter 5. Scaling has the effect of reducing the number of distinct costs and,

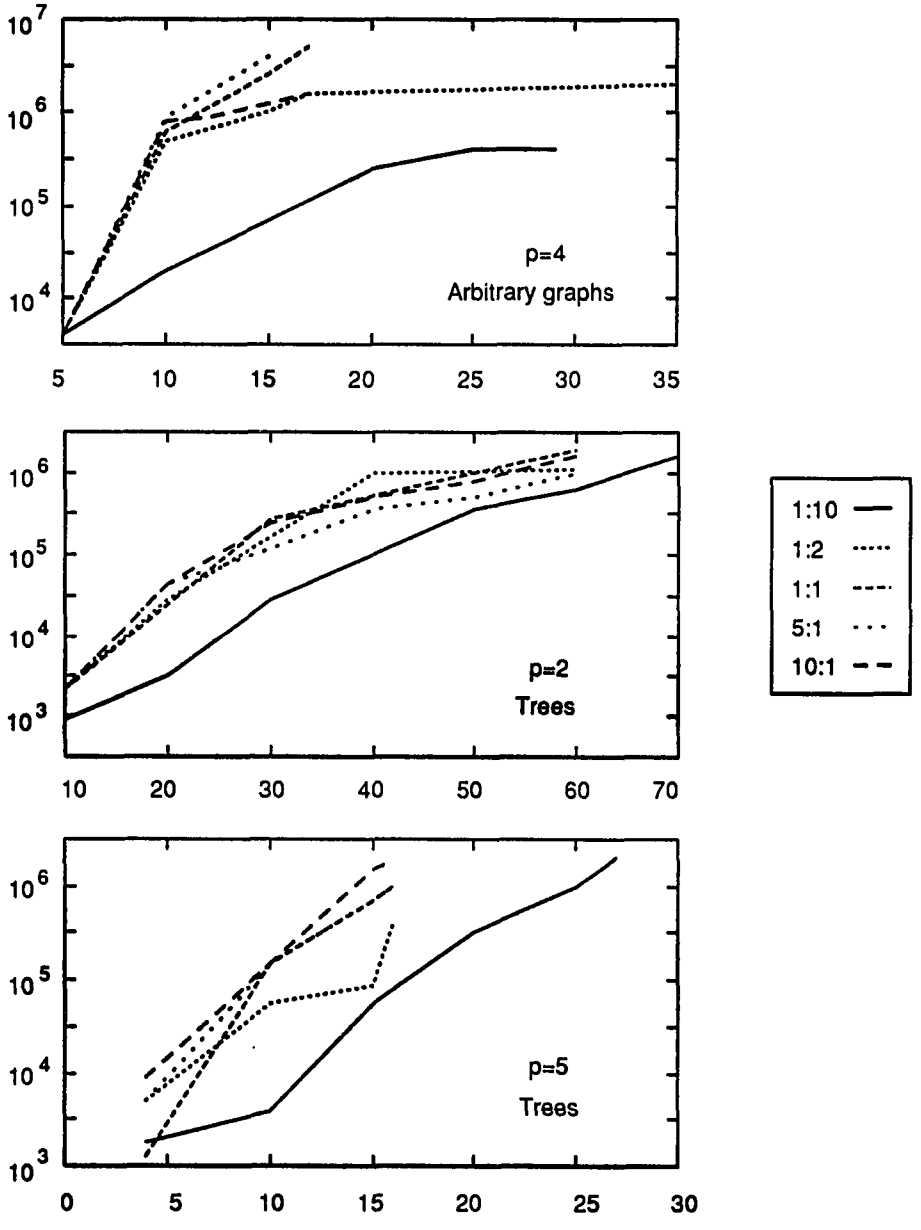


Figure 6.4: Some of the experimental results for CMA. The plots show the number of list operations versus number of modules for various $e : c$ ratios.

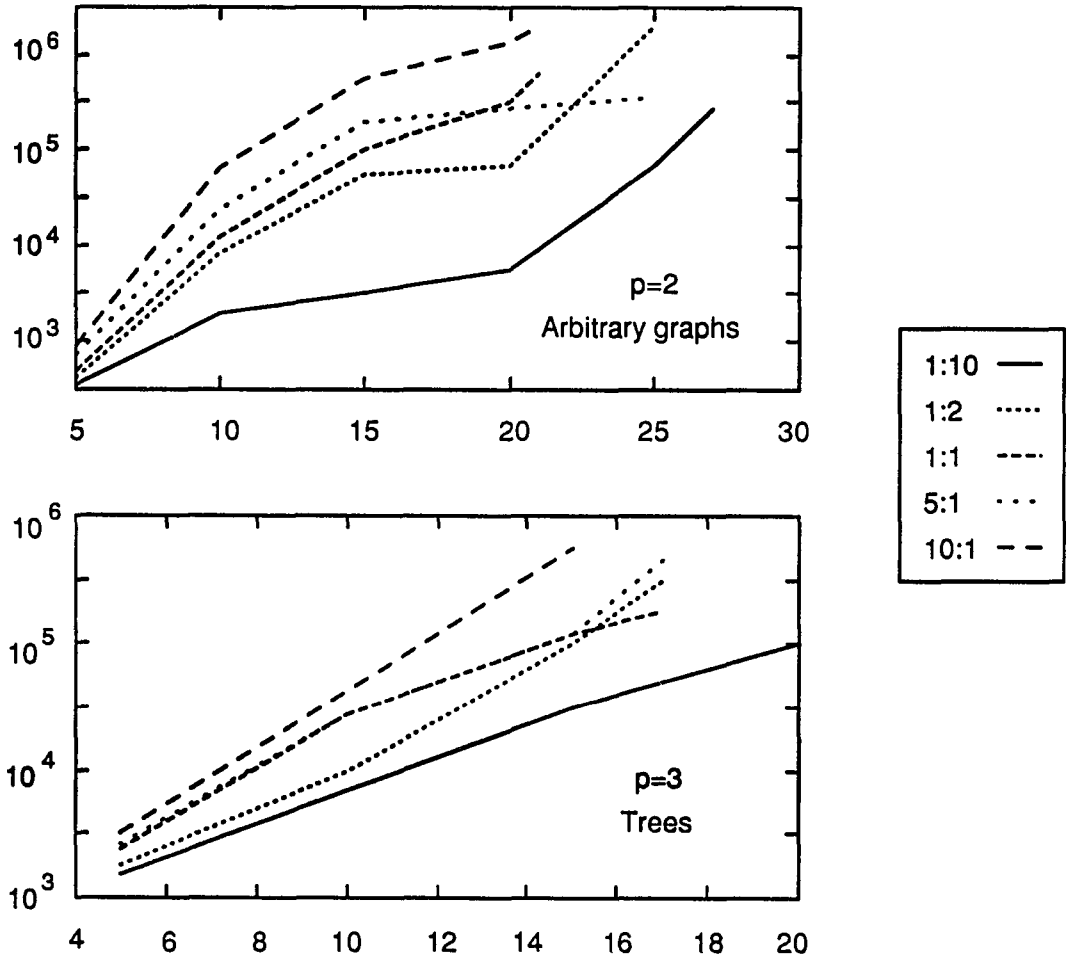


Figure 6.5: Some of the experimental results for BMA. The plots show the number of list operations versus number of modules for various $e : c$ ratios.

consequently, the list sizes. Thus scaling may have the practical advantage of allowing us to solve instances where our program takes too long or simply fails to run due to lack of space. To a limited extent, this observation seems to be borne out by our experiments. Unfortunately, choosing a good scaling factor is not simple. Too large a factor yields unacceptably inaccurate solutions, while too low a factor will not improve the efficiency of the program sufficiently. Our experiments seem to indicate that scaling, while appealing in theory, is not a practical tool for obtaining good approximation algorithms. This seems to be due partly to the fact that the scale factor J used in Section 5 tends to be very low. For instance, in a 2–tree with 5 vertices and $U_1 = 20$, J equals 1, which, in effect, implies that we are solving the original instance of the problem without scaling. In general, it appears that graphs must be enormous, and the ϵ's extremely large before scaling pays off as a practical algorithmic tool. Still, we cannot rule out the usefulness of scaling until a more thorough study is conducted.

Discussion

Of Lo's requirements for task assignment algorithms (see [57]), ours meet with the monotonicity and sensitivity requirements, i.e., with increasing number of processors, the cost (load) goes down and our algorithms are not sensitive to small changes in costs.

On a more practical note, we believe that it is possible to improve our program's memory management, in order to enable us to solve larger problem instances. We chose not to pursue this issue, since our focus was to determine the feasibility of implementing our algorithms, rather than how best to implement them. Note that our

programs do not return the optimal assignments which give rise to the optimal costs. The backpointer techniques of [55] could be used, but then this would be prohibitive from memory management point of view. It should be an interesting exercise to see if indeed there is an efficient way to construct the optimum assignments.

The actual *C* codes for implementing the dynamic programming algorithms and the codes for the random graph and tree generation are given in the appendices. We also provide the sample input and outputs for the programs.

CHAPTER 7. CONCLUSIONS AND FUTURE DIRECTIONS

In this dissertation, we focussed on some of the optimization problems that arise in distributed computing. In their most general form, these problems deal with the question of assigning the modules of a program to the processors of a distributed computer system in order to minimize the cost of running the program. This cost depends on the module execution times and the inter-processor communication costs. The module allocation problem (MA) arises in several situations and has several applications, as seen in Chapter 1.

We investigated a parametric problem, where all costs are allowed to vary over time (PMA); the module allocation problem when one of the processors has a limited memory (CMA) and finally, the allocation problem with the goal of balancing the loads on the processors (BMA).

MA has been studied extensively. We saw that even some seemingly simple cases are extremely difficult to solve; in fact, we investigated their intractable nature in Chapter 2. These negative results have been strengthened in this dissertation. We showed that, both CMA and BMA are strongly NP-complete and hence, unless P = NP, no FPTAS exist to solve them. On the positive side, however, we showed that FPTAS exist for partial k-trees. Faster algorithms have been developed for trees with uniform costs. Exact, but exponential time, algorithms using dynamic programming

have been developed and implemented to solve both CMA and BMA. We observed that these algorithms work well for small instances of the problem. Better memory management techniques could very well improve the running of these algorithms. To our knowledge, our work on CMA presents the first significant progress on the problem since Rao, et al. [68] published their results in 1979.

As for the future, it should be noted that even though MA has been studied extensively over the past decade or so, it and all its variants are far from being solved completely. As such, this field is still quite open and several unanswered questions remain. To begin with, the open problems and questions raised at the end of Chapters 4 and 5 could be investigated. In addition, one could consider several other directions and we list some of them.

Parametric Problems. The parametric problem we investigated in Chapter 4 assumed linear costs. The non-linear cost case would be a challenging problem to look at. Also, one could also investigate the parametric versions of CMA and BMA.

Dual Algorithms. Hochbaum and Shmoys [46] used dual algorithms to solve scheduling problems. The aim of dual algorithms is to find super-optimal, but infeasible solutions, and the performance is measured by the degree of infeasibility allowed. This seems a promising approach. Can it be used for any of the problems discussed in this dissertation ?

Lagrangian Relaxation. Billionet [11, 12] has recently investigated MA and a variant of CMA using this approach. This approach provides sub-optimal solutions. This approach is definitely worth looking into, especially since several techniques are available to solve 0–1 integer programming problems.

Dynamic Problems and Sensitivity Analysis. Module allocation problems tend to

occur in dynamic settings, where processor loads vary over time. To achieve optimum performance, modules may have to be reassigned from time to time. It would thus be interesting to design algorithms that identify the points in time at which these reassignments are necessary. A related problem is to investigate the sensitivity of the optimum assignment to changes in cost functions. This is especially important if the costs are only approximate values, which of course is closer to reality.

Additional Constraints. In a real system, in addition to the overall cost, other issues must be taken into account. Among the prominent ones are precedence relationships among the modules, the queuing delays at the processors and the reliability of the system. It is important to be able to integrate some or all of these constraints into one tractable model.

BIBLIOGRAPHY

[1] S. ARNBORG, D.G. CORNEIL AND A. PROSKUROWSKI, "Complexity of finding embeddings in a k–tree," *SIAM J. Alg. Discr. Methods*, Vol. 8, No. 2, pp. 277-284 (1987).

[2] S. ARNBORG AND A. PROSKUROWSKI, "Characterization and recognitions of partial 3–trees," *SIAM J. Alg. Discr. Methods*, Vol. 7, pp. 305-314 (1986).

[3] S. ARNBORG AND A. PROSKUROWSKI, "Linear time algorithms for NP-hard problems restricted to partial k–trees," *Discr. Appl. Math.*, Vol. 23, pp. 11-24 (1989).

[4] R.K. ARORA AND S.P. RANA, "On module assignment in two–processor distributed systems," *Info. Processing Letters*, Vol. 9, No. 3, pp. 113-117 (1979).

[5] F. BARAHONA, "A solvable case for quadratic 0–1 programming," *Discrete Appl. Math.*, Vol. 13, pp. 23-26 (1986).

[6] J. BAXTER AND J.H. PATEL, "The LAST Algorithm: A Heuristic-Based Static Task Allocation Algorithm," *Proceedings of the 1989 International Conference on Parallel Processing*, Vol. 2, pp. 217-222. The Pennsylvania State University Press, University Park and London.

[7] M.J. BERGER AND S.H. BOKHARI, "A partitioning strategy for PDE's across multiprocessors," *Proceedings of the 1985 International Conference on Parallel Processing*, pp. 166-170. IEEE Computer Society Press, Washington D.C.

[8] M.J. BERGER AND S.H. BOKHARI, "A partitioning stategy for non-uniform problems across multiprocessors," *IEEE Trans. Computers*, Vol. C-36, pp. 570-580 (1987).

[9] U. BERTELÈ AND F. BRIOSCHI, *Nonserial Dynamic Programming*. Academic Press, New York (1972).

[10] A. BILLIONNET, "Allocating tree structured programs in a distributed system with uniform communication costs," *Research Report CEDRIC* No. 90-14 (1989).

[11] A. BILLIONNET, M.C. COSTA AND A. SUTTER, "An efficient algorithm for a task allocation problem," Manuscript.

[12] A. BILLIONNET AND S. ELLOUMI, "Placement de taches dans un systeme distribue et dualite Lagrangienne," Manuscript.

[13] H.L. BODLAENDER, "Classes of graphs with bounded tree-width," *Tech. Report RUU-CS-86-22*, Dept. of Computer Science, University of Utrecht, The Netherlands (1986).

[14] H.L. BODLAENDER, "Some classes of graphs with bounded tree-width," *Bulletin of the European Association for Theoretical Computer Science (EATCS)*, Vol. 36, pp. 116-126 (1988).

[15] S.H. BOKHARI, "Dual processor scheduling with dynamic reassignment," *IEEE Trans. Software Eng.*, Vol. SE-5, No. 5, pp. 341-349 (1979).

[16] S.H. BOKHARI, "A shortest tree algorithm for optimal assignments across space and time in a distributed processor system," *IEEE Trans. Software Eng.*, Vol. SE-7, No. 6, pp. 583-589 (1981).

[17] S.H. BOKHARI, "On the mapping problem," *IEEE Trans. Computers*, Vol. C-30, pp. 207-214 (1981).

[18] S.H. BOKHARI, *Assignment Problems in Parallel and Distributed Computing.* Kluwer Academic Publishers, Boston (1987).

[19] S.H. BOKHARI, "Partitioning problems in parallel, pipelined, and distributed computing," *IEEE Trans. Computers*, Vol. C-37, pp. 48-57 (1988).

[20] G. BOLCH, et al., "A multiprocessor system for simulating data transmission systems (MUPSI)," *Microprocessing and Microprogramming*, Vol. 12, No. 5, pp. 267-277 (1983).

[21] P.J. CARTENSEN, *The complexity of some problems in parametric linear and combinatorial programming*, Ph.D. Thesis, University of Michigan, 1983.

[22] J. CHERIYAN, T. HAGERUP AND K. MEHLHORN, "Can a maximum flow be computed in $o(nm)$ time?," *Proceedings ICALP 1990*, Lecture notes in Computer Science, Vol. 443, pp. 235-248, Springer Verlag, New York.

[23] Y.C. CHOW AND W.H. KOHLER, "Models for dynamic load balancing in a heterogeneous multiple processor system," *IEEE Trans. Computers*, Vol. C-28, No. 5, pp. 354-361 (1979).

[24] W.W. CHU, et al., "Task Allocation in distributed data processing," *IEEE Computer*, Vol. 13, No. 11, pp. 57-69 (1980).

[25] W.W. CHU AND L.M. LAN, "Task allocation and precedence relations for distributed real-time systems," *IEEE Trans. Computers*, Vol. C-36, No. 6, pp. 667-679 (1987).

[26] M.A. CHUGTHAI, "Complete binary spanning trees of the eight nearest neighbor array," *IEEE Trans. Computers*, Vol. C-34, No. 6, pp. 547-549 (1985).

[27] T.H. CORMEN, C.E. LEISERSON, AND R.L. RIVEST, *Introduction to Algorithms*. McGraw-Hill Book Company, New York (1990).

[28] E. DALHAUS, D.S. JOHNSON, C.H. PAPADIMITRIOU, P. SEYMOUR, AND M. YANNAKAKIS, "The complexity of multiway cuts," *Proceedings of 24th Annual Symposium on Theory of Computing*, pp. 241-251 (1992), held in Victoria, British Columbia, Canada. Sponsored by ACM special interest group for Automata and Computability Theory.

[29] E.W. DIJKSTRA, "A note on two problems in connection with graphs," *Numerische Mathematik*, Vol. 1, pp. 269-271 (1959).

[30] K.W. DOTY, P.L. MCENTIRE, AND J.G. O'REILLY, "Task allocation in a distributed computer system," *Proceedings of the 1982 IEEE Infocom*, pp. 33-38. IEEE Computer Society Press, Piscataway, New Jersey.

[31] M.J. EISNER AND D.G. SEVERANCE, "Mathematical techniques for efficient record segmentation in large shared databases," *JACM*, Vol. 23, pp. 619-635 (1976).

[32] D. FERNÁNDEZ-BACA, "Allocating modules to processors in a distributed system," *IEEE Trans. Software Eng.*, Vol. 15, No. 11, pp. 1427-1436 (1989).

[33] D. FERNÁNDEZ-BACA AND A. MEDEPALLI, "Parametric module allocation on partial k-trees," Tech. Report 90-25, Dept. of Computer Science, Iowa State University (1990). To appear in *IEEE Trans. Computers.*

[34] D. FERNÁNDEZ-BACA AND A. MEDEPALLI, "Exact and approximate algorithms for assignment problems in distributed systems," Tech. Report 92-29, Dept. of Computer Science, Iowa State University (1992).

[35] D. FERNÁNDEZ-BACA AND G. SLUTZKI, "Solving parametric problems on trees," *J. Algorithms*, Vol. 10, pp. 381-402 (1989).

[36] D. FERNÁNDEZ-BACA AND S. SRINIVASAN, "Constructing the minimization diagram of a two-parameter problem," *Operations Research Letters*, Vol. 10, No. 2, pp. 87-93 (1991).

[37] A.M. FRIEZE AND M.R.B. CLARKE, "Approximation algorithms for the m-dimensional and 0–1 knapsack problem: Worst case and probabilistic analyses," *European J. of Oper. Res.*, Vol. 15, pp. 100-109 (1984).

[38] G. GALLO, M.D. GRIGORIADES, AND R.E. TARJAN, "A fast parametric maximum flow algorithm," *SIAM J. Comput.*, Vol. 18, No. 1, pp. 30-55 (1989).

[39] M.R. GAREY AND D.S. JOHNSON, *Computers and Intractability.* W.H. Freeman, New York (1979).

[40] V.P. GULATI, S.K. GUPTA, AND A.K. MITTAL, "The unconstrained quadratic bivalent programming problem," *European J. of Oper. Res.*, Vol. 15, pp. 121-125 (1981).

[41] Y. GUREVICH, L. STOCKMEYER, AND U. VISHKIN, "Solving NP-hard problems on graphs that are almost trees and an application to facility location problems," *JACM* Vol. 31, No. 3, pp. 459-473 (1984).

[42] D. GUSFIELD, "Parametric combinatorial computing and a problem of module distribution," *JACM*, Vol. 30, No. 3, pp. 551-563 (1983).

[43] V.B. GYLYS AND J.A. EDWARDS, "Optimal partitioning of workload for distributed systems," *Dig. Papers COMPCON*, pp. 353-357 (Fall 1976).

[44] P. HANSEN AND K. LIH, "Improved algorithms for partitioning problems in parallel, pipelined, and distributed computing," Manuscript.

[45] L.H. HARPER, et al., "Sorting X + Y," *Communications of the ACM*, Vol. 18, No. 6, pp. 347-349 (1975).

[46] D.S. HOCHBAUM AND D.B. SHMOYS, "Using dual approximation algorithms for scheduling problems: Theoretical and practical results," *JACM*, Vol. 34, No. 1, pp. 144-162 (1987).

[47] O.H. IBARRA AND C.E. KIM, "Fast approximation algorithms for the knapsack and sum of subset problems," *JACM*, Vol. 22, No. 4, pp. 463-468 (1975).

[48] M.A. IQBAL, " Approximate algorithms for partitioning and assignment problems," *ICASE Report 86-40, NASA Contractor Report 178130* (June 1986). Available from ICASE, NASA Langley Research Center, Hampton, Virginia 23665.

[49] M.A. IQBAL, "Efficient algorithms for partitioning problems," *Proceedings of the 1990 International Conference on Parallel Processing*, Vol. 3, pp. 123-127. The Pennsylvania State University Press, University Park and London.

[50] M.A. IQBAL, J.H. SALTZ, AND S.H. BOKHARI, "A comparitive analysis of static and dynamic load balancing strategies," *Proceedings of the 1986 International Conference on Parallel Processing*, pp. 1040-1047, IEEE Computer Society Press, Washington D.C.

[51] D.S. JOHNSON AND K.A. NIEMI, "On knapsacks, partitions, and a new dynamic programming technique for trees," *Math. Oper. Res.*, Vol. 8, No. 1, pp. 1-14 (1983).

[52] O. KARIV AND S.L. HAKIMI, "An algorithmic approach to network location problems I : The p-centers," *SIAM J. Appl. Math.* Vol. 37, pp. 513-538 (1979).

[53] F. KAUDEL, "Comments on allocating programs containing branches and loops within a multiple processor system," *IEEE Trans. Software Eng.*, Vol. 16, No. 4, pp. 471 (1990).

[54] J. LAGERGREN, "Efficient parallel algorithms for tree-decomposition and related problems," *Proceedings of 31st Annual Symposium on Foundations of Computer Science*, IEEE Computer Society Press, Los Alamito, California, pp. 173-182 (1990).

[55] E.L. LAWLER, "Fast approximation algorithms for knapsack problems," *Math. of Oper. Res.*, Vol. 4, No. 4, pp. 339-356 (1979).

[56] R. LIPTON AND R. TARJAN, "Applications of a planar separator theorem," *SIAM J. Comput.*, Vol. 9, No. 3, pp. 615-627 (1980).

[57] V.M. LO, "Heuristic algorithms for task assignments in distributed systems," *IEEE Trans. Computers*, Vol. 37, No. 11, pp. 1384-1397 (1988).

[58] U. MANBER, *Introduction to Algorithms: A Creative Approach.* Addison Wesley, New York (1989).

[59] R. MARCOGLIESE AND R. NOVARESE, "Module and data allocation methods in distributed systems," *Proceedings of the second International Conference on Distributed Computing Systems*, IEEE Computer Society Press, Los Alamito, California, pp. 50-59 (1981).

[60] J. MATOUŠEK AND R.THOMAS, "Algorithms finding tree-decompositions of graphs," *J. Algorithms*, Vol. 12, pp. 1-22 (1991).

[61] S. MORAN, "General approximation algorithms for some arithmetical combinatorial problems," *Theoretical Computer Science*, Vol. 14, pp. 289-303 (1981).

[62] L.M. NI AND K. HWANG, "Optimal load balancing strategies for a multiple processor operating system," *Proceedings of the IEEE Conference on Parallel Processing*, pp. 352-357 (1981). Held in Bellaire, Michigan.

[63] D.M. NICOL AND D.R. O'HALLARON, "Improved algorithms for mapping pipelined and parallel computations," *IEEE Trans. on Computers*, Vol. 40, No. 3, pp. 295-306 (1991).

[64] E.M. PALMER, *Graphical Evolution.* Wiley, New York (1985).

[65] C.H. PAPADIMITROU AND K. STEIGLITZ, *Combinatorial Optimization.* Prentice-Hall, Englewood Cliffs, New Jersey (1982).

[66] C.C. PRICE AND S. KRISHNAPRASAD, "Software allocation models for distributed computing systems," in *Proceedings of the fourth International Conference on Distributed Computing Systems*, IEEE Computer Society Press, Silver Spring, Maryland, pp. 40-48 (1984).

[67] C.C. PRICE AND U.W. POOCH, "Search techniques for a non-linear multiprocessor scheduling problem," *Naval Research Logistics Quarterly*, Vol. 29, No. 2, pp. 213-233 (1982).

[68] G.S. RAO, H.S. STONE, AND T.C. HU, "Assignment of tasks in a distributed processor system with limited memory," *IEEE Trans. Computers*, Vol. C-28, No. 4, pp. 291-299 (1979).

[69] B.A. REED, "Finding approximate separators and computing tree width quickly," *Proceedings of 24th Annual Symposium on Theory of Computing*, pp. 221-228 (1992), held in Victoria, British Columbia, Canada. Sponsored by ACM special interest group for Automata and Computability Theory.

[70] N. ROBERTSON AND P.D. SEYMOUR, "Graph minors II: Algorithmic aspects of treewidth," *J. Algorithms*, Vol. 7, pp. 309-322 (1986).

[71] G. SAGAR, A.K. SARJE, AND K.U. AHMED, "On module assignment in two-processor distributed systems: A modified Algorithm," *Info. Processing Letters*, Vol. 32, No. 3, pp. 151-153 (1989).

[72] S. SAHNI, "Approximate algorithms for the 0–1 knapsack problem," *JACM*, Vol. 22, No. 1, pp. 115-124 (1975).

[73] S. SAHNI, "Algorithms for scheduling independent tasks," *JACM*, Vol. 23, No. 1, pp. 116-127 (1976).

[74] J.H. SALTZ, *Parallel and adaptive algorithms for problems in scientific and medical computing*, Ph.D. Thesis, Duke University (1985).

[75] K.G. SHIN AND M. CHEN, "On the number of acceptable task assignments in distributed computing systems," *IEEE Trans. Computers*, Vol. 39, No. 1, pp. 99-110 (1990).

[76] J.B. SINCLAIR, "Efficient computation of optimal assignments for distributed tasks," *J. Parallel and Distributed Computing*, Vol. 4, No. 4, pp. 342-362 (1987).

[77] J.B. SINCLAIR, "Optimal assignments in broadcast networks," *IEEE Trans. Comput.*, Vol. 37, No. 5, pp. 521-531 (1988).

[78] S.R. STERNBERG, "Biomedical image processing," *IEEE Computer*, Vol. 16, No. 1, pp. 22-34 (1983).

[79] H. STONE, "Multiprocessor scheduling with the aid of network flow algorithms," *IEEE Trans. Software Eng.*, Vol. SE-3, No. 1, pp. 85-93 (1977).

[80] H. STONE, "Critical load factors in two-processor distributed systems," *IEEE Trans. Software Eng.*, Vol. SE-4, No. 3, pp. 254-258 (1978).

[81] H. Stone and S.H. Bokhari, "Control of distributed processes," *IEEE Computer*, Vol. 11, No. 7, pp. 97-106 (1978).

[82] A. TAMIR, Personal Communication (1990).

[83] D. TOWSLEY, "Allocating programs containing branches and loops within a multiple processor system," *IEEE Trans. Software Eng.*, Vol. SE-12, No. 10, pp. 1018-1024 (1986).

[84] D. TOWSLEY, "Correction to allocating programs containing branches and loops within a multiple processor system," *IEEE Trans. Software Eng.*, Vol. 16, No. 4, pp. 472 (1990).

[85] J. TURNER, "The structure of modular programs," *Communications of the ACM*, Vol. 23, No. 5, pp. 272-277 (1980).

[86] J. VAN LEEUWEN, *Graph Algorithms*, In J. van Leeuwen (ed.) *Handbook of Theoretical Computer Science*, MIT press, Cambridge, Massachusetts (1980).

APPENDIX A. THE C CODE FOR THE EXACT CMA ALGORITHM

```
/* This program implements the exact algorithm for CMA.
   The constants are defined assuming the input graph
   of the example in Chapter 5 */

#include <stdio.h>

#define n 5        /* Number of modules */
#define p 3        /* Number of processors */
#define q 9        /* q = p^2 : To read in the edge information */

int m, U, M;       /* Number of edges, cost upper bound
                      and maximum memory on processor 1 */

int listops;       /* This variable counts the number of list
                      operations performed by the program */

int assign[n];     /* This array stores the partial assignments */

FILE *fp;          /* A file pointer to the input file */
```

```
/* The various data-structures used in the program are defined next */

struct vector
 /* contains the actual cost or memory value */
{
  int  key;
  struct vector  *next;
}

struct node
 /* This defines the lists -- they are linked lists of vectors */
{
  struct vector  *hdr;
  struct node  *next;
}

struct vlist
  /* This defined the variable lists in a term */
{
  int vtx;
  struct termptr *ptr;
  struct vlist *next;
};

struct term
  /* This defined the actual term */
{
  struct vlist *vars;
  struct node  **termhd;
};

struct termptr
  /* This defines the actual pointers to terms and also
     connects the function descriptors of a single variable */
{
  struct termptr *prev;
  struct term *ptr;
  struct termptr *next;
};
```

```
struct vector *vechead()
  /* This defines a dummy header for vector lists */
{
  struct vector *head;
  head = (struct vector *) malloc(sizeof(struct vector));
  head->key = -1;
  head->next = NULL;
  return head;
}

struct node *initlist()
  /* This defines a list containing dummy header and tail nodes */
{
  struct node *start, *tail;

  start = (struct node *) malloc(sizeof(struct node));
  tail  = (struct node *) malloc(sizeof(struct node));

  tail->hdr = NULL ;
  tail->next = tail;

  start->hdr = NULL ;
  start->next = tail;

  listops++;
  return start;
}

struct vector *addvects(a,b)
     struct vector *a;
     struct vector *b;
  /* This adds two vectors */
{
  struct vector *sum, *x, *y, *z;

  sum = vechead();
  x = a->next;
  y = b->next;
  z = sum;
```

```
  while ( x != NULL)
    {
      struct vector *temp;
      temp = (struct vector*) malloc(sizeof(struct vector));
      temp->key = x->key + y->key;
      temp->next = NULL;
      z->next = temp;
      x = x->next;
      y = y->next;
      z = z->next;
    }
  return sum;
}

struct node *addelt(lp, vect)
     struct node *lp;
     struct vector *vect;
  /* This appends a vector to an existing list */
{
  struct node *temp;
  temp = (struct node*) malloc(sizeof(struct node));
  temp->hdr = vect;
  temp->next = lp->next;
  lp->next = temp;

  listops++;
  return lp;
}

int compare(a,b)
     struct vector *a, *b;
  /* This compares two vectors lexicographically */
{
  if (a != NULL && b != NULL)
    {
      if ( a->key < b->key )
return -1;    /* a < b */
      else
{
  if ( a->key > b->key )
```

```
    return 1;                    /* a > b */
  else  /* a->key = b->key */
    return compare(a->next, b->next);
}
    }
  else
    {
      if (a == NULL && b != NULL)
return -1;
      else
{
  if (a != NULL && b == NULL)
    return 1;
  else        /* both 'a' and 'b' are NULL */
    return 0;
}
    }
}

struct node *merge(x,y)
     struct node *x, *y;
  /* This merges two sorted lists into a single sorted list */
{
  int i;
  struct node *c, *z, *a, *b;

  listops++;

  a = x->next;
  b = y->next;
  c = initlist();
  z = c;

  while (a->hdr != NULL && b->hdr != NULL)
    {
        i = compare(a->hdr, b->hdr);
if (i == -1 || i == 0)
  {
    c->next = a;
    c = a;
```

```
    a = a->next;
  }
else /* i == 1 */
  {
    c->next = b;
    c = b;
    b = b->next;
  }
      }

  if (a->hdr == NULL && b->hdr != NULL)
    c->next = b;

  if (a->hdr != NULL && b->hdr == NULL)
    c->next = a;

  return z;
}

struct node *mergesort(a)
     struct node *a;
  /* This sorts a given list of vectors */
{
  struct node *p1, *q1, *r, *s;

  listops++;

  if ( a->next->next->hdr != NULL )
            /* Check if list has 0 or 1 element */
    {
      p1 = a;
      q1 = a->next->next;
      while (q1->hdr != NULL)
{
  a = a->next;
  q1 = q1->next->next;
}

      s = (struct node *) malloc(sizeof(struct node));
      s->hdr = NULL;
```

```
      s->next = a->next;

      r = (struct node *) malloc(sizeof(struct node));
      r->hdr = NULL;
      r->next = r;
      a->next = r;

      return merge(mergesort(p1), mergesort(s));
    }
  return a;
}

struct node *addlists(a,b)
     struct node *a, *b;
  /* This adds two lists and the addition is done pairwise */
{
  struct node *a1, *b1, *c;

  c = initlist();

  if (a->next->hdr == NULL || b->next->hdr == NULL)
    return c;
  else
    {
      for (a1 = a->next; a1->hdr != NULL; a1 = a1->next)
for (b1 = b->next; b1->hdr != NULL; b1 = b1->next)
  c = addelt(c, addvects(a1->hdr, b1->hdr));
      return mergesort(c);
    }
}

int dominates(a,b)           /* returns 0  if 'a' and 'b' are incomparable*/
     struct vector *a, *b;        /* returns  1 if 'b' dominates 'a' */
{
  if (a->next->key == b->next->key)  /* if costs are equal */
    {
      if (a->next->next->key <= b->next->next->key)
return 1;
    }
  else
```

```
    return 0;
}

struct node *reduce(a)
     struct node *a;
  /* This implements the procedure REDUCE */
{
  struct node *x, *y, *z;
  struct vector *v;

  listops++;

  if (a->next->hdr == NULL) /* empty list */
    return a;
  else
    {
      z = a;

      while (a->next->hdr != NULL)
{      /* get rid of the costly and heavy pairs */
  v = a->next->hdr->next;
  if  ( v->key > U || v->next->key > M )
    a->next = a->next->next;
  else
    a = a->next;
}

      x = z->next;
      y = z->next->next;

      while ( x->hdr != NULL && y->hdr != NULL)
{
  if ( dominates(x->hdr, y->hdr) == 1)
    {
      y = y->next;
      x->next = y;
    }
  else
    {
      x = x->next;
```

```
      y = y->next;
    }
}
      return z;
    }
}

struct vector *singlevec(a)
     int a;
  /* This defines a vector with key = a */
{
  struct vector *vp;

  vp = (struct vector *) malloc(sizeof(struct vector));

  vp->key = a;
  vp->next = NULL;

  return vp;
}

struct vector *vecpair(a,b)
     int a, b;
  /* This creates a pair of vectors with keys a and b */
{
  struct vector *vp1, *vp2, *vp3;

  vp1 = vechead();
  vp2 = singlevec(a);
  vp3 = singlevec(b);

  vp1->next = vp2;
  vp2->next = vp3;

  return vp1;
}
```

```
struct vlist *initvlist()
  /* This creates a variable list with a dummy header and tail */
{
  struct vlist *vl1, *vl2;
  vl1 = (struct vlist *) malloc(sizeof(struct vlist));
  vl2 = (struct vlist *) malloc(sizeof(struct vlist));

  vl1->vtx = -2;
  vl1->ptr = NULL;
  vl1->next = vl2;

  vl2->vtx = -3;
  vl2->ptr = NULL;
  vl2->next = vl2;

  return vl1;
}

printlist(a)
     struct node *a;
  /* This prints a list of vectors */
{
  int i = 1;
  struct vector *vect;

  if (a->next->hdr == NULL)
    printf("Empty list of pairs\n");
  else
    {
      a = a->next;
      while ( a != a->next)
{
  vect = a->hdr->next;
  printf("Vector%d\n",i++);
  while ( vect != NULL )
    {
      printf("%d\n",vect->key);
      vect = vect->next ;
    }
```

```
  a = a->next;
}
    }
}

printvlist(v)
     struct vlist *v;
  /* This prints a variable list */
{
  if ( v->next->next == v->next )
    printf("Empty list of vertices\n");
  else
    {
      printf("Vertices\n");
      while (v->next->next != v->next)
{
  printf("%d\n", v->next->vtx);
  v = v->next;
}
    }
}

printtp(t)
     struct termptr **t;
 /* This prints an entire list of terms;
    the initial data structure for instance */
{
  int i, j, limit;
  struct termptr *t1;
  struct vlist *v;

  if ( t == NULL )
    printf("Empty list of terms\n");
  else
    {
      for (i = 0; i < n; i++)
{
  t1 = t[i]->next;
  while ( t1->next != t1)
    {
```

```
      v = t1->ptr->vars;
      printvlist(v);

      limit = q;
      if (v->next->next->next == v->next->next)
limit = p;

      for (j = 0; j < limit; j++)
printlist((t1->ptr->termhd)[j]);

      t1 = t1->next;
    }
}
    }
}

int countvar(a)
     struct vlist *a;
  /* Returns the number of variables in a list */
{
  if (a->vtx == -2)
    a = a->next;

  if (a->next == a)
    return 0;
  else
    return (1 + countvar(a->next));
}

struct vlist *makevtx(i)
     int i;
  /* creates a variable list containing i */
{
  struct vlist *vl;
  vl =  (struct vlist *) malloc(sizeof(struct vlist));
  vl->vtx = i;
  vl->ptr = NULL;
  vl->next = NULL;

  return vl;
```

```
}

struct vlist *unite(vl1, vl2)
     struct vlist *vl1, *vl2;
  /* Returns the union of two variable lists */
{
  struct vlist *vl3, *vl4, *vl5;
  vl3 = initvlist();
  vl4 = vl3;

  vl1 = vl1->next;
  vl2 = vl2->next;

  while (vl1->next != vl1  && vl2->next != vl2)
    {
      if (vl1->vtx < vl2->vtx)
{
  vl5 = makevtx(vl1->vtx);
  vl5->next = vl3->next;
  vl3->next = vl5;
  vl1 = vl1->next;
}
      else
if (vl1->vtx > vl2->vtx)
  {
    vl5 = makevtx(vl2->vtx);
    vl5->next = vl3->next;
    vl3->next = vl5;
    vl2 = vl2->next;
  }
else  /* duplicate elements */
  {
    vl5 = makevtx(vl1->vtx);
    vl5->next = vl3->next;
    vl3->next = vl5;
    vl1 = vl1->next;
    vl2 = vl2->next;
  }
      vl3 = vl3->next;
    }
```

```
 /* copy remainder of one list */

  if( vl1->next == vl1 )
    vl1 = vl2;

  while ( vl1->next != vl1)
    {
      vl5 = makevtx(vl1->vtx);
      vl5->next = vl3->next;
      vl3->next = vl5;
      vl3 = vl3->next;
      vl1 = vl1->next;
    }

  return vl4;
}

struct vlist *vertex(i)
     int i;
  /* makes a vertex */
{
  struct vlist *vl1, *vl2;
  vl1 = initvlist();
  vl2 = makevtx(i);
  vl2->next = vl1->next;
  vl1->next = vl2;

  return vl1;
}

struct vlist *edge(i,j)
     int i, j;
  /* creates an edge */
{
  struct vlist *vl1, *vl2, *vl3;

  vl1 = initvlist();
  vl2 = makevtx(i);
  vl3 = makevtx(j);
```

```
  vl2->next = vl3;
  vl3->next = vl1->next;
  vl1->next = vl2;

  return vl1;
}

struct vlist *readvtxnum()
  /* reads in vertex value from the input file */
{
  int i;
  struct vlist *vl;

  vl = initvlist();

  if ( fscanf(fp, "%d", &i) == 1 )
    vl = unite(vl, vertex(i));
  else
    {
      printf("Program Aborted: Check input\n");
      exit(0);
    }

  return vl;
}

struct vlist *readedgenum()
  /* reads the vertices that make up the edge from the input file */
{
  int i, j;
  struct vlist *vl;

  vl = initvlist();

  if ( fscanf(fp, "%d %d", &i, &j) == 2)
    vl = unite(vl, edge(i,j));
  else
    {
      printf("Program Aborted: Check input\n");
```

```
      exit(0);
    }

  return vl;
}

struct node **readvertex()
  /* read in vertex information from the file */
{
  int i, j, r;
  struct node **np;

  np = (struct node **) malloc(p*sizeof(struct node *));

  for (j = 0; j < p; j++)
    np[j] = initlist();

  if ( fscanf(fp, "%d %d", &r, &i) == 2 )
    np[0] = addelt(np[0], vecpair(i,r));
  else
    {
      printf("Program Aborted: Check input\n");
      exit(0);
    }

  for (j = 1; j < p; j++)
    {
      if ( fscanf(fp, "%d", &i) == 1)
np[j] = addelt(np[j], vecpair(i,0));
      else
{
  printf("Program Aborted: Check input\n");
  exit(0);
}
    }
  return np;
}
```

```
struct node **readedge()
  /* reads in edge information from the input file */
{
  int i, j, r;
  struct node **np;

  np = (struct node **) malloc(q*sizeof(struct node *));

  for (j = 0; j < q; j++)
    np[j] = initlist();

  for ( j = 0; j < q; j += (p+1) )   /* same processor case */
    np[j] = addelt(np[j], vecpair(0,0));

  if ( fscanf(fp, "%d", &i) == 1)
    for ( j = 1; j < q; j++ )
      {
if ( np[j]->next->hdr == NULL )
  np[j] = addelt(np[j], vecpair(i,0));
      }
  else
    {
      printf("Program Aborted: Check input\n");
      exit(0);
    }

  return np;
}

struct termptr *inittermptr()
  /* creates an initial term pointer with dummy nodes */
{
  struct termptr *start, *tail;

  start = (struct termptr *) malloc(sizeof(struct termptr));
  tail  = (struct termptr *) malloc(sizeof(struct termptr));

  start->prev = NULL;
  start->ptr = NULL;
  start->next = tail;
```

```
  tail->prev = start;
  tail->ptr = NULL;
  tail->next = tail;

  return start;
}

struct termptr **maketp(tp,f)
     struct termptr **tp;
     struct term *f;
  /* creates a new term */
{
  int j;
  struct termptr *t, *z;
  struct vlist *vl;

  vl = f->vars->next;          /* Here is where a vertex/edge is stored */

  while (vl->next != vl)
    {
      j = vl->vtx;

      z = tp[j];

      if (z->next->next != z->next)
z = z->next;

      t = (struct termptr *) malloc(sizeof(struct termptr));
      t->ptr = f;
      t->next = z->next;
      t->prev = z;
      z->next->prev = t;
      z->next = t;

      vl->ptr = t;
      vl = vl->next;
    }
  return tp;
}
```

```
struct termptr **maketerm(file)
     char *file;
  /* creates the initial data structure from the input file */
{
  struct termptr **start, **s;
  struct term *t, *f;
  struct node *z;
  int i, j, r;
  char c;        /* c gets 'V' or 'E' or something else */

  start = (struct termptr **) malloc(n*sizeof(struct termptr *));

  for (j = 0; j < n; j++)
    start[j] = inittermptr();

  s = start;

  fp = fopen(file, "r");

  while ( !feof(fp) )
    {
      if ( fscanf(fp, "%c", &c) == 1 )
{
  if ( !isspace(c) )   /* ignore blank spaces in the file */
    {
      if ( c == 'V' )
{
  f = (struct term *) malloc(sizeof(struct term));
  f->vars = readvtxnum();
  f->termhd = readvertex();
  s = maketp(s,f);
}
      else
{
  if (  c == 'E' )
    {
      f = (struct term *) malloc(sizeof(struct term));
      f->vars = readedgenum();
      f->termhd = readedge();
```

```
      s = maketp(s,f);
    }
  else
    {
      printf(" Program Aborted: Check input\n");
      exit(0);
    }
}
    }
}
    }
  fclose(fp);
  return start;
}

void index(v, r)
     struct vlist *v;
     int r;
 /* This figures the partial assignments */
{
  int j;
  while (v->next != v)
    {
      j = v->vtx;
      assign[j] = (r%p);
      r = (r/p);
      v = v->next;
    }
  return;
}
```

```
int value(v, r)
     struct vlist *v;
     int r;
  /* The indices of old terms are figured out here */
{
  int c, d, e, k[n], i;
  struct vlist *vl;

  e = 0;
  c = 0;
  vl = v->next;

  while (vl->next != vl)
    {
      d = vl->vtx;
      k[c++] = assign[d];
      vl = vl->next;
    }

  for (i = r-1; i > 0; i--)
    e = p*(e + k[i]);

  return (e + k[0]);
}

int elim(tp)
     struct termptr **tp;
  /* The actual elimination of variables takes place here */
{
  int i, j, k, l, r, a, b, c, d, x, y, val1, val2, cost, temp;
  struct vlist *v1, *v2, *v3, *fv, *vtemp, *tv;
  struct node **g, ***h, **newh;
  struct termptr *tp1, *tp2, *tpr, *tpr1, *tpr2;
  struct term *f;

  for ( i = 0; i < n; i++ )
    {
      h = (struct node ***) malloc(m*sizeof(struct node **));
```

```
        a = 0;

        tp1 = (tp[i])->next;
        tp2 = tp1->next;

        h[0] = tp1->ptr->termhd;
        v3 = tp1->ptr->vars;

        while (tp2 != tp2->next)
{
  a++;
  v1 = v3;
  v2 = tp2->ptr->vars;

  g = tp2->ptr->termhd;

  v3 = unite(v1, v2);
  vtemp = v3->next;

  c = countvar(v3);

  b = p;
  for (k = 1; k < c; k++)
    b = (b*p);

  h[a] = (struct node **) malloc(b*sizeof(struct node *));

  for (k = 0; k < n; k++)
    assign[k] = -10;

  for (k = 0; k < b; k++)
    {
      r = k;
      index(vtemp, r);
      val1 = value(v1, countvar(v1));
      val2 = value(v2, countvar(v2));
      (h[a])[k] = addlists( (h[a-1])[val1], g[val2] );
      (h[a])[k] = reduce( (h[a])[k] );
    }
  free(h[a-1]);
```

```
  tv = v2->next->next;
  while (tv->next != tv)
    {
      tpr1 = tv->ptr;
      tpr1->prev->next = tpr1->next;
      tpr1->next->prev = tpr1->prev;
      tv = tv->next;
    }

  tp2 = tp2->next;
}

      d = (b/p);
      newh = (struct node **) malloc(d*sizeof(struct node *));

      l = 0;
      for (j = 0; j < d; j++)
{
  newh[j] = initlist();
  for (k = 0; k < p; k++)
    newh[j] = merge( newh[j], (h[a])[l+k] );

  newh[j] = reduce( newh[j] );
  l += p;
}

      free(h[a]);

      v3->next = v3->next->next; /* i eliminated */

      f = (struct term *) malloc(sizeof(struct term));
      f->vars = v3;
      f->termhd = newh;

      fv = f->vars->next;
      while (fv->next != fv)
{
  x = fv->vtx;
```

```
  tpr = (struct termptr *) malloc(sizeof(struct termptr));
  tpr->ptr = f;
  tpr->next = tp[x]->next->next;
  tpr->prev = tp[x]->next;
  tp[x]->next->next->prev = tpr;
  tp[x]->next->next = tpr;

  fv->ptr = tpr;
  fv = fv->next;
}
    } /* the i loop */


if(newh[0]->next->hdr == NULL)  /* empty list */
  return (U+1);
else
  return newh[0]->next->hdr->next->key;
}

main()
{
  int cost;
  char graph[50]; /* This stores the input graph name */
  struct termptr **tp;

  listops = 0;
  M = 100;   /* arbitrarily chosen */

  scanf("%s", graph);  /* The input graph name */
  scanf("%d", &m);     /* The number of edges */
  scanf("%d", &U);     /* The starting value of U */

  tp = maketerm(graph);

  printf("\n");
  printf("Current value of U is %d\n", U);

  cost = elim(tp);

  printf("Minimum Cost = %d\n", cost);
```

```
  printf("The number of list operations were %d\n", listops);

  if (cost == (U+1))  /* too costly */
      return 1;

  return 0;       /* The optimum is reached */
}
```

APPENDIX B. THE C CODE FOR THE EXACT BMA ALGORITHM

```
/* This program implements the exact algorithm for BMA.
   The comments which are the same as in CMA are avoided here */

#include <stdio.h>

#define n 5
#define p 3
#define q 9

int m, U;

int listops;

int assign[n];

FILE *fp;

struct vector
{
  int  key;
  struct vector  *next;
};

struct node
{
  struct vector  *hdr;
```

```
  struct node  *next;
};

struct vlist
{
  int vtx;
  struct termptr *ptr;
  struct vlist *next;
};

struct term
{
  struct vlist *vars;
  struct node  **termhd;
};

struct termptr
{
  struct termptr *prev;
  struct term *ptr;
  struct termptr *next;
};

void index(v, r)
     struct vlist *v;
     int r;
{
  int j;
  while (v->next != v)
    {
      j = v->vtx;
      assign[j] = (r%p);
      r = (r/p);
      v = v->next;
    }
  return;
}

int value(v, r)
     struct vlist *v;
```

```
    int r;
{
  int c, d, e, k[n], i;
  struct vlist *vl;

  e = 0;
  c = 0;
  vl = v->next;

  while (vl->next != vl)
    {
      d = vl->vtx;
      k[c++] = assign[d];
      vl = vl->next;
    }

  for (i = r-1; i > 0; i--)
    e = p*(e + k[i]);

  return (e + k[0]);
}

struct vector *vechead()
{
  struct vector *head;
  head = (struct vector *) malloc(sizeof(struct vector));
  head->key = -1;
  head->next = NULL;
  return head;
}

struct node *initlist()
{
  struct node *start, *tail;

  listops++;

  start = (struct node *) malloc(sizeof(struct node));
  tail  = (struct node *) malloc(sizeof(struct node));
```

```
  tail->hdr = NULL ;
  tail->next = tail;

  start->hdr = NULL ;
  start->next = tail;

  return start;
}

struct vector *posn(vp,k)
     struct vector *vp;
     int k;
  /* returns pointer to kth position in a tuple */
{
  if ( k == 0 )
    return (vp->next);
  else
    return posn(vp->next, k-1);
}

struct vector *makepvector()
  /* creates a p-tuple with all zeros */
{
  int j;
  struct vector *vp1, *vp2;

  vp1 = vechead();
  vp2 = vp1;
  vp1->next = (struct vector *) malloc(sizeof(struct vector));

  for ( j = 0; j < (p-1); j++)
    {
      vp1->next->key = 0;
      vp1->next->next = (struct vector *) malloc(sizeof(struct vector));
      vp1 = vp1->next;
    }
  vp1->next->key = 0;
  vp1->next->next = NULL;

  return vp2;
```

```
}

struct vector *addvects(a,b)
     struct vector *a;
     struct vector *b;
{
  struct vector *sum, *x, *y, *z;

  sum = vechead();
  x = a->next;
  y = b->next;
  z = sum;
  while ( x != NULL)
    {
      struct vector *temp;
      temp = (struct vector*) malloc(sizeof(struct vector));
      temp->key = x->key + y->key;
      temp->next = NULL;
      z->next = temp;
      x = x->next;
      y = y->next;
      z = z->next;
    }
  return sum;
}

struct node *addelt(lp, vect)
     struct node *lp;
     struct vector *vect;
{
  struct node *temp;

  listops++;

  temp = (struct node*) malloc(sizeof(struct node));
  temp->hdr = vect;
  temp->next = lp->next;
  lp->next = temp;
  return lp;
}
```

```
int compare(a,b)
     struct vector *a, *b;
{
  if (a != NULL && b != NULL)
    {
      if ( a->key < b->key )
return -1;
      else
{
  if ( a->key > b->key )
    return 1;
  else  /* a->key = b->key */
    return compare(a->next, b->next);
}
    }
  else
    {
      if (a == NULL && b != NULL)
return -1;
      else
{
  if (a != NULL && b == NULL)
    return 1;
  else        /* both 'a' and 'b' are NULL */
    return 0;
}
    }
}

struct node *merge(x,y)
     struct node *x, *y;
{
  int i;
  struct node *c, *z, *a, *b;

  listops++;

  a = x->next;
  b = y->next;
```

```
  c = initlist();
  z = c;

  while (a->hdr != NULL && b->hdr != NULL)
    {
        i = compare(a->hdr, b->hdr);
if (i == -1 || i == 0)
  {
    c->next = a;
    c = a;
    a = a->next;
  }
else /* i == 1 */
  {
    c->next = b;
    c = b;
    b = b->next;
  }
      }

  if (a->hdr == NULL && b->hdr != NULL)
    c->next = b;

  if (a->hdr != NULL && b->hdr == NULL)
    c->next = a;

  return z;
}

struct node *mergesort(a)
     struct node *a;
{
  struct node *p1, *q1, *r, *s;

  listops++;

  if ( a->next->next->hdr != NULL )
            /* Check if list has 0 or 1 element */
    {
      p1 = a;
```

```
      q1 = a->next->next;
      while (q1->hdr != NULL)
{
  a = a->next;
  q1 = q1->next->next;
}

      s = (struct node *) malloc(sizeof(struct node));
      s->hdr = NULL;
      s->next = a->next;

      r = (struct node *) malloc(sizeof(struct node));
      r->hdr = NULL;
      r->next = r;
      a->next = r;

      return merge(mergesort(p1), mergesort(s));
    }
  return a;
}

struct node *addlists(a,b)
     struct node *a, *b;
{
  struct node *a1, *b1, *c;

  c = initlist();

  if (a->next->hdr == NULL || b->next->hdr == NULL)
    return c;
  else
    {
      for (a1 = a->next; a1->hdr != NULL; a1 = a1->next)
for (b1 = b->next; b1->hdr != NULL; b1 = b1->next)
  c = addelt(c, addvects(a1->hdr, b1->hdr));
      return mergesort(c);
    }
}

struct node *trim(a)
```

```
    struct node *a;
  /* This implements the procedure TRIM */
{
  struct node *x, *y, *z;
  struct vector *v;
  int heavy;

  listops++;

  z = a;

  while (a->next->next != a->next)
    {
      heavy = 0;
      for(v = a->next->hdr->next; v != NULL; v = v->next)
if (v->key > U)
  {
    a->next = a->next->next;
    heavy = 1;
    break;
  }
      if(!heavy)
a = a->next;
    }

  x = z->next;
  y = z->next->next;

  while ( x->hdr != NULL && y->hdr != NULL)
    {
      if ( compare(x->hdr, y->hdr) == 0)
{
  y = y->next;
  x->next = y;
}
      else
{
  x = x->next;
  y = y->next;
}
```

```
    }
  return z;
}

int weight(v)
     struct vector *v;
  /* returns the weight of a tuple */
{
  int w;
  if (v == NULL)
    return 0;
  else
    {
      w = weight(v->next);
      return (((v->key) > w) ? (v->key) : w);
    }
}

int minmax(a)
     struct node *a;
  /* returns the minimum processor load */
{
  int w, x;
  struct vector *v;

  listops++;

  v = a->next->hdr;

  if (v == NULL)        /* empty list */
    return (U+1);
  else
    {
      w = weight(v);
      x = minmax(a->next);
      return ( (w <= x) ? w : x );
    }
}

struct vector *singlevec(a)
```

```
    int a;
{
  struct vector *vp;

  vp = (struct vector *) malloc(sizeof(struct vector));

  vp->key = a;
  vp->next = NULL;

  return vp;
}

struct vector *vecpair(a,b)
    int a, b;
{
  struct vector *vp1, *vp2, *vp3;

  vp1 = vechead();
  vp2 = singlevec(a);
  vp3 = singlevec(b);

  vp1->next = vp2;
  vp2->next = vp3;

  return vp1;
}

struct vlist *initvlist()
{
  struct vlist *vl1, *vl2;
  vl1 = (struct vlist *) malloc(sizeof(struct vlist));
  vl2 = (struct vlist *) malloc(sizeof(struct vlist));

  vl1->vtx = -2;
  vl1->ptr = NULL;
  vl1->next = vl2;

  vl2->vtx = -3;
  vl2->ptr = NULL;
  vl2->next = vl2;
```

```
  return vl1;
}

printlist(a)
     struct node *a;
{
  int i = 1;
  struct vector *vect;

  if (a->next->hdr == NULL)
    printf("Empty list of pairs\n");
  else
    {
      a = a->next;
      while ( a != a->next)
{
  vect = a->hdr->next;
  printf("Vector%d\n",i++);
  while ( vect != NULL )
    {
      printf("%d\n",vect->key);
      vect = vect->next ;
    }
  a = a->next;
}
    }
}

printvlist(v)
     struct vlist *v;
{
  if ( v->next->next == v->next )
    printf("Empty list of vertices\n");
  else
    {
      printf("Vertices\n");
      while (v->next->next != v->next)
{
  printf("%d\n", v->next->vtx);
```

```
  v = v->next;
}
    }
}

printtp(t)
     struct termptr **t;
{
  int i, j, limit;
  struct termptr *t1;
  struct vlist *v;

  if ( t == NULL )
    printf("Empty list of terms\n");
  else
    {
      for (i = 0; i < n; i++)
{
  t1 = t[i]->next;
  while ( t1->next != t1)
    {
      v = t1->ptr->vars;
      printvlist(v);

      limit = q;
      if (v->next->next->next == v->next->next)
limit = p;

      for (j = 0; j < limit; j++)
printlist((t1->ptr->termhd)[j]);

      t1 = t1->next;
    }
}
    }
}

int countvar(a)
     struct vlist *a;
{
```

```
  if (a->vtx == -2)
    a = a->next;

  if (a->next == a)
    return 0;
  else
    return (1 + countvar(a->next));
}

struct vlist *makevtx(i)
     int i;
{
  struct vlist *vl;
  vl =  (struct vlist *) malloc(sizeof(struct vlist));
  vl->vtx = i;
  vl->ptr = NULL;
  vl->next = NULL;

  return vl;
}

struct vlist *unite(vl1, vl2)
     struct vlist *vl1, *vl2;
{
  struct vlist *vl3, *vl4, *vl5;
  vl3 = initvlist();
  vl4 = vl3;

  vl1 = vl1->next;
  vl2 = vl2->next;

  while (vl1->next != vl1  && vl2->next != vl2)
    {
      if (vl1->vtx < vl2->vtx)
{
  vl5 = makevtx(vl1->vtx);
  vl5->next = vl3->next;
  vl3->next = vl5;
  vl1 = vl1->next;
}
```

```
      else
if (vl1->vtx > vl2->vtx)
  {
    vl5 = makevtx(vl2->vtx);
    vl5->next = vl3->next;
    vl3->next = vl5;
    vl2 = vl2->next;
  }
else  /* duplicate elements */
  {
    vl5 = makevtx(vl1->vtx);
    vl5->next = vl3->next;
    vl3->next = vl5;
    vl1 = vl1->next;
    vl2 = vl2->next;
  }
      vl3 = vl3->next;
    }

 /* copy remainder of one list */

  if( vl1->next == vl1 )
    vl1 = vl2;

  while ( vl1->next != vl1)
    {
      vl5 = makevtx(vl1->vtx);
      vl5->next = vl3->next;
      vl3->next = vl5;
      vl3 = vl3->next;
      vl1 = vl1->next;
    }

  return vl4;
}

struct vlist *vertex(i)
     int i;
{
  struct vlist *vl1, *vl2;
```

```
  vl1 = initvlist();
  vl2 = makevtx(i);
  vl2->next = vl1->next;
  vl1->next = vl2;

  return vl1;
}

struct vlist *edge(i,j)
     int i, j;
{
  struct vlist *vl1, *vl2, *vl3;

  vl1 = initvlist();
  vl2 = makevtx(i);
  vl3 = makevtx(j);

  vl2->next = vl3;
  vl3->next = vl1->next;
  vl1->next = vl2;

  return vl1;
}

struct vlist *readvtxnum()
{
  int i;
  struct vlist *vl;

  vl = initvlist();

  if ( fscanf(fp, "%d", &i) == 1 )
    vl = unite(vl, vertex(i));
  else
    {
      printf("Could not read the vertex\n");
      exit(0);
    }

  return vl;
```

```
}

struct vlist *readedgenum()
{
  int i, j;
  struct vlist *vl;

  vl = initvlist();

  if ( fscanf(fp, "%d %d", &i, &j) == 2 )
    vl = unite(vl, edge(i,j));
  else
    {
      printf("Could not read the edge\n");
      exit(0);
    }

  return vl;
}

struct node **readvertex()
{
  int i, j, r;
  struct node **np;
  struct vector *vp;

  np = (struct node **) malloc(p*sizeof(struct node *));

  for (j = 0; j < p; j++)
    {
      np[j] = initlist();
      if ( fscanf(fp, "%d", &i) == 1 )
{
  vp = makepvector();
  (posn(vp,j))->key = i;
  np[j] = addelt(np[j], vp);
}
      else
{
  printf("Could not read vertex information: Check p\n");
```

```
  exit(0);
}
    }
  return np;
}

struct node **readedge()
{
  int i, j, k, r, a[2];
  struct node **np;
  struct vector *vp;

  np = (struct node **) malloc(q*sizeof(struct node *));

  for (j = 0; j < q; j++)
    np[j] = initlist();

  for ( j = 0; j < q; j += (p+1) )   /* same processor case */
    np[j] = addelt(np[j], makepvector());

  if ( fscanf(fp, "%d", &i) == 1 )
    for ( j = 1; j < q; j++ )
      {
if ( np[j]->next->hdr == NULL )
  {
    vp = makepvector();
    r = j;
    for (k = 0; k < 2; k++)
      {
a[k] = (r%p);
(posn(vp,a[k]))->key = i;
r = (r/p);
      }
    np[j] = addelt(np[j], vp);
  }
      }
  else
    {
      printf("Could not read communication cost\n");
      exit(0);
```

```
    }

  return np;
}

struct termptr *inittermptr()
{
  struct termptr *start, *tail;

  start = (struct termptr *) malloc(sizeof(struct termptr));
  tail  = (struct termptr *) malloc(sizeof(struct termptr));

  start->prev = NULL;
  start->ptr = NULL;
  start->next = tail;

  tail->prev = start;
  tail->ptr = NULL;
  tail->next = tail;

  return start;
}

struct termptr **maketp(tp,f)
     struct termptr **tp;
     struct term *f;
{
  int j;
  struct termptr *t, *z;
  struct vlist *vl;

  vl = f->vars->next;          /* Here is where a vertex/edge is stored */

  while (vl->next != vl)
    {
      j = vl->vtx;

      z = tp[j];

      if (z->next->next != z->next)
```

```
z = z->next;

      t = (struct termptr *) malloc(sizeof(struct termptr));
      t->ptr = f;
      t->next = z->next;
      t->prev = z;
      z->next->prev = t;
      z->next = t;

      vl->ptr = t;
      vl = vl->next;
    }
  return tp;
}

struct termptr **maketerm(file)
     char *file;
{
  struct termptr **start, **s;
  struct term *t, *f;
  struct node *z;
  int i, j, r;
  char c;        /* c gets 'V' or 'E' or something else */

  start = (struct termptr **) malloc(n*sizeof(struct termptr *));

  for (j = 0; j < n; j++)
    start[j] = inittermptr();

  s = start;

  fp = fopen(file, "r");

  while ( !feof(fp) )
    {
      if ( fscanf(fp, "%c", &c) == 1 )
{
  if ( !isspace(c) )   /* ignore blank spaces in the file */
    {
      if ( c == 'V' )
```

```
{
  f = (struct term *) malloc(sizeof(struct term));
  f->vars = readvtxnum();
  f->termhd = readvertex();
  s = maketp(s,f);
}
      else
{
  if (  c == 'E' )
    {
      f = (struct term *) malloc(sizeof(struct term));
      f->vars = readedgenum();
      f->termhd = readedge();
      s = maketp(s,f);
    }
  else
    {
      printf(" Program Aborted as a number was read instead of a V or an E\n");
      exit(0);
    }
}
    }
}
    }
  fclose(fp);
  return start;
}

int elim(tp)
     struct termptr **tp;
{
  int i, j, k, l, r, a, b, c, d, x, y, val1, val2, cost, temp;
  struct vlist *v1, *v2, *v3, *fv, *vtemp, *tv;
  struct node **g, ***h, **newh;
  struct termptr *tp1, *tp2, *tpr, *tpr1, *tpr2;
  struct term *f;

  for ( i = 0; i < n; i++ )
    {
      h = (struct node ***) malloc(m*sizeof(struct node **));
```

```
        a = 0;

        tp1 = (tp[i])->next;
        tp2 = tp1->next;

        h[0] = tp1->ptr->termhd;
        v3 = tp1->ptr->vars;

        while (tp2 != tp2->next)
{
  a++;
  v1 = v3;
  v2 = tp2->ptr->vars;

  g = tp2->ptr->termhd;

  v3 = unite(v1, v2);
  vtemp = v3->next;

  c = countvar(v3);

  b = p;
  for (k = 1; k < c; k++)
    b = (b*p);

  h[a] = (struct node **) malloc(b*sizeof(struct node *));

  for (k = 0; k < n; k++)
    assign[k] = -10;

  for (k = 0; k < b; k++)
    {
      r = k;
      index(vtemp, r);
      val1 = value(v1, countvar(v1));
      val2 = value(v2, countvar(v2));
      (h[a])[k] = addlists( (h[a-1])[val1], g[val2] );
      (h[a])[k] = trim( (h[a])[k] );
    }
```

```
  free(h[a-1]);

  tv = v2->next->next;
  while (tv->next != tv)
    {
      tpr1 = tv->ptr;
      tpr1->prev->next = tpr1->next;
      tpr1->next->prev = tpr1->prev;
      tv = tv->next;
    }

  tp2 = tp2->next;
}

      d = (b/p);
      newh = (struct node **) malloc(d*sizeof(struct node *));

      l = 0;
      for (j = 0; j < d; j++)
{
  newh[j] = initlist();
  for (k = 0; k < p; k++)
    newh[j] = merge( newh[j], (h[a])[l+k] );
  newh[j] = trim( newh[j] );
  l += p;
}

      free(h[a]);

      v3->next = v3->next->next; /* i eliminated */

      f = (struct term *) malloc(sizeof(struct term));
      f->vars = v3;
      f->termhd = newh;

      fv = f->vars->next;
      while (fv->next != fv)
{
  x = fv->vtx;
```

```
  tpr = (struct termptr *) malloc(sizeof(struct termptr));
  tpr->ptr = f;
  tpr->next = tp[x]->next->next;
  tpr->prev = tp[x]->next;
  tp[x]->next->next->prev = tpr;
  tp[x]->next->next = tpr;

  fv->ptr = tpr;
  fv = fv->next;
}
    } /* the i loop */

return minmax(newh[0]);
}

main()
{
  int load;
  char graph[50];
  struct termptr **tp;

  scanf("%s", graph);
  scanf("%d", &m);
  scanf("%d", &U);

  listops = 0;

  tp = maketerm(graph);
  load = elim(tp);

  printf("optimum load = %d\n", load);
  printf("The number of list operations were %d\n", listops);

  if (load == (U+1))  /* too heavy */
    return 1;

  return 0;
}
```

APPENDIX C. A GENERIC UNIX SHELL PROGRAM THAT IMPLEMENTS THE ALGORITHMS

Both CMA and BMA are driven by a shell program. CMA is comiled as "cm" and BMA as "bm". When this shell program is invoked, it prompts the user for the input graph name, the number of edges and the beginning value of U. If an optimum solution is found with the present value of U, then the optimum value is found. Otherwise, U is multiplied by 1.5, shown as 3/2 here, and the program runs again. In the program below "xm" must be replaced by "cm" or "bm" as the case may be.

```
#! /bin/sh

echo  "Enter graph name: "
read graph
echo  "Enter number of edges: "
read m
echo  "Enter the value of U: "
read U

while true
do
  if echo "$graph
    $m
    $U" | xm
then
break
else
U=`expr $U \* 3 / 2`
fi
done
```

APPENDIX D. THE C CODE FOR RANDOM GRAPH GENERATION

We describe the code that generates random graphs for both CMA and BMA. The code given below actually generates the random graphs for CMA. However, the corresponding code for BMA is easily obtained by not generating the memory information of a module in the subroutine "vertices()" — simply remove that piece of code.

To make the graph generated as random as possible, we employ the following strategy. We use the unix system command called "rand()", which generates a sequence of pseudo-random numbers between 0 and $2^{31} - 1 = 2147483647$. Since we are interested in a probability, we divide the generated numbers by this number, to get random numbers between 0 and 1. The generator is reinitialized each time by calling the system command "srand(seed)" with "seed" as its integer argument. We get random starting point by calling srand with a random seed. We choose "process id's" as the seed. So each time we run the program on even the same input, we are guaranteed a different graph, since the seed is the "id" of the process we just created when we called the program.

The code is self-explanatory and comments are included where appropriate. Note that the constant "edgeprob" stands for the edge probability, while "maxexec",

"maxcom" and "maxmem" represent respectively the maximum execution cost , maximum communication cost and maximum memory available on processor 1. All of these are pre-determined.

```
#include<stdio.h>
#include<sys/types.h>

#define p 3                 /* number of processors */
#define n 10                /* number of vertices */
#define edgeprob (0.15)     /* probability of an edge between two vertices */

#define maxexec 20     /* The e:c ratio */
#define maxcom 200

#define maxmem 100

#define randmax (2147483647.0)     /* 2^31 - 1 */

int visited[n];     /* To mark a vertex visited in checking
                       connectivity of a graph */

struct node
{
  int vtx;
  struct node *next;
};

float random()
  /* This generates a random number between 0 and 1 */
{
  int j;
  float r;

  r = rand()/randmax;
  return r;
}
```

```
void vertices()
  /* This generates the vertex information */
{
  int j, k, execcost, memreq;
  for ( j = 0; j < n; j++)
    {
      printf("V\n");
      printf("%d\n", j);

/* The next four lines generate  the memory requirement of a module */

      memreq = maxmem*random();
      while (memreq == 0)
memreq = maxmem*random();
      printf("%d\n", memreq);

/* The following loop generates the execution costs of a module
   on the p processors */

      for ( k = 0; k < p; k++)
{
  execcost = maxexec*random();
  while (execcost == 0)
    execcost = maxexec*random();
  printf("%d\n", execcost);
}
    }
  return;
}

struct node *makenode(i)
     int i;
  /* creates a node in the adjacency list */
{
  struct node *np;
  np = (struct node *) malloc(sizeof(struct node));
  np->vtx = i;
  np->next = NULL;
```

```
  return np;
}

struct node *addnode(np, i)
     struct node *np;
     int i;
  /* appends a node to an adjacency list */
{
  struct node *temp;

  if (np == NULL)
    return makenode(i);
  else
    {
      temp = np;
      while (temp->next != NULL)
temp = temp->next;
    }

  temp->next = makenode(i);

  return np;
}

void visit(el,k)
     struct node **el;
     int k;
  /* visits the vertices of the graph */
{
  struct node *t;
  visited[k] = 1;
  for (t = el[k]; t!= NULL; t = t->next)
    if (visited[t->vtx] == 0)
      visit(el,t->vtx);
}
```

```
int connected(el)
     struct node **el;
  /* returns 1 if graph is connected, else 0 */
{
  int k;

  for (k = 0; k < n; k++)
    visited[k] = 0;

  for (k = 0; k < n; k++)
    if (visited[k] == 0)
      {
if (k == 0)
  visit(el, k);
else
  return 0;
      }
  return 1;
}

void edges()
  /* creates the edge information */
{
  int j, k, comcost[n][n], numedges;
  struct node *np, **edgelist;

 start:
  numedges = 0;
  edgelist = (struct node **) malloc(n*sizeof(struct node *));

  for ( j = 0; j < n; j++)
    edgelist[j] = NULL;

  for (j = 0; j < (n-1); j++)
    for (k = (j+1); k < n; k++)
      {
comcost[j][k] = 0;
if ( random() <= edgeprob )
     /* generate an edge between two vertices with a
```

```
      certain probability */
  {
    numedges++;

    edgelist[j] = addnode(edgelist[j], k);
    edgelist[k] = addnode(edgelist[k], j);

    while (comcost[j][k] == 0)
      comcost[j][k] = maxcom*random();
  }
      }

/* The following condition checks if the graph generated above
   is connected; if not it starts all over again */

  if ( !connected(edgelist) )
    {
      free(edgelist);
      goto start;
    }

/* The following prints out the edges if the
   generated graph is connected */

  printf("\n");
  for (j = 0; j < (n-1); j++)
    for (k = (j+1); k < n; k++)
      if (comcost[j][k] != 0)
{
  printf("E\n");
  printf("%d\n", j);
  printf("%d\n", k);
  printf("%d\n", comcost[j][k]);
}

/* The number of edges generated is also printed out */

  printf("\n");
  printf("numedges = %d\n", numedges);
```

```
}

main()
{
pid_t getpid();

srand(getpid()); /* initiate the random sequence at a random point */

vertices();
edges();
}
```

APPENDIX E. THE C CODE FOR RANDOM TREE GENERATION

Given below is the code for generating random trees. As with the code for generating random graphs given in Appendix D, the code below is for CMA. However, as observed there, we can easily modify the code to get the random tree generator for BMA. The constants are all same as in Appendix D. The code is adapted from the algorithm for random tree generation given in [64].

```
#include<stdio.h>
#include<sys/types.h>

#define p 4                  /* number of processors */
#define n 50                  /* number of vertices */

#define maxexec 5
#define maxcom  50

#define maxmem 100

#define TRUE 1
#define FALSE 0

#define randmax (2147483647.0)     /* 2^31 - 1 */
```

```
float random()
{
  int j;
  float r;

  r = rand()/randmax;
  return r;
}

void vertices()
{
  int j, k, execcost, memreq;
  for ( j = 0; j < n; j++)
    {
      printf("V\n");
      printf("%d\n", j);

      memreq = maxmem*random();
      while (memreq == 0)
memreq = maxmem*random();
      printf("%d\n", memreq);

      for ( k = 0; k < p; k++)
{
  execcost = maxexec*random();
  while (execcost == 0)
    execcost = maxexec*random();
  printf("%d\n", execcost);
}
    }
  return;
}

void reorder(x, k)
     int x[], k;
  /* This sorts the first k elements in an array of integers */
{
  int i, item, temp;
  for (item = 0; item < k-1; ++item)
```

```
    for (i = item+1; i < k; ++i)
      if ( x[i] < x[item] )
{
  temp = x[item];
  x[item] = x[i];
  x[i] = temp;
}
  return;
}

void edges()
{
  int i = 0, j, k, comcost, found, sequence[n-2], x[n];

  for (j = 0; j < (n-2); j++)
    sequence[j] = (n-1)*random();

  for (k = 0; k < n; k++)
    x[k] = n+1;

  for (k = 0; k < n; k++)
    {
      found = FALSE;
      for (j = 0; j < n-2; j++)
if ( k == sequence[j] )
  {
    found = TRUE;
    break;
  }

      if (!found)
x[i++] = k;
    }

  k = 0;

 start:
  if (x[0] != n+1)
    {
      comcost = maxcom*random();
```

```
      while (comcost == 0)
comcost = maxcom*random();
      if ( x[0] < sequence[k] )
{
  printf("E\n");
  printf("%d\n", x[0]);
  printf("%d\n", sequence[k]);
  printf("%d\n", comcost);
}
      else
{
  printf("E\n");
  printf("%d\n", sequence[k]);
  printf("%d\n", x[0]);
  printf("%d\n", comcost);
}
    }

      k++;

      if (k == n-2)
{
  numedges++;
  comcost = maxcom*random();
  while (comcost == 0)
    comcost = maxcom*random();
  if ( x[0] < sequence[k] )
    {
      printf("E\n");
      printf("%d\n", x[0]);
      printf("%d\n", sequence[k]);
      printf("%d\n", comcost);
    }
  else
    {
      printf("E\n");
      printf("%d\n", sequence[k]);
      printf("%d\n", x[0]);
      printf("%d\n", comcost);
    }
```

```
  goto end;
}

      for (j = k; j < n-2; j++)
if ( sequence[k-1] == sequence[j] )
  {
    x[0] = n+1;
    reorder(x,n);
    goto start;
  }

      x[0] = sequence[k-1];
      reorder(x,n);
      goto start;

 end:
  return;
 }

main()
{
pid_t getpid();

srand(getpid()); /* initiate the random sequence at a random point */

vertices();
edges();
}
```

APPENDIX F. A SAMPLE INPUT GRAPH FOR CMA

This is a sample input graph; in fact it is the same graph given as an example in Chapter 5. The letter "V" stands for a vertex (module) and "E" for an edge. The number immediately following V is the vertex number. Note that the vertices are numbered from 0 to (n-1). This is to maintain consistentency with the computer during implementation. The second number after a V is the memory requirement of the module. The numbers following the memory requirement are all execution costs. For example, module 1 (vertex 0 here) has a memory requirement of 20 and its execution costs are 40, 20 and 25 on the three processors respectively. Similarly, the first two numbers following an E are the vertices which make up the edge. The third number is the communication cost between the two modules making the edge up. For example, the edge (2,3) ((1,2) here) has a uniform communication cost of 2.

```
V
0
20
40
20
25
V
1
30
25
25
20
V
2
25
20
45
25
V
3
15
30
35
30
V
4
35
15
30
40
```

E
0
2
8
E
1
2
2
E
1
4
E
2
3
3
E
2
4
1
E
3
4
4

APPENDIX G. A SAMPLE INPUT GRAPH FOR BMA

This is the same input graph as in Appendix F. The only difference from that graph is that there are no memory requirements here.

```
V
0
40
20
25
V
1
25
25
20
V
2
20
45
25
V
3
30
35
30
V
4
15
30
40
```

E
0
2
8
E
1
2
2
E
1
4
5
E
2
3
3
E
2
4
1
E
3
4
4