PARALLEL PROCESSING OF BEST-FIRST BRANCH AND BOUND ALGORITHMS ON DISTRIBUTED MEMORY MULTIPROCESSORS

by

Tarek Saad Abdel-Rahman

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in The University of Michigan
1989

Doctoral Committee:

Associate Professor Trevor N. Mudge, Chairman
Assistant Professor Chaitanya K. Baru
Professor John P. Hayes
Professor William R. Martin

© Tarek S. Abdel-Rahman 1989
All Rights Reserved

To my family

ACKNOWLEDGEMENTS

I would like to thank my committee members for their useful comments and suggestions, and their careful review of this manuscript. I would like to specially thank my advisor Trevor N. Mudge for his help during the many stages of my studies at the University of Michigan.

I would like to extent my deepest gratitude to my family, whom without their help and support it would not have been possible to complete this work. I wish to thank my wife Suad for her love, care, support and patience during the years of our marriage.

Financial support has been partially provided by the Robot Systems Division (RSD), Center for Research on Integrated Manufacturing (CRIM) at the University of Michigan; by the Materials Laboratory, Air Force Wright Aeronautical Laboratories, Aeronautical Systems Division (AFSC), United States Air Force, Wright-Patterson AFB, Ohio 45433-6503; by the Army Research Office under Contract DAAG29-84-K-0070; and by a pre-doctoral Fellowship from the Hoarce H. Rackham School of Graduate Studies at the University of Michigan.

The computing facilities of the Advanced Computer Architecture Laboratory and the Robot Systems Division were instrumental in the completion of this work.

TABLE OF CONTENTS

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF APPENDICES

CHAPTER 1

INTRODUCTION

The continuing advances in VLSI technology are making possible the construction of highly parallel multiprocessor systems that are capable of delivering high performance computing at relatively low costs. These systems are based on a large ensemble of relatively simple and relatively inexpensive processors connected together by a message passing network. There is no globally shared memory in these systems; rather, each processor has its own private memory. The processors communicate to share data and/or synchronize by passing messages to one another over the network. Consequently, these multiprocessors are known as *Distributed Memory Multiprocessors (DMMs)*, or *Message Passing Multiprocessors (MPMs).* Examples include the hypercubes of Intel [Inte85], Ametek [Amet85], and NCUBE [NCUB85], and the mesh-connected systems of Ametek [Amet88] and Goodyear [Pott85].

In contrast to more conventional high-performance supercomputers, which rely on pipelined and vectored processing, DMMs achieve their high performance potential through the use of a large number of processors cooperating in parallel to perform a particular computation. This approach is referred to as *parallel processing* and is becoming an increasingly attractive alternative to achieving high performance computing. The effective use of a large number of processors, however, relies on the extent to which a computation can be *mapped* onto the multiprocessor, i.e., divided into smaller components which can be assigned to, and independently executed by, individual processors. There are no general techniques for mapping applications onto multiprocessors; rather, the mapping of applications is studied on a case by case basis. This study deals with the mapping of a class of heuristic search problems that can be described by the Branch and Bound (BB) algorithm onto DMMs.

1.1 Motivations and Objectives

DMMs have been successfully applied to a large number of problems from many application areas of science and engineering (see [FJLO88] for a survey). Examples include: image processing, computer vision, matrix computations, simulation of particle transport, and computer-aided design. The results obtained often reflect substantial improvements over the performance of a single processor. A notable example is the use of a 1024-processor hypercube at Sandia National Labs to solve large scale problems in fluid dynamics and structural analysis [GuMB88]. Experimentally obtained results indicate an impressive gain in performance, not only over a single processor, but also over supercomputers of higher cost.

However, the success of a DMM in improving the performance of a particular computation depends to a large extent on the nature of that computation. In many cases a computation lends itself to parallelism and substantial performance improvements are possible. In some other computations, however, this is not the case, and special care must be taken with the mapping of that computation onto the DMM if any improvements in performance are to be obtained. Therefore, it is safe to say that the performance gains obtained by applying a DMM to a computation depends not only on the performance potential of the DMM, but also on the nature of that computation. Hence, to obtain a valid view of the true potential of DMMs as high performance computers, applications of diverse computational nature must be studied. The majority of applications to which DMMs have been applied to date tend to have common properties which lend them to parallel processors in general, and to DMMs in particular. The applications tend to have regular and homogeneous data sets, typically consisting of arrays of data elements of the same type. Operations are usually applied to the elements of the data sets almost independently of each other, implying large degrees of inherent parallelism. The computations are also structured in such a way that a static load-balancing strategy can be employed to generate uniform workloads across the processors. This is usually accomplished by dividing the data sets equally among the processors. Therefore, it seems that only a restricted set of computation have been considered to reflect performance gains that can be obtained from DMMs. A key motivation behind this study was to extend the scope of the computations to which DMMs have been applied by considering the parallel processing of the BB algorithm on this class of multiprocessors.

The BB algorithm generates a data set that is highly irregular and dynamic. This data set, referred to as the BB tree, will be described later in Chapter 3. The structure of this data set cannot be predicted in advance and hence, dynamic balancing of workloads must be employed. Furthermore, the BB algorithm, as will be seen later, requires global knowledge of some attributes of this data set in order to maintain its computational efficiency. This global knowledge requirement can pose problems in a distributed memory environment and presents a challenge for DMMs. Therefore, an objective of this study is to investigate the suitability of a distributed memory architecture when both global knowledge and dynamic load balancing are required.

There are other factors that motivate us to study the parallel processing of BB algorithms on DMMs and make this study significant. The BB algorithm is an interesting algorithm in its own right. It is used to solve a large number of fundamental problems in science and engineering that otherwise would have no efficient solution methods [LaWo66, Nils80]. Examples of such problems include the well-known traveling salesman problem, the integer programming problem, and state-space search. BB techniques have also been used in heuristic search algorithms [KuKa83], floor planning of VLSI integrated circuits [WiKC88], placement of electronic components [BrKa88] and robot path planning [KaNa85]. The use of BB to solve many of the above problems generally results in intensive computations that can benefit from the high performance potential of DMMs. One objective of this research is to find out the extent to which BB can benefit from these multiprocessors.

In most scientific and engineering applications that have been successfully parallelized to date, the amount of computations performed by the parallel algorithm to complete the solution of a particular problem instance has not been influenced by the introduction of parallelism. That is, the number of computational steps needed to obtain a solution to a given problem instance is independent of, and hence, is not affected by the parallelism introduced through the use of parallel processors. Parallelism affects the performance of the application only through overheads incurred by the processors in communicating data and/or synchronizing. This is not actual in the case of parallel BB algorithms. The introduction of parallelism to BB can, and indeed often does, change the amount of computations needed to obtain a solution to a given problems instance. This can have considerable effects on the performance of the parallel BB algorithm when compared to the performance of the sequential one. One objective of this study is to investigate the effect of this change in the computational characteristics introduced by parallelism on the performance of the parallel BB algorithm. This will, in effect, lead to the design of a more efficient mapping of BB onto DMMs.

1.2 Research Overview

The mapping of BB algorithms on DMMs will be investigated in this study. This will be done by identifying the factors that affect the performance of this class of algorithms on DMMs. These factors will be shown to give rise to three types of overhead that degrade the performance of the algorithm. The first is communication-overhead that results from the need to exchange information about the search among the processors. The second is computation-overhead that results as a consequence of the expansion of non-essential subproblems; i.e, subproblems not expanded by the sequential algorithm. The third is imbalance-overhead that results as processors idle due to load imbalance. A computation-communication tradeoff exists between the first type of overhead on one hand and the other two types of overhead on the other. In order to reduce the expansion of non-essential subproblems and obtain a more balanced workload, more exchange of information is needed, which gives rise to more communication-overhead. On the other hand, communication-overhead can be minimized by eliminating communications among the processors, which can be done only at the expense of expanding non-essential subproblems and an imbalanced workload, giving rise to more computation-overhead.

Analytical modeling and algorithm analysis techniques are limited in predicting the extent to which these types of overhead affect performance, and in further characterizing the tradeoff among them. While these techniques are successful in predicting the performance of sequential BB algorithms [Smit84, WaYu85], their applicability to parallel BB algorithms is limited by the change in computational characteristics of BB algorithms which caused by the introduction of parallelism. This change is dependent not only on the problem instance to be solved, but also on the nature of the parallel algorithm and the characteristics of the DMM, making it difficult to model without unrealistic simplifying assumptions.

Consequently, the extent to which the factors described above, the three types of overhead, and the computation-communication tradeoff affect the performance of parallel BB algorithms will be demonstrated for actual applications using three parallel algorithms that map the sequential BB algorithm on a hypercube multiprocessor. The third algorithm also demonstrates the performance of a new load balancing strategy that utilizes the computation-communication tradeoff to obtain good performance.

1.3 Thesis Organization

This remainder of this thesis is organized as follows:

- In Chapter 2 the key features of DMMs are reviewed.
- In Chapter 3 a formulation of the BB algorithm outlining its main components is presented. Examples illustrating the application of the algorithm to mathematical programming and artificial intelligence are given.
- In Chapter 4 the mapping of the BB algorithm on DMMs is examined. The factors that affect the performance of a parallel BB algorithm on this class of multiprocessors are presented. The tradeoff among the three types of overhead these factors give rise to is described, and a simple model that captures how this tradeoff affects performance is presented. A review of past work on parallel BB algorithms is also given in this chapter.
- In Chapter 5 the three parallel algorithms used to illustrate the effect of the factors identified in Chapter 4 are described. The third algorithm uses the new load balancing strategy.
- In Chapter 6 experimental results obtained by implementing the algorithms on a commercial hypercube multiprocessor are presented and discussed.
- In Chapter 7 acceleration anomalies that can occur in the execution of parallel BB algorithms are described.
- In Chapter 8 concluding remarks and future research are presented.
- In Appendix A the NCUBE/ten hypercube multiprocessor used to obtain experimental results is described.
- In Appendix B the test problems used in the experimental runs are described.

- In Appendix C implementation details of the experiments are described.

CHAPTER 2

DISTRIBUTED MEMORY MULTIPROCESSORS

In this chapter an overview is presented of distributed memory multiprocessors (DMMs). In section 2.1 the general organization of this class of multiprocessors is reviewed and the key elements of their architecture are discussed. In section 2.2 examples of DMMs are presented.

2.1 The Architecture of DMMs

A DMM consists of N processing elements or nodes that are interconnected together using an interconnection network. There exists no globally shared memory in a DMM; instead, each node contains its own local memory. The nodes communicate to each other through the interconnection network. A block diagram of a DMM is shown in figure 2.1.

The interconnection network consists of a set of dedicated communication links. Each communication link is used to connect a pair of processing elements to each other. In general, the communication links interconnect the nodes so that each node is directly connected to only a subset, referred to as its *neighbors*, of the other nodes in the system.

The nodes and the communication links interconnecting them can be represented by a graph $G = (N, E)$ that consists of a set of vertices N and a set of edges E. The vertices of the graph represent the processing elements in the system, and the edges of the graph represent the communication links. The overall configuration in which the processors and

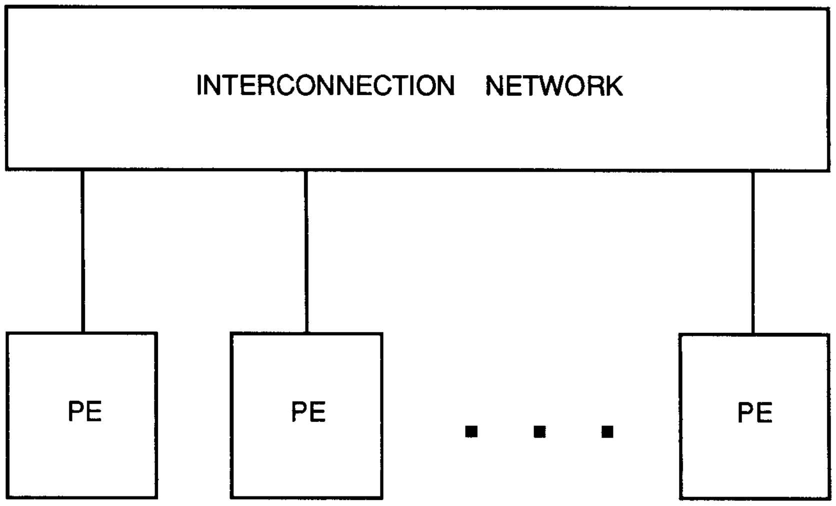

Figure 2.1. A distributed memory multiprocessor system.

the communication links are interconnected is referred to as the *interconnection topology.*

A processing element or node consists of four major components: a processor P, a local memory M, an I/O module, and a communication module CM. These components interface to a local bus as shown in figure 2.2. P is a general purpose instruction set processor, and the local memory is used to store both programs and data for P. The I/O module provides an interface to external devices and peripherals. The communication module connects the node to the communication links, which in turn connects the node to its neighbors in the DMM.

The processing elements communicate by passing messages to one another over the communication links. This is facilitated by a Message Transfer System (MTS) which operates in each node [HwBr84]; this is depicted in figure 2.3. A process in node A makes a call to the MTS when it wants to send a message to another process in node B. The MTS receives the request, determines the destination, computes the route if needed, and initiates the transmission of the message. When the message arrives at B, the MTS

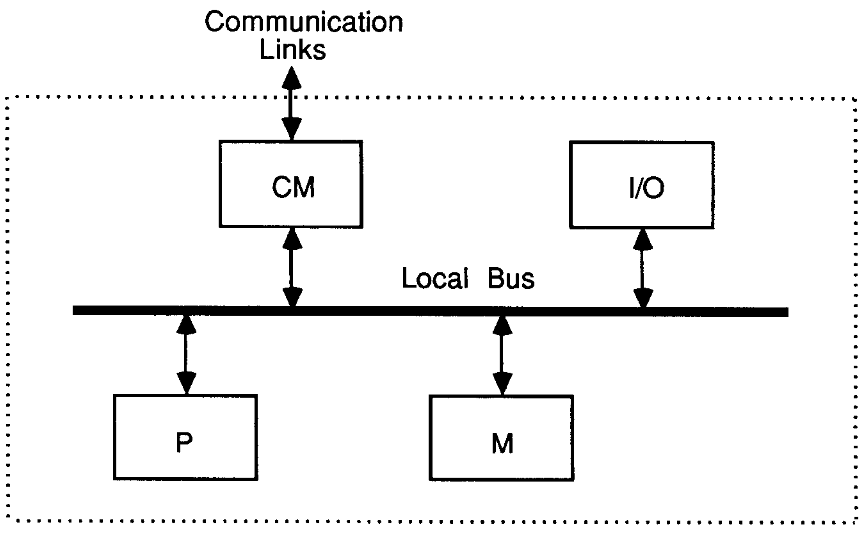

Figure 2.2. The architecture of a node.

interrupts the process on B to inform it that the message has arrived. Such communication protocol also is referred to as the *mail box* communication protocol [PTLP85].

DMMs generally support the *loosely coupled* model of computation. In this model, concurrent processes in the computation are assumed to be independent of one another. The degree of interaction among the processes is small. Each process executes its own code operating on its own data set. Processes communicate by passing messages to each other. Hence, DMMs are also referred to as *loosely coupled* multiprocessors.

The architecture of DMMs offers a number of advantages over other architectures for interconnecting processors to processors as well as processors to memory. First, the contention for memory as a shared resource is avoided by distributing the memory resources across the nodes. Second, since there is no contention for memory, the latency in memory access is reduced to a minimum. This is an important factor since memory access is a significant part of any computation. Finally, with a properly designed interconnection network, the number of nodes can be increased to several times that which is possible in

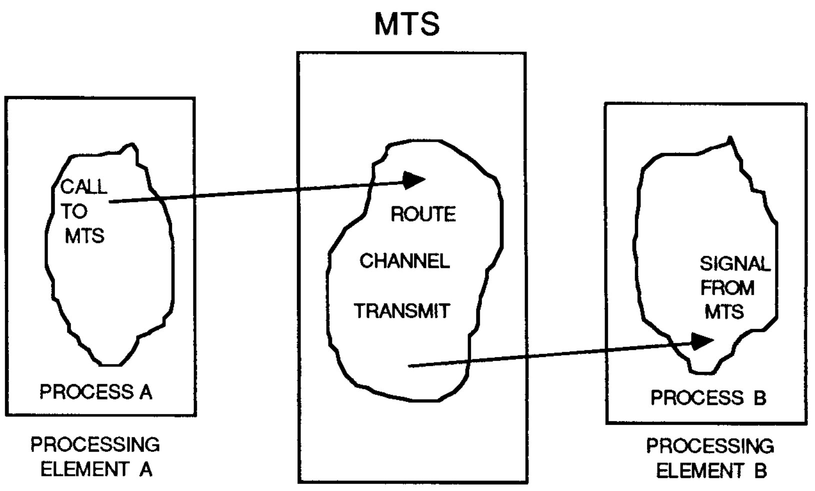

Figure 2.3. The Message Transfer System.

alternative architectures based on shared-buses or multistage interconnection networks. This provides the potential for massive parallelism and very high performance computing.

The architecture of DMMs can be characterized by three main elements: the multiplicity of instruction and data streams, the interconnection topology, and the granularity of the node.

2.1.1 Multiplicity of Instructions and Data Streams

Following Flynn's classification scheme [Flyn66], DMM systems can be split into two groups with respect to the multiplicity of instruction and data streams: Single Instruction Multiple Data streams, or SIMD multiprocessors and Multiple Instruction Multiple Data streams, or MIMD multiprocessors.

In SIMD multiprocessors, there is a single instruction stream to all the processing elements, or nodes, in the system. A global control unit broadcasts the instructions to all nodes, and each node executes the instructions using the data in its local memory. The nodes are synchronized and execute in lock-step, and each node can be optionally activated or deactivated. Only activated nodes execute the broadcasted instructions allowing the conditional execution of instructions.

In MIMD multiprocessors, there is a separate instruction stream in each node of the system. Each node has a control unit which issues instructions allowing the node to execute asynchronously with respect to other nodes on the data in its local memory. In general, MIMD multiprocessors offer the potential for more flexibility and performance over SIMD multiprocessors for a wide range of applications. Furthermore, MIMD multiprocessors can efficiently emulate SIMD multiprocessors by executing the same instructions in each node. This mode of operation is referred to as Single Code Multiple Data execution (SCMD) [Buzz88], or sometimes as Single Program Multiple Data (SPMD) [Karp87].

2.1.2 Interconnection Topology

The interconnection network is used to transfer messages from one node in the system to another. The topology of the interconnection network plays an important role in the efficiency with which the network can achieve that goal. Topologies for interconnecting processors in a DMM system can be characterized by a number of properties [AgJa86].

1. *Diameter.* The diameter D of an interconnection topology is defined as the maximum of the minimum distances between any two nodes in the topology. The distance is measured by the number of links. That is,

$$D = max\{d_{min}(i,j) \quad \forall i,j \in V\},$$

where $d_{min}(i,j)$ is the minimum distance between any two nodes i and j. The diameter can be used as a measure of the communication distance between pairs of nodes in a given interconnection topology. The smaller the diameter, the shorter the communication distance and, consequently, the closer the nodes are to each other.

Hence, it is desirable to interconnect the nodes in a topology that has as small a diameter as possible.

2. *Degree.* The degree δ of a node in a topology is defined as the number of edges that are connected to that node. A topology in which all nodes have the same degree δ is referred to as a *regular* topology of degree δ. The degree of the node can be used as a measure of the number of wires that fanout from a node. The smaller the degree, the smaller the fanout of the node, simplifying its design and increasing its fault tolerance. Furthermore, a regular topology implies that all the nodes can be identical. Hence, it is desirable to interconnect the nodes in a regular topology that has as small a degree as possible. However, it is important to realize that minimizing the degree of the topology is typically in conflict with minimizing the diameter of the topology. In order to get the nodes closer together by minimizing the diameter, the number of connections from each node must be increased. Therefore, it is not possible to minimize both the degree and the diameter independently.

3. *Expandability.* The expandability of a topology is a measure of the ability of that topology to accommodate more nodes and links. A distributed memory multiprocessor system with an expandable topology requires little or no modification to the topology to add new nodes to the system. Topologies that have borders are generally hard to expand since border processors are usually connected to I/O. It is often desirable to have a system with an expandable topology.

4. *Scalability.* The scalability of a topology refers to the extent to which it is possible to scale the topology to systems containing a large number of nodes without serious performance degradation. It is desirable to have a topology that is scalable. Topologies that have small node degrees can normally be expanded more easily than those with large degrees. However, since topologies with small degrees typically tend to

have larger diameters, the data communication time becomes large and the topology becomes unusable.

5. *Fault tolerance.* The fault tolerance of a topology refers to its ability to tolerate faulty nodes or links. Some topologies can map out faulty nodes and/or links without affecting the basic operation of the topology. These topologies are fault tolerant and are also desirable for a DMM system.

Examples of topologies that have been, or are being used in DMM systems include: rings, multi-dimensional grids or meshes, trees, completely connected arrays, pyramids and hypercubes. The more common ones are reviewed briefly below.

2-D Grids. In this topology, the nodes are interconnected together to form a two-dimensional grid. Each node is directly connected to only four neighbors in the grid; this is depicted in figure 2.4. This topology has a small degree, 4, which is constant and independent of the number of PEs in the system. This results in simple and regular interconnections. However, the diameter of the topology is relatively large, $O(\sqrt{N})$. Data movements beyond the four neighbors can become time consuming.

Completely Connected Arrays. In this topology, every node is connected to every other node. An example is shown in figure 2.5. Each node is exactly one edge away from any other node. The advantage of this topology is the high connectivity of its nodes. The diameter of the topology is the minimum possible diameter. The major disadvantages of this topology are the large degree of the topology and the fact that the number of links grows quadratically with respect to the number of the nodes.

Pyramids. In this topology the nodes are interconnected together to form a pyramid, as shown in figure 2.6. Each node is connected to four neighbors, a parent and four children nodes (some boundary nodes have fewer connections). The advantage of this topology is that the degree of each node is small (≤ 9) and that no two nodes are more than $O(\log_2 N)$

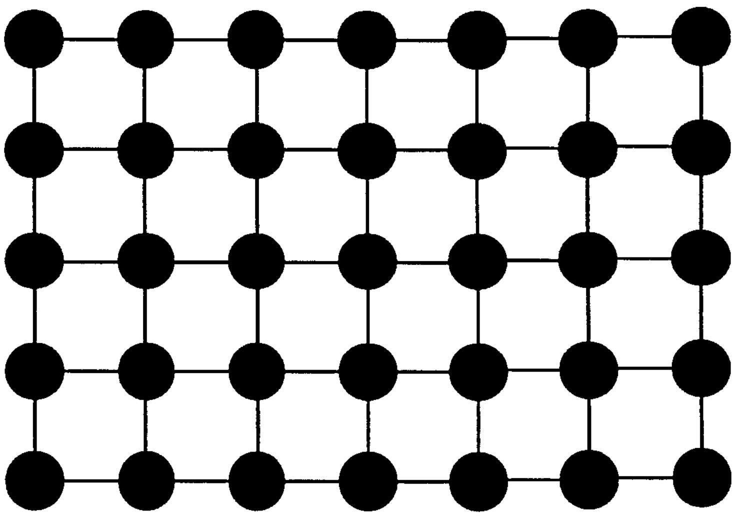
Figure 2.4. The 2-D grid topology.

steps apart. The disadvantage is congestion in the upper levels during system wide data transfers.

Hypercubes. In this topology, $N = 2^n$, identical processing elements are interconnected together using $n2^{n-1}$ communication links to form an n-dimensional hypercube graph. An n-dimensional hypercube array of nodes, or an n-cube, can be constructed and its nodes uniquely labeled using the following recursive procedure. First, a 1-cube is formed by connecting two nodes with a single communication link. One of the nodes is labeled with a 0 and the other is labeled with a 1. This is the basis step of the procedure. An n-cube is formed from two $(n-1)$-cubes using the general step of the procedure as follows: node labels in one of the $(n-1)$-cubes are prefixed with a 0 so they are of the form $0xx\ldots xx$. Similarly, node labels in the other $(n-1)$-cube are prefixed with a 1 so they are of the form $1xx\ldots xx$. Finally, the two $(n-1)$-cubes are connected with communication links between nodes that have labels differing only in their most significant bit. This labeling scheme results in a unique n-bit binary address for each node in the resulting hypercube array. The address for each node differs in exactly one bit position from that of any of its

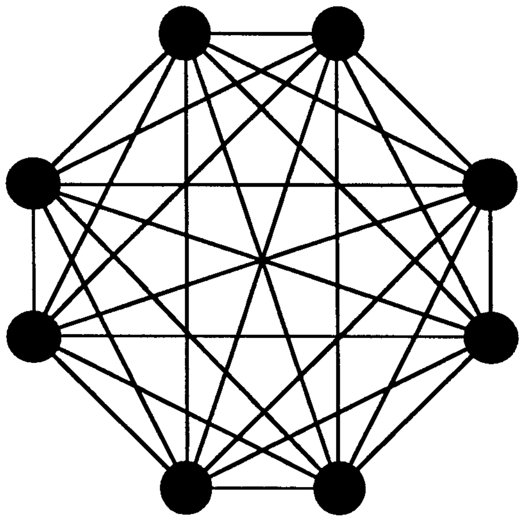

Figure 2.5. The completely connected topology.

neighbors. Since there are n such neighbors, each bit in the address can be thought of as corresponding to one dimension in the hypercube array. The hypercube array topology is illustrated in figure 2.7 for $n \leq 4$. The zero-dimensional hypercube is the conventional single processor computer.

The hypercube topology for interconnecting nodes has a number of features that makes it an attractive topology for DMMs.

1. The hypercube topology offers a unique balance between the diameter and the degree of the topology. The nodes of the hypercube topology are no more than $\log_2 N$ steps apart. Consequently, any two nodes in the hypercube array can communicate efficiently even if they are not directly connected to each other. At the same time, the degree of each node in the graph is relatively small— $\log_2 N$. This balance makes the hypercube topology compare favorably with other topologies such as grids and completely connected arrays. It also allows the topology to be scalable to a large number of processors.

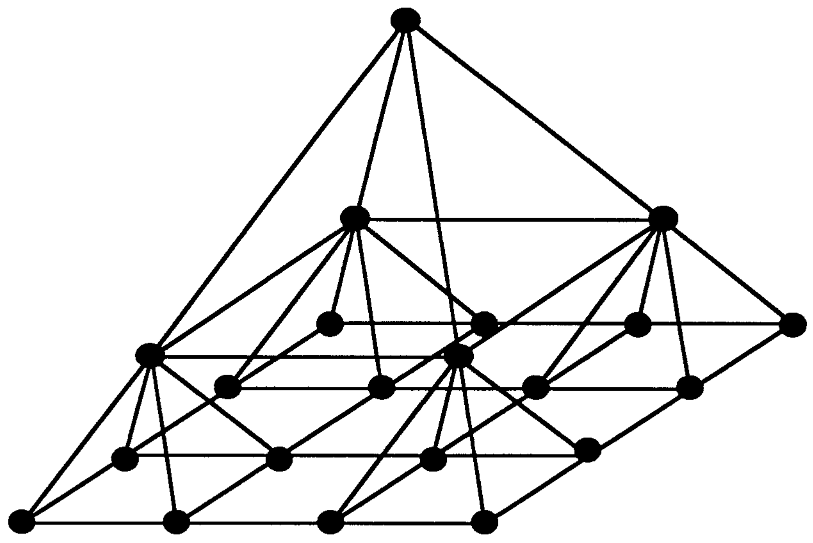

Figure 2.6. The pyramid topology.

2. The hypercube topology results in a regular and homogeneous array of nodes. All of the processors are identical and have the same view of the array. There are no special nodes that must be designated as the "borders" of the topology, as is the case in most other topologies (two-dimensional grids for example). Homogeneity makes it natural to attach an I/O channel to each node, providing the potential for high I/O bandwidth.

3. A large dimension hypercube array can be divided into smaller dimension hypercube arrays, or subcubes. Each subcube is completely independent of the other subcubes. This feature of the hypercube topology facilitates multiprogramming in which each user can be assigned a dedicated subcube [NCUB85]. It also increases the fault tolerance of the hypercube array since a faulty node can be mapped out by mapping out a subcube that contains it. Furthermore, a user program can be developed and debugged on a small size hypercube while production runs can be made on larger dimension hypercubes.

4. Finally, the hypercube topology can efficiently embed other regular topologies such that neighboring processors in those topologies are also neighboring, or at least close, in the hypercube topology. Examples include: multi-dimensional grids [ChSa86], trees [BCLR86] and pyramids [Stou86]. Since many of these regular topologies are used in a variety of applications, this feature makes the hypercube topology a good candidate for a general purpose parallel architecture.

2.1.3 Granularity of Processing Elements

The granularity of a DMM refers to the number of nodes it contains and the characteristics of their main features. It is generally difficult to quantify granularity, so the concept will be described qualitatively here.

At one end of the spectrum there are fine-grained DMMs which are characterized by a very large number of nodes. Each node is a relatively simple processor consisting of a CPU and a limited size local memory. The ratio of communication to computation tends to be large in fine-grained multiprocessors. Therefore, this granularity of multiprocessors is generally suitable for *data parallel* applications, in which sequences of simple operations are applied across very large sets of data (SIMD). Examples of fine-grained multiprocessors include the MPP [Batc80] and the Connection Machine [Hill85].

At the other end of the spectrum there are coarse-grained multiprocessors which are characterized by a small number of very powerful nodes. Each node contains a large memory and a sophisticated CPU that may even support vector operations. The ratio of communication to computation in coarse-grained multiprocessors is generally low. Coarse-grained multiprocessors are usually suitable for *control parallel* applications, in which different segments of the code execute simultaneously (MIMD). Such applications appear frequently in many areas of science and engineering. Examples of coarse-grained multiprocessors include the Mark III hypercube [PTLP85] and the Intel iPSC-VX hypercube multiprocessor which incorporates vector co-processors with each node [Inte86].

In the middle of the spectrum lie medium-grained multiprocessors, which offer a compromise between the two extremes. Medium-grained multiprocessors are characterized by a moderate number of nodes, in the range 100–1000, each a reasonably sized CPU and memory. This configuration has the potential of efficiently handling applications that are either data or control parallel. Furthermore, with the current technology, such granularity provides the maximum possible performance. Applications for medium-grained multiprocessors range from AI to scientific applications. Examples of this granularity include the Intel iPSC [Inte85] and the NCUBE/ten [NCUB85], both hypercube DMMs.

2.2 Examples of DMMs

In this section three examples of DMM systems are presented. The examples represent existing multiprocessor systems, and an overview of each system is given emphasizing the key architectural features described above.

2.2.1 The Connection Machine

The Connection Machine is a very-fine-grained DMM [Hill85]. It consists of up to 65,536 nodes that contain a bit-serial processor with 4096 bits of memory. The nodes are interconnected using a dual network topology. Groups of 16 nodes are interconnected in a two-dimensional grid topology. These groups are then connected by a 12-cube interconnection topology. The bit-serial processors execute instructions that are broadcast from a set of four control processors. Each control processor is connected to a subset of 16,384 processors in the system. The control processors are asynchronous, and each can broadcast a different set of instruction streams. The system is hence an MSIMD (Multiple SIMD) system.

The connection machine is designed for AI applications and is particularly well-suited for low-level vision applications, and for searching databases in parallel [HiSt86].

2.2.2 The Massively Parallel Processor (MPP)

The MPP is a fine-grained DMM that consists of 16,384 bit serial processors that are interconnected in a two dimensional grid topology. Each processor has 4096 bits of local memory. The processors execute instructions that are broadcast from a single control unit. A global 320 Mbytes/sec I/O channel is used to transfer programs and data to and from the MPP. The MPP is designed for real-time, low-level image processing and scene analysis. These and other applications are discussed in [Pott85].

2.2.3 The Mark III

The Mark III is a coarse-grained distributed memory multiprocessor. It consists of up to 1024 nodes that are interconnected in a 10-dimensional hypercube topology. The nodes operate asynchronously, making the Mark III an MIMD system. Each node consists of a 16 MHz Motorola 68020 microprocessor with an associated 68881 floating point co-processor. A separate 68020 is used for inter-node communication support. Each node can include up to 4 Mbytes of memory and can use special function units such as vector processors.

Two types of message transfer systems are available for the Mark III. The first is the Crystalline Operating System (CrOS–III), which provides synchronous communication between the nodes; the second is the Mercury operating system, which supports asynchronous communication between nodes.

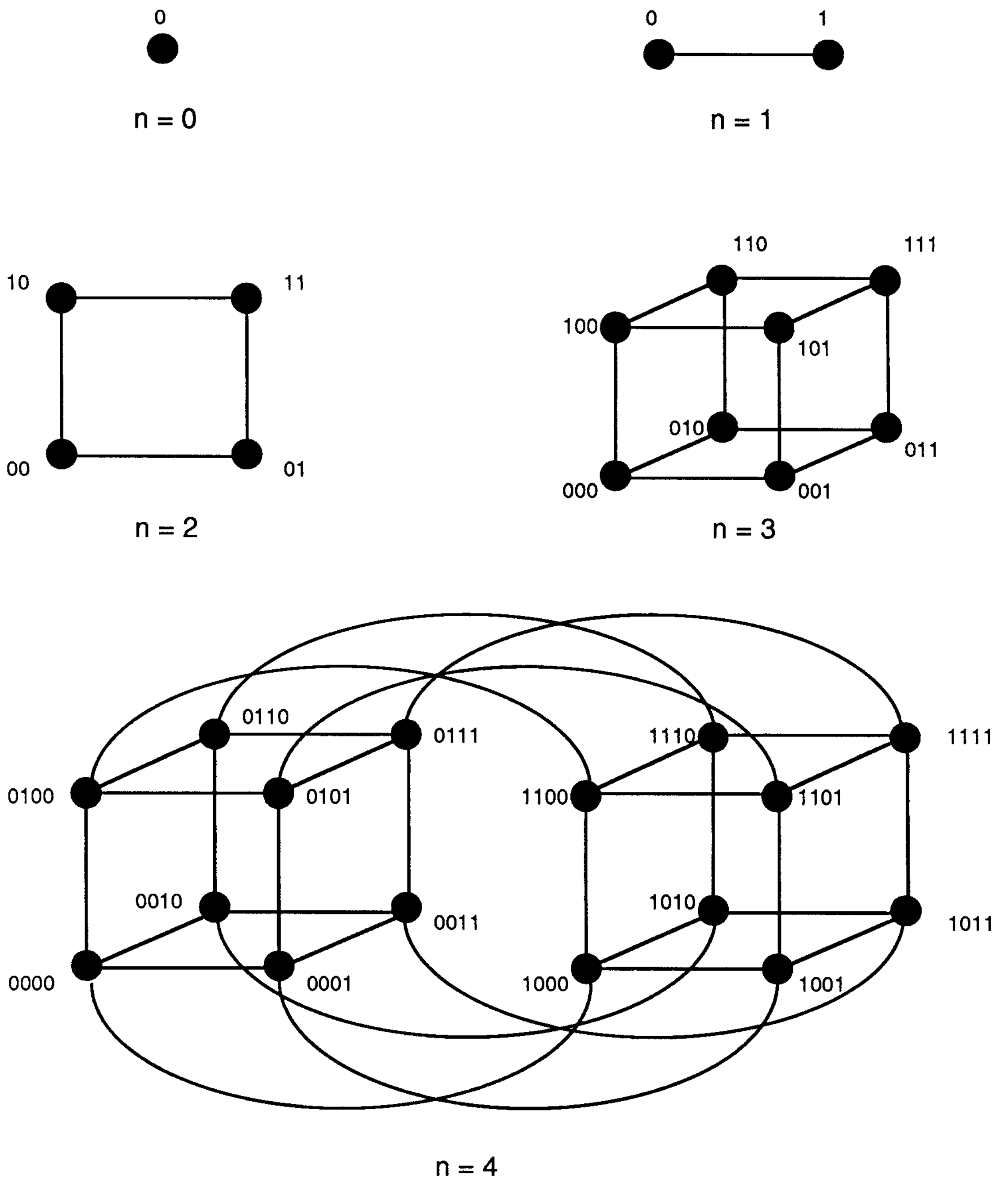

Figure 2.7. The hypercube topology.

CHAPTER 3

THE BRANCH AND BOUND ALGORITHM

There is a large class of problems in the fields of Operations Research (OR) and Artificial Intelligence (AI) for which there exist no "direct" methods of solution or only inefficient ones. Techniques for solving such problems generally involve the search for solutions in a large problem space. Examples include the traveling salesman problem [LMSK63] and heuristic search problems [Nils80].

Unguided search through the problem space is usually inefficient and impractical. This is particularly true since many of these problems are NP-hard, and the size of the problem space increases exponentially with the size of the problem. Several techniques have been developed to improve the average efficiency of the search. The most general of these techniques is the Branch and Bound (BB) algorithm [LaWo66].

The BB algorithm, as its name suggests, consists of two processes: a branching process and a bounding process. The branching process partitions the problem space, or subspaces of it, into smaller subspaces until the subspaces are small enough to be searched exhaustively for the desired solution. The bounding process of the algorithm acts to reduce the number of subspaces partitioned by the branching process. A subspace is examined by the bounding process before it is partitioned. If it can be proved that the subspace does not contain the desired solution, the subspace is *pruned* or eliminated from further consideration from the branching process.

The combined action of the branching and bounding processes reduces the extent of the search and improves the average efficiency of the BB algorithm. The branching process guides the search towards a solution by partitioning subspaces that are a more likely to contain a solution before subspaces that are less likely to contain it. The bounding process assists the search by eliminating subspaces that cannot lead to a solution before they are actually partitioned.

The branching process applied to the problem space of a given problem is performed by building a search tree, called the *BB tree*, over the problem space. The root of the tree represents the complete problem space. The nodes of the tree represent subspaces of the problem space. The branching process proceeds from the root of the tree to its leaves, partitioning subspaces into smaller and smaller subspaces. The leaf nodes represent subspaces that are small enough to be completely searched for solutions.

Subspaces of the problem space represent partial solutions to the problem. Consequently, a node of the BB tree represents a partial solution to the original problem. The branching process proceeds from the root of the tree to its leaves extending partial solutions towards more complete solutions. Each child node represents one possible way of extending its parent's partial solution towards a more complete one.

In most problems, it is impractical, if not impossible, to explicitly represent the problem space or its subspaces. A more practical representation is to use a problem specific data structure that implicitly represents the problem space. This data structure representation is referred to as a *subproblem.* Hence, a subproblem is a representation of a problem subspace or equivalently, a partial solution to the problem. The BB algorithm is generally expressed and formulated in terms of subproblems rather than in terms of problem subspaces.

The above process of building a BB tree is illustrated in figure 3.1. The figure shows the BB tree for a simple example. The original problem P_0 is at the root of the tree.

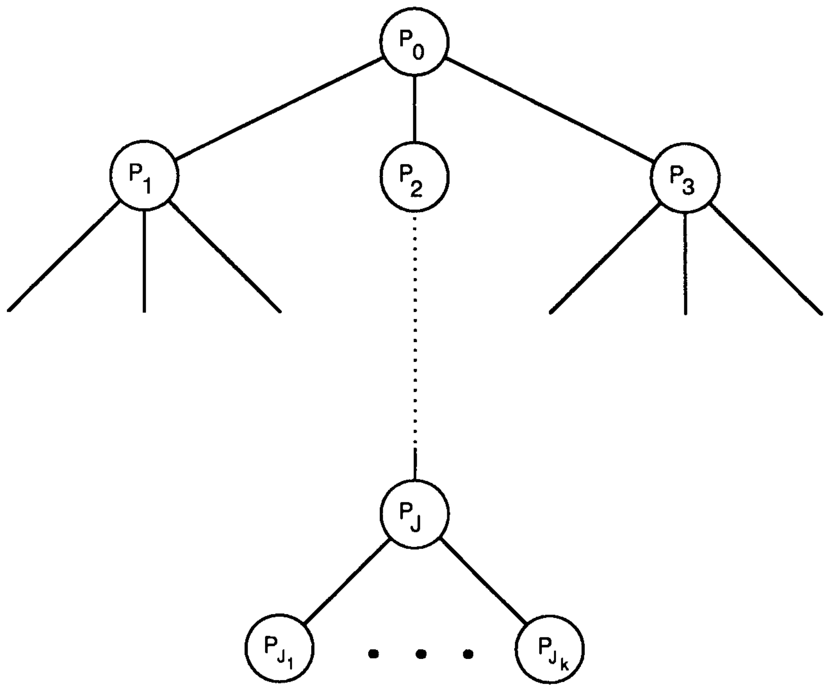

Figure 3.1. BB tree of a simple example.

P_0 is then partitioned into three smaller subproblems P_1, P_2 and P_3. These subproblems are represented as the three children of P_0. Each one of the three subproblems is further partitioned into yet smaller subproblems, as shown in the figure. In general, a subproblem P_j is partitioned into k smaller subproblems $P_{j_1}, \ldots, P_{j_k}$ which are represented as the k children of P_j. The process of partitioning a subproblem into smaller subproblems and adding the new subproblems to the BB tree is referred to as *expanding* that subproblem.

The remainder of this chapter is devoted to the formulation of the BB algorithm and its use in a number of application problems. In the next section a more formal description of the BB algorithm is presented, and in section 3.2 the use of the BB to solve four application problems is given.

3.1 Formulation of the BB Algorithm

The general class of problems that are solved by the BB algorithm can be divided into two subclasses: *decision* problems and *optimization* problems. In decision problems the objective is to determine the existence of one solution that satisfies a set of constraints. Examples of decision problems include: theorem proving, game playing and rule-based expert systems. In optimization problems, on the other hand, it is desired to optimize an objective function subject to a number of given constraints. Examples of optimization problems include: the traveling salesman problem, integer programming and job-shop scheduling.

A problem instance can be equivalently represented in the form of a decision problem or an optimization problem [GaJo79]. Hence, we represent our problem instances in the generic form of constrained optimization problems without any loss of generality. Furthermore, for the consistency of presentation, the minimization form of the problem is assumed. Hence, the constrained optimization problem takes the form:

$$\begin{array}{lll}\text{Minimize} & C_o(\mathbf{x}) & \\[1ex] \text{subject to} & G_i(\mathbf{x}) \geq 0 & i = 1, 2, \ldots, m \\ \text{and} & \mathbf{x} \in \mathbf{X} & \end{array}$$

where $\mathbf{X}$ is the permissible domain of optimization, often the Euclidean n-space, and $\mathbf{x}$ denotes a vector $(x_1, x_2, \ldots, x_n)$. A solution vector $\mathbf{x}$ that satisfies the constraints and lies within the domain of optimization $\mathbf{X}$ is called a *feasible solution.* A feasible solution for which the objective function $C_o(\mathbf{x})$ is a minimum is called an *optimal solution.* In general, there may be more than one optimal solution.

The formulation of the BB algorithm consists of five major components: the BB tree, which is the basic data structure used by the algorithm, and four procedures that are used to implement the branching and bounding processes of the algorithm: the *selection* procedure, the *branching* procedure, the *elimination* procedure, and the *termination* procedure.

The first two procedures implement the branching process while the last two implement the bounding process.

3.1.1 The BB Tree

The branching process in a BB algorithm can be represented by a rooted tree $\mathcal{B} = (\mathcal{P}, \mathcal{E})$. The tree consists of a set of nodes $\mathcal{P}$ and a set of edges $\mathcal{E}$. The nodes represent partitioned subproblems that are generated by the branching process. The edges represent the action of the branching process on the subproblems. The original problem, denoted by P_0, is at the root of the tree $\mathcal{B}$. For any two subproblems P_i and $P_j \in \mathcal{P}$, the directed arc $(P_i, P_j) \in \mathcal{E}$ if and only if P_j is directly generated from P_i by the branching process. P_i is called the *parent* of P_j, and P_j is, hence, called a *child* of P_i. A subproblem P_j is said to be a *descendant* of a subproblem P_i (or equivalently, P_i is said to be the *ancestor* of P_j) if there exists a sequence of S subproblems $P_{i_1}, P_{i_2}, \ldots, P_{i_S}$ such that $(P_i, P_{i_1}) \in \mathcal{E}$, $(P_{i_1}, P_{i_2}) \in \mathcal{E}$, ..., and $(P_{i_S}, P_j) \in \mathcal{E}$. A set of subproblems is said to be *independent* if no subproblem is a descendant of any other subproblems in the set.

The level of a subproblem P_i, denoted by $L(P_i)$, is defined as the length of the path, measured by the number of arcs, from the root P_0 to P_i in $\mathcal{B}$. P_0 is defined to be at level 1. The set of leaf nodes in the BB tree is denoted by $\mathcal{T}$.

The set of subproblems that has been generated by the branching process but not yet examined by that process nor deleted by the bounding process is referred to as the *set of active subproblems* and is denoted by $\mathcal{A}$. The branching process examines that set, removes a subproblem and expands it, adding new subproblems back to the set. The set of active subproblems is, therefore, an independent set of subproblems.

The real valued cost function $f : \mathcal{P} \rightarrow \mathbf{E} \cup \{\infty\}$, where $\mathbf{E}$ is the set of non-negative real numbers, denotes the value of the best solution that can be obtained from any subproblem

$P_i \in \mathcal{P}$. The function f can be defined recursively as follows:

$$f(P_i) \;=\; \min\{f(P_{i_j}),\; j = 1,\ldots,k\},$$

where P_{i_j} denotes the j^{th} subproblem partitioned directly from P_i, and k is the total number of these subproblems. That is,

$$k = |\{(P_i, x) \mid (P_i, x) \in \mathcal{E} \;\; \forall x\}|.$$

The value of f is computed at the leaf subproblems by directly evaluating the solutions represented by these subproblems. Consequently, the value of the function f for any subproblem P_i is not known until all leaf subproblems in the subtree rooted at P_i have been evaluated. The value of f at the root node in the BB tree, therefore, denotes the value of the objective function at the optimal solution vector. That is, $f(P_0) = C_o(\mathbf{x})$. The function $f(P_i)$ takes the value ∞ when there are no feasible solutions from P_i.

Each subproblem P_i is also characterized by a value that is computed from a lower bound function $g : \mathcal{P} \rightarrow \mathbf{E} \cup \{\infty\}$. The lower bound function must have the following properties:

1. $g(P_i) \leq f(P_i)$ for all $P_i \in \mathcal{P}$,
2. $g(P_i) = f(P_i)$ for $P_i \in \mathcal{T}$, and
3. $g(P_j) \geq g(P_i)$ for $(P_i, P_j) \in \mathcal{E}$

In other words, the function g is a lower bound estimate of the cost function f, is exact when a subproblem is terminal (i.e., the subproblem represents a feasible solution or the subproblem can never lead to one), and never decreases for descendant subproblems.

The function g can be used to guide the search of the BB algorithm by providing an estimate of f, as will be described in the following section. Hence, it is desired to make g as close an estimate to f as possible so as to guide the search to a solution as quickly as possible; the more accurate g is, the smaller is the number of subproblems expanded by the BB algorithm [Pear84]. However, making g more accurate usually implies that it becomes more difficult to evaluate. Clearly the evaluation of g should not amount to the complete evaluation of f, or even be close to it. However, g should provide a good estimate of f to guide the search effectively. There is a tradeoff between the accuracy of g and its evaluation difficulty [Pear84].

3.1.2 The Selection Procedure

The selection procedure examines the set of active subproblems $\mathcal{A}$ and selects one subproblem from that set for expansion. The procedure selects the subproblem that is most likely to lead to the optimal solution among those that belong to the set of active subproblems.

The selection procedure is defined in terms of a *selection function* $s_h : \pi \rightarrow \mathcal{P}$ such that $s_h(\mathcal{A}) \in \mathcal{A}$, where π denotes the family of all independent sets in $\mathcal{P}$. The selection function is based on the heuristic function $h : \mathcal{P} \rightarrow \mathbf{E} \cup \{\infty\}$ which assigns a figure of merit to each subproblem in $\mathcal{P}$. The selection function s_h always selects the subproblem with the smallest value of h. That is,

$$h(s_h(\mathcal{A})) = \min\{h(P_i) \mid P_i \in \mathcal{A}\}.$$

The heuristic function h is assumed to be *monotone*. That is,

$$h(P_j) \geq h(P_i) \qquad \text{if } P_j \text{ is a descendant of } P_i. \tag{3.1}$$

The heuristic function is also assumed to be *unambiguous*. That is,

$$h(P_i) \neq h(P_j) \qquad \text{for} \quad P_i \neq P_j \text{ and } P_i, P_j \in \mathcal{A}. \tag{3.2}$$

The selection function determines the order in which the subproblems are selected for expansion from the set of active subproblems. In other words, it determines the order in which the nodes of the BB tree are expanded. Therefore, the selection function is in effect a *search strategy* for the BB algorithm. Three search strategies are generally used in BB algorithms: *best-first* search, *depth-first* search and *breadth-first* search.

In best-first search, the heuristic function h is the same as the lower bound function g. That is,

$$s_g(\mathcal{A}) = \{P_i \in \mathcal{A} \mid g(P_i) = \min\{g(P_j) \mid P_j \in \mathcal{A}\}\}.$$

Therefore, in this search strategy, subproblems with smaller lower bounds are selected before subproblems with larger lower bounds. In depth-first search, the selection function is defined such that

$$s_d(\mathcal{A}) = \{P_i \in \mathcal{A} \mid L(P_i) = \max\{L(P_j) \mid P_j \in \mathcal{A}\}\}.$$

That is, subproblems that are deeper in the tree are selected before subproblems that are closer to the root of the tree. Finally, in breadth-first search, the selection function is defined such that

$$s_b(\mathcal{A}) = \{P_i \in \mathcal{A} \mid L(P_i) = \min\{L(P_j) \mid P_j \in \mathcal{A}\}\}.$$

That is, subproblems with smaller level numbers (subproblems that are closer to the root) are selected before subproblems that are deeper in the tree.

The above three strategies are illustrated using the BB tree shown in figure 3.2. The number inside each subproblem in the tree designates the number of that subproblem. The number to the right of each subproblem denotes the value of the lower bound function for that subproblem. The optimal solution is obtained at subproblem 6. If depth-first strategy is used, then the order of subproblems expanded is $0 \rightarrow 1 \rightarrow 4 \rightarrow 9 \rightarrow 11 \rightarrow 5 \rightarrow 10 \rightarrow 2$. If breadth-first strategy is used, then the order of subproblems expanded is $0 \rightarrow 1 \rightarrow 2 \rightarrow 4$. Finally, if best-first strategy is used, then the order becomes $0 \rightarrow 1 \rightarrow 4 \rightarrow 2$.

It is generally the case that equation 3.2 does not hold for the above search strategies, and more than one active subproblem may have the same lower bound or the same level

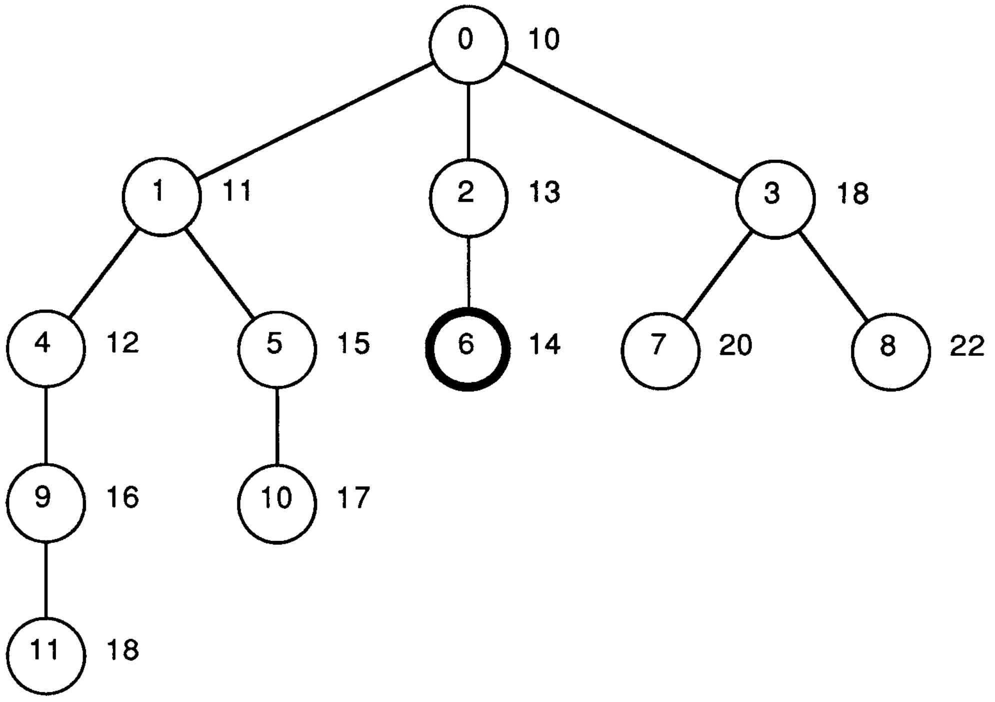

Figure 3.2. A BB tree.

number. Therefore, a tie-breaking mechanism must be be employed. A path number is defined to uniquely identify each subproblem in the set of active subproblems [Li85]. The path number, $e_1e_2 \dots e_d$, of a subproblem P_i, denoted by $p(P_i)$, is a sequence of d integers representing the path from the root to P_i, where d is the maximum level in the BB tree. The path number for a subproblem in the BB tree is defined recursively as follows:

1. The root P_0, which is at level 1, has a path number $p(P_0) = 00 \dots 0$.

2. If a subproblem P_i at level l has a path number $p(P_i) = e_1e_2 \dots e_l 0 \dots 0$, then P_{i_j}, the j^{th} child of P_i counting from the left, has a path number $p(P_{i_j}) = e_1e_2 \dots e_l(j-1)0 \dots 0$.

A path number $p(P_1) = e_1^1 e_2^1 \dots e_d^1$ can be compared to a path number $p(P_2) = e_1^2 e_2^2 \dots e_d^2$ using the relations '<' and '=' as follows:

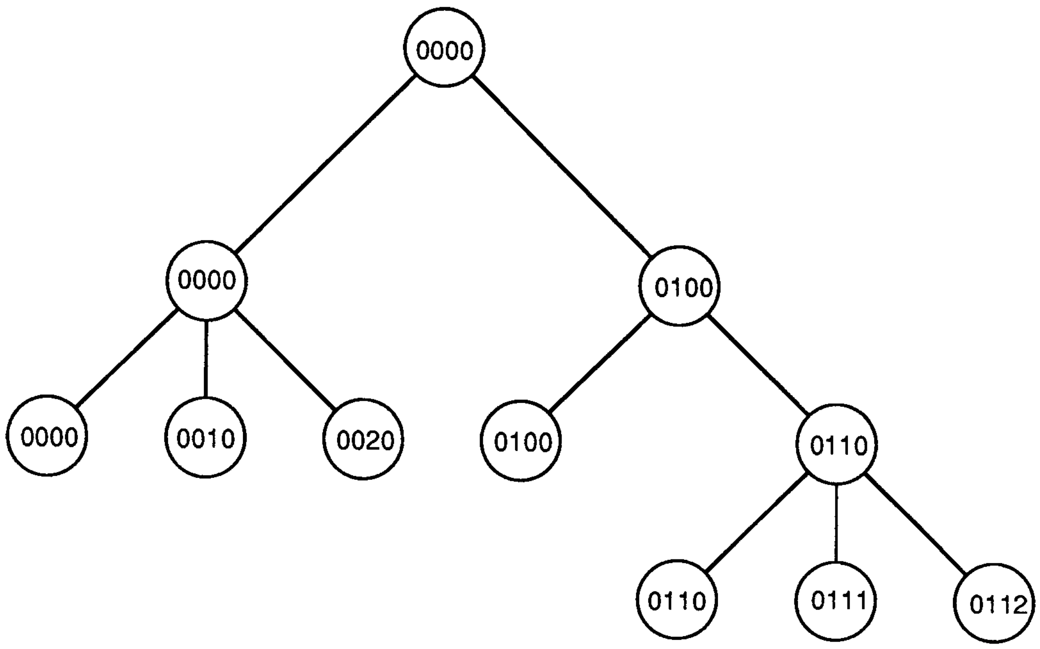

Figure 3.3. A BB tree with path numbers.

1. $p(P_1) = p(P_2)$ if $e_i^1 = e_i^2$ for $1 \leq i \leq d$.

2. $p(P_1) < p(P_2)$ if there exits j, $1 \leq j \leq d$ such that $e_i^1 = e_i^2, 1 \leq i < j$ and $e_j^1 < e_j^2$.

An example showing a BB tree with its nodes labeled by path numbers is depicted in figure 3.3. Subproblems can have equal path numbers only if they have ancestor-descendant relationship. Since these subproblems cannot coexist in the set of active subproblems, each active subproblem will have its own unique path number.

The path number can be included in the definition of the heuristic function h in order for h to become unambiguous. In the case of depth-first, the heuristic function h becomes

$$h(P_i) = p(P_i).$$

In the case of breadth-first,

$$h(P_i) = (L(P_i), p(P_i)).$$

In this case, subproblems that are at the same level in the tree will be searched left to right. Finally, in the case of best-first,

$$h(P_i) = (g(P_i), L(P_i), p(P_i)). \tag{3.3}$$

In this case, subproblems that have the same lower bound are searched in a breadth-first fashion. Alternatively, the heuristic function for the case of best-first can be defined as

$$h(P_i) = (g(P_i), p(P_i)), \tag{3.4}$$

in which case subproblems that have the same lower bound are searched in a depth-first fashion.

It is often convenient to view the set of active subproblems as being arranged in a list $\mathcal{L}$ referred to as the *list of active subproblems.* Different search strategies of the BB algorithm can then be viewed as different ways of maintaining that list. For breadth-first search, the list is maintained in a first-in-first-out order. For depth-first search, the list is maintained in a last-in-first-out order. For best-first strategy, the list is maintained in increasing order of lower bounds. In the general heuristic search, the list can be viewed as being maintained in increasing order of the heuristic function h. Therefore, the subproblem that is selected by the selection procedure is always the first subproblem on the list of active subproblems.

3.1.3 The Branching Procedure

The branching procedure is used by the BB algorithm to decompose problem subspaces into smaller subspaces. The branching procedure examines the subproblem selected for expansion by the selection procedure and creates new subproblems from it. The procedure performs this function by heuristically selecting some unassigned parameters in the subproblem representation and then assigning alternative values for these parameters.

In general, the branching procedure is highly problem dependent and is critical to the performance of the BB algorithm.

3.1.4 The Elimination Procedure

The elimination procedure is used by the BB algorithm to bound the number of subproblems examined by the branching process of the algorithm. It achieves this goal by eliminating subproblems that cannot lead to better optimal solutions than ones already known. This is accomplished by employing one or more bounding tests. These tests generally can be divided into three types: *lower bound* tests, *dominance* tests, and *equivalence* tests.

Lower Bound Tests

A lower bound test employs a special subproblem which is referred to as the *incumbent*, and which will be denoted by z. The incumbent is used to store the best feasible solution discovered during the search process at any point in time.

The lower bound test examines the lower bound of a new subproblem generated by the branching procedure, and compares it to the value of the best solution of the incumbent. If the lower bound of that subproblem exceeds that of the incumbent, this subproblem can be eliminated from further consideration, as it can never lead to a better solution than the one already found. That is, a subproblem P_i is eliminated if

$$g(P_i) \geq g(z), \qquad P_i \in \mathcal{A}.$$

When a new subproblem represents a feasible solution, the value of that feasible solution is compared to that of the incumbent. If the new subproblem represents a better solution, then the incumbent is replaced by the new subproblem.

It is possible to obtain a suboptimal solution with some guaranteed accuracy by relaxing the above lower bound test [Ibar76a]. BB algorithms with such relaxed lower bound tests are referred to as *approximate* BB algorithms. The lower bound test is relaxed as follows: a subproblem P_i is eliminated from further consideration if

$$g(P_i) \geq g(z) - \epsilon(z),$$

where $\epsilon(z) : \mathbf{R} \rightarrow \mathbf{R}$ ($\mathbf{R}$ is the set of real numbers) is referred to as the *allowance function*, and it must satisfy the following conditions,

1. $\epsilon(z) \geq 0$.

2. $g(z_1) \leq g(z_2) \Longrightarrow g(z_1) - \epsilon(z_1) \leq g(z_2) - \epsilon(z_2)$.

The allowance function specifies the allowable deviation of the suboptimal solution value from the optimal solution value. In fact, the suboptimal solution z_A obtained using approximate BB algorithms can be shown to deviate from the exact optimal solution z_o by:

$$g(z_A) - \epsilon(z_A) \leq g(z_o) \leq g(z_A).$$

Examples of allowance functions include:

1. Absolute error deviation allowance function: $\epsilon(z) = \epsilon, \ \ \epsilon \geq 0$. The suboptimal solution is guaranteed to deviate at most by ϵ from the exact one.

2. Relative error deviation allowance function: $\epsilon(z) = \frac{\epsilon g(z)}{1+\epsilon}, \ \ \epsilon \geq 0, \ \ z \geq 0$. This guarantees a relative deviation of $\frac{\epsilon}{1+\epsilon}$.

Dominance Tests

A dominance test is performed to eliminate subproblems that cannot lead to better optimal solutions compared to other subproblems already examined. More formally, a subproblem P_i is said to *dominate* a subproblem P_j if P_j is known not to provide a better feasible solution than the one that can be obtained from P_i. This binary relation between P_i and P_j is called a *dominance* relation and is denoted by $P_i\mathbf{D}P_j$. A test based on the dominance relation eliminates a subproblem P_j from further consideration from the selection process if there exists a subproblem P_i such that $P_i\mathbf{D}P_j$. A dominance relation must satisfy the following conditions [Ibar77]:

1. $P_i\mathbf{D}P_j$ implies that $f(P_i) \leq f(P_j)$ and that P_i is not a proper descendant of P_j.
2. $\mathbf{D}$ is a partial ordering. (i.e., reflexive, antisymmetric, and transitive).
3. $P_i\mathbf{D}P_j$ and $P_i \neq P_j$ imply that there exists a descendant P_i' (including P_i) of P_i such that $P_i'\mathbf{D}P_j'$ for any proper descendant P_j' of P_j. That is, if $P_i\mathbf{D}P_j$, then there exists a subproblem in the subtree rooted at P_i that dominates all subproblems in the subtree rooted at P_j.

Since a dominance relation is only a partial ordering, it is possible that neither the $P_i\mathbf{D}P_j$ nor the $P_j\mathbf{D}P_i$ holds. In this case P_i and P_j are said to be *incomparable.* A subproblem P_i is said to be a *current dominating* subproblem if it has been generated but has not been dominated so far. It then follows that all current dominating subproblems are incomparable to each other. In order to apply dominance tests, a set of current dominating subproblems (denoted by $\mathcal{D}$) has to be maintained. When a subproblem P_j is generated, it is compared against the subproblems in $\mathcal{D}$. If it is incomparable to each one of them, then it is added to $\mathcal{D}$. If it is dominated by $P_i \in \mathcal{D}$, then it is deleted. Finally, if it dominates $P_i \in \mathcal{D}$, then P_i is deleted from $\mathcal{D}$ and P_j is added to $\mathcal{D}$.

Dominance tests can greatly reduce the number of subproblems expanded, and hence the execution time of a BB algorithm. This is accomplished by storing more information, the set $\mathcal{D}$, and represents a typical time-space tradeoff. This tradeoff must be taken into consideration when dominance tests are employed, particularly if the dominance relation is weak and, hence, most of the subproblems are incomparable. Furthermore, in NP-hard problems, the size of $\mathcal{D}$ can be exponentially large [Li85]. In this study, dominance tests are assumed to be inactive.

Equivalence Tests

Equivalence tests are special cases of dominance tests that are frequently used in AI applications [Ibar78a]. An equivalence test eliminates a subproblem P_j from further consideration of the branching process if there exists a subproblem P_i that is equivalent to P_j and has already been examined by the branching process. That is, P_j is deleted if it is equivalent to P_i which has already been expanded by the BB algorithm. The equivalence of two subproblems is determined by the equivalence of their representation.

3.1.5 The Termination Procedure

The termination procedure is used by the BB algorithm to eliminate subproblems that will not eventually lead to any feasible solutions. The termination procedure employs techniques that are highly problem dependent and require considerable knowledge about the problem domain. This knowledge is used to determine if a subproblem represents a partial solution that can be extended to a complete feasible solution.

3.1.6 Outline of the BB Algorithm

The following is an outline of how the algorithm uses the selection, branching, elimination and termination procedures to obtain the optimal solution to a problem.

1. **Initialization.**
   (a) The set of active subproblems is initialized to contain the original problem.
   (b) The lower bound of the incumbent is initialized to ∞. The subproblem defined by the incumbent is initially undefined.

2. **Selection.**

   (a) The subproblem with the smallest value of the heuristic function h is selected from the set of active subproblems.

   (b) The subproblem is removed from the set.

3. **Branching.**

   The branching procedure is used to generate new smaller subproblems from the one selected in (2). The lower bounds of the new subproblems are calculated.

   Steps 4 to 7 are repeated for each of the new subproblems generated by the branching procedure above.

4. **Termination test.**

   The subproblem is evaluated to determine if it can lead to a feasible solution. If not, it is deleted.

5. **Feasibility.**

   (a) The subproblem is evaluated to determine if it is a feasible solution. If it is, and its lower bound is smaller than that of the incumbent, it replaces the incumbent. Otherwise, it is deleted.

   (b) If the incumbent is updated in 5(a), then all the subproblems in the set of active subproblems $\mathcal{A}$ whose lower bounds are greater than that of the new incumbent are deleted from the set of active subproblems.

6. **Lower bound test.**

   If the lower bound of a new subproblem is greater than the lower bound of the incumbent, the subproblem is deleted. Otherwise it is added to $\mathcal{A}$ if dominance tests are not being employed.

7. **Dominance test.**

   (a) If a new subproblem is dominated by a subproblem in $\mathcal{D}$, then the subproblem is deleted.

   (b) If a new subproblem dominates a subproblem $P_j \in \mathcal{D}$, then P_j is deleted. The subproblem is added to $\mathcal{D}$ and $\mathcal{A}$.

   (c) If a new subproblem is incomparable, it is added to $\mathcal{D}$ and $\mathcal{A}$.

8. **Algorithm termination.**

   If the set of active subproblems is not empty, steps (2)-(7) are repeated. Otherwise the algorithm terminates. The optimal solution is stored in the incumbent.

A number of researchers have examined BB algorithms and their properties. Theoretical comparisons of search strategies in BB algorithms and their effects on the average performance of BB algorithms are developed in [Ibar76b]. Various search strategies are shown to be special cases of the general heuristic search. It is also shown that when the value of the lower bound function of a subproblem is unique (i.e., the lower bound function is one-to-one), then the performance of a BB algorithm under a best-first search strategy is better than the performance of the same algorithm under breadth-first or depth-first strategies.

The effect of the accuracy of the lower bound function on the average performance of some BB algorithms for decision problems was conducted by Pearl in [Pear84]. The analysis shows that linear errors in the lower bound function cause the number of nodes generated by the BB algorithm to grow exponentially.

The computational efficiency of approximate BB algorithms is studied in [Ibar76a]. It is shown that under proper conditions the number of subproblems examined using the approximate algorithm is smaller than the number of nodes examined using the exact one. In [WaYu85] it is shown that a linear reduction in the accuracy of the solution can result in an exponential reduction in the total number of subproblems examined by the BB algorithm. This result seems to be consistent with Pearl's result in [Pear84], since changes in the accuracy of the solution can be modeled as changes in the accuracy of the lower bound function and vice versa.

Dominance tests in BB algorithms and their effect on the performance of the BB algorithm are discussed in [Ibar77]. It is shown that under certain conditions dominance tests can enhance the performance of BB algorithms. In fact, a stronger dominance relation implies improved performance.

The above formulation of BB algorithms is also recognized as a general formulation for many heuristic procedures for searching AND/OR graphs, game trees and state-space representations in the area of AI [KuKa83].

Approximate stochastic models of BB algorithms under best-first and depth-first search strategies are developed in [WaYu82, WaYu85]. The models allow the estimation of the average number of subproblems examined by the BB algorithm as well as the average memory space required. The results are used to evaluate the performance of BB algorithms in virtual memory environments and to aid the design of virtual memory support for BB algorithms in [YuWa83, YuWa84].

3.2 Examples of the BB Algorithm

The BB algorithm has been used to solve a variety of problems in many application areas. A survey of BB techniques and their applications in the area of mathematical programming is given in [LaWo66], and BB algorithms for solving a variety of AI problems are described in [Nils80]. The BB algorithm has been used to solve the traveling salesman problem [LMSK63], integer linear programming problems [GeMa72], the knapsack problem [InKo77], the facility allocation problem [EfRa66], and scheduling problems [Lens76], to cite just a few examples.

In this section, four examples of BB algorithms are described; three examples from the Mathematical Programming area, and one example from the Artificial Intelligence area. The BB algorithm used to solve each problem is described with its basic components illustrated using the general formulation given in the previous section. Emphasis is given to illustrating these basic components rather than problem specific details.

3.2.1 The 0–1 Integer Linear Programming (ILP) Problem

The 0–1 ILP problem is an optimization problem in which it is desired to minimize the value of a linear objective function $f(x_1, x_2, \ldots, x_n)$ subject to a set of constraints. The variables $(x_1, x_2, \ldots, x_n)$, which are referred to as the decision variables, can take only the values 0 or 1. The problem can be more formally stated as follows:

$$\begin{array}{lll} \text{Minimize} & f = \sum_{j=1}^{n} c_j x_j & \\ \text{subject to} & \sum_{j=1}^{n} a_{ij} x_j \geq b_i & i = 1, 2, \ldots, m \\ & x_j \in \{0, 1\} & j = 1, 2, \ldots, n. \end{array}$$

It can be assumed, with no loss of generality, that the coefficients c_j, $j = 1, 2, \ldots, n$ are non-negative.

The BB algorithm used to solve the 0–1 ILP problem is known as *implicit enumeration* [Taha75, WuCo80]. There are n binary variables and the problem could, conceivably, be solved by enumerating all of the 2^n possible solutions. The bounding process of the BB algorithm can, however, discard many of these 2^n solutions and not explicitly enumerate them; hence the name "implicit enumeration."

The implicit enumeration algorithm can be described using the following simple terminology: the assignment of a 0 or 1 value to each one of the decision variables gives one of the 2^n possible *solutions*; the assignment of values to some but not all of the decision variables gives a *partial solution.* A partial solution represents a subspace of the solution space or a *subproblem* of the original problem. The decision variables that are assigned values in a partial solution are said to be *fixed.* In contrast, the decision variables with no assigned values are said to be *free.* A *completion* is made by assigning a value of 0 or 1 to one of the free variables.

Since $c_j \geq 0$ for all j, a lower bound f_L on the value of the objective function for any subproblem can be computed by assigning the value of 0 to each free variable. Hence,

$$f_L = \sum_{\substack{fixed \\ variables}} c_j x_j. \tag{3.5}$$

Furthermore, a constraint can be satisfied if and only if

$$\sum_{\substack{free \\ variables}} max(a_{ij}, 0) \geq b_i - \sum_{\substack{fixed \\ variables}} a_{ij} x_j \qquad i = 1, 2, \ldots, m. \tag{3.6}$$

Therefore, it is possible to check the infeasibility of any subproblem by applying equation 3.6 to the constraints of the problem. Assigning the value of 0 to each free variable in a subproblem makes a special completion that is referred to as the *lower bound completion.* The feasibility of the lower bound completion can be checked using equation 3.6, which reduces to:

$$\sum_{\substack{fixed \\ variables}} a_{ij} x_j \geq b_i \qquad i = 1, 2, \ldots, m. \tag{3.7}$$

The implicit enumeration BB algorithm for the 0–1 ILP problem can be formulated in the following steps:

Step 1 The incumbent, denoted by z, is created to contain the best feasible solution found during the search. The lower bound of z, denoted here by f_z, is initialized to ∞. The initial subproblem, in which all the variables are free, is created. A list of active subproblems is created, and the initial subproblem is inserted on it.

Step 2 The subproblem whose lower bound is the smallest among all subproblems on the list of active subproblems is selected.

Step 3 A free variable x_k in the selected subproblem is chosen and is used to generate two new subproblems. The first subproblem is generated by making the completion $x_k = 0$. The second is generated by making the completion $x_k = 1$. The variable x_k is now fixed. The variable selected is the free variable with the smallest index k.

Step 4 The lower bound of each new subproblem is calculated using equation 3.5. The possibility of leading to a feasible solution for each subproblem is checked using equation 3.6. The feasibility of the lower bound completion also is checked using equation 3.7.

Step 5 A subproblem is deleted if any one of the following conditions is true:

a. $f_L \geq f_z$.

b. The subproblem cannot lead to a feasible solution.

c. There are no remaining free variables.

d. The lower bound completion is feasible. In this case, the incumbent is replaced by the lower bound completion if $f_L < f_z$, and all subproblems on the list of active subproblems with $f_L \geq f_z$ are deleted. This is done to speed the discovery of feasible solutions.

A subproblem that is not deleted is added to the list of active subproblems.

Step 6 Steps 2–5 are repeated as long as there are subproblems on the list of active subproblems. When the list is empty, the algorithm terminates. The optimal solution is the current incumbent.

The implicit enumeration algorithm illustrates the steps of the general BB formulation. Step 1 of the algorithm implements the initialization step. Step 2 implements selection. The lower bound of a subproblem is used as the selection heuristic function making the search strategy of the algorithm best-first. Steps 3 and 4 implement branching. Step 5.a implements the lower bound test. Steps 5.b and 5.c implement the termination test. Step 5.d implements the feasibility test. Finally, step 6 implements algorithm termination.

3.2.2 The Integer Linear Programming (ILP) Problem

The ILP problem is an optimization problem in which it is desired to minimize the value of an objective function subject to a set of constraints, where the variables of optimization can take only integer values. The problem can be formulated as follows:

$$\begin{array}{lll} \text{Minimize} & \sum_{j=1}^{n} c_j x_j & \\ & & \\ \text{subject to} & \sum_{j=1}^{n} a_{ij} x_j \geq b_i & i = 1, 2, \ldots, m \\ \text{where} & x_j \text{ are non–negative integers} & j = 1, 2, \ldots, n. \end{array}$$

The problem is similar to the ordinary Linear Programming (LP) problem except that the variables are restricted to have non-negative integer values. The LP problem obtained by dropping the integrality constraints from the ILP problem is referred to as the *corresponding* LP problem. The dropping of the integrality constraints is referred to as a *relaxation.*

The BB algorithm for solving the ILP problem is described in [GeMa72, Taha75]. The algorithm is a modification of the Land-Doig algorithm and is referred to here as the *relaxation* algorithm because of its technique of relaxing the integrality constraints on the variables of optimization and then of solving the corresponding LP problem using the *simplex* method [Dant63]. If the resulting solution is integral, then it is feasible. Otherwise, a variable x_j that has a non-integral value a is selected and is used to partition the problem into two new subproblems. In the first subproblem, the variable x_j is restricted to the next lower integral value (i.e., $x_j \leq \lfloor a \rfloor$ is added to the set of constraints of the parent subproblem). In the second subproblem, the same variable x_j is restricted to the next higher integral value (i.e., $x_j \geq \lceil a \rceil$ is added to the set of constraints of the parent subproblem). The lower bound for any subproblem is the value of the optimal simplex solution to its corresponding LP problem.

The relaxation BB algorithm for the ILP can be outlined as follows:

1. **Initialization.** The set of active subproblems is initialized to contain the original ILP problem. The lower bound of the incumbent is initialized to ∞.
2. **Selection.** The subproblem with the smallest lower bound in the set of active subproblems is selected.
3. **Termination.**
   (a) The corresponding LP is solved using the simplex method.
   (b) If there is no feasible solution to the corresponding LP subproblem, the subproblem is deleted.
4. **Feasibility.**
   (a) If the simplex solution of the LP subproblem is integral, then it is feasible. If the value of the integral solution is smaller than that of the incumbent, then the incumbent is replaced by the new feasible solution.
   (b) If the incumbent is updated, then all subproblems on the list of active subproblems with the lower bounds greater than or equal to the new incumbent are deleted.
5. **Branching.** If the simplex solution of the LP subproblem is not integral, a non-integer variable is used to generate two new subproblems, as described above. The lower bounds of the new subproblems are calculated.

6. **Bounding.** For each newly generated subproblem, if the lower bound of the subproblem is less than the incumbent, the subproblem is inserted on the list of active subproblems. Otherwise, the subproblem is deleted.

Steps 2–6 are repeated until the list of active subproblems is empty. The optimal solution is stored in the incumbent.

A number of methods exist for the selection of the non-integer variable to branch from. These methods have been reviewed in [Taha75] and will not be described here. In this implementation of integer programming, the non-integer variable with the smallest index is selected to branch from.

The relaxation BB algorithm can be illustrated using the following simple ILP problem:

$$\begin{array}{ll}
\text{Minimize} & 6x_1 + 5x_2 \\[1em]
\text{subject to} & -3x_1 + 3x_2 \geq 6 \\
 & 3x_1 + 2x_2 \geq 10 \\
\text{where} & x_1 \text{ and } x_2 \text{ non--negative integers.}
\end{array}$$

The result of applying the relaxation algorithm to that example is shown in figure 3.4.

3.2.3 The Traveling Salesman Problem (TSP)

The TSP can be described as follows. A salesman takes a tour traveling through n cities, visiting each city once and only once, and returning to his starting city. He incurs a cost c_{ij} in traveling from city i to city j. He is required to minimize the total cost of his entire tour.

The problem can be more formally defined in terms of a weighted graph $G = (V, E)$. The graph consists of a set of n vertices V, which correspond to the n cities, and a set of arcs E, which correspond to the routes between various pairs of cities. Associated with each arc $(v_i, v_j) \in E$ is the non-negative cost c_{ij} for traveling between the cities v_i and

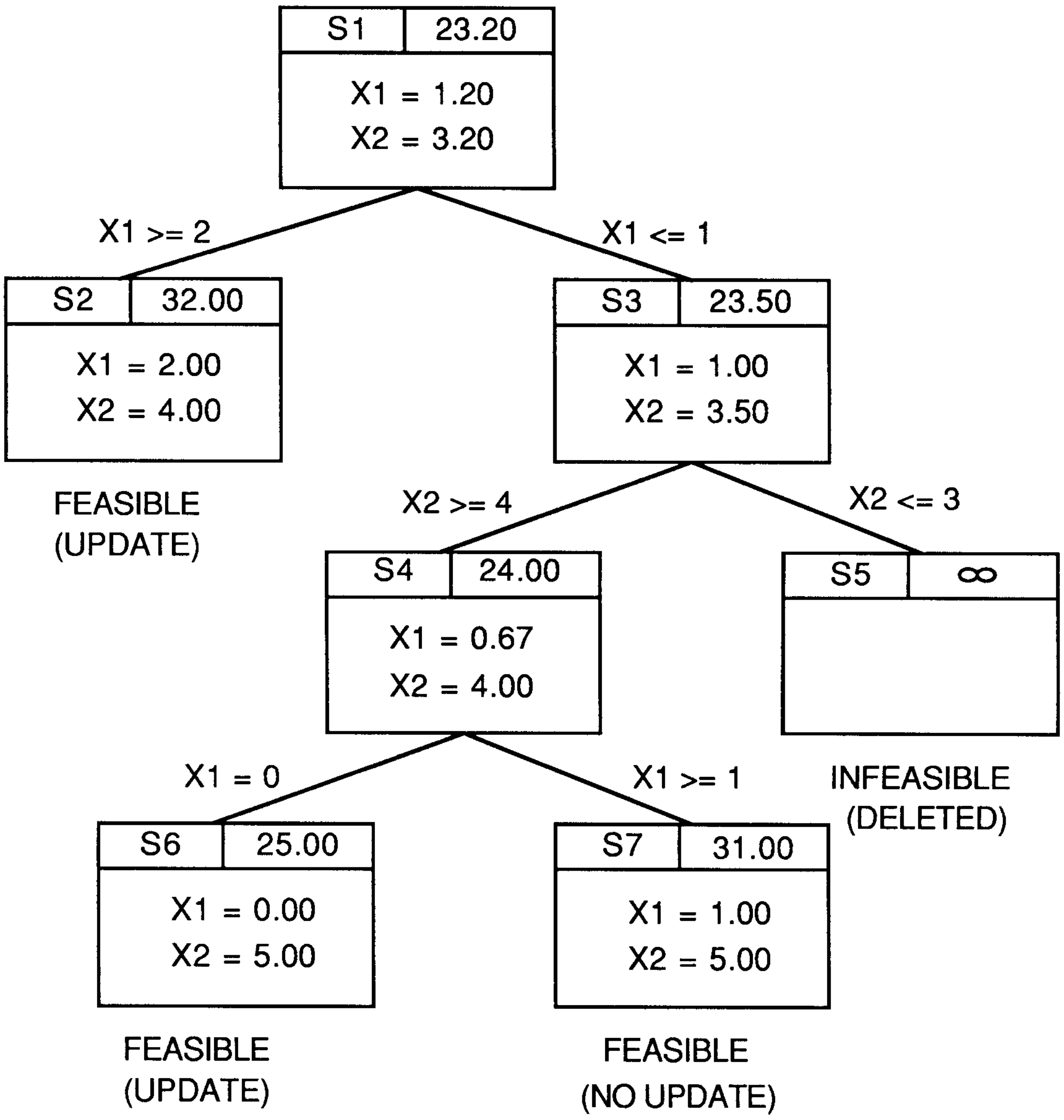

Figure 3.4. BB tree for the ILP example.

v_j. The objective is to find a minimal cost tour t consisting of a sequence of arcs, or equivalently, a sequence of ordered city pairs, which forms a closed path going through each vertex in the graph once and only once. That is, the objective is to find a tour

$$t = [(i_1, i_2), (i_2, i_3), \ldots, (i_{n-1}, i_n), (i_n, i_1)] ,$$

that minimizes the total cost of the tour, given by

$$\text{cost} = \sum_{(i,j)\in t} c_{ij}.$$

The costs of traveling between various pairs of cities can be represented by a *cost matrix* $C = [c_{ij}]$. The entry in row i and column j of the matrix is the cost of traveling from city i to city j. The cost of a tour t under a matrix C is the sum of the matrix elements picked out by t. That is,

$$\text{cost} = \sum_{(i,j)\in t} C(i,j).$$

Since t can include a city once and only once, t picks out one and only one cost in each row and in each column of the cost matrix.

A cost matrix that has non-negative elements and has at least one zero in each row and each column is referred to as a *reduced* matrix. Given any cost matrix C, there is a corresponding reduced matrix C_r. The process of subtracting the smallest element of a row from each element in that row is called *reducing* that row. Similarly, the process of subtracting the smallest element of a column from each element in that column is called reducing that column. A matrix C can be reduced by first reducing its rows and then reducing its columns. It should be pointed out that it is possible to obtain more than one reduced matrix depending on the order in which the rows and columns are reduced.

To illustrate the reduction process, the cost matrix C

$$\begin{bmatrix} \infty & 27 & 43 & 16 \\ 7 & \infty & 16 & 1 \\ 20 & 13 & \infty & 35 \\ 5 & 13 & 24 & \infty \end{bmatrix}$$

is reduced by reducing its rows first to get

$$\begin{bmatrix} \infty & 11 & 27 & 0 \\ 6 & \infty & 15 & 0 \\ 7 & 0 & \infty & 22 \\ 0 & 8 & 19 & \infty \end{bmatrix},$$

and then by reducing its columns to obtain a reduced matrix C_r

$$\begin{bmatrix} \infty & 11 & 12 & 0 \\ 6 & \infty & 0 & 0 \\ 7 & 0 & \infty & 22 \\ 0 & 8 & 4 & \infty \end{bmatrix}.$$

When a constant k is subtracted from each element in a row or a column of a cost matrix, the cost of any tour under the new matrix is k less than the cost of the same tour under the old one. This is true since any tour must contain one and only one element in each row and each column. It follows then that the relative costs of tours are unchanged and that an optimal tour under the new matrix is also optimal under the old one. In fact, if z_1 is the optimal tour cost under the old cost matrix and z_2 is the optimal tour cost under the new one, then

$$z_1 = k + z_2.$$

Consequently, if h is the sum of all the values used in reducing a given cost matrix C, then h can be regarded as a lower bound on the optimal tour cost of C. This will always be true since C contains only non-negative elements.

An efficient BB algorithm for solving the TSP is given by Little et al. [LMSK63]. It will be referred to here as the LMSK algorithm. It uses the process of reduction described above to calculate the lower bounds on subproblems. In the algorithm, if a problem cannot be solved directly, it is broken down into two subproblems. An arc is selected, according to the heuristic rule below, and two subproblems are created. The first subproblem represents all tours that *include* the arc. The second subproblem represents all tours that *exclude* the arc.

A heuristic rule is used to select an arc to decompose a subproblem. The arc selected is the one whose exclusion causes the maximum increase in the cost of the tour. That is, the heuristic rule examines each possible arc that can be excluded in a subproblem. It then evaluates the increase in the tour length when that arc is excluded and then selects the arc whose exclusion will cause the maximum increase.

The algorithm can be outlined as follows:

1. **Initialization.** The list of active subproblems is initialized to contain the original problem. The incumbent is initialized to ∞.
2. **Selection.** The subproblem with the smallest lower bound is selected.
3. **Feasibility.** If the subproblem is small enough to be solved directly, it is. The incumbent is updated accordingly.
4. **Branching.** An arc whose exclusion causes the largest increase in the tour length is selected as described above. That is, the possible arcs that can be selected in the partial tour are excluded one arc at a time, and the resulting matrix is reduced to determine the increase in the tour length caused by the exclusion. The arc that causes the maximum increase in the tour length is the one used to create the two new subproblems are created as described above. The lower bounds of the two subproblems are calculated.
5. **Lower bound test.** A subproblem is deleted if its lower bound is larger than that of the incumbent.

The cost matrix of a 6-city TSP is shown below:

$$\begin{bmatrix} \infty & 27 & 43 & 16 & 30 & 26 \\ 7 & \infty & 16 & 1 & 30 & 25 \\ 20 & 13 & \infty & 35 & 5 & 0 \\ 21 & 16 & 25 & \infty & 18 & 18 \\ 12 & 46 & 27 & 48 & \infty & 5 \\ 23 & 5 & 5 & 9 & 5 & \infty \end{bmatrix}.$$

The LMSK algorithm is used to solve that problem, and the resulting BB tree problem is shown in figure 3.5. A node containing (i, j) represents a subproblem that includes the arc (v_i, v_j). A node containing $\overline{(i, j)}$ represents all tours that do not include that arc. The

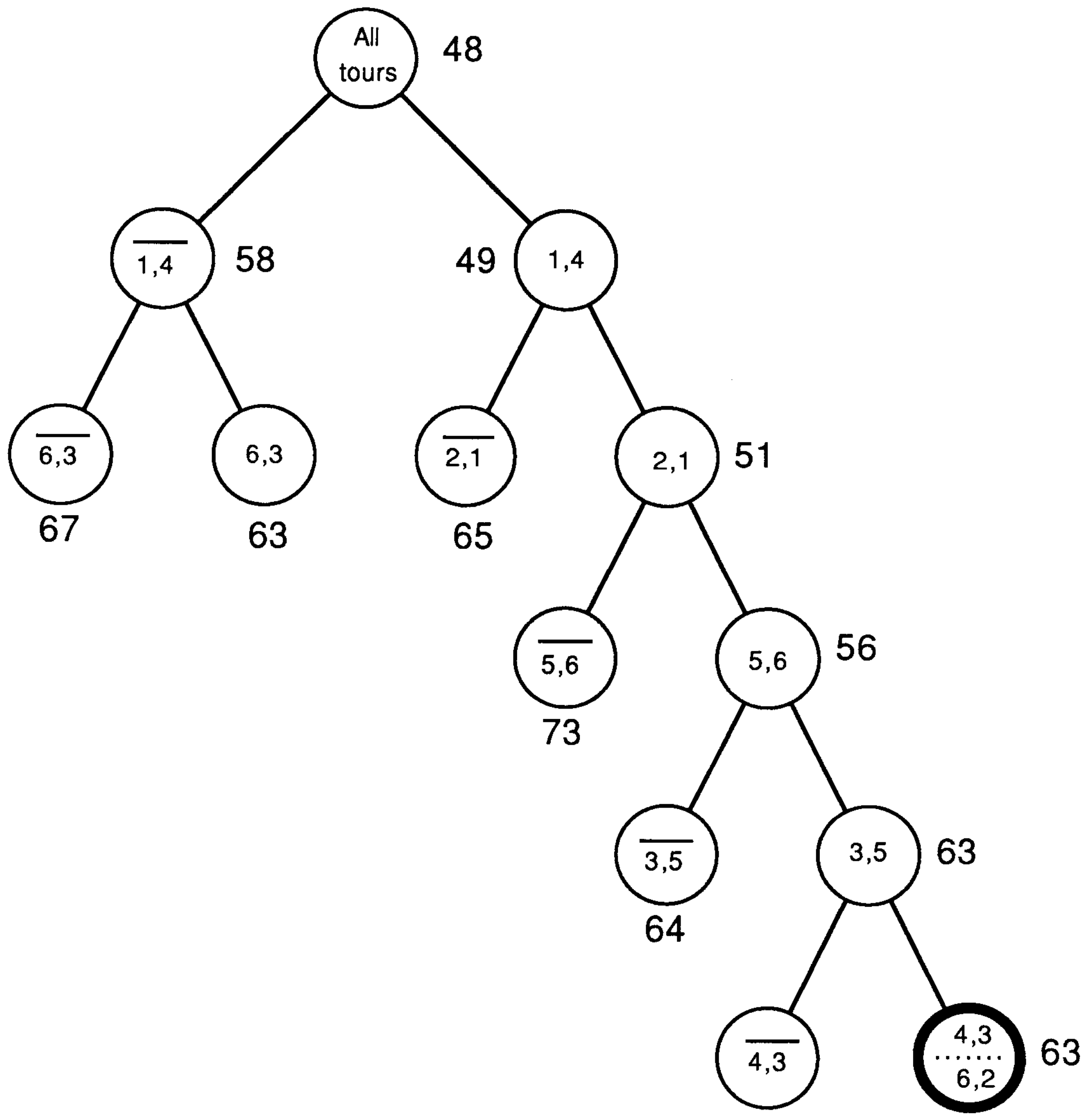

Figure 3.5. The BB tree for the TSP example.

lower bound for each subproblem is shown next to it. After 7 subproblem expansions, the optimal tour $1 \rightarrow 4 \rightarrow 3 \rightarrow 5 \rightarrow 6 \rightarrow 2 \rightarrow 1$ is found costing 63. The last subproblem at which the optimal solution is found includes two arcs, $(4,3)$ and $(6,2)$. This is because the subproblem at this point becomes small enough to be solved directly, or in other words, all possible tours can be investigated exhaustively.

3.2.4 The A^* Algorithm

State space search forms the basis of many AI applications. A state space representation of a problem consists of two basic components: the *state* and the *operators.* The state is a data structure that defines all conditions of the problem at any instant during the search process. The *operators* define the set of rules by which one state can be transformed into another. Associated with each operator is a measure of the cost of applying that operator to a state. Given an initial starting state of the problem and a final goal state, a state space search is conducted by repeatedly applying one or more of the operators to the state of the problem to transform it from the initial state to the goal state.

State space searches have been recognized as being BB algorithms in [KuKa83, Ibar78a]. In this section, we illustrate this by the A^* algorithm. The objective of the algorithm is to find a goal state G from an initial starting state S while minimizing the cost incurred in using the operators.

The A^* algorithm is usually used to solve decision problems in which it is desired to determine the existence of at least one solution (the goal state) in the presence of constraints (implied by the operators). However, the algorithm may also be viewed as an optimization algorithm. The objective function in this case is the sum of the costs of the operators used to reach the goal state from the initial state.

The A^* algorithm uses a function f to measure the merit of each state in the state space. Hence, the function f is defined as the sum of two components g and h. That is,

$$f = g + h.$$

The function g is a measure of the cost of getting from the initial state to the current state. The value of g is computed as the sum of the costs along the path from the starting state to the current one. The function h is a heuristic function that estimates the additional cost of getting from the current node to the goal state.

The A^* algorithm is described in [Nils80] and is only outlined here. Two lists are maintained by the algorithm. The first is called OPEN that contains unexamined nodes or subproblems, and the second is called CLOSED that contains examined subproblems. Initially, OPEN contains the starting state while CLOSED is empty. The following steps are then repeated. If there are no nodes on OPEN, then there is no goal state and the algorithm terminates with failure. Otherwise, the node with the smallest value of the function f is selected and is called BESTNODE. The node is removed from OPEN and placed on CLOSED. If BESTNODE is the goal state, then the algorithm terminates in success. Otherwise, all successors of BESTNODE are generated by applying all possible operators to BESTNODE. For each successor, compute the sum of g(successor) and the estimate of the cost of getting from the successor to the goal state. If the successor is not already on OPEN or CLOSED, then it is placed on OPEN, otherwise it is deleted. The value of the function f of successor is computed as

$$f(successor) = g(successor) + h(successor).$$

The A^* formulated as a BB algorithm can be outlined as follows:

1. **Initialization.**
   (a) The OPEN list is initialized to contain the starting state S.
   (b) The CLOSED list is initialized to be empty.
   (c) The incumbent z is initialized to ∞.
2. **Selection.** The subproblem with the smallest value of the function f is selected. This node is referred to as BESTNODE. It is removed from the OPEN list and placed on the CLOSED list.

3. **Feasibility.** If BESTNODE is the same as the goal state G, then a solution is found. The incumbent is updated.

4. **Branching.** If BESTNODE is not the goal state, the set of operators are used to generate the successors of that subproblem. The function f is calculated for each successor as described above.

5. **Equivalence test.** For each successor generated, if the successor is on CLOSED or on OPEN, then it is deleted.

6. **Lower Bound test.** For each successor, if the value of f is less than that of the incumbent, it is inserted on OPEN, otherwise, the successor is deleted.

The A^* algorithm illustrates the basic constituents of a BB algorithm. The function f can be treated as the lower bound function of the BB algorithm. The search strategy of the algorithm is best-first since the algorithm always selects a node with the smallest value of f. A branching procedure is used to generate the successors of a node. However, since the first solution discovered by the algorithm is the optimal one, the lower bound elimination test can be considered to be inactive. The algorithm, however, uses an equivalence test to delete subproblems that have been already examined by the algorithm (the set of equivalent subproblems is OPEN plus CLOSED). The termination procedure is inactive in the algorithm.

To illustrate the A^* algorithm, the 8-puzzle problem is used as an example. The 8-puzzle is a square tray in which are placed eight square tiles. The remaining ninth square is left uncovered. Each tile has a number from 1 to 8 on it. A tile that is adjacent to the blank space can be slid into the space. The objective is to slide the tiles around to transform an initial arrangement of tiles into a final one.

The state of the problem may be defined as a 3×3 matrix that contains numbers representing the tiles. The blank space is denoted by a 0. Each move of a numbered tile

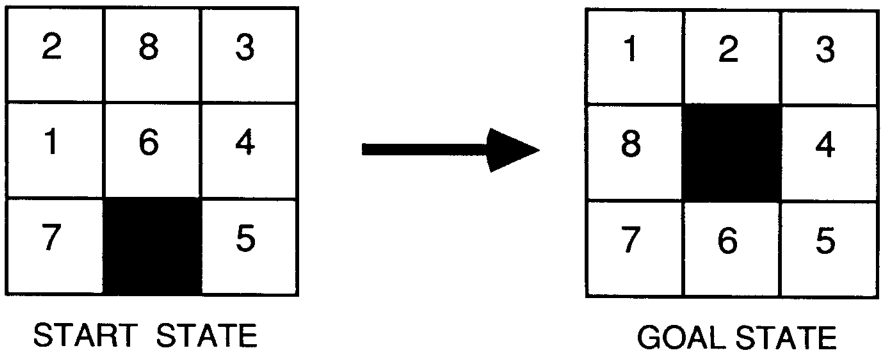

Figure 3.6. An 8-puzzle example.

into the space can be viewed as a move of the space into one of four possible directions: up (U), down (D), left (L) and right (R). Hence, the set of operators can be defined in terms of the movement of the space as

$$\{U, D, L, R\}.$$

In some states, only some of the above 4 operators are allowed. For example, if the empty space is in the left lower corner of the puzzle, only U and R are allowed. A unit cost may be assigned to each operator since the cost of applying any of them is the same. In this case, the first component of the merit function (i.e., g) becomes equivalent to the depth d of the node in the BB tree.

The branching rule generates successors of a subproblem by applying all the allowed operators to the state of the subproblem. The heuristic function h may be defined as W, the number of misplaced tiles from the goal state. Hence, the lower bound function f is

$$f = d + W.$$

The heuristic function in this case is monotone (as defined by 3.2 [Nils80]), and therefore, the use of the equivalence test is not necessary. Consequently, there is no need to use the CLOSED list [Pear84].

An example of the problem is shown in figure 3.6. The resulting BB tree using the A^* algorithms is shown in figure 3.7. The number shown to the right of each node in the tree represents the iteration of the BB algorithm in which this node is expanded. The number shown to the left of each node inside a circle represents the lower bound of that node.

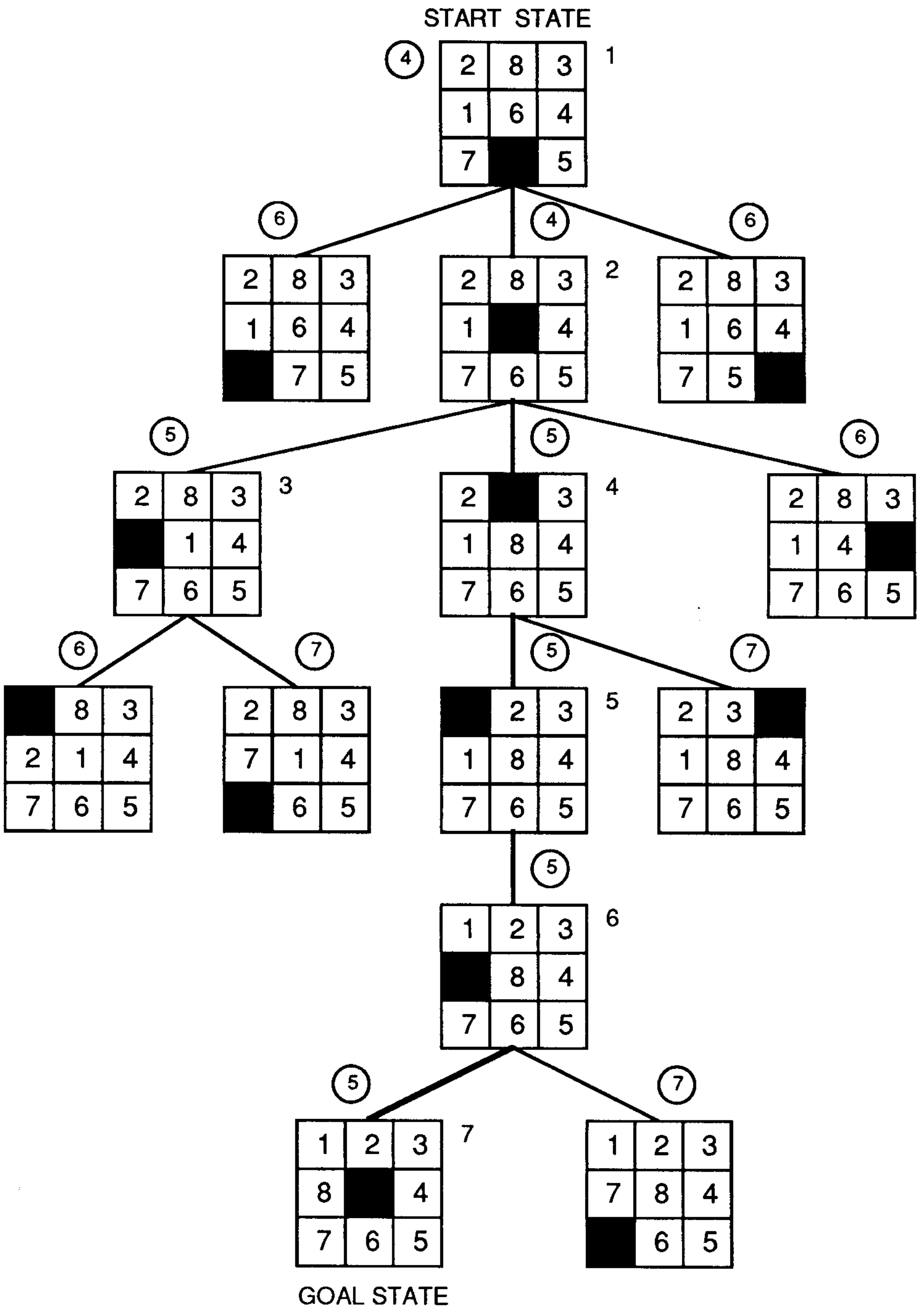

Figure 3.7. The BB tree for the 8-puzzle example.

CHAPTER 4

MAPPING THE BB ALGORITHM ONTO DMMs

In this chapter the mapping of the sequential BB algorithm onto DMMs is examined. Parallelism that exists in the sequential BB algorithm, and that can be exploited by a multiprocessor is described in section 4.1. An algorithm that illustrates how this parallelism can be exploited with no particular assumptions about the architecture of the multiprocessor is described in section 4.2. The algorithm represents an abstract framework that has been used by researchers in the past to study many of the properties of parallel BB algorithms, particularly on shared memory multiprocessors. However, since this algorithm and the framework it represents are not adequate to describe parallel BB algorithms on DMMs nor to identify the factors that affect their performance, we present a distributed model algorithm in section 4.3. The algorithm is used to explain and illustrate the factors that degrade the performance of parallel BB algorithms on DMMs, as well as the tradeoffs among these factors. A simple model is used to identify the extent to which these factors affect the performance of this algorithm. A critical review of previous work on parallel BB algorithms in light of the two models is then given in section 4.4.

4.1 Parallelism in BB Algorithms

A parallel BB algorithm employs multiple processors in order to perform the computations of the sequential BB algorithm and obtain performance gains that are proportional to the number of processors used. This performance gain can only be realized by exploiting

UNIVERSITY OF MICHIGAN LIBRARIES

parallelism that exists in the sequential BB algorithm. A parallel BB algorithm can make use of the multiple processors to exploit two main sources of such parallelism [Trie86].

A parallel algorithm can exploit parallelism in the computations performed on a subproblem by the sequential BB algorithm. That is, a parallel algorithm can employ the multiple processors to evaluate a subproblem, calculate its lower bound, determine it feasibility, and decompose it into smaller subproblems. In other works, all processors work on one subproblem at a time. Consequently, this source of parallelism in the sequential BB algorithm is referred to as *subproblem-level* parallelism.

The overall execution of a parallel BB algorithm that exploits subproblem-level parallelism is essentially the same as the overall execution of the sequential algorithm, with the exception that the computations performed on a subproblem are performed in parallel. Therefore, the general algorithmic characteristics of the sequential algorithm are not altered. For example, the set of subproblems expanded by the parallel algorithm is the same as the set of subproblems expanded by the sequential one, and furthermore, the order in which these subproblems are expanded is the same for both algorithms. Consequently, the performance gain that can be realized by the parallel algorithm is only that which can be obtained by performing the computations on a subproblem in parallel. The amount of this performance gain depends on the exact nature of these computations, and therefore, is highly problem dependent. Furthermore, since the amount of those computations may be limited in some cases, as in the case of the 8-puzzle problem, for example, it may not be possible for a parallel BB algorithm to employ a large number of processors to exploit this source of parallelism and obtain scaling performance.

A parallel BB algorithm can also employ multiple processors to exploit parallelism in the set of subproblems expanded by the sequential algorithm. The set of active subproblems during any iteration of the sequential algorithm contains other subproblems that will be expanded in the subsequent iterations of the algorithm. Therefore, a parallel algorithm can expand multiple subproblems from the set of active subproblems during each iteration rather than a single subproblem as in the sequential algorithm. This source of parallelism is referred to as *algorithm-level* parallelism.

The overall behavior of a parallel BB algorithm that exploits algorithm-level parallelism can change from that of the sequential one since there is no guarantee that the multiple subproblems expanded during an iteration of the parallel algorithm all belong to the set of subproblems expanded by the sequential algorithm. Consequently, unnecessary expansion of subproblems may result. Furthermore, the order of the expansion of subproblems by the parallel algorithm may be different from the order of their expansion by the sequential one, which can lead to anomalies [LaSa83].

However, the performance gain that can be realized by a parallel BB algorithm exploiting algorithm-level parallelism is not directly affected by the computations performed on a subproblem, and is, consequently, less problem dependent. The performance gain is limited only by factors that will be discussed later in this chapter, and it is possible to obtain performance that scales to a large number of processors.

Consequently, only parallel BB algorithms that exploit algorithm-level parallelism are considered in this study. Parallel algorithms that exploit subproblem-level parallelism, and parallel algorithms that exploit both sources of parallelism combined can be considered as an extension to this research.

4.2 The Logical Model Algorithm

In this section a parallel algorithm that exploits algorithm-level parallelism is presented. The algorithm is referred to as the *Logical Model (LM)* algorithm, and it is so called because it assumes no model for the architecture of the multiprocessor used to execute the algorithm. It serves only to describe a framework of operation that illustrates how algorithm-level parallelism can be exploited by a parallel BB algorithm. This framework has been used by many researchers to study performance aspects of parallel BB algorithms, particularly for shared memory multiprocessors.

In the LM algorithm, a global data set is maintained, and P processors are used to execute the algorithm as will be described below. The global data set contains a list of active subproblems and a single incumbent. The list is maintained in an increasing order of values of the selection heuristic function h in order to implement a best-first search strategy as described in Chapter 3. The P processors are assumed to have an overhead-free access to the global data set. This model is diagramed in figure 4.1.

The set of subproblems $\mathcal{S}$, is defined to be a subset of the set of active subproblems that contains the P subproblems with the smallest values of h among all subproblems in the set of active subproblems. These subproblems are also the first P subproblems on this list of active subproblems due to the order in which the list is maintained. This set of subproblems is expanded by the processors of the LM algorithm.

The P processors are assumed to be synchronized into cycles that are referred to as *iterations*. In any iteration, a processor executes the steps that implement the LM algorithm:

1. A subproblem from the list of active subproblems is removed. The subproblem is one of the subproblems that belong to the set $\mathcal{S}$ described above. It is not important which subproblem is selected by the processor, as long as it belongs to that set. Therefore, collectively, the processors select the P subproblems with the globally best values of h. This step is referred to as the *parallel selection* of subproblems.

2. The subproblem is decomposed into new smaller subproblems by applying the branching procedure. A processor applies the branching procedure to decompose the subproblem independently from other processors. This step is referred to as *parallel branching* of subproblems.

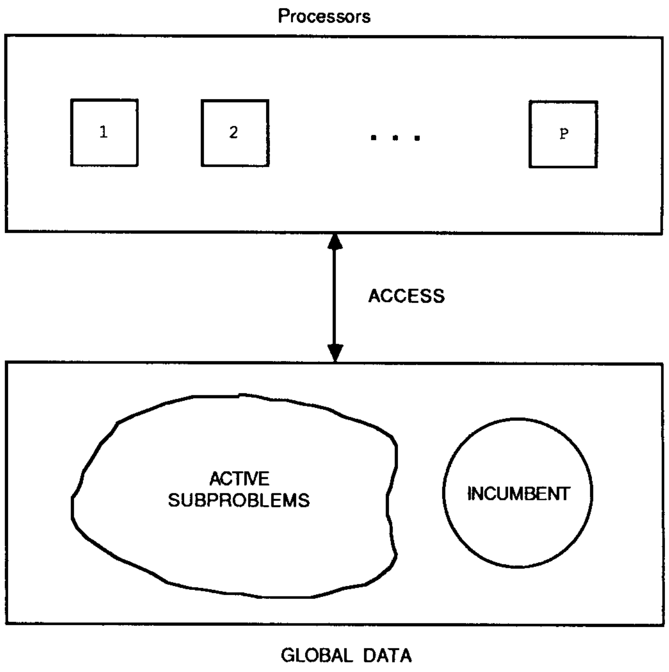

Figure 4.1. The Logical Model.

3. The termination procedure is applied to new subproblems in order to delete ones that can never lead to feasible solutions. Similar to parallel branching, a processor applies the termination procedure independently from other processors. This step is referred to as *parallel termination* of subproblems.

4. Feasible solutions among the new subproblems generated are checked for. A subproblem representing a feasible solution whose lower bound is less than that of the global incumbent is used to replace that incumbent, making the new solution available to all other processors for the next step of the iteration. In the event that more than one feasible solution is discovered by a processor, or by multiple processors, only the one with the smallest lower bound becomes the candidate to update the incumbent in the manner described above.

5. The lower bound elimination test is used to delete subproblems that cannot lead

to better feasible solutions than the incumbent. This step is referred to as *parallel elimination* of subproblems.

6. The remaining subproblems are then inserted on the list of active subproblems, endig the iteration.

The iterations of the LM algorithm are repeated until the global list of active subproblem is empty at the end of an iteration. This terminates the operation of the LM algorithm. The optimal solution is stored in the incumbent.

The above strategy in selecting subproblems that belong to the set $\mathcal{S}$ during each iteration of the LM algorithm is based on the premise that this set of subproblems contains ones that are most likely to be expanded by the sequential algorithm. That is, a subproblem that belongs to the set $\mathcal{S}$ in an iteration of the LM algorithm is more likely to belong to the set of subproblems that would have been selected and expanded by the sequential algorithm in its subsequent iterations. In fact, as will be described below, the selection of this set of subproblems is sufficient to select subproblems that are expanded by the sequential algorithm before subproblems that are not in any iteration of the LM algorithm.

The strategy is also a natural extension to the best-first strategy that is used by the sequential algorithm. If the subproblem with the smallest value of h in any iteration of the sequential algorithm is the subproblem most likely to lead to the optimal solution, then the P subproblems with the smallest values of h in any given iteration of the LM algorithm are the subproblems most likely to lead to the optimal solution. Hence, this strategy for search is referred to as a *parallel-best-first* search strategy [WaLY85].

The performance of the LM algorithm is measured by the number of iterations the algorithm takes to obtain the optimal solution. The speedup of the LM algorithm reflects the overall gain in performance of the algorithm that is obtained by using P processors.

It is defined as the ratio of the number of iterations taken by the algorithm using one processor (i.e., the sequential algorithm) to the number of iterations taken by the algorithm using P processors. That is,

$$S_P = \frac{I(1)}{I(P)},$$

where $I(P)$ denotes the number of iterations of the LM algorithm using P processors. The factors that affect the performance of the LM algorithm are described next.

The set of subproblems expanded by the sequential algorithm is referred to as the set of *essential* subproblems, and a subproblem that belongs to this set is referred to as an *essential* subproblem. All other subproblems that are generated, but are not expanded by the sequential algorithm are referred to as *non-essential* subproblems. In its quest for the optimal solution, the LM algorithm *must* expand all essential subproblems before terminating. Therefore, the set of essential subproblems represents the workload that must be executed by the algorithm before it terminates. Furthermore, when the selection heuristic function is defined by equation 3.3, or equation 3.4, the LM algorithm during any of its iterations selects all essential subproblems available on the list of active subproblems (up to P). That is, the LM algorithm does not select a non-essential subproblem for expansion in an iteration unless there are no enough essential subproblems to expand in that iteration. This is a consequence of the parallel-best-first search strategy used by the LM algorithm. Formal proofs of the above statements can be found in [LaSp85, Li85, QuDe85], and are not presented here.

Consequently, the performance of the LM algorithm is affected by the number of essential subproblems expanded by the algorithm during each of its iterations. In an iteration in which the LM algorithm expands only essential subproblems, the speedup of the algorithm during that iteration is P, and in this case the iteration is said to be *perfect*. However, in an iteration in which the number of essential subproblems expanded by the algorithm is less than P, the speedup of the algorithm during that iteration is less than

P, and the iteration is said to be *imperfect*.

The possible lack of essential subproblems in an iteration is referred to as *lack of parallelism* in the BB tree since it reflects a lack of workload that can be performed by the processors. Although a processor can be active expanding a non-essential subproblem in an imperfect iteration, the effort of the processor is wasted since the expansion of this non-essential subproblem is not performed by the sequential algorithm, and is not needed to obtain the optimal solution.

This effect of lack of parallelism in degrading the speedup of the LM algorithm is illustrated using the example in figure 4.2, which shows a BB tree generated by the sequential algorithm. The sequential algorithm expands the root into subproblems 2 and 3 during its first iteration. The algorithm expands subproblem 2 during the second iteration into subproblems 4 and 5. The algorithm then expands subproblem 5 during the third iteration to find a feasible solution at subproblem 6. The feasible solution updates the incumbent and terminates the algorithm by eliminating all remaining subproblems. Therefore, three iterations are needed to find the optimal solution, and subproblems 1, 2, and 5 are the essential subproblems in this example, while subproblems 3 and 4 are the non-essential subproblems.

The BB tree generated by the LM algorithm for the same example using two processors is shown in figure 4.3. The algorithm expands the root during its first iteration to generate subproblems 2 and 3. This iteration is imperfect since only one essential subproblem (subproblem 1) is expanded. The LM algorithm then expands subproblems 2 and 3 during its second iteration to generate subproblems 4, 5, 8 and 9. This iteration is again imperfect since only one essential subproblem (subproblem 2) is expanded. During the third iteration, the algorithm expands subproblems 4 and 5 to find the feasible solution and terminate the algorithm, as in the case of the sequential algorithm. This iteration is also imperfect since only one essential subproblem is expanded (subproblem 5). The LM

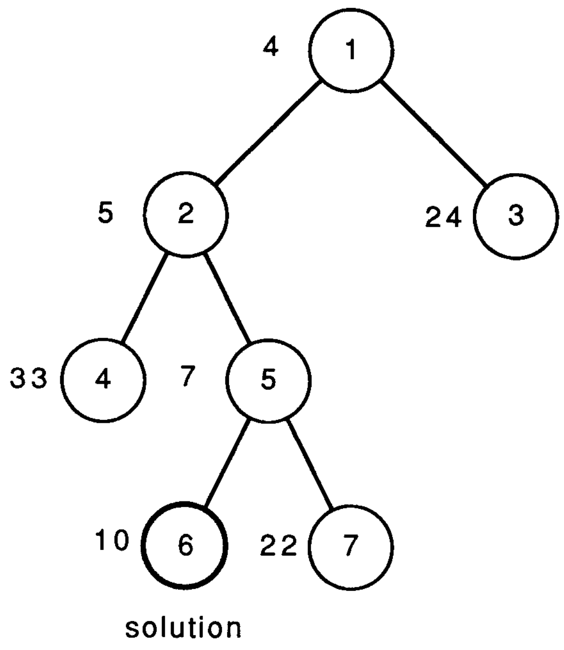

Figure 4.2. The BB tree generated by the sequential BB algorithm.

algorithm also takes three iterations to complete the problems resulting in no speedup (i.e. speedup = 1.0). This loss of speedup is a result of the lack of essential subproblems during the iterations of the algorithm, or equivalently due to the lack of parallelism in the BB tree.

The LM algorithm has been used by many researchers to study the execution of parallel BB algorithms in the context described by the framework of that algorithm. Quinn and Deo [QuDe85], Li [Li85], and Lai and Sprague [LaSp85] have studied the effect of the structure of the BB tree on lack of parallelism, and hence, on performance, and have derived theoretical bounds on the speedup of the LM algorithm. They suggest that if a problem generates a "large enough" tree, then the speedup of the LM algorithm is near linear (i.e., close to P when P processors are used). Lai and Sahni [LaSa83], Li [Li85], and Li and Wah [LiWa84a, LiWa84b, LiWa86] have used the LM algorithm to study the effect

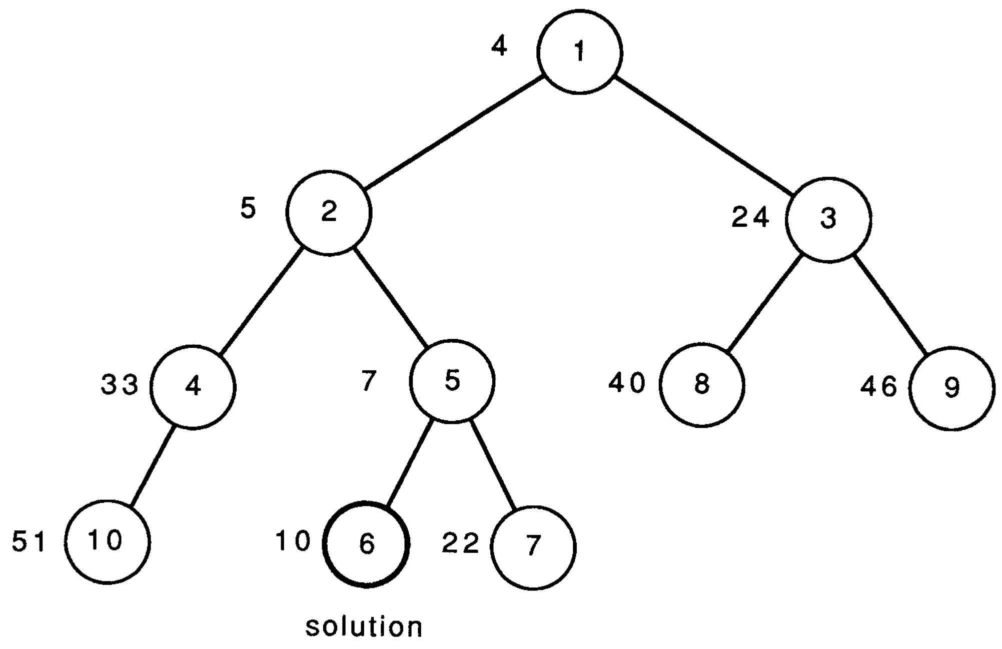

Figure 4.3. The BB tree generated by the LM algorithm using two processors.

of anomalies on performance and to derive conditions needed to limit their occurrence.

4.3 The Distributed Model Algorithm

The LM algorithm discussed in the previous section is suitable for studying parallel BB algorithms and the factors that affect their performance on shared memory multiprocessors. The global shared memory of the multiprocessor can be used to maintain the global data set while its processors can assume the role of the processors in the LM algorithm. However, the LM algorithm is not adequate to study a parallel BB and the factors that affect its performance on a DMM. The lack of a global shared memory in a DMM makes it difficult to efficiently maintain a global data set that is accessible to all the processors. The global data set must, therefore, be divided into smaller components that are maintained by the individual processors. This is also necessary to efficiently utilize the memory resources of the DMM. A processor must access the components of the data set that are not maintained locally by it using message passing.

Accordingly, we propose an algorithm that is more suitable to describe the operation of a parallel BB algorithm and to study the factors that affect its performance on DMMs. Similar to the LM algorithm, this algorithm represents only a framework of operation and does not assume any particular architecture for parallel processor used by the algorithm, except its distributed memory nature, and the use of message passing to accomplish communication among processors. This algorithm is referred to as the *Distributed Model (DM)* algorithm. The operation of the algorithm is first presented followed by a description of the factors that affect its performance.

A processor in the DM algorithm maintains its own list of active subproblems and its own incumbent; this is depicted in figure 4.4. The local list of each processor maintains the subproblems of the set of active subproblems in an increasing order of the selection heuristic function h, similar to a sequential BB algorithm with a best-first search strategy. The processors operate asynchronously, and therefore, there is no concrete concept of an iteration similar to that of the LM algorithm; instead, each processor operates in its own iterations. An iteration consists of two components: a *compute* component and a *load-balance* component.

In the compute component of its iteration, a processor selects and expands the subproblem with the smallest value of h from its local list of active subproblems in a manner similar to the sequential BB algorithm. That is, the subproblem is decomposed into smaller subproblems using the branching procedure. Subproblems that can never lead to feasible solutions are deleted using the termination procedure. The lower bounds of subproblems that represent feasible solutions are compared to the lower bound of the local incumbent, and a feasible solution that has a smaller lower bound than that of the local incumbent replaces that incumbent. All subproblems on the local list of active subproblems whose

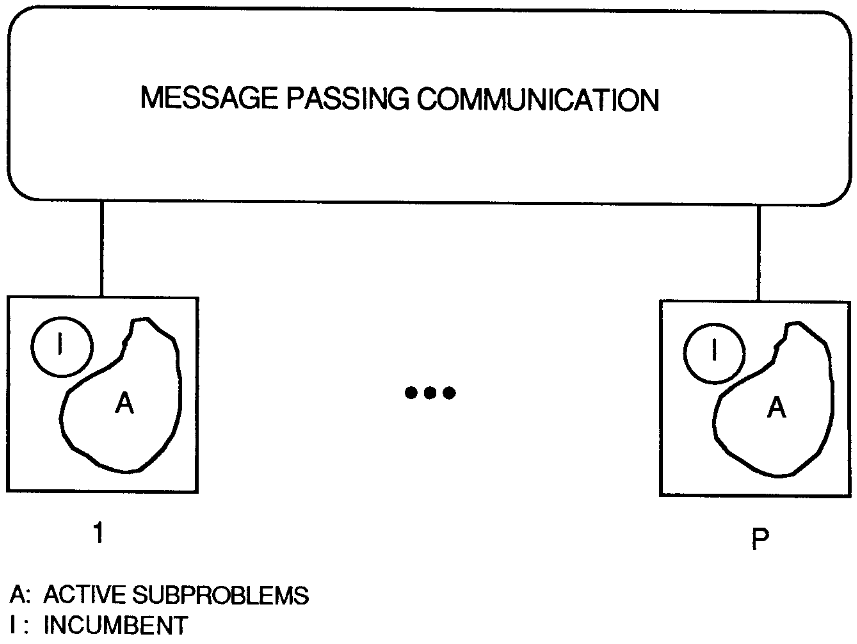

Figure 4.4. The Distributed Model.

lower bounds are larger than or equal to that of the new local incumbent are deleted. The local incumbent is then used to perform the lower bound elimination test on subproblems, and all remaining subproblems are inserted on the local list of active subproblems.

In the load-balance component of its iteration, a processor employs a strategy of communication with other processors in order to perform three functions. The first is to broadcast any feasible solution that updated the incumbent during the compute component of the iteration; this is referred to as communication of *pruning* information. The second is to exchange information regarding values of h of the subproblems on its local list of active subproblems with other processors; this is referred to as communication of *selection* information. The third function of the load-balance component is to perform load balancing of subproblems among the processors; i.e., to move subproblems from processors with more workload of subproblems to processors with less workload of subproblems. The processor must use message passing in order to perform these functions.

The processors repeat their iterations until all local lists become empty At that point in time, the algorithm terminates and the optimal solution is the minimum of all the local incumbents in the processors.

The performance of the DM algorithm is measured by its speedup $S(P)$, which reflects the gain obtained by using P processors relative to a single one (i.e., the sequential algorithm). It is defined as the ratio of the execution time of the DM algorithm using a single processor, denoted by $T(1)$ to the execution time of the DM algorithm using P processors, denoted by $T(P)$. That is

$$S(P) = \frac{T(1)}{T(P)}$$

The performance of the DM algorithm is degraded by amounts of time that a processor spends performing an activity that is not performed by the sequential BB algorithm. These amounts of time are referred to as *overhead*, and there are three types of overhead that a processor can incur: *computation-overhead*, *communication-overhead*, and *imbalance-overhead*.

Computation-overhead refers to amounts of time a processor spends in expanding non-essential subproblems. These subproblems are not expanded by the sequential algorithm and, therefore, represent additional and unnecessary computations. Communication-overhead refers to the amounts of time a processor spends in the process of communication with other processors during the load-balance component of an iteration. These amounts of time are not incurred by the sequential algorithm and, hence, degrade performance. Finally, imbalance-overhead refers to amounts of time a processor spends idle due to the lack of subproblems on its local list, which is caused by possible load imbalance among the processors.

The factors that give rise to these three types of overhead are described in detail in the following three sections. A tradeoff between computation-overhead and imbalance-overhead on one hand and communication-overhead on the other is then described in section 4.3.4

4.3.1 Computation-Overhead

Computation-overhead reflects wasted computational effort performed by a processor in expanding non-essential subproblems. In the LM algorithm, overhead of a similar nature arise due to lack of parallelism. However, in the DM algorithm two additional factors contribute to the expansion of non-essential subproblems by a processor: the first is lack of selection information; the second is lack of pruning information.

Lack of selection information refers to the lack of complete knowledge by a processor about the values of the selection heuristic function h of subproblems in other processors. A processor expands the subproblem with the smallest values of h on its local list, which may or may not be essential. It is possible, due to the lack of knowledge described above, that a processor expands a non-essential subproblem while another processor has multiple subproblems on its local list. In this case, the first processor incurs a computation-overhead that is caused by lack of selection information.

The effect of lack of selection information is illustrated using figure 4.5, which shows the BB tree of a hypothetical problem that has been expanded using two processors. In the case of the LM algorithm, the root is expanded during the first iteration to give subproblems 2 and 3. These two subproblems are then expanded by the two processors during the next iteration to generate subproblems 4, 5, 6, and 7. Since the two processors have global knowledge about lower bounds, subproblems 4 and 5 are selected for expansion during the third iteration, resulting in the optimal solution at subproblem 9, which terminates the algorithm by eliminating all subproblems from the list of active subproblems. Therefore, three iterations are needed to obtain the optimal solution.

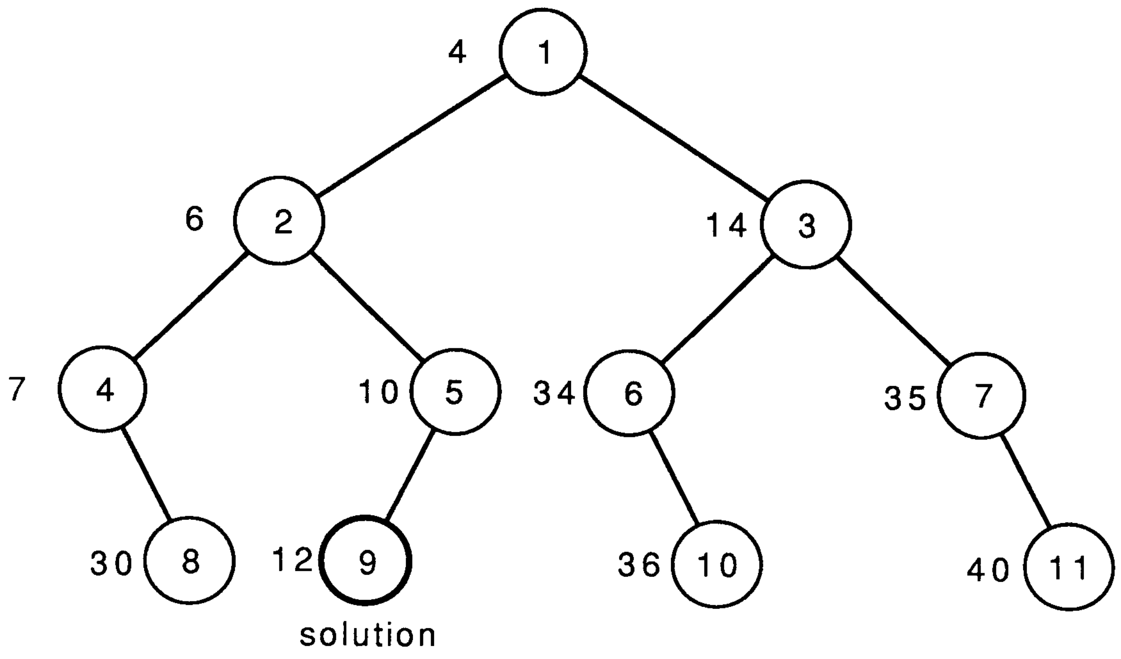

Figure 4.5. Example of computation-overhead caused by lack of selection information.

In the case of the DM algorithm, it is assumed that the two processors do not communicate selection information. (Partial communication has the same effect but requires a larger example to illustrate.) Global knowledge about pruning information is assumed. The root of the BB tree is expanded into subproblems 2 and 3. Subproblem 2 is inserted on the local list of processor 1, and subproblem 3 is inserted on the local list of processor 2. Concurrently the two processors expand subproblems 2 and 3 to generate subproblems 4, 5, 6 and 7 as before. However, in this case, subproblems 4 and 5 are inserted on the local list of processor 1, while subproblems 6 and 7 are inserted on the local list of processor 2. Each processor executes the next iteration by selecting the subproblem with the smallest lower bound on its own local list. Processor 1 selects subproblem 4 while processor 2, lacking knowledge about the existence of 5, selects subproblem 6 for expansion. This results in the generation of subproblem 8 in processor 1 and subproblem 10 in processor 2. The two subproblems are inserted back on the local lists of the respective processors. Processor 1 expands subproblem 5, and processor 2 expands subproblem 7 during their fourth iteration. This generates the optimal solution and terminates the algorithm. Therefore, for the distributed list algorithm each processor expands four subproblems (or executes four iterations) to find the optimal solution. This additional expansion of a subproblem by each processor is computational overhead that results from the lack of knowledge by processor 2 about the lower bounds of subproblems in processor 1.

The lack of pruning information is the second contributor to computation-overhead. This refers to the lack of global knowledge about the information that is used to eliminate subproblems that cannot lead to better optimal solutions than any already found. That is, it refers to the lack of knowledge by one processor about the values of other incumbents in other processors. If a processor does not have knowledge about the incumbent in another processor that is better than its own, then an additional expansion of subproblems that should be pruned or deleted will occur.

Computation-overhead caused by the lack of pruning information can be illustrated using figure 4.6, which shows the BB tree of a problem expanded by the distributed model algorithm using two processors. The processors are assumed not to communicate any pruning information. Initially, the root is expanded and subproblem 2 is inserted on the local list of processor 1, and subproblem 3 is inserted on the local list of processor 2. Processor 1 expands subproblem 2 finding the optimal solution and making the local list of active subproblems for this processor empty. Processor 2 expands subproblem 3 resulting in subproblem 5. This subproblem should be pruned since its lower bound is larger than that of the optimal solution discovered by processor 1. However, the lack of knowledge about the existence of that solution by processor 2 causes it to expand this subproblem to generate two new subproblems. These are subsequently eliminated by the termination test, making the local list of active subproblems in processor 2 empty. Therefore, an

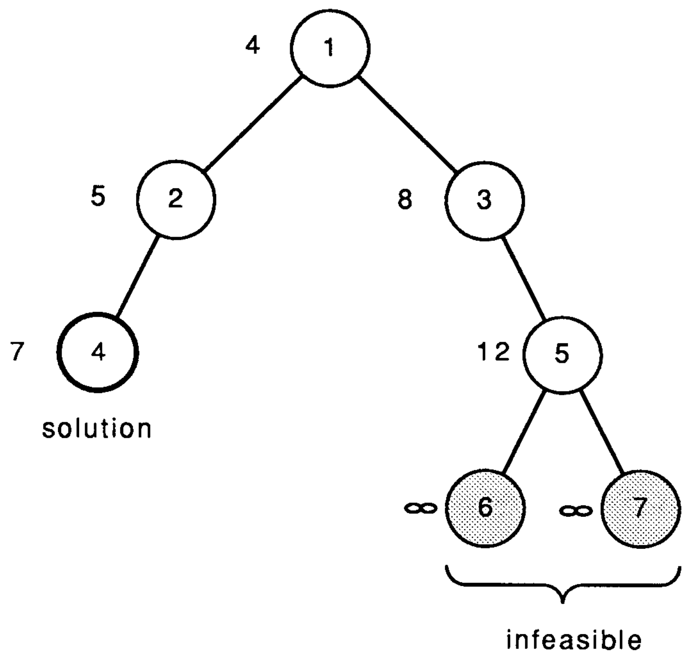

Figure 4.6. Example of computation-overhead caused by lack of pruning information.

additional iteration is needed in order for this algorithm to terminate, resulting in a computational-overhead.

In summary, the lack of global selection information and global pruning information by the processors can result in additional computations for the DM algorithm in the form of non-essential subproblem expansions. This overhead reflects computations in excess of those performed by the sequential BB algorithm or by the LM algorithm, and hence causes the performance of the algorithm to degrade. These overheads are not present in the LM since that model assumes global knowledge of the entire data set. Lack of parallelism overhead can contribute to the computation-overhead incurred by the DM algorithm, as is the case in the LM algorithm.

4.3.2 Communication-Overhead

Communication-overhead reflects the costs incurred by the processors to communicate selection and pruning information, and to perform load balancing. Each processor must send and receive this type of information, and since all communications must be performed by message passing, a time penalty is incurred by each processor. This results in what is referred to as communication-overhead.

It is possible to distinguish between two types of strategies a processor can use in its load-balance component. The first type can be described as synchronous, while the second can be described as asynchronous. In a synchronous load-balance strategy, a processor does not proceed with its next iteration, until the communication with the other processors is complete. In an asynchronous strategy, a processor may proceed with the next iteration without waiting for its communication with the other processors to complete. For example, a processor can start the exchange of selection information with another processor, and proceed with its next iteration, without waiting to receive a response to the exchange. The idle time spent by a processor in synchronization during the load-balance component is the second contributor to communication-overhead.

4.3.3 Imbalance-Overhead

Imbalance-overhead reflects the amount of idle time spent by a processor due to the lack of subproblems on its local list of active subproblems. This idle time is not incurred by the sequential algorithm, and hence, degrades the performance of the DM algorithm.

The lack of subproblems on the local list of active subproblems of a processor is caused by the action of the load-balance component of an iteration in the DM algorithm, coupled with the irregular nature of the BB tree generated by the BB algorithm. This is illustrated in the example shown in figure 4.7 which shows a BB tree generated by the DM algorithm using two processors. The root of the BB tree is expanded into subproblems 2 and 3 as shown in the figure. Subsequently, subproblem 2 is expanded by processor 1 and generates no subproblems that can be inserted on the local list of active subproblems of that processor. Similarly, subproblem 3 is expanded by processor 2 and generates subproblems 4 and 5. In the case of lack of communication between the two processors, processor 1 remains idle, while processor 2 continues to expand subproblems on its local list of active subproblems. The lack of communication between the two processors causes one processor to idle, which leads to imbalance-overhead. It is the irregular structure of the BB tree in this case that makes the lack of communication lead to imbalance-overhead rather than the computation-overhead, which would have occurred had subproblem 2 generated non-essential subproblems.

The effect of the load imbalance generated by the irregular structure of the BB tree on the performance of a parallel BB algorithm for solving the 0–1 integer programming problem is demonstrated in [AbMu88].

4.3.4 Computation-Communication Tradeoff

The amounts of computation-overhead and imbalance-overhead incurred by the DM algorithm can be minimized by perfectly communicating selection and pruning information among the processors. This, for instance, can be performed by broadcasting such information from each processor to all other processors in the DMM. Therefore, a processor can have a global knowledge about the values of h and incumbents in other processors before selecting and expanding a subproblem. This consequently enables the processors to expand subproblems that belong to the set $\mathcal{S}$ and reduce the amount of computation-overhead and communication-overhead. However, this global broadcasting of information in a DMM can be expensive in terms of communication and synchronization costs, and can lead to considerable communication-overhead, which degrades the performance of the

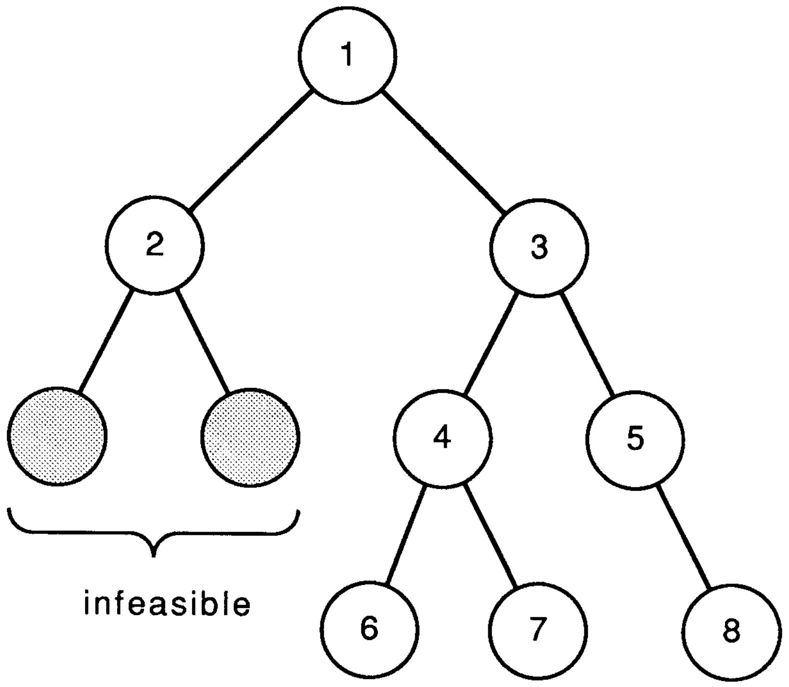

Figure 4.7. Example illustrating possible load imbalance.

DM algorithm.

On the other hand, the amount of communication-overhead can be minimized by eliminating all communication and synchronization among the processors. This, however, implies that no selection and pruning information can be communicated among the processors, and that no load balancing can be performed. This can lead to considerable computation-overhead and imbalance-overhead as illustrated above, and consequently degrade the performance of the DM algorithm.

Therefore there exists a tradeoff between the amounts of computation-overhead and imbalance-overhead on one hand, and the amount of communication-overhead that affects the overall performance of the DM algorithm as shown in figure 4.8. The figure depicts the amounts of the three types of overhead and the total execution time for the algorithm as a function of the amount of selection and pruning information communicated between the processors. At one end of the spectrum, computation-overhead and imbalance-overhead are minimal with complete knowledge of selection and pruning information. However, this is achieved at a high cost of communication, resulting in high communication-overhead. This makes the total execution time taken by the algorithm to find the optimal solution high. At the other end of the spectrum, communication-overhead is minimal since the number of communication steps used to communicate selection and pruning information is small. This, however, results in high computation-overhead and high imbalance-overhead that increases the execution time of the DM algorithm. In the middle of the spectrum, a point exists at which the execution time of the algorithm is minimal. This point is referred to as the *the point of optimal tradeoff*. It represents a tradeoff that results in the best possible performance for the DM algorithm.

The point of optimal tradeoff depends on a number of factors, including the characteristics of the problems being solved, the characteristics of the DMM, and the method that is used to communicate the selection and pruning information. This makes it difficult, if not impossible to analytically determine this point, especially with the highly non-deterministic nature of the BB algorithm that depends to a large extent on the exact problem instance being solved [Trie86].

This tradeoff between computation-overhead and imbalance-overhead on one hand and communication-overhead on the other is referred to as the *computation-communication tradeoff* since it represents a tradeoff between the amount of time spent by a processor in communication and the amount of time spent by the processor either performing additional computations due to non-essential subproblems, or idle performing no computations due to the lack of subproblems on its local list.

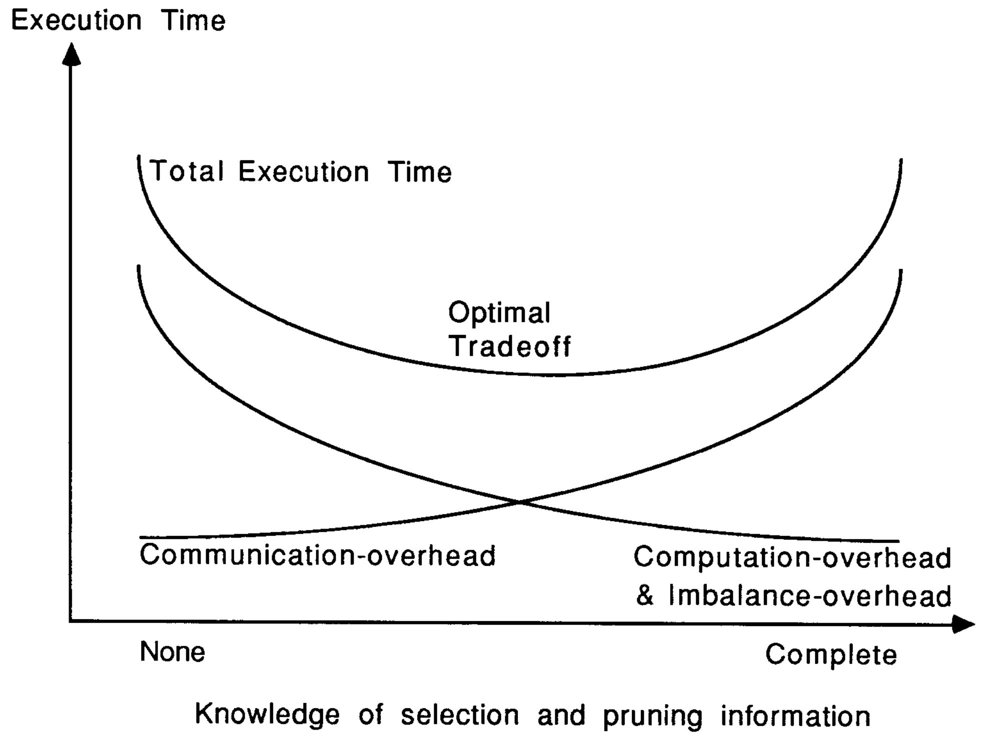

Figure 4.8. The computation-communication tradeoff.

4.3.5 The Model

In order to further characterize the factors that affect the performance of the distributed model algorithm, a simple model for the performance of the algorithm has been developed. The objective of the model is not to quantitatively estimate the performance of the algorithm, but rather to gain insight into the factors that affect its performance in a qualitative manner. A quantitative analysis of the performance of the DM algorithm is difficult to conduct due to the irregular nature of the BB tree that cannot be predicted in advance, to the dependence of the performance of the algorithm on the problem instance to be solved, and to the change in computational characteristics caused by the introduction of parallelism.

In order to simplify the model, the processors in the distributed model are assumed to be synchronized. Although this assumption represents only an approximate picture of the operation of the model, it is adequate for the objectives of the model.

The average time taken by a processor to expand a subproblem is denoted by t_c, and the average time taken by a processor to perform a single communication step such as sending or receiving a message is denoted by t_{comm}. The number of such communication steps performed by a processor during one of its iterations is denoted by C. The number of subproblems expanded by the sequential BB algorithm is denoted by S, which is also the number of essential subproblems. The number of subproblems expanded by processor i in the DM algorithm is denoted by S_i, and it reflects both essential and non-essential subproblems expanded by that processor. The execution time of the sequential BB algorithm can, therefore, be expressed as

$$T(1) = S \ t_c. \tag{4.1}$$

The execution time of the DM algorithm using P processors can be expressed as

$$T(P) = \max\{S_i\} \ \ [t_c + C \ \ t_{comm}] \,. \tag{4.2}$$

Therefore, the speedup of the DM algorithm using P processors is

$$S(P) = \frac{T_s}{T_p} = \frac{S}{\max\{S_i\}} \times \frac{t_c}{t_c + C \ \ t_{comm}} \times P, \tag{4.3}$$

or, equivalently,

$$S(P) = \frac{S/P}{\max\{S_i\}} \times \frac{1}{1 + C\frac{t_{comm}}{t_c}} \times P. \tag{4.4}$$

The factor $\frac{t_c}{t_{comm}}$ is referred to as the *granularity* of computation, and is denoted by g. Therefore, the granularity of computation is a measure of the amount of computations per subproblem performed by the BB algorithm per "unit" communication time. The granularity of computation is coarse (large) when the time taken to expand a subproblem is large compared to the time taken to perform a single communication step, and is fine (small) when the time taken to expand a subproblem is small compared to the time taken to perform a communication step.

The factor $\frac{S/P}{\max\{S_i\}}$ is referred to as the *selection ratio* of the distributed model algorithm, and is denoted by γ. Therefore, the speedup of the algorithm can be expressed as

$$S(P) = \frac{\gamma}{1 + C/g} P. \tag{4.5}$$

The factors that affect the performance of the DM algorithm can be seen from equation 4.5. The speedup of the algorithm is reduced from the perfect value of P by the selection ratio, and by the factor $\frac{1}{1+C/g}$.

The selection ratio of the algorithm reflects computational-overhead and imbalance-overhead incurred by the DM algorithm. Since the number of essential subproblems is S, and these subproblems must be expanded by the algorithm before it terminates, a lower bound on the number of iterations is S/P. The number of iterations taken by the DM algorithm is $\max\{S_i\}$. Both essential and non-essential subproblems are expanded during these iterations. When the communication of selection and pruning information is complete, and the workload is balanced, only essential subproblems are expanded by the algorithm, and the selection ratio of the algorithm is at its maximum value of one. In this case, $\max\{S_i\}$ is close to $\sum_i S_i/P$, which is also close to S/P. However, when the communication of selection and pruning information is not complete, the number of non-essential subproblems expanded by the algorithm increases, and the load becomes more imbalanced, and therefore, the selection ratio of the algorithm drops.

The factor $\frac{1}{1+C/g}$ reflects the effect of communication-overhead on performance. The number of communication steps performed by the algorithm per subproblem to communicate selection and pruning information, and to perform load balancing causes this factor to be larger than one, which reduces the speedup. However, it can be seen that the effect of the communication-overhead is limited by the granularity of computation. The coarser the granularity, the smaller the factor C/g and the smaller the effect of the factor $\frac{1}{1+C/g}$ is on performance.

4.4 Relevant Work on Parallel BB

A number of researchers have studied parallel BB algorithms, and in this section, their relevant work on the subject is reviewed.

An early attempt to parallelize BB algorithms is that made by Imai, Fukumura and Yoshida [ImFY79]. They studied the parallelization of a depth-first BB algorithm on a shared memory multiprocessor architecture. The objective of their study was twofold: (1) to examine the computational efficiency of parallel BB on a shared memory architecture, and (2) to show that it is possible to obtain acceleration anomalies in which the number of subproblems expanded by the parallel algorithm using P processors is less than the number of subproblems expanded by the sequential one (see Chapter 7 for detailed description of acceleration anomalies). The global memory of the multiprocessor is used to store the incumbent and a shared list of active subproblems, maintained in a last-in-first-out order to implement depth-first search. All processors execute the same algorithm and operate asynchronously. A processor executes the same set of steps executed by a processor in the LM algorithm. A processor that is accessing the shared list is given exclusive access to that list through the use of semaphores. The processor then expands the subproblem and inserts the results back on the shared list. The algorithm was not implemented, however, and a simulation was used to study its performance. The simulation, however, ignores all possible overhead due to contention for the shared memory. The simulated parallel depth-first algorithm was used to solve two problems. The first was the minimum covering problem, used to study the performance of the parallel algorithm for a real application. The second was a synthetic minimization problem with n k-valued variables in which the lower bounds of subproblems were generated using an exponentially distributed random distribution. This second problem primarily was used to study possible acceleration anomalies in the execution of the parallel algorithm. The simulation results do not show that the execution time of the parallel algorithm for the two problems considered and only show the number of subproblems expanded in order to determine the existence of anomalies. The simulation results indicate that it is possible for the parallel algorithm to display an acceleration anomaly for both the minimum covering problem and the synthetic problem. That is, the simulation indicates that the number of subproblems that are expanded by the parallel depth-first algorithm can be less in some problem instances than the number of subproblems expanded by the sequential one. An analysis is also given that estimates that the memory requirements of the parallel algorithm increases linearly with the number of processors used by the multiprocessor.

Boehning [Boeh85] implements three parallel algorithms that employ a depth-first search strategy to solve integer linear programming problems on the HEP shared memory multiprocessor [Smit78]. The algorithms are similar to the LM algorithm in that the incumbent and the list of active subproblems are maintained globally in the shared memory of the HEP. The processors operate asynchronously, but execute the same steps the processors in the LM algorithm execute. Each processor selects a subproblem from the list of active subproblems in the shared memory. The processor expands the subproblem it selected and inserts the results back on the list of active subproblems. The three algorithms differ in the action taken by the processors to insert the results back on the list. In the first algorithm, a processor selects a subproblem, expands it, and inserts the two new subproblems after both subproblems have been evaluated using the simplex method (see [Dant63]). Therefore, results of expanding subproblems are not made available to other processors until both new subproblems are evaluated. In the second algorithm, a processor inserts the first of the two new problems on the list once it is evaluated and continues to evaluate the second problem. In this case, the results of expanding a subproblem are made available to other processors one subproblem at a time. Finally, in the third algorithm, a processor expands a subproblem and partially evaluates the new subproblems before inserting them back on the list of active subproblems. Another processor can continue to evaluate the subproblem when it removes that subproblem from the list. It is not clear from the description of the algorithm the extent to which a subproblem is partially evaluated before it is inserted back on the list.

The performance of each of the three algorithms was reported using 7 to 16 processors. Results were given for a set of standard benchmarks for the ILP problem known as the IBM and HALDI problems [Taha75]. All three algorithms show almost the same performance with the second algorithm showing slightly better performance than the other two. This is attributed to a reduced overhead in synchronizing the processors. It is shown that acceleration anomalies happen in which the number of operations performed by the parallel algorithm is less than the number of operations performed by the sequential algorithm. The metrics used to report the performance of the algorithms, however, are not indicative of the true performance of the parallel algorithms. The number of subproblems expanded by the parallel algorithms and the number of pivoting operations performed by the simplex algorithm (see [Dant63]) were the only two metrics used to measure the performance of the algorithms. While they are indicative of some aspects of the performance of the parallel algorithms, they do not reflect overheads in memory access and contention nor do they reflect the actual speedup, or overall performance of the algorithms.

Marz and Seward [MaSe87] implement a parallel depth-first BB algorithm for solving the N-queens problem on the Intel iPSC hypercube multiprocessor [Inte85]. Their research had three main objectives: to exercise an object-oriented approach to writing programs for the iPSC, to measure the performance of a parallel BB algorithm on a parallel computer, and to investigate the suitability of the hypercube architecture for parallel search. Their implementation is object-oriented and uses processing node 0 of the hypercube multiprocessor to execute a *Control* process and each of the remaining $P - 1$ processing nodes to execute a *Worker* process. The Control process monitors the execution of the Worker processes and maintains a global incumbent and a global list of active subproblems. Each Worker process maintains its own local list of active subproblems and its own local incumbent. The Control process initially creates a set of subproblems using a depth-first sequential BB algorithm and inserts them on its global list. It then sends one subproblem from its list to each Worker. (It is not clear how many subproblems are initially created by the Control process nor how the subproblems are assigned to the Workers. A possible assumption, however, is that the number of subproblems is larger than the number of Workers, and that their assignment to the processors is irrelevant). Upon receiving a subproblem, the Worker uses its local list of active subproblems and its local incumbent to apply a depth-first sequential BB algorithm to find the required solution(s) in the subtree rooted at the received subproblem. The Worker reports the results to the Control process when it is done, and requests another subproblem. The Control process updates its incumbent and sends another subproblem to the Worker if one is available on the list. The Control process continues to monitor the execution of the Worker processes until there are no more subproblems on its own list and all the Workers are done, at which point the execution of the algorithm terminates. The performance of the parallel algorithm was reported for the case in which the first solution to the N-queens problem is required and for the case in which all solutions to the problem are required. The results are reported in terms of the performance gain over a VAX 11/780 executing the sequential depth-first algorithm. The results indicate that the performance gain is not significant for the case in which only one solution is required. This is attributed to the communication-overhead incurred by the parallel algorithm and to the small amount computations needed to obtain the first solution. The performance gain increases, however, when all solutions to the problem are required as a result of the increased amount of computation involved, but no scaling performance could be obtained. It is concluded that a more tightly coupled architecture is more suitable for the parallel processing of search problems. This conclusion, however, is based only on the N-queens problem which has a very small granularity, making the communication time between the Control process and the Worker processes significant. Furthermore, since the Worker processes do not communicate, a large number of subproblems that are not expanded by the sequential algorithm are expanded by the parallel algorithm, leading to computation-overhead. This is particularly true in the case where only the first solution to the problem is required. This explains why the performance is better in the case where all solution to the N-queens problem are needed; the number of subproblems expanded by the sequential algorithm is large which makes the effect of the additional subproblems expanded by the parallel algorithm small, hence improving the speedup, which is a measure of the performance of the parallel algorithm relative to the sequential one.

Mohan [Moha83] presents experimental results of two best-first parallel algorithms for solving the traveling salesman problem on the Cm* multiprocessor [GeSS87]. The first algorithm uses a master process to maintain a global list of active subproblems and a global incumbent. The master process controls the operation of P slave processes that perform the computations necessary to evaluate lower bounds of subproblems. The master operates in synchronized iterations with the slaves. At the beginning of an iteration the master selects the subproblems with the smallest lower bound in the list of active subproblems. It then selects $\log_2 P$ edges in that subproblem to generate P subproblems that correspond to all possible combinations of edge inclusion and exclusion (see chapter 3 for a description of the traveling salesman problem). The master then assigns one subproblem to each of the P slaves which computes its lower bound. The slaves return the results back to the master, which inserts the subproblems on the global list. The master process repeats its iteration until there are no subproblems on its list, at which point the algorithm terminates. The speedup of this algorithm was reported to be reasonable for a small number of processors. However, as the number of processors is increased, the speedup degrades and eventually remains constant. This is attributed to the increasing number of non-essential subproblem expanded, which increases the amount of work performed by the parallel algorithm compared to the sequential one, and to the saturation of the communication resources of the Cm*. It is important to note that this algorithm does not exploit algorithm-level parallelism in the sequential BB algorithm for solving the traveling salesman problem. The algorithm performs a parallel branching of a subproblem rather than parallel selection of subproblems.

The second algorithm uses P independent asynchronous processes to implement a parallel asynchronous BB algorithm. Each process maintains its own list of active subproblems and its own incumbent. A process operates in iterations, and during each of these iterations: the process selects the subproblem with the smallest lower bound from its own local list of active subproblems, selects one edge in that subproblem, and then expands the subproblem into two new subproblems; one corresponding to the inclusion of the edge and the other corresponding to the exclusion of the edge. The new subproblems are inserted on the local list of active subproblems. That is, each processor executes the sequential algorithm for solving the traveling salesman problem using its local list and its local incumbent. The processes do not interact in any way during the execution of the algorithm, which terminates when an optimal tour is found. The results, in the form of the speedup of the parallel algorithm, indicate a performance that is close to linear for a small number of processors but degrades as the number of processors is increased. At 16 processors, the speedup is 8. The results also indicate that the performance of the second algorithm is better that the first one. This is attributed mainly to the smaller number of non-essential subproblems expanded by the second algorithm compared with the first. That is, the second algorithm expanded less subproblems than the first, leading to improved performance.

Quinn [Quin86] proposes a number of parallel BB algorithms that use a best-first search strategy for solving the traveling salesman problem on hypercube multiprocessors.

The algorithms attempt to meet two conflicting goals. The first is to keep the processors evaluating essential subproblems, i.e., subproblems that would have been expanded by the sequential algorithm; the second is to minimize the communication overhead along the critical path from the root of the BB tree to the solution node. This communication overhead is incurred when a subproblem on that path is generated by one processor and then later examined by another. The two goals are conflicting because the first requires the redistribution of subproblems over the processors, thereby introducing communication overheads, while the second requires keeping subproblems in one processor, thereby minimizing communication overheads. Four algorithms are proposed that are specific to both the traveling salesman problem and the hypercube architecture. The four algorithms illustrate different approaches for balancing the two goals.

The first algorithm is similar to the LM algorithm and it maintains a global list of active subproblems and a global incumbent in processor 0 of the hypercube. The remaining $P-1$ processors act like slaves to processor 0 and execute the steps of the LM algorithm. Processor 0 initially executes a sequential best-first BB algorithm to generate subproblems for the slaves. When there are as many unexamined subproblems as processors, processor 0 begins each iteration of the parallel algorithm by sending $P-1$ unexamined subproblems such that each processor receives a unique subproblem. Processor 0 then completes the iteration by collecting two newly created subproblems from each of the remaining $P-1$ processors.

In the second algorithm, a list of active subproblems is maintained by each of the processors, and the processors are synchronized in iterations. In each iteration, a processor expands the subproblem with the smallest lower bound in its local list. However, rather than inserting both newly created subproblems on its local list of active subproblems, the processor sends the subproblem with the excluded edge to one of its neighboring processors in the hypercube, and inserts the subproblem with the included edge on its local list. The third algorithm is similar to second, except that when a processor expands a subproblem, it keeps the one with the smaller lower bound and sends the one with the higher lower bound to one of its neighbors. In the fourth algorithm, a processor expands a subproblems and then keeps the two newly created subproblems on its local list. It sends the active subproblem with the second lowest lower bound to one of its neighbors.

The performance of these algorithms was compared using simulation. The results indicate that the least promising algorithm is the first and that the most promising is the fourth. The poor performance of the first algorithm is attributed to the excessive communication between processor 0 and the remaining $P-1$ processors. The promising performance of the fourth algorithm is attributed to its ability to distribute essential subproblems across the processors and to strike a balance between the two goals.

However, as should be clear from the discussion of the factors that affect the performance of parallel BB algorithms on DMMs described in the previous sections, the attempt to achieve the first of the two goals described above is unnecessary. A parallel BB algorithm does not terminate until *all* essential subproblems are expanded. Therefore, minimizing the communication cost along the critical path from the root of the BB tree to the optimal solution can lead to a faster discovery of that solution, but the algorithm must continue to expand all other essential subproblems to verify that the solution is indeed the optimal one. Therefore, attempting to achieve that goal can only lead to additional unnecessary effort. The promising performance of the fourth algorithm should be attributed to its ability to distribute essential subproblems more evenly than the first three algorithms, rather than to its ability to strike a balance between the two conflicting goals.

Anderson and Chen [AnCh86] propose a parallel BB algorithm for a hypercube multiprocessor. The algorithm implements a distributed strategy in which each processor maintains its own list of active subproblems and its own incumbent. Each processor expands subproblems from its local list and inserts new subproblems back on that list. A load balancing strategy is employed to minimize the expansion of non-essential subproblems that would not have been expanded by the sequential algorithm. A figure of merit reflecting the goodness of the subproblems in each processor is computed based on the lower bound values of subproblems in that processor and all subproblems in the neighbors of the processor. This figure of goodness defines a goodness gradient that is used to offload subproblems from processors with a high figure of goodness to processors with a low figure of goodness. The values of the local incumbents in each processor are also exchanged with neighboring processors to update the local incumbents with the minimum value of all incumbents. The process of load balancing, which includes the exchange of lower bounds, the computation of goodness measures, the offloading of subproblems among neighbors and the exchange of incumbents, is performed in an interleaved fashion with the expansion of subproblems. The termination of the algorithm is detected using a set of *completion* messages. When all subproblems in the subtree rooted at a given subproblem are expanded, the subproblem is said to be *terminated.* When a subproblem is terminated in a processor, the processor sends a completion message to the processor containing the parent of the terminated subproblem. When a parent receives completion messages from all its children, it becomes terminated itself and sends a completion message to its own parent. When the root of the BB tree receives completion messages from all its children, the algorithm terminates.

There were no performance results reported for the above algorithm; in fact, the steps of the algorithm as described in [AnCh86] were not specified to the extent that allows its implementation. However, it can be seen that the algorithm is not expected to result in scaling performance due to the excessive communication and synchronization of the processors during the load balancing process. The amount of effort spent in performing the load balancing process can easily overweigh any gains in performance due to a balanced load. Furthermore, the method used for detecting the termination of the algorithm is inefficient and consumes considerable memory space to maintain a list of all expanded subproblems that are not terminated yet. The method can also lead to considerable communication as a completion message may have to travel to several processors before finding the needed parent since subproblems often move from one processor to another due to load balancing.

Wah et al. [WaMa84, WaLY84] propose the design of a special purpose multiprocessor system for the parallel processing of combinatorial optimization problems, including BB problems. It is referred to as *MANIP*, and its general architecture is depicted in figure 4.9. It consists of five major components: M subproblem memory controllers (SMCs); each of which is connected to N general purpose processors; a selection and redistribution network; a global data register; and a secondary storage subsystem that connects to the SMCs through a secondary storage redistribution network. Each of the M SMCs maintains a list of active subproblems (ordered by ascending lower bounds of subproblems) and communicates with one another through the selection and redistribution network. The system operates synchronized in iterations. At the beginning of an iteration the SMCs select $M \times N$ subproblems with the minimum lower bounds and assign one subproblem to each processor. The processors expand and evaluate the $M \times N$ subproblems and insert the results back on the local lists of their SMCs. The processors use a global incumbent that is stored in the global data register to perform elimination tests. Feasible solutions that are found during the iteration are used to update the global incumbent. Concurrent access to the incumbent is facilitated by special hardware associated with the register.

The selection and redistribution network is used to select subproblems for expansion. The network is a unidirectional ring network that connects each SMC to two neighbors, allowing each SMC to send information to one neighbor and receive information from the other. To select the subproblems with the minimum lower bounds, N subproblems with the minimum lower bounds are selected from the local list in each controller and are sent to the neighboring controller. The subproblems received from the neighboring controller are inserted into the local list. This is referred to as *shifting* subproblems. The shifting of subproblems is repeated $M-1$ times. It is shown that each SMC will have N subproblems of the $M \times N$ with the minimal bounds after this number of shifts. This shifting process forms the basis of one of the parallel algorithms that is discussed in Chapter 5. It will be described in greater detail later. However, the use of a special purpose architecture for the parallel processing of the special class of BB algorithms may be impractical; the hardware of MANIP has not be built. Furthermore, there are a number of architectural inefficiencies in MANIP that will become more clear as the its operation is described in Chapter 5. The synchronized operation of the processors can be a major source of inefficiency since the amounts of time taken to expand subproblems vary in many applications. This, as will be seen later, can be a major source of inefficiency. The use of a global data register that facilitates access to the incumbent is impractical, especially that the processors operate in synchronized iterations, and require both read and write access to the contents of that register simultaneously.

Finkel and Manber [FiMa85, FiMa87] describe a distributed implementation of backtracking (DIB) on the crystal multicomputer that consists of a small number of VAX-like workstations connected by a ring network [DeFS87]. They implement a general-purpose package that allows a wide range of problems that employ backtracking and depth-first search to be implemented on a multicomputer. A main objective of the implementation is to make the distributed algorithm transparent to the application programmer by hiding the details of the implementation and providing the application programmer with a well-defined interface.

In the implementation, each machine in the network maintains its own list of active subproblems (represented as a stack) to implement depth-first search. Initially, the problem to be solved is given to one of the machines, and the rest of the machines are idle. The machines expand subproblems independently from one another until a machine runs out of subproblems and becomes idle. The idle machine generates a request for subproblems from another machine in the system. A machine that receives the request examines it own stack of subproblems and sends the requesting machine a portion of its subproblems. This portion always consists of one half the list of active subproblems. Two strategies were used to decide on a machine to request subproblems from. The first is to request subproblems from the successor of the idle machine on the ring. That is, if machine i becomes idle, then work is requested from machine $j = (i + 1) \bmod P$, where P is the total number of machines on the ring. If machine j cannot grant subproblems to machine i, then the request is forwarded to the successor of j. The second strategy is to send a fixed number of requests randomly to other machines on the ring. Experimental results are reported using three problems. The first is the N-queens problem seeking all solution to a problem, which results in excellent performance in terms of both the speedup and the load balance. This excellent performance can be attributed to the large number of N-queens subproblems generated by the sequential algorithm in its search for all solutions which allows the processors to expand the tree without generating many non-essential subproblems (similar to the implementation of Marz and Seward described above). The second problem is the traveling salesman problem. The sequential algorithm used, however, is not an efficient algorithm for solving that problem and generates a large number of unnecessary subproblems. The performance of the parallel implementation is reported as good, which can also be attributed to the generation of many unnecessary subproblems by the sequential algorithm. The third problem is an alpha-beta game tree search for the NIM game for which moderate performance is reported. The performance for the alpha-beta game tree is poor because of better performance of that algorithm in the sequential case which causes the parallel algorithm to expand a larger number of non-essential subproblems compared to the first two cases the performance is reported for.

Kumar, Rao and Ramesh [KuRR88] extend the work of Finkel and Manber [FiMa87] by introducing a modified strategy to determine which machines to request subproblems from. They implement their strategy on an Intel iPSC hypercube multiprocessor [Inte85], which was used to embed a ring, for iterative deepening depth-first search. However, only the last iteration of the algorithm is implemented and a search for all solutions is made. The strategy assigns a special machine as a coordinator to which idle machines send their requests for subproblems. The coordinator maintains a counter I which is used to indicate the number, or identity of the machine to request work from. When the coordinator receives a request for subproblems from an idle machine, it returns the value of I to the requesting machine and increments that value of I. The requesting machine then re-submits its request to machine I. Theoretical analysis and experimental results for the traveling salesman problem indicate that their modified strategy results in better performance than the one presented in [FiMa87]. The performance of the algorithm, however, is biased by the fact that only the last iteration of the algorithm is implemented, which practically makes the value of the optimal solution is known while the search is conducted, and hence little or no computation-overhead is incurred by the algorithm. The search for all solutions also causes the performance of the algorithm to look good when compared to the performance of the sequential one as described earlier.

Schwan, Gawkowski and Blake [ScGB88] discuss a similar scheme for solving the traveling salesman problem on the Intel iPSC hypercube. They employ best-first search, however, and investigate a number of issues relating to their implementation which is similar to that of Kumar et al. They employ one processor as a coordinator that initially decomposes the original problem into subproblems and assigns one subproblem to each processor. The coordinator is also responsible for receiving and broadcasting feasible solutions, load balancing and determining the termination of the algorithm. The processors expand their subproblems independently. A processor that becomes idle after exhausting its subproblems sends a signal to the coordinator, which determines a busy processor and instructs it to send subproblems to the idle one. When all of the processors become idle, the algorithm terminates. Results are reported for the traveling salesman problem for up to 20 processors. The performance of the algorithm degrades after a few processors even though it is given only for the best case problems: this was attributed to the contention for the coordinator. The work is extended to include two level coordinators. The master coordinator assigns subproblems to the coordinators which in turn assigns problems to the remainder of the processors. The performance results reported indicate that the use of this multi-level coordination further degrades the performance of the algorithm. Although the reported poor performance was attributed to the contention for the coordinator, the independent operation of the processors without any exchange of selection information can lead to significant computation-overhead, which also can account for the poor performance.

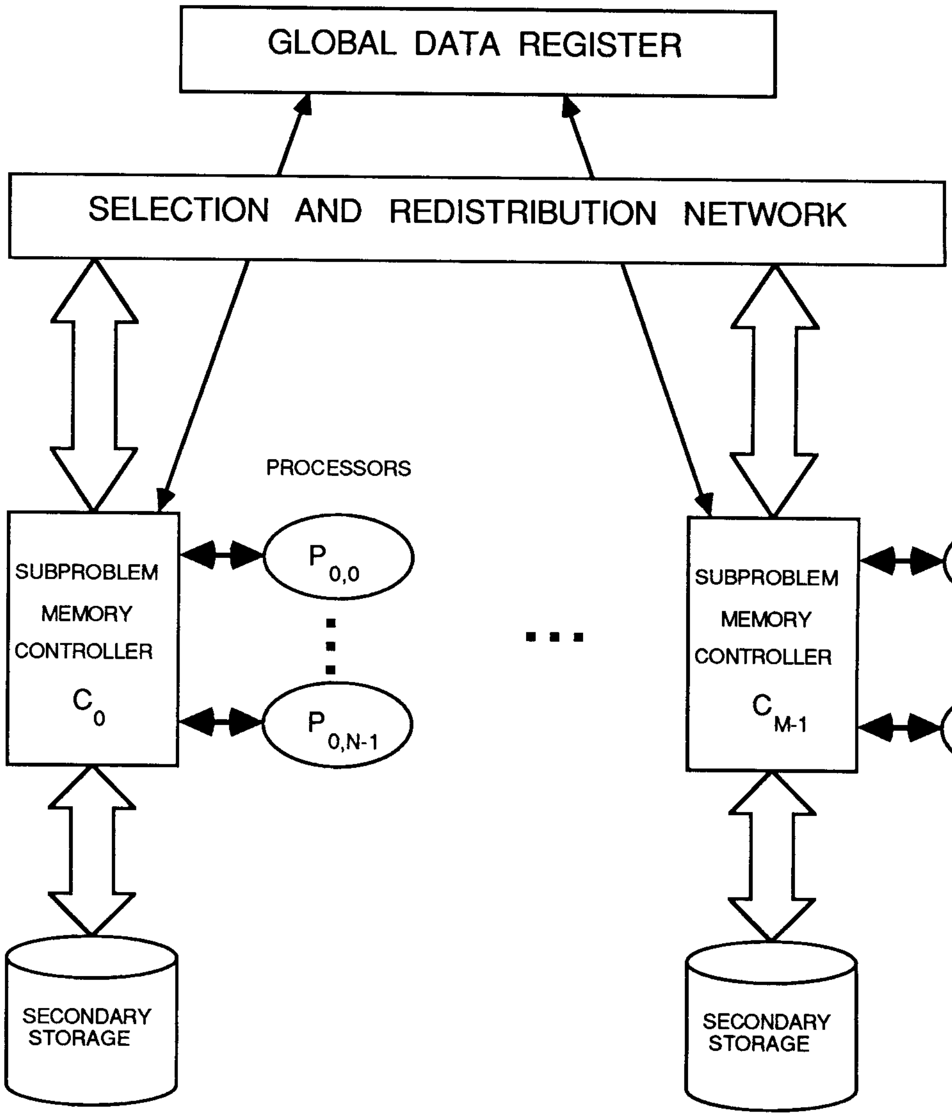
GLOBAL DATA REGISTER
SELECTION AND REDISTRIBUTION NETWORK
PROCESSORS
SUBPROBLEM MEMORY CONTROLLER C_0
$P_{0,0}$
$P_{0,N-1}$
SUBPROBLEM MEMORY CONTROLLER C_{M-1}
$P_{M-1,0}$
$P_{M-1,N-1}$
SECONDARY STORAGE
SECONDARY STORAGE

Figure 4.9. The architecture of MANIP.

CHAPTER 5

THE PARALLEL ALGORITHMS

In this chapter three parallel algorithms that map the sequential BB algorithm on hypercubes are described. The algorithms employ different strategies for mapping the algorithm on multiprocessors with no shared memory. In particular, the three algorithms reflect different computation-communication tradeoffs.

The first algorithm is referred to as the Central List (CL) algorithm. It is an asynchronous master-slave algorithm that employs a centralized strategy similar to that used by the LM algorithm, which, as noted before, has been used by most researchers for parallel BB in the past. The algorithm reflects a computation-communication tradeoff in which computation-overhead and imbalance-overhead are minimized at a cost of high communication-overhead. The algorithm represents a common strategy that has been used to map BB algorithms onto multiprocessors [ImFY79, Boeh85, MaSe87, Moha83, ScGB88].

The second algorithm is referred to as the SHIFT algorithm, and it employs a load balancing strategy that is based on the operation of MANIP (see previous chapter), a proposed special purpose multiprocessor architecture for the parallel processing of combinatorial optimization problems. The load balancing strategy aims to expand subproblems with the globally best values of the selection heuristic function h, and hence to reduce computation-overhead and imbalance-overhead. This is achieved using a high degree of communication and processor synchronization, which both lead to considerable communication-overhead. Consequently, the algorithm represents a computation-communication tradeoff in which computation-overhead and imbalance-overhead are reduced at a cost of high communication-overhead, similar to the CL algorithm. The strategy of the algorithm is a representative of many strategies used in the past to map the BB algorithm onto DMMs [WaMa84, AnCh86, Quin86, PaWo88].

Finally, the third algorithm is referred to as the Distributed List (DL) algorithm and it uses a new load balancing strategy that attempts to improve performance by reducing communication-overhead in two aspects of the operation of the algorithm. First, the load balancing strategy aims to select and expand essential subproblems rather than subproblems with the globally best values of h, requiring less communication of selection information. Second, the load balancing strategy allows the processors of the hypercube to operate completely asynchronously, eliminating communication-overhead due to processor synchronization. This reduction in communication-overhead increases the amounts of computation-overhead and imbalance-overhead incurred by the DL algorithm in comparison to the first two algorithms. Hence, the DL algorithm represents a computation-communication tradeoff in which communication-overhead is reduced at the cost of higher computation-overhead and higher imbalance-overhead.

The three algorithms are presented in the following three sections. In each section, the operation of one algorithm is first described followed by a discussion of the factors that lead to computation-overhead, imbalance-overhead, and communication-overhead in that algorithm.

5.1 The CL Algorithm

In the CL algorithm, the P processors of the hypercube are used such that processor 0 is a *master* and the remaining $P - 1$ processors are *slaves.* The master controls the operation of the slaves, and maintains a global list of active subproblems and a global incumbent. In addition, the master maintains the status of the individual slaves. A slave is *busy* if it is active computing, or *idle* if it is not. A slave does not maintain a local list of active subproblems, but does maintain a local incumbent.

The master operates in iterations examining the list of active subproblems and the status of the slaves in the beginning of each iteration. The master removes the subproblem with the smallest value of the selection heuristic function h from the list of active subproblems and sends it to an idle slave. The status of that slave changes from idle to busy. The master then re-examines the list of active subproblems and the status of the slaves, and repeats sending subproblems to idle slaves until all slaves are busy or the list of active subproblems becomes empty. The master then idles until new subproblems are returned by a slave.

The subproblem received by a slave from the master is decomposed into smaller subproblems using the branching procedure of the BB algorithm. The termination procedure is then applied to delete new subproblems that can never lead to feasible solutions. The new subproblems are then examined and the lower bound of a new subproblem representing a feasible solution is compared to that of the local incumbent. If smaller, the local incumbent is replaced by the new feasible solution. Similar to the sequential BB algorithm, a subproblem that represents a feasible solution but does not update the incumbent is discarded. The local incumbent of the slave is then used to perform the lower bound elimination test on the new subproblems. All remaining new subproblems and a copy of the best feasible solution that updated the local incumbent are sent back to the master.

The master receives the new subproblems and any feasible solutions from the slave and changes the status of that slave back to idle. The lower bound of a feasible solution that updated the local incumbent of the slave is compared to that of the global incumbent and if smaller, the feasible solution updates the global incumbent. Similar to the sequential BB algorithm when the incumbent is updated, all subproblems on the list of active subproblems whose lower bounds are greater than or equal to that of the new global incumbent are deleted. The master uses the global incumbent to perform the lower bound elimination test on the new subproblems. The remaining subproblems are then inserted on the list of active subproblems.

The iteration of the master is completed after the new subproblems are inserted on the global list of active subproblems. A new iteration immediately begins. It is important to note that the master can start an iteration by sending a subproblem to an idle slave, but ends that same iteration by inserting on its list subproblems that have been received from another slave.

The master continues its iterations until the list of active subproblems is empty and all the slaves are idle, at which time the algorithm terminates. The optimal solution is stored in the global incumbent of the master. An outline of the major steps of the CL algorithm is depicted in figure 5.1.

The use of the global incumbent by the master to perform the lower bound elimination test on new subproblems received from a slave is necessary and is not duplicative of that performed by the slave using its local incumbent. The incumbent maintained by the slave is updated only by feasible solutions discovered by that slave, and hence, represents the best feasible solution in a local sense and not in a global sense. On the other hand, the global incumbent maintained by the master is updated by feasible solutions discovered by all slaves and, therefore, represents the best feasible solution in the global sense. Accordingly, a subproblem that escapes deletion by the slave may be deleted by the master if the global incumbent of the master represents a better feasible solution than that represented by the local incumbent of the slave. Therefore, it is necessary to perform the lower bound elimination test using the global incumbent to avoid the insertion of such a subproblem on the list of active subproblems.

```
Master
  1. Examine list of active subproblems and the status of the slaves.
  2. Repeat until the list is empty or all slaves are busy:
       send to an idle slave the subproblem with the smallest
       value of the selection heuristic function.
  3. Idle until a slave returns its results.
  4. Check for feasible solutions and update global incumbent
     if necessary.
  5. Perform lower bound elimination test on new subproblems using
     the global incumbent and insert remaining subproblems on list.
  6. Repeat steps 2-5 until list is empty and all slaves are idle.


A slave
  1. Idle until a subproblem is received from master.
  2. Apply branching procedure.
  3. Apply termination procedure
  4. Check for feasible solutions and update local incumbent
     if necessary.
  5. Perform lower bound elimination procedure using local incumbent.
  6. Send results back to master, including any feasible solutions
     that updated the local incumbent.
  7. Repeat steps 1-6.
```

Figure 5.1. Outline of the CL algorithm.

The use of the local incumbent by a slave to perform the lower bound elimination test is strictly unnecessary since any subproblem that is deleted by that test using the local incumbent of the slave can be deleted by the same test using the global incumbent of the master. Nevertheless, the test is performed using local incumbents to reduce communication-overhead between the master and the slaves. The deletion by the slave of subproblems that will be deleted when sent back to the master reduces the length of the communication messages between the master and the slaves. This can lead to a reduction in overhead due to communication, particularly when the size of a subproblem is large.

The operation of the CL algorithm is more similar to the operation of the LM algorithm than to the operation of the DM algorithm. In the CL algorithm a global list of active subproblems and a global incumbent are maintained by the master, similar to the LM algorithm. Moreover, the master operates in iterations also similar, but not identical, to those of the LM algorithm. Nevertheless, the performance of the CL algorithm is affected by the same factors that affect the performance of the DM algorithm. These factors are described below using a simple model for the performance of the CL algorithm. The model is similar in nature and scope to the one described in section 4.3.5 for the DM algorithm, but is more specific to the CL algorithm. Although the model is similar to that of the DM algorithm, it is presented here in the context of the CL algorithm to clarify the factors that affect the performance of this algorithm.

The sequential BB algorithm is assumed to expand S_1 subproblems. The CL algorithm using P processors is assumed to expand $S_P \geq S_1$ subproblems. The subproblems are assumed to be equally expanded by the $P-1$ slaves. That is, a uniform and a well balanced workload is assumed. The average time taken to expand a subproblem is denoted by T_E. The execution time of the sequential algorithm is given by

$$T(1) = S_1 \times T_E. \tag{5.1}$$

The execution time and speedup of the CL algorithm using P processors can be expressed as

$$T(P) = \frac{S_P}{P-1}\left[T_O(P) + T_E\right], \tag{5.2}$$

and

$$S(P) = \left[\frac{S_1}{S_P} \times \frac{1}{1 + \frac{T_O(P)}{T_E}}\right] \; P-1 \tag{5.3}$$

respectively.

The number of subproblems expanded by each slave is $\frac{S_P}{P-1}$. The maximum speedup that can be attained by the algorithm is $P-1$. This is due to the use of processor 0 as the master, which only controls the operation of the slaves but does not participate in the expansion of subproblems. The speedup of the algorithm is degraded by two factors.

The first is represented in equation 5.3 by the term $\frac{1}{1+\frac{T_O(P)}{T_E}}$, which reflects the effect of communication-overhead incurred by the algorithm. The term $T_O(P)$ denotes the average amount of overhead incurred per subproblem by a slave to receive a subproblem from the master, to send new subproblems to the master, and to wait idle for the master to send the next subproblem to expand. This overhead reflects the time spent in sending and receiving subproblems, and the time spent in performing the lower bound elimination test using the global incumbent by the master. That is, $T_O(P)$ reflects overhead due to communication of subproblems between the master and the slaves, as well as overhead due to serial processing by the master.

The amount of overhead due to communication is affected by the number of the slaves used. This is due to possible contention for the communication resources of the master by the slaves. This contention affects the time taken to send or receive a subproblem to or from a slave. When the number of slaves is small, the contention for the master is negligible, and has little or no effect on the time to communicate a subproblem. However, when the number of slaves is large, the contention for the master is high and the time taken to send or receive a subproblem increases. This, in turn, increases the amount of overhead due to communication of subproblems.

The extent of the effect of communication-overhead on the overall performance of the CL algorithm is influenced by the average time for expanding a subproblem T_E. The term $\frac{1}{1+\frac{T_O(P)}{T_E}}$ indicates that it is the ratio of the $T_O(P)$ to T_E that directly affects the overall performance of the algorithm rather than the absolute amount of $T_O(P)$. Therefore, when T_E is large compared to $T_O(P)$, the effect of $T_O(P)$ is reduced and the effect of communication-overhead on overall performance also is reduced. However, when T_E is small compared to $T_O(P)$, the effect of communication-overhead on overall performance is more significant.

The second factor that affects the performance of the CL algorithm is computation-overhead and is indicated by the term $\frac{S_1}{S_P}$ in equation 5.3 above. This term reflects the expansion of non-essential subproblems by the CL algorithm due to lack of parallelism as well as lack of selection and lack of pruning information. Although a global list of active subproblems and a global incumbent are maintained by the master, there is lack of selection and lack of pruning information in the CL algorithm, due to the lack of knowledge on part of the master about selection and pruning information in the slaves. This lack of knowledge is caused by the asynchronous operation of the slaves coupled with communication delays between the master and the slaves, which is illustrated by the following two situations.

In the first, it is possible that a non-essential subproblem be removed from the list of active subproblems and be sent to an idle slave even though a new incumbent has been generated by another slave, but has not been received by the master due to communication delays. This subproblem would have been deleted by the master had it not been for the delay in receiving the incumbent. This subproblem is also deleted by the LM algorithm since the processors in that algorithm operate in synchronized iterations and subproblems are not selected for a new iteration until the computations of the previous iteration in all processors are completed. Therefore, a computation-overhead is introduced due to the lack of pruning information on part of the master.

In the second situation, it is possible that a non-essential subproblem be removed from the list of active subproblems and be sent to an idle slave even though an essential subproblem has been generated by another slave, but also has not been received by the master due to communication delays. Therefore, a computation-overhead is introduced due to the lack of knowledge on part of the master about selection information in the second slave.

The master in the CL algorithm maintains the workloads of subproblems balanced across the processors using a *self-scheduling* load balancing technique. The master maintains the status of all the slaves, and assigns a new subproblem to a slave as soon as that slave returns the results of its previous computation and becomes idle. Therefore, subproblems are assigned to slaves based on the demand by the slaves for subproblems and the availability of the subproblems, which keeps all the slaves as busy as possible and results in an even workload across the processors [Poly88]. Consequently, little or no imbalance-overhead is incurred by the CL algorithm.

It is important to note that the CL algorithm has a serious disadvantage; it requires a large memory on the master to maintain the global list of active subproblems. This can be limiting to the size of problems that can be solved by this algorithm when the size of memory on the master processor is limited, and no support for virtual memory is provided. Furthermore, the algorithm poorly utilizes the memory resources of the hypercube multiprocessor. While the memory of processor 0 is well utilized, the memories of the remaining processors are not.

5.2 The SHIFT Algorithm

The SHIFT algorithm is based on the operation of MANIP— the special purpose multiprocessor proposed by Wah and Ma for the solution of combinatorial optimization problems using the BB algorithm [WaMa84, WaLY84]. The general architecture of MANIP has already been described in Chapter 4. In this section the operation of MANIP and the algorithm it uses are reviewed before the SHIFT algorithm is described.

The architecture of MANIP is diagramed in figure 4.9. Central to that architecture are the M subproblem memory controllers that are used to maintain M independent lists of active subproblems, one in each controller. The memory controllers are connected by the selection and redistribution network, which connects each controller to two others in a ring topology. There are N processors attached to each memory controller. The N processors expand subproblems from the list of active subproblems maintained by the controller they are attached to. A global data register is used to store the incumbent, making it accessible to all processors.

The operation of MANIP is described here for the special configuration in which only one processor is attached to each memory controller (i.e., $N = 1$). This configuration is suggested to be the most cost-effective for the architecture, especially when the number of controllers is large [WaMa84]. Moreover, this is the configuration upon which the SHIFT algorithm is based. A complete description of the operation of MANIP for general configurations can be found in [WaMa84, WaLY84].

The subproblem memory controllers are synchronized and operate in iterations. All the controllers perform the same set of operations in an iteration, which consists of two main phases. In the first phase, $M - 1$ *shifts* are performed in order to redistribute subproblems in the memory controllers. The subproblems are redistributed such that each controller contains one subproblem from the set $\mathcal{S}$ consisting of the M subproblems with the globally smallest values of the selection heuristic function h. This is referred to as a *complete distribution* of subproblems. In the second phase of an iteration, a memory controller removes the subproblem with the smallest value of h from its local list of active subproblems, and assigns the subproblem to its associated processor for expansion. The subproblem removed by the controller is also the one that belongs to the set $\mathcal{S}$ in that controller. The subproblem is expanded by the processor and the results are returned to the memory controller, where they are inserted on the local list of active subproblems maintained by the controller.

In a shift a subproblem memory controller removes from its local list of active subproblems the subproblem with the smallest value of h and sends it to its right neighboring controller on the ring. The subproblem memory controller also receives a subproblem from its left neighbor and inserts it on its local list. This shift is repeated $M - 1$ times in each iteration to achieve a complete distribution.

The complete distribution of subproblems in the memory controllers by repeated shifts is illustrated in figure 5.2 for $M = 4$. The four memory controllers are represented by the rectangular boxes and are connected by the ring as shown in the figure. The four subproblems that belong to the set $\mathcal{S}$ are shown as black dots inside the boxes. The remaining subproblems in each memory controller are not shown in the figure since their existence will not affect the redistribution of the four subproblems. The initial distribution of the subproblems in the controllers is shown in figure 5.2(a). There are three subproblems in processor 2 and one subproblem in processor 4. The distribution of the four subproblems in the processors after each of the three shifts needed for a complete distribution is shown in figure 5.2(b), (c) and (d) respectively. The shifts are performed in a counter clockwise direction.

The $M - 1$ shifts performed in each iteration by the memory controllers represent the maximum number of shifts that are needed to achieve a complete distribution of subproblems (the $M - 1$ shifts are needed when the M subproblems that belong to the set $\mathcal{S}$ are all in one memory controller). That is, this number of shifts is sufficient but is not necessary to achieve a complete distribution. Indeed, it is possible to have a complete distribution with a smaller number of shifts, depending on the initial distribution of the M subproblems that belong to the set $\mathcal{S}$ in the memory controllers. Furthermore, a number of shifts that is less than $P - 1$ can be used in each iteration to achieve a close to complete distribution of subproblems. That is, a redistribution of subproblems in which *most* of the controllers receive one of the subproblems that belong to the set $\mathcal{S}$. This is important since performing $M - 1$ shifts in each iteration can be expensive in terms of communication costs. Theoretical analysis and simulation of MANIP were used in [WaMa84] to determine the effect of the number of shifts per iteration on performance. The performance was measured in terms of the number of iterations needed to solve a given problem. The simulation used the vertex covering problem to obtain the performance results. The results indicated that

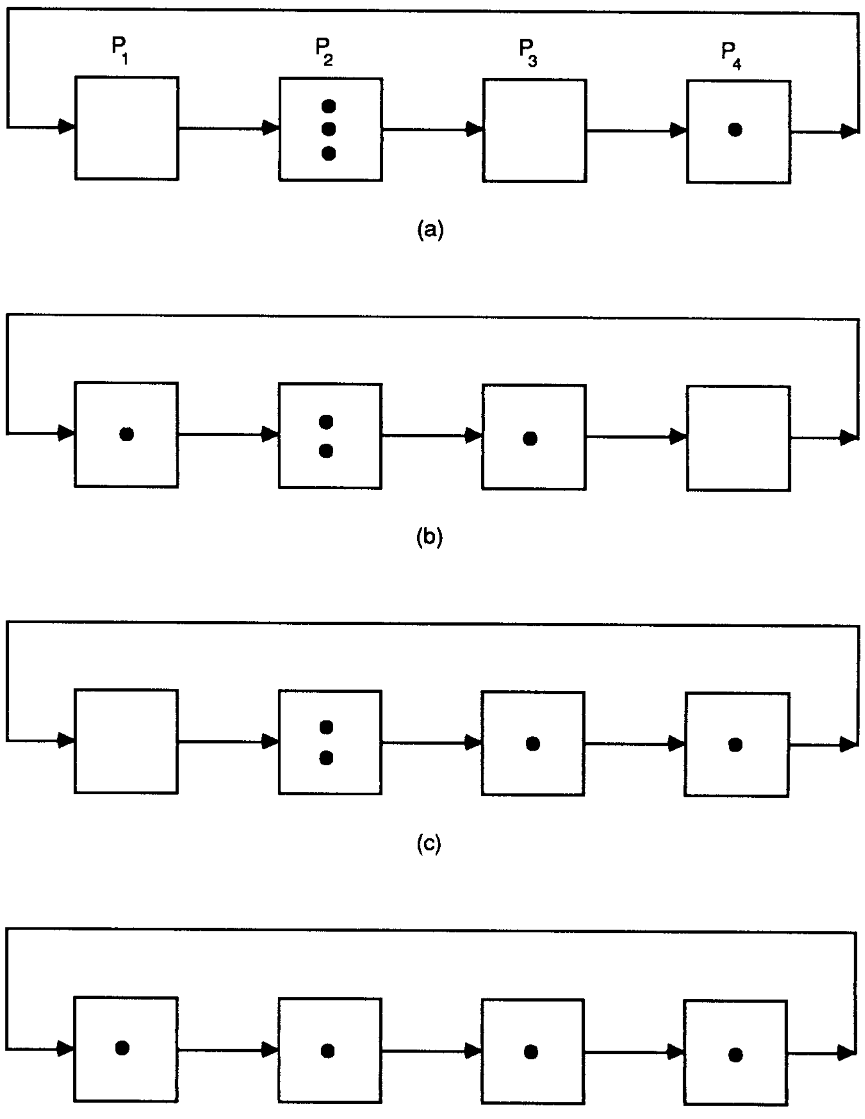

Figure 5.2. The process of shifting subproblems to achieve complete distribution.

only a small number of shifts per iteration is sufficient to achieve a very close to complete distribution. Therefore, in the operation of MANIP, only one shift in each iteration is performed. It is important to note that the simulation conducted by Wah and Ma used a mixed best-first/depth-first search strategy due to memory limitations of the machine the simulation was conducted on [WaMa84]. The search strategy is best-first until memory is exhausted, at which time the search strategy is switched to depth-first to conserve memory. The details of that change were not described in [WaMa84], nor was its effect on performance.

The expansion of a subproblem by a processor during the second phase of an iteration proceeds in a manner similar to that of a sequential BB algorithm. The subproblem is decomposed into smaller subproblems using the branching procedure. New subproblems that can never lead to feasible solutions are deleted by the termination procedure. The lower bounds of subproblems that represent feasible solutions are compared to the lower bound of the global incumbent maintained in the global data register. If the lower bound of a new feasible solution is smaller than that of the incumbent, the incumbent is replaced by the new feasible solution, and all subproblems on the local list of active subproblems that have lower bounds greater than or equal to that of the new incumbent are deleted. The incumbent is used to perform the lower bound elimination test and the remaining subproblems are sent to the subproblem memory controller, where they are inserted on the list maintained by that controller.

The processors have a contention-free access to the contents of the global data register that is facilitated by the hardware of MANIP. Since it is possible for multiple feasible solutions to be discovered by multiple processors and to update the incumbent in the same iteration, only the one with the smallest lower bound is used to update the incumbent and to become immediately available to all processors to perform their lower bound elimination tests. This mechanism for updating the incumbent is also overhead-free and is facilitated by the hardware of MANIP. The architectural design that facilitates this type of access was not described in [WaMa84, WaLY84]. However, as indicated earlier in Chapter 4, the cost of realizing this access can be expensive and impractical.

The iterations of the subproblem memory controllers continue until there are no subproblems to be expanded in all the processors. The detection of this condition is not described in [WaMa84] and is assumed to be performed by the hardware of MANIP.

In the SHIFT algorithm, the role of a subproblem memory controller and its attached processor is assumed by a processor in the hypercube array. There are P independent lists of active subproblems maintained by the P processors of the hypercube, one in each processor. There is no global incumbent due to the lack of global memory; rather, a local incumbent is maintained by each processor. This is unlike MANIP in which a single incumbent is maintained in the global data register. The processors operate asynchronously, again unlike the operation of the subproblem memory controllers of MANIP. A processor operates in iterations and all processors perform the same set of operations in an iteration. There are two phases in an iteration. In the first phase subproblems are redistributed by a process of repeated shifts similar to that used in MANIP. In the second phase a processor removes the subproblem with the smallest value of h from its local list and expands it.

The shifts are performed by the processors over a ring that is embedded in the hypercube array. The embedding of rings into hypercubes is described in Appendix A. In each of its iterations, a processor performs $s \leq P - 1$ shifts over the embedded ring. In each shift, the processor removes the subproblem with the smallest value of h from its local list and sends it to its right neighbor on the embedded ring. The processor then receives a subproblem from the left neighbor on the embedded ring and inserts it on the local list. It is possible that a processor has an empty list of active subproblems. In this case, the processor also participates in the shift, but rather than sending a subproblem it sends a special null message indicating that no subproblems are available on its local list.

A processor expands subproblems from its local list in a manner similar to a sequential BB algorithm. The subproblem with the smallest value of h is selected from the local list of active subproblems. The subproblem is decomposed using the branching procedure. Subproblems that can never lead to feasible solutions are deleted using the termination procedure. The lower bounds of subproblems that represent feasible solutions are compared to the lower bound of the local incumbent, and a feasible solution that has a lower bound smaller than that of the local incumbent replaces that incumbent. All subproblems on the local list of active subproblems whose lower bounds are larger than that of the new local incumbent are deleted. The local incumbent is used to perform the lower bound elimination test on subproblems. The remaining subproblems are finally inserted on the local list of active subproblems.

A feasible solution that updates the local incumbent of a processor is broadcast to all other processors in the hypercube. The broadcast is performed using a spanning tree algorithm that is described in detail in Appendix A. When a processor receives the feasible solution, it compares the lower bound of the solution to the lower bound of its local incumbent. If the lower bound of the feasible solution is less than that of the local incumbent, the feasible solution replaces the incumbent; otherwise the feasible solution is discarded. That is, a received feasible solution does not automatically update the local incumbent of a processor. This is necessary since multiple feasible solutions can be discovered by different processors, and it is possible for a processor receiving the feasible solution discovered by another processor to have discovered a better one and updated its own incumbent accordingly.

The termination of the SHIFT algorithm is detected by performing $P-1$ shifts in a special iteration. At regular intervals, a processor executes a special iteration in which $P-1$ shifts are performed instead of the s shifts performed in regular iterations. A sufficient condition for the processors to terminate is that a processor i with an empty local list of active subproblems receives no subproblems from its left neighbor on the ring during all of the $P-1$ shifts. This is easily deduced since a subproblem in any processor takes at most $P-1$ shifts to reach processor i. Therefore, not receiving any subproblems for that many shifts implies that all the remaining lists of active subproblems are also empty. It is possible that other methods for detecting the termination of the algorithm be

```
1. Repeat s < P times the process of shifting subproblems
     - remove the subproblem with the smallest value of the selection
       heuristic function from the local list of active subproblems.
     - send the subproblem to the right neighbor in the ring.
     - receive a subproblem from the left neighbor in the ring and
       insert on the local list of active subproblems.
2. Remove the subproblem with the smallest value of the selection
   heuristic function from the local list of active subproblems.
3. Apply the branching procedure.
4. Apply the termination procedure.
5. Check for feasible solutions, and update incumbent if necessary.
   If incumbent is updated, the new incumbent is broadcasted to
   all processors.
6. Apply the elimination procedure using local incumbent.
7. Repeat steps 1-7 until all lists of active subproblems are empty.
```

Figure 5.3. Outline of the SHIFT algorithm.

employed, as will be described in the following section. However, this method, although expensive, is used as it also contributes to the redistribution of subproblems among the processors. An outline of the operation of the SHIFT algorithm is depicted in figure 5.3.

The processors in the SHIFT algorithm execute using the Single Code Multiple Data (SCMD) model of operation (see [Buzz88]), and do not synchronize at the end of each iteration. Therefore, the processors can operate in an out-of-phase fashion with respect to each other in their iterations. For instance, a processor can be starting a new iteration while another processor is still in the middle of its current one. This out-of-phase execution results from the unequal amounts of time taken to expand different subproblems coupled with the asynchronicity of the processors. A processor that takes a shorter period of time to expand its subproblem proceeds with its next iteration, leaving the remaining processors behind in their iterations, and becoming out-of-phase with them.

It is important to realize, however, that although the processors operate asynchronously and do not explicitly synchronize at the end of each iteration, they effectively synchronize through their action of repeated shifts. A processor in the SHIFT algorithm does not start a new shift until the current one is complete. Therefore, while a processor can send the subproblem with the smallest value of h from its local list to its right neighbor without synchronization, the processor must idle if necessary until its receives a subproblem from its left neighbor. In other words, each processor is synchronized with its left neighbor during the shift and is forced not to proceed with the rest of its iteration until the subproblem sent by that neighbor is received. This synchronization "ripples" through the ring and causes the P processors to indirectly synchronize, and imposes the restriction that the same set of subproblems that are shifted in a synchronous iteration of MANIP also be shifted by the processors in the SHIFT algorithm. Furthermore, since a processor does not expand a subproblem until the process of repeated shifts is complete, the processor must idle if necessary until it receives the subproblem it would have received had the operation of the processors been synchronized in iterations. Therefore, the SHIFT algorithm expands the same set of subproblems expanded by MANIP, albeit in an out-of-phase fashion due to the asynchronicity of the processors. Consequently, when the number of shifts in each iteration are $P-1$, the processors in the SHIFT algorithm expand subproblems with the globally smallest values of h, which is the desired set of subproblems to be expanded. However, this synchronization, as well be seen later, is a major source of communication-overhead.

The above mechanism for synchronizing the processors in the SHIFT algorithm has the advantage of reducing the amount of idle time spent by the processors compared with a synchronization mechanism in which all the processors are synchronized at the end of each iteration. In the SHIFT algorithm, a processor is allowed to proceed as much as possible into its iteration until it becomes necessary for it to wait for other processors. Although a processor must wait until its left neighbor finishs its iteration and sends it a subproblem before proceeding with its own iteration, the processor may send a subproblem to its right neighbor in the beginning of a new iteration without having to wait for its left neighbor, which may still be in the middle of its iteration. In contrast, in a mechanism that requires all the processors to synchronize at the end of each iteration, a processor that completes its computation must wait for the remaining processors to complete their iterations before it can proceed with a new one, although, as described above, part of that new iteration can be performed without having to wait for the rest of the processors, which can cause mode idle time for the processors.

There are two main factors that can lead to computation-overhead in the SHIFT algorithm in addition to possible lack of parallelism in problems. The lack of a global incumbent and the existence of multiple local incumbents in the processors is the first. A feasible solution that updates the local incumbent of a processor cannot immediately update the local incumbents of other processors. This is due to the number of communication steps needed to broadcast the feasible solution from the processor that discovered it to another processor in the hypercube ($O(\log_2 P)$ on the average, see Appendix A). These communication steps take a measurable amount of time, leading to time delays in incumbent updates. Consequently, a processor that receives the new feasible solution delayed in time may expand a non-essential subproblem that would have been deleted had the feasible solution been received with no time delay. This gives rise to computation-overhead caused by lack of pruning information.

The second factor that can lead to computation-overhead in the SHIFT algorithm is the number of shifts performed by a processor in each iteration. The use of $P-1$ shifts in each iteration guarantees a complete distribution, which in turn guarantees that a processor selects one of the P subproblems with the globally smallest values of h in any of its iterations. Such a selection of subproblems is sufficient for the selection of essential subproblems (as has been described in section 4.3). Therefore, no computation-overhead caused by lack of selection information can occur when $P-1$ shifts are used. However, when the number of shifts is less than $P-1$, a complete distribution of subproblems is not guaranteed and a processor may select and expand a non-essential subproblem although multiple essential subproblems exist in another processor. This gives rise to computation-overhead due to lack of selection information.

The number of shifts used by the SHIFT algorithm can also lead to load imbalance, and consequently, to imbalance-overhead. The use of $P-1$ shifts to achieve a complete distribution also guarantees that each processor receives a subproblem to expand when at least P subproblems exist. However, when a smaller number of shifts is used, there is no guarantee that an idle processor receives a subproblem to expand after the shifts are performed. This causes the workload across the processors to be imbalanced and leads to imbalance-overhead.

There are two factors that lead to communication-overhead in the SHIFT algorithm. The first is the time spent by the processors in sending and receiving subproblems to and from neighbors over the ring. This communication time leads to communication-overhead. The amount of the communication time increases with the number of shifts since more subproblems must be communicated when the number of shifts is increased. Hence, the contribution of this factor to communication-overhead is small when the number of shifts is small, and is large when the number of shifts is large.

The second factor that leads to communication-overhead is the idle time spent by the processors due to synchronization. The processors in the SHIFT algorithm operate asynchronously, but effectively synchronize in the manner described above while distributing subproblems. The idle time is incurred by a processor waiting to receive a subproblem from its left neighbor during a shift.

The amount of idle time spent by the processors is particularly affected by the uneveness in the amounts of time taken to expand subproblems. This uneveness causes some processors to finish the expansion of their subproblems before others, and hence, to idle waiting to participate in the shifts as described above. When the subproblems expanded take the same amount of time, the amount of idle time is negligible. However, when different subproblems take different times to expand, many processors idle during the shifts and the effect of the idle time can be significant.

The amount of idle time is also affected by the number of processors used by the algorithm. The larger the number of processors P, the graeter the potential for uneveness in the time to expand P subproblems, and consequently, the larger the potential of idle time and the more significant is the effect of synchronization delays.

A tradeoff between computation-overhead and imbalance-overhead on one hand, and communication-overhead on the other, can be seen in the operation of the SHIFT algorithm, and is caused by the number of shifts s. A complete distribution of subproblems, which eliminates any computation-overhead due to lack of selection information and any imbalance-overhead, is possible using $P-1$ shifts in each iteration. However, the communication and synchronization time needed to perform the $P-1$ shifts can be high and can severely degrade the overall performance of the algorithm. On the other hand, this communication-overhead can be reduced by using only a small number of shifts in each iteration. This, however, is at the expense of an increase in computation-overhead due to lack of selection information resulting from the small number of shifts, and at the expense of higher imbalance-overhead. The computation-overhead and imbalance-overhead also can severly degrade the overall performance of the algorithm. Therefore, there is a tradeoff between the two types of overhead that is caused by the number of shifts. The effect of the number of shifts on the performance of the SHIFT algorithm will be demonstrated experimentally in Chapter 6. This tradeoff has not been identified by Wah and Ma in the design of MANIP since performance has been measured only in terms of the number of iterations needed to obtain a solution with no regard to the time taken by each iteration. Hence, the reported performance did not degrade by increasing the number of shifts from one or two towards $P-1$, and while the advantage of a small number of shifts was recognized, the disadvantage of the large number of shifts was not reflected in the performance measure they used.

5.3 The DL Algorithm

In both the CL and SHIFT algorithms, a strategy has been used that attempts to select and expand a set that contains subproblems with the globally best values of the selection heuristic function h. This strategy reduces the amount of computation-overhead to a minimum, and is implemented by maintaining a global list of active subproblems and a global incumbent in case of the CL algorithm, and by a process of repeated shifts in case of the SHIFT algorithm. However, the selection of this desired set of subproblems is achieved at the expense of high communication-overhead due to communication and contention in the case of the CL algorithm, and due to communication and synchronization in the case of the SHIFT algorithm. The overall performance of the two algorithms can be severly degraded by that overhead.

In contrast, the DL algorithm employs a strategy that cuts down on the amount of communication-overhead by reducing the amount of communication and by eliminating synchronization. This, however, introduces a lack of selection and pruning information which leads to higher computation-overhead. Therefore, the DL algorithm represents a computation-communication tradeoff in which communication-overhead is reduced at the expense of higher computation-overhead and imbalance-overhead.

In the DL algorithm, the P processors of the hypercube array are used to maintain P independent lists of active subproblems, one in each processor. There is no global incumbent; rather, a local incumbent is maintained in each processor. A number of variables and flags (whose use will be described below) are also maintained in each processor. The processors operate asynchronously, each performing its own iterations. All the processors execute the same set of operations in an iteration.

There are two main components to the iteration of a processor. In the first component, the processor removes from its local list of active subproblems the one with the smallest value of h and expands it. This, similar to the DM algorithm, is referred to as the *compute* component of the iteration. In the second component, the processor communicates with one of its neighbors in the hypercube array in order to exchange values of h as will be described below. This is referred to as the *load-balance* component of the iteration. These two components are described in details below.

In the compute component of the iteration, the processor selects and expands the subproblem with the smallest value of h from its local list of active subproblems in a manner similar to the sequential BB algorithm, and similar to a processor in the SHIFT algorithm during the second phase of its iteration. That is, the subproblem is decomposed into smaller subproblems using the branching procedure. Subproblems that can never lead to feasible solutions are deleted using the termination procedure. The lower bounds of subproblems that represent feasible solutions are compared to the lower bound of the local incumbent, and a feasible solution that has a smaller lower bound than that of the local incumbent replaces that incumbent. All subproblems on the local list of active subproblems whose lower bounds are larger than or equal to that of the new local incumbent are deleted. The local incumbent is then used to perform the lower bound elimination test on subproblems, and all remining subproblems are inserted on the local list of active subproblems.

Again similar to the SHIFT algorithm, a feasible solution that updates the local incumbent of a processor is broadcast to all other processors in the hypercube array. The same procedure used by the processors in the SHIFT algorithm to receive the broadcast feasible solution and to update their local incumbents is also used by the processors in the DL algorithm. This procedure has been described in the previous section.

In the load-balance component of its iteration, a processor communicates with one its $n = \log_2 P$ neighbors in the hypercube array. For convenience we will assume these neighbors are numbered 0 to $n-1$, such that neighbor i is the processor's neighbor along dimension i of the hypercube topology (see Chapter 2). A variable NEIGHBOR is maintained by each processor to indicate the neighbor the processor is communicating with during its current iteration. The variable takes values from 0 to $n-1$ and is initialized to 0 in the first iteration. The processor interacts with one of its neighbors by sending it a request for subproblems. The neighbor receives that request and responds by sending back a subset of its subproblems to the requesting processor.

When a processor sends a request for subproblems to its neighbor, it includes with that request the lower bound of the subproblem expanded during the compute component of its current iteration. The processor sends the request to the neighbor whose number is indicated by NEIGHBOR and then increments (modulo n) the value of NEIGHBOR by one. The processor then continues with the remainder of the load-balance component of its iteration, and starts a new iteration without waiting for a response to its request. However, the processor can have only one request pending at any given point of time. That is, a processor does not send a new request except after a response to the previous one has been received. A flag PENDING is maintained by each processor and is used to indicate the status of the previous request made by the processor. This flag is set to TRUE when the processor sends a request and is set to FALSE only when a response to that request has been received. A processor cannot send a new request while the PENDING flag is TRUE. When the processor sends requests and receives responses to these requests to and from all its neighbors in the hypercube array (i.e., a total of n requests and n responses), a *cycle* of requests is said to be complete. That is, the processor interacts with its neighbors in cycles, once a neighbor in each cycle.

The processor can send two types of requests in the load-balance component of its iteration. In the first type, which is referred to as ESSENTIAL, the request is made for subproblems whose lower bounds are less than the lower bound of the subproblem expanded during the compute component of the same iteration. In the second type, which is referred to as **ANY**, the request is made for any subproblems regardless of their lower bounds.

A processor also checks for requests for subproblems from other processors during its load-balance component. The processor may receive requests for subproblems from several processors (limited to one request per processors, however, since a processor may have only one request pending at any time), and the processor responds to all requests during the current load-balance component. The processor, however, handles each request one at a time independently, and without consideration to other requests. The processor responds to a request either by sending subproblems to the requesting processor, or by sending a special message indicating that no subproblems are available. In the first case, the request is said to be granted. In the second case the request is said to be denied.

A processor send requests of the type **ESSENTIAL** in order to balance essential subproblems as will be described below. A processor that has all its **ESSENTIAL** requests denied for a complete cycle, changes the type of its request to **ANY**.

A processor responds to a received request for subproblems after examining the type of the request, the lower bound received with the request, and its local list of active subproblems. The processor responds differently depending on the type of the received request. If the request is of the type **ESSENTIAL**, the processor responds by sending to the requesting processor one half of the subproblems on its local list whose lower bounds are less than the lower bound received with the request. If no such subproblems exist, the processor responds by sending the requesting processor the special message indicating that the request is denied.

If the request is of the type **ANY**, the processor responds by sending to the requesting processor one half of all the subproblems on its local list of active subproblems. In the response to both types of requests, a threshold is used to avoid a thrashing situation on which subproblems are continually exchanged between processors. If the number of subproblems to be sent to the requesting processor is less than the value of the threshold, the subproblems are not sent, and the special message is sent indicating that the request is denied.

The processor also checks for a response to a previous request in the load-balance component of its iteration. If the response indicates that the request has been granted, the processor inserts the received subproblems on its local list of active subproblems. In this case a new request is not generated until the next iteration of the processor. This is performed to allow the processor to expand a new subproblem before issuing a new request, hence, updating the lower bound to be sent with the request. However, if the received response indicates that the request is denied, the processor sends a new request during the same load-balance component to another processor as described above.

The iterations of the processor continue until the termination of the algorithm is detected, at which point the processors forward their incumbents to processor 0 and stop. Processor 0 reports the optimal solution as the the incumbent with the smallest lower bound among the received incumbents and its own. The method used to detect the termination of the algorithm is described later in this section. An outline of the DL algorithm is shown in figure 5.4.

The detection of termination of the DL algorithm is more complex than the detection of termination of the CL and SHIFT algorithms. A necessary and sufficient condition for the DL algorithm to terminate is that all local lists of active subproblems in the P processors be empty *and* no messages from load-balance be in transit among the processors. This condition is difficult to detect due to the asynchronous nature of the communication among the processors coupled with the lack of any global knowledge about the status of the processors at any given point in time.

The detection of termination of distributed algorithms in general is non-trivial and

```
A. Compute Component
   1. Remove the subproblem with the smallest value of the selection
      heuristic function from the local list of active subproblems.
   2. Apply the branching procedure.
   3. Apply the termination procedure.
   4. Check for feasible solutions, and update incumbent if necessary.
      If incumbent is updated, the new incumbent is broadcasted to
      all processors.
   5. Apply the elimination procedure using local incumbent.
B. Load-balance Component
   6. If PENDING is FALSE then send a request including the lower
      bound of the subproblem removed in step 1.
   7. Else if PENDING is TRUE then check for response to previous
      request. Insert received subproblems, if any, on local list.
   8. Check for requests from other processors and respond to each
      one (see text).
C. Repeat steps A and B until the termination of the algorithm is
   detected (see text).
```

Figure 5.4. Outline of the DL algorithm.

has been studied extensively in distributed computing literature. A partial survey can be found in [Rayn88]. The majority of these methods for detecting termination usually require the processors to perform a special detection of termination cycle in which a "probe" message is initiated by a designated processor and is forwarded from one processor to the next (usually using a ring topology) recording the status of the processors. That status varies from one method to the other, and can include time stamps, processor activity, and message counts. Finally, when the probe message is received by the initiating processors, it becomes possible to determine if the distributed algorithm has terminated. These methods are suitable for general distributed algorithms and not only parallel BB algorithms such as the DL algorithm, and may involve elaborate communications that can degrade performance. Consequently, a method is used in the DL algorithm that is specific to the BB algorithm, and is integrated into the load-balance component of the iteration to reduce any overhead due to special probe messages. The method is heuristic in the sense that it does not guarantee that a processor will not terminate before the condition for termination described above becomes true. The time taken by the algorithm to detect its termination, however, is small compared to the total time taken by the algorithm to find the optimal solution to a problem. This is particularly true when the problem size is large. Therefore, it is unlikely that the method used for detecting the termination will affect the overall performance of the algorithm. The problem of detecting termination in the case of the DL algorithm becomes a problem of insuring that the algorithm terminates correctly rather than a problem of efficiency.

The method used by the DL algorithm to detect its termination is motivated by the pattern in which the number of subproblems on a list of active subproblems grows or shrinks during the execution of the BB algorithm. There are three phases in that pattern: the startup phase, the steady state phase and the wind-down phase. These phases are shown in figure 5.5 which depicts the growth of the number of subproblems with the number of iterations. There is only one subproblem on the list of active subproblems initially (the root of the BB tree). The number of subproblems the list then grows rapidly during the startup phase. The number of subproblems then peaks and remains fairly constant during the steady state phase. Finally, due to the discovery of feasible solutions, the number of subproblems declines and eventually reaches zero, at which point the algorithm terminates. The rate by which the number of subproblems declines during the wind-down phase is usually high and only but a few iterations are executed during that phase. In most BB applications, including the 8-puzzle and the traveling salesman problems, the first solution discovered by the algorithm is usually the optimal one, and all subproblems on the list of active subproblems are deleted during the iteration in which the solution is discovered. In other cases, including the integer programming and 0–1 integer programming problems, when the first feasible solution is discovered, the optimal solution is discovered shortly after. Hence, only a relatively few iterations are executed during the wind-down phase. This pattern is also observed by [WaYu82].

Therefore, when the optimal solution is discovered in a processor the number of subproblems in the processors declines rapidly as that solution is broadcast to all the processors in the hypercube array. Consequently, a processor whose local list of active subproblems is empty and whose load-balance component cannot obtain subproblems from the processor's neighbors for two consecutive cycles, assumes that the number of subproblems in the processors is declining rapidly, and that the BB algorithm is in its wind-down phase. Therefore, the processor terminates its iterations sending its local incumbent to processor 0. The processor continues to participate in feasible solution broadcast originating in other processors and responds to a request for subproblems from other processors by sending the special message indicating that no subproblems are available in its local list. When all other $P - 1$ incumbents are received by processor 0, a message is sent to stop all processors, at which point the algorithm terminates.

It is important to note, then, that there is no guarantee that a processor does not terminate before all the lists of active subproblems in the processors are empty. Indeed, it is likely that a processor would terminate before such a condition occurs since the information a processor bases it decision to terminate on is only local to its neighbors. However, since the duration of the wind-down phase is small, and since the neighbors of the processor interact with their own neighbors as well, the effect of this process in terminating is not significant on performance.

It is possible to combine some of the more formal methods for detecting the termination of the algorithm with the above observation of the phases of execution of the BB algorithm to obtain a detection of termination mechanism that is guaranteed to detect the termination of the algorithm only after the condition described above for termination becomes true. For example, a probe message can be sent to the processors in the manner described above after a processor has not received any subproblems for two con-

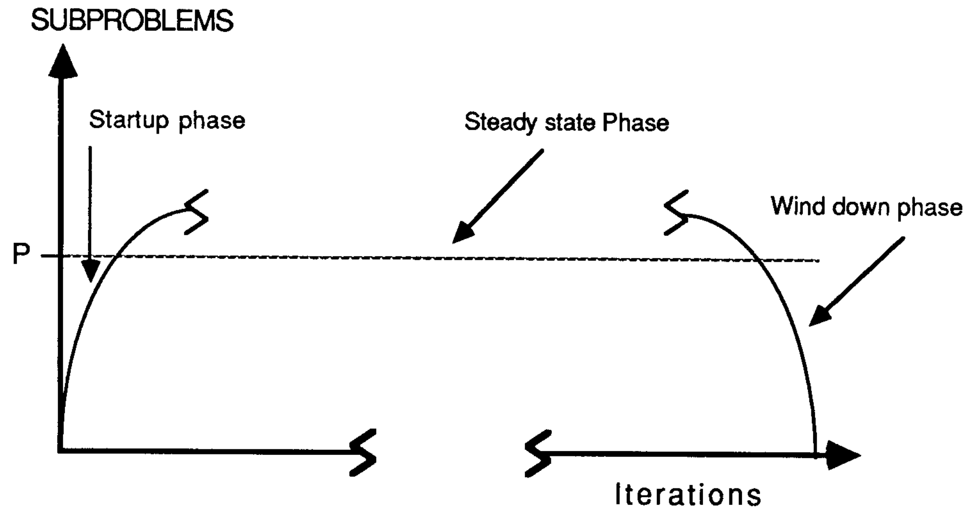

Figure 5.5. The phases of execution of a BB algorithm.

secutive loops. The probe message. This method, however is not implemented by the DL algorithm.

There are a number of features that distinguish the load-balance mechanism used by the DL algorithm from other mechanisms used by parallel BB algorithms that have been proposed in the past. The load-balance used by the DL algorithm is completely asynchronous and does not requires the processors to synchronize in their iterations. This is unlike the proposed algorithms of [Quin86, AnCh86, PaWo88] in which mechanisms that require full synchronization of the processors are employed. This synchronization can result in considerable communication-overhead as has been described for the SHIFT algorithm.

The load-balance mechanism used by the DL algorithm communicates selection information among the processors. This is unlike the algorithms of [AbMu88, MaTM88] in which the load-balance is used only when a processor exhausts its local list of active subproblems and becomes idle. This approach has the disadvantage of permitting a processor that is not idle to expand non-essential subproblems without sharing selection information with other processors, leading to considerable computation-overhead.

In communicating selection information, the load-balance strategy used by the DL algorithm attempts to distribute essential subproblems to the P processors rather than the P subproblems with the globally best values of h. A processor sends the value of the selection heuristic function for the last subproblem it expanded to its neighbor. The neighbor responds by sending back on half of the subproblems that can be called "more essential" than that subproblem since they all have values of h that are less than the lower bound received with the request. This allows the two processors to share the "more essential" subproblems. The strategy uses less communication information that reduces communication-overhead. This is unlike the asynchronous strategy used by [Felt88] in which a random communication strategy is used that does not necessarily distribute essential subproblems.

The use of a hypercube to implement the DL algorithm has the advantage of increasing the number of neighbors of a processor as the total number of processor are increased (see Chapter 2). This offers the potential of a scaling performance, since as the number of processors are increased, a processor performs load balancing with a larger number of processors.

The main factor that leads to communication-overhead in the DL algorithm is the time spent by a processor in performing the load-balance component of its iteration. This time is consumed in sending and receiving requests as well as the removal and insertion of subproblems on the local list. There is no communication-overhead in the DL algorithm due idle time as a result of synchronization, since the processors operate completely asynchronously, as described earlier. Furthermore, the amount of communication effort performed by a processor is independent from the number of subproblems expanded by that processor. In the CL and SHIFT algorithms, the number of communication steps performed by a processor depend on the number of subproblems expanded by that processor. In the CL algorithm, two steps are needed; one to send a subproblem to a slave, and another to receive results from the slave. Similarly, in the SHIFT algorithm, $2s$ communication steps are needed to perform the s shifts before a subproblem can be expanded. In contrast, in the DL algorithm, the number of communication steps depends on the number of processors used by the algorithm. This in effect reduces the number of communication steps per subproblem expanded and, hence, reduces the total amount of communication-overhead in the DL algorithm compared to the first two algorithms.

There two main factors that lead to computation-overhead in the DL algorithm. The first is the lack of a global incumbent and the existence of multiple incumbents that leads to lack of pruning information, that in turn leads to computation-overhead. This effect of this factor has been described in the previous section for the SHIFT algorithm.

The second factor that leads to computation-overhead in the DL algorithm is the lack of selection information due to the asynchronousity of the processors. A processor in the DL algorithm sends a request for subproblems to one of its neighbors in the hypercube array, and then proceeds to expand subproblems from its local list. However, there is no guarantee that the subproblems expanded from that local list are essential. Furthermore, the strategy implemented by the load-balance component of the algorithm communicates selection information between the processor and only one of its neighbors during an iteration. This does not guarantee that the set of subproblems selected by the P processors belongs to the set $\mathcal{S}$ that contains subproblems with the globally best values of h. Therefore, a processor in the DL algorithm expands non-essential subproblems, which leads to computation-overhead.

It is possible for load imbalance to occur in the DL algorithm. This is again due to the asynchronicity of the processors and the action of the load-balance component of the algorithm. A processor that is idle may send a request for subproblems to a number of its neighbors before that request is granted. Hence, the processor remains idle for a number of iterations, which leads to imbalance-overhead.

CHAPTER 6

EXPERIMENTAL RESULTS

The parallel algorithms presented in the previous chapter have been implemented on a commercial hypercube multiprocessor in an effort to determine the extent to which the factors discussed in Chapter 4 affect the performance of the parallel algorithms for realistic applications, and to compare the performance of the third algorithm which employs the new load balancing strategy to the performance of the first two. The comparison of the performance of the three algorithms also shows the extent to which the computation-communication tradeoff can be effectively used to obtain performance gains.

The four applications described earlier in Chapter 3 were implemented using the three parallel algorithms. The applications are: the 8-puzzle problem, the 0–1 integer programming problem, the traveling salesman problem, and the integer programming problem. The applications represent BB problems with diverse characteristics, especially their granularity of computation g.

The algorithms have been implemented on an NCUBE/ten hypercube multiprocessor with 64 processors. Experimental performance measurements have been performed using the standard hardware and software of the system. The NCUBE/ten is briefly described in Appendix A, and a more detailed description of its architecture, software, and programming environment appears in [NCUB85, HMSP86].

The performance results have been obtained by executing the algorithms on the

NCUBE/ten using a number of test problems for each application. The test problems have been selected from among standard benchmark problems for each application or, when such problems were not available, randomly generated. The test problems for the four applications are described in Appendix B. The performance measures that we present for an algorithm applied to a given application are average over all test problems for that application. Furthermore, the size of each test problem has been chosen so that the problem generates the largest number of subproblems possible under the memory limitation constraint of a single processor of the NCUBE/ten. The size of the test problem is not increased as the number of processors is increased in the hypercube.

The experimental results are presented using a number of metrics that reflect different aspects of the performance of the algorithms. The *speedup* of the parallel algorithm, denoted by $S(P)$, is used to measure the overall gain in performance obtained by using P processors relative to a single processor. The speedup is defined as the ratio of the execution time of the algorithm using a single processor, $T(1)$, to the execution time of the algorithm using P processors, $T(P)$. That is,

$$S(P) = \frac{T(1)}{T(P)}.$$

The speedup is said to be *linear* when the gain in performance using P processors is equal to P (i.e. when the speedup is equal to P). The speedup is normally *sublinear* (i.e., less than P) due to the various types of overhead incurred by the algorithm. However, in the case of a parallel BB algorithm, it is possible for the speedup to be *superlinear* (i.e., greater then P) due to an *acceleration anomaly* that occurs in the execution of the parallel BB algorithm [LiWa86].

The *computation ratio* of the parallel algorithm, denoted by $C_r(P)$, is used to indicate the extent of the computation-overhead incurred by the parallel algorithm at P processors. It is defined as the ratio of the number of subproblems expanded by the sequential algorithm to the number of subproblems expanded by the parallel algorithm using

P processors. That is,

$$C_r(P) = \frac{S}{\sum_{i=1}^{P} S_i}, \tag{6.1}$$

where S denotes the number of subproblems expanded by the sequential algorithm, and S_i denotes the number of subproblems expanded by processor i. The value of C_r is equal to 1 when no computation-overhead is incurred by the algorithm, and declines to 0 as more computation-overhead is incurred. The value of C_r normally cannot exceed 1 since the number of subproblems expanded by the sequential algorithm is less than or equal to the number of subproblems expanded by the parallel algorithm. However, it is possible for C_r to exceed 1 when an acceleration anomaly occurs. In this case, the number of subproblems expanded by the parallel algorithm is less than the number of subproblems expanded by the sequential one, and hence C_r becomes greater than one.

The *communication ratio* of the parallel algorithm, denoted by $CM_r(P)$, is used to indicate the amount of communication effort spent by a processor in performing load balancing and communication of selection information. It is defined as the ratio of the time spent by a processor in communication to the total execution time of that processor, averaged over all processors. That is,

$$CM_r(P) = \frac{1}{P} \sum_{i=1}^{P} \frac{CM_i}{T_i}, \tag{6.2}$$

where CM_i and T_i are respectively the communication time and total execution time spent by processor i.

Finally, the *load imbalance factor*, denoted by $\ell(P)$, is used to indicate the extent by which the workload (measured by the number of subproblems expanded) is imbalanced across the P processors. It is defined as

$$\ell(P) = \frac{\sigma_s}{\mu_s} \times 100\%, \tag{6.3}$$

where μ_s is the mean of the number of subproblems expanded by the P processors, i.e.,

$$\mu_s = \frac{1}{P} \sum_{i=1}^{P} S_i, \tag{6.4}$$

and σ_s is the standard deviation of the number of subproblems expanded from their mean, i.e.,

$$\sigma_s = \sqrt{\frac{1}{P-1} \sum_{i=1}^{P} (S_i - \mu_s)^2}. \tag{6.5}$$

The load imbalance factor has the value of 0 when the workload is perfectly balanced and all processors expand exactly the same number of subproblems, and increases as the load becomes more imbalanced.

It is important to note that the speedup is the only metric that reflects the overall performance of the parallel algorithm. The other metrics reflect aspects, but are not indicative, of that performance. For instance, the load imbalance factor can be 0, indicating a perfectly balanced workload, but the overall performance of the algorithm can be poor due to considerable communication-overhead. Similarly, the computation ratio can be 1, indicating that the parallel algorithm does not incur any computation-overhead, yet the overall performance of the algorithm can be poor due to severe load imbalance.

The granularity of computation g affects the performance of the parallel algorithms as indicated by the model presented in section 4.3.5; the effect of communication-overhead on performance is influenced by g. The applications used in the experiments exhibit a wide range of values of g, and hence, the performance of the three parallel algorithms for the four applications can be used to illustrate the effect of the granularity on performance, and on the computation-communication tradeoff. The average time taken to expand a subproblem for each of the four applications on a single NCUBE/ten processor has been experimentally measured and is shown in Table 6.1. This time can be used as an indication of the granularity of each application assuming that the time taken to perform a single communication step is the same for all applications (recall the definition of granularity

Application	Time (milliseconds)
8-puzzle	2.48
0–1 integer programming	25.89
Traveling salesman	289.60
Integer programming	2627.00

Table 6.1. The average time to expand a subproblem in the four applications.

from section 4.3.5; it is the ratio of the average time taken to expand a subproblem to the average time taken to perform a unit communication step, such as send or receive a subproblem). This is only an approximation, however, since the time taken to perform the single communication step depends on the size of a subproblem (i.e., the number of words used to represent the subproblem), which affects the time to communicate the subproblem from one processor to the other, and is not the same for the four applications. However, the approximation is acceptable since the time to perform a communication step between two processors in the NCUBE/ten is dominated by the initial latency of establishing the communication (see [Buzz88]), and since the size of a subproblem in all four applications is relatively small. Consequently, the application with the finest (smallest) granularity is the 8-puzzle problem, and the application with the coarsest (largest) granularity is the integer programming problem. The granularity, hence, increases from fine (small) to coarse (large) as shown in the table.

The remainder of this chapter is devoted to the presentation and discussion of the performance of the three parallel algorithms presented in Chapter 5. The performance of the Logical Model (LM) algorithm, obtained by means of simulation, is first presented in section 6.1 to indicate the effect of lack of parallelism on performance. The performance of the CL algorithm is then presented in section 6.2. The performance results for the SHIFT and DL algorithms are given in sections 6.3 and 6.4 respectively. The performance of the three algorithms is then compared in section 6.5 to show the effect of the computation-communication tradeoff on performance. Finally, summary and highlights of the results are given in section 6.6.

6.1 The LM Algorithm

The operation of the LM algorithm is described in section 4.3. Its performance is affected only by lack of parallelism, and is not affected by lack of selection and lack of pruning information, by load imbalance, nor by any overhead due to communication. Therefore, the performance of the LM algorithm can be used to reflect the extent by which lack of parallelism exists in the test problems, and consequently, the extent by which it can affect the performance of the three parallel algorithms to be discussed. Therefore, as a preliminary to our experimental work, the performance of the LM algorithm was measured using simulation. The results are presented in this section.

The speedup of the LM algorithm is shown in figure 6.1 for the four applications. The speedup is close to linear for all values of P in the cases of the 8-puzzle problem and the 0–1 integer programming problem. The speedup of the algorithm is close to linear up to 32 processors in the cases of the traveling salesman problem and the integer programming problem, but drops slightly at 64 processors. This indicates that sufficient parallelism exists in the first two applications for all values of P, while sufficient parallelism exist in the latter two only up to 32 processors. Lack of parallelism exits at 64 processors for the traveling salesman problem and the integer programming problem.

Therefore, the performance of any of the three parallel algorithms to be discussed is not affected by lack of parallelism for the 8-puzzle problem and the 0–1 integer programming problem. Consequently, the only two factors that can lead to computation-overhead become lack of selection and lack of pruning information. The computation ratio for that algorithm can be used to reflect the effect of these two factors alone on performance.

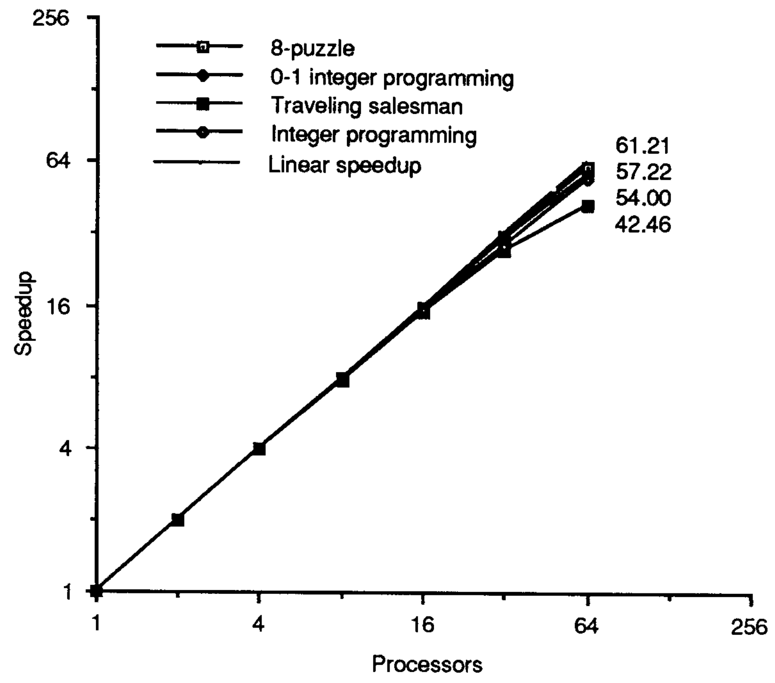

Figure 6.1. The speedup of the logical model algorithm.

The effect of lack of parallelism on the performance of a parallel BB in the case of the traveling salesman problem and the integer programming problem is more apparent than in the case of the other two problems, particularly at 64 processors. Hence, the computation ratio of the three parallel algorithms reflects overhead due to lack of parallelism in addition to lack of selection and lack of pruning information. This lack of parallelism must be taken into consideration when examining the performance of the three parallel algorithm for these two applications.

It is important to note that the results shown above for the traveling salesman problem and the integer programming problems reflect lack of parallelism in the test problems used and for the number of processors indicated, and do not reflect lack of parallelism in the these two problems in general. This lack of parallelism exists only because of the memory constraint of an NCUBE/ten processor which does not permit "sufficiently" large test problems to be solves for these two applications.

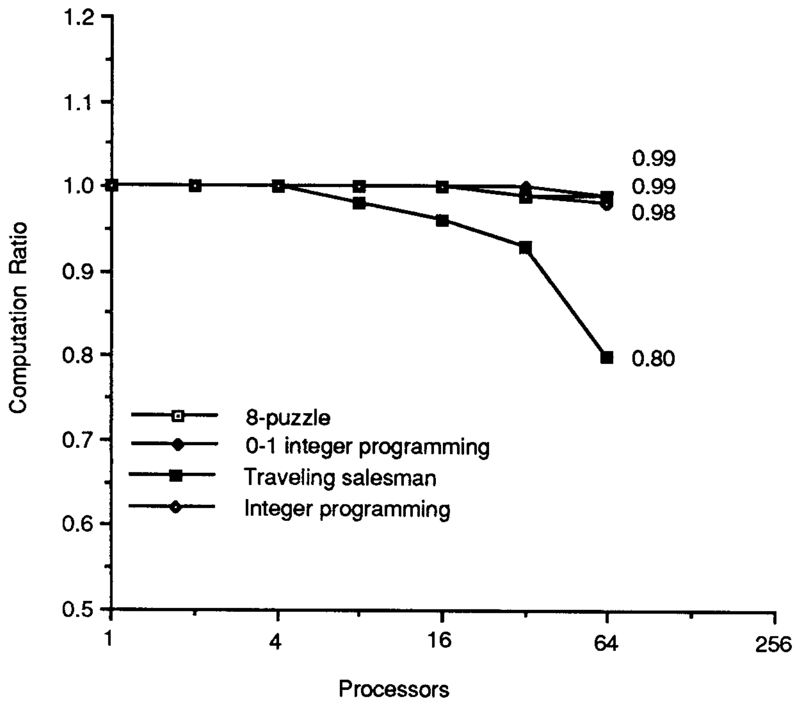

Figure 6.2. Computation ratio of the CL algorithm.

6.2 The CL Algorithm

The performance of the CL algorithm for the four applications is presented and is discussed in this section. The computation ratio of the algorithm is shown in figure 6.2. The load balance factor and the communication ratio of the algorithm are shown in figures 6.3 and 6.4 respectively. The speedup of the algorithm is depicted in figure 6.5. The performance of the algorithm is shown in the figures for values of P greater than 1 since at least two processors must be used to execute the algorithm; processor 0 being used as the master. The figures indicate the overall performance of the algorithm, the factors that degrade it, and how it is affected by the granularity of computation.

The computation ratio of the algorithm is close to 1 for the 8-puzzle problem and the 0–1 integer programming problem for all values of P. This indicates that no computation-overhead is incurred by the algorithm for these two applications. The same is observed for

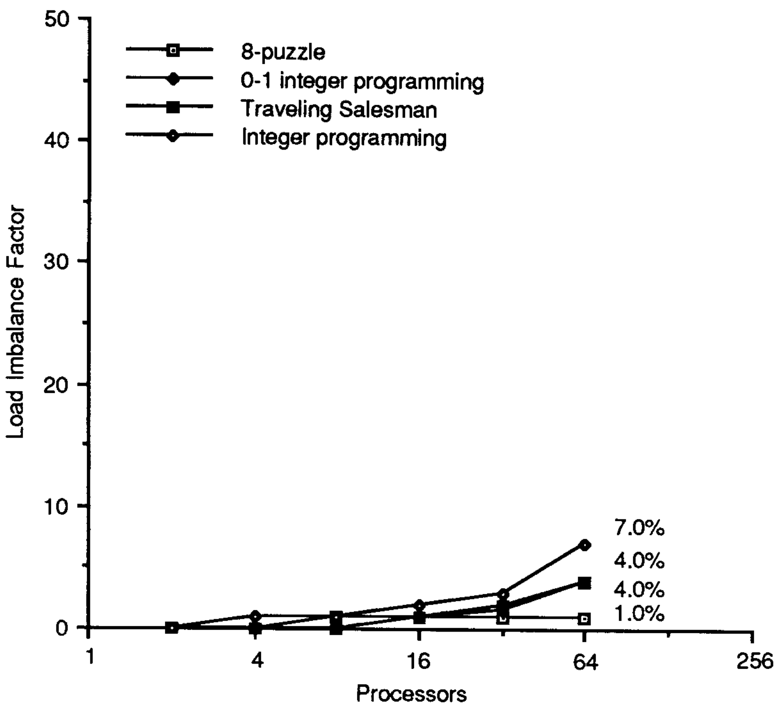

Figure 6.3. Load imbalance factor of the CL algorithm.

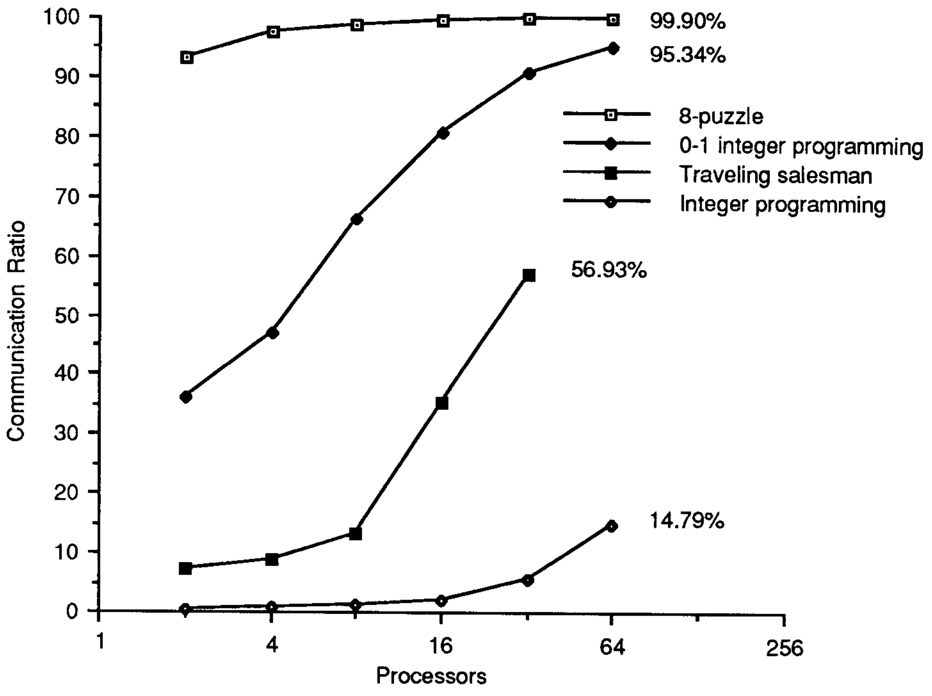

Figure 6.4. Communication ratio of the CL algorithm.

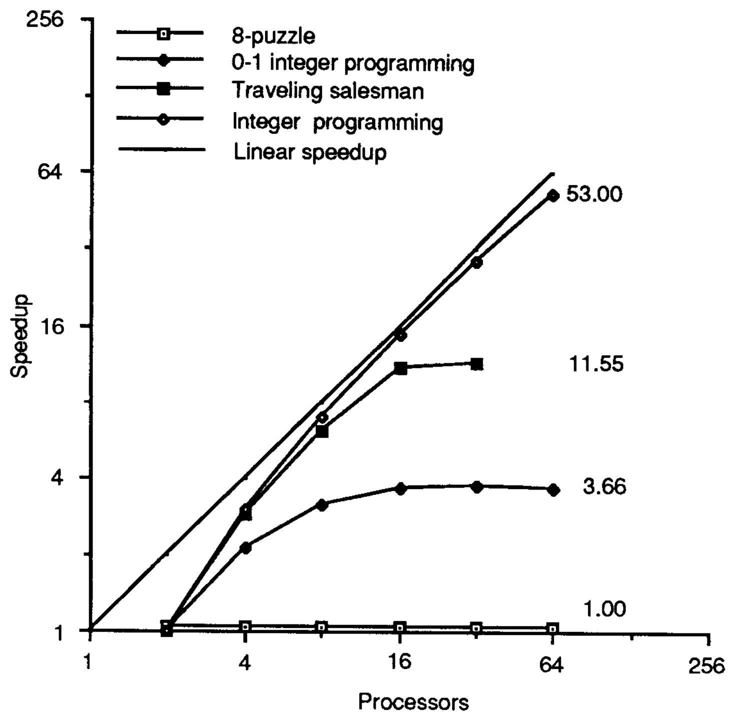

Figure 6.5. Speedup of the CL algorithm.

the traveling salesman problem and the integer programming problem up to 32 processors. The computation ratio of the algorithm is close to that of the LM algorithm for these two applications at 64 processors. This indicates that no overhead is incurred by the algorithm beyond that which is caused by lack of parallelism in the test problems. Therefore, it can be concluded that for all four applications, lack of selection and lack of pruning information are non-significant and result in no significant computation-overhead, and that the effect of this type of overhead on performance is negligible. This is a result of the use of a global list of active subproblems and a global incumbent by the master, which allow the expansion of a set of subproblems close to that expanded by the LM algorithm.

The load imbalance factor of the CL algorithm indicates that the algorithm maintains a well balanced workload across all the processors for all values of P. In other words, all processors expand almost the same number of subproblems. Therefore, the performance of the CL algorithm is not significantly affected by any imbalance-overhead. This well balanced load is a result of the self-scheduled strategy used by the master for distributing subproblems to the slaves. The load imbalance factor indicates that a slight load imbalance exists at 64 processors, which is explained later in this section.

The factor that has a significant effect the performance of the CL algorithm is communication-overhead. The average amount of communication-overhead incurred by a processor is indicated by the communication ratio, which is shown in figure 6.4 for the four applications. The amount of this overhead is the highest in the cases of the 8-puzzle problem and the 0–1 integer programming problem, indicating that performance is significantly affected by this factor for these two applications. The amount of this overhead is less for the traveling salesman and integer programming problems, but is still a major source of overhead. The model presented in section 5.2 for the performance of the CL algorithm indicates that the effect of communication-overhead is influenced by the granularity of computation. This effect can be seen from the communication ratio and from the overall performance of the algorithm, which is measured by its speedup.

The speedup of the CL algorithm for the 8-puzzle problem remains negligible as the number of processors is increased. This poor performance, as will be shown below, is attributed to the very fine granularity of the 8-puzzle problem. The amount of time taken to expand a subproblem in that application is small compared to the amount of time incurred in overhead due to communication and contention for the master. This can be seen from figure 6.4. The communication ratio shown in the figure is large making any gains in performance impossible.

The speedup of the algorithm for the 0–1 integer programming problems shows an improvement over that of the 8-puzzle problem. The speedup increases with the number of processors but shows no gain in performance after 16 processors. The improvement in performance is attributed, for the most part, to the coarser granularity of of the 0–1 integer programming problem, which is almost one order of magnitude larger than that of the 8-puzzle problem (see Table 6.1). However, when the number of processors becomes large, the amount of communication-overhead increases and becomes dominant, limiting further gains in performance.

The speedup of the traveling salesman problem shows further improvements as the granularity of that problem increases by almost another order of magnitude. The speedup is not only higher, due to the coarser granularity, but also keeps increasing up to 32 processors. The speedup at 64 processors was not possible to obtain due to lack of memory on processor 0, the master. The number of subproblems on the global list of active subproblems increased due to computation-overhead such that the memory on processor 0 became insufficient to maintain it.

The speedup of the algorithm for the integer programming problem is almost linear with the number of processors used. This is mainly attributed to the very coarse granularity of this application which makes the amount communication-overhead incurred by the algorithm almost negligible compared to the time taken to expand a subproblem.

The small load imbalance incurred by the CL algorithm at 64 processors can be attributed to the unequal amounts of communication time between the master (processor 0) and different slaves. The distance, measured by the number of communication links, between processor 0 and each of the remaining $P-1$ processors in the hypercube array is not the same for all processors, and consequently some slaves are at a further distance of communication from the master than others. Therefore, a longer amount of time is taken to send and receive a subproblem to and from a slave that is further away from the master. Consequently, a slave that is closer to the master receives subproblems more frequently during the execution of the algorithm, and therefore expands more subproblems. This leads to the slight load imbalance reflected by the load imbalance factor.

The effect of this unequal distance of communication between the master and the slaves on the load balance of the CL algorithm for a typical problem can be more clearly

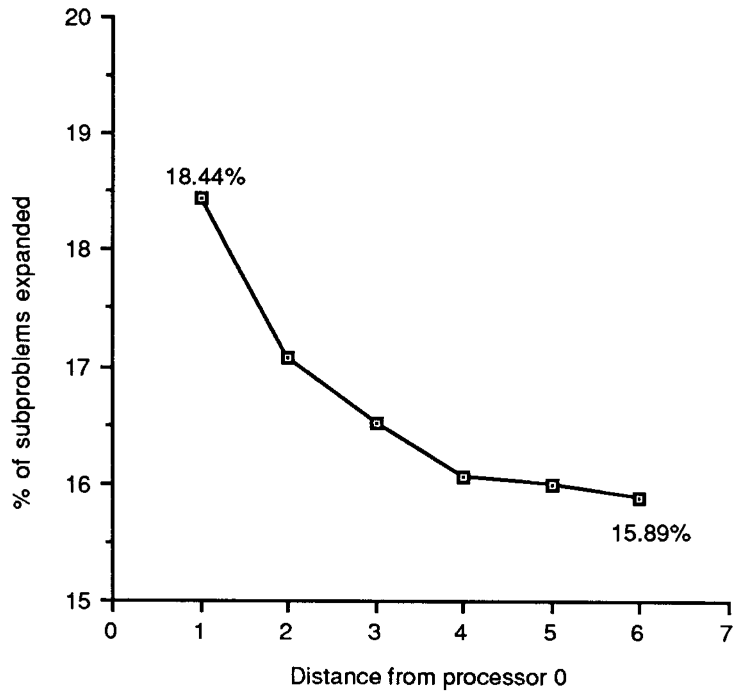

Figure 6.6. The effect of communication distance on load balance.

seen in figure 6.6. The ratio of the average number of subproblems per processor expanded by the processors at distance *i* from the master to the total number of subproblems expanded is shown for the possible values of *i* in a 64 processor (6-dimensional) hypercube. The number of subproblems steadily declines as the distance between the master and the slaves increases, indicating the load imbalance. The difference in the number of subproblems expanded by processors at distance 1 and the processor at distance 6 is small, however, and the effect of this load imbalance on performance is not significant.

In summary, the performance of the CL algorithm is not significantly affected by computation-overhead, as indicated by the computation ratio. The performance of the algorithm also is not significantly affected by imbalance-overhead, as indicated by the load imbalance factor. The performance of the CL algorithm is affected, however, by the communication-overhead that results from communication to and contention for the master. The effect of this overhead on performance is significant when the granularity of computation is small, and is negligible when the granularity is large.

6.3 The SHIFT Algorithm

The performance of the SHIFT algorithm for the four applications on the NCUBE/ten is presented in this section. The computation-communication tradeoff caused by the number of shifts used by the algorithm is first demonstrated. The overall performance of the algorithm is then discussed to indicate the factors that cause it to degrade, and the effect of the granularity of computation on it.

The effect of the number of shifts s used by a processor in the SHIFT algorithm in each of its iterations can be seen from the speedup, the computation ratio and the load balance factor, and the communication ratio of the algorithm, which are all shown for possible values of s for the 0–1 integer programming problem using 32 processors in figures 6.7, 6.8, and 6.9 respectively. The effect of the number of shifts on the remainder of the applications is similar, but is not shown here.

The computation ratio and the load imbalance factor both indicate that the amount of computation-overhead and the imbalance-overhead incurred by the algorithm are large when the number of shifts is small (e.g. 1). This is due to the unability of the algorithm to achieve a close to complete distribution of subproblems with this small number of shifts, as described earlier in section 5.3. The amount of communication-overhead is small as indicated by the communication ratio. The overall performance of the algorithm is poor, as indicated by its small speedup, due to the large amount of computation-overhead and imbalance-overhead.

The computation ratio and the load imbalance factor of the algorithm improve when the number of shifts per iteration is increased and a closer to complete distribution of subproblems is achieved. The overall performance improves as indicated by the increase in the speedup of the algorithm shown in figure 6.7. The amount of communication-

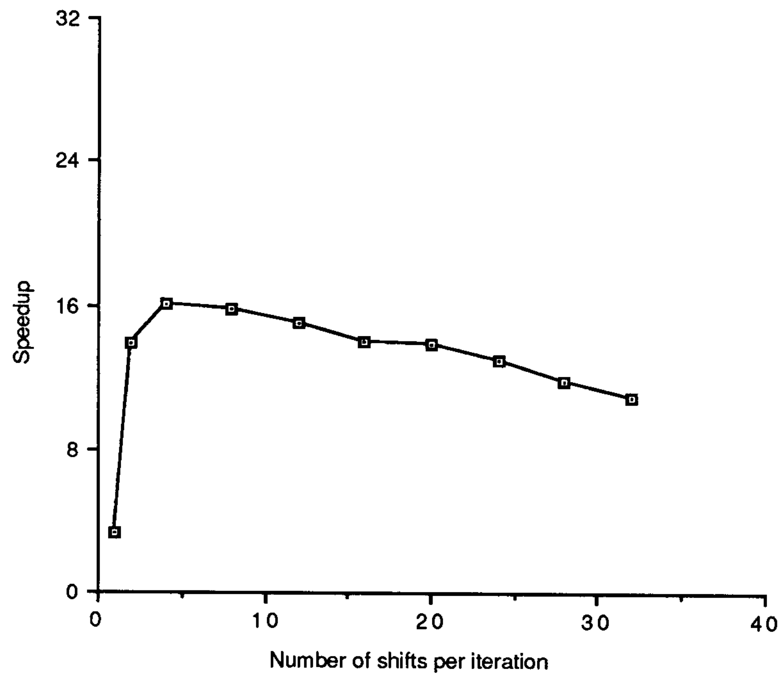

Figure 6.7. The speedup of the SHIFT algorithm versus s.

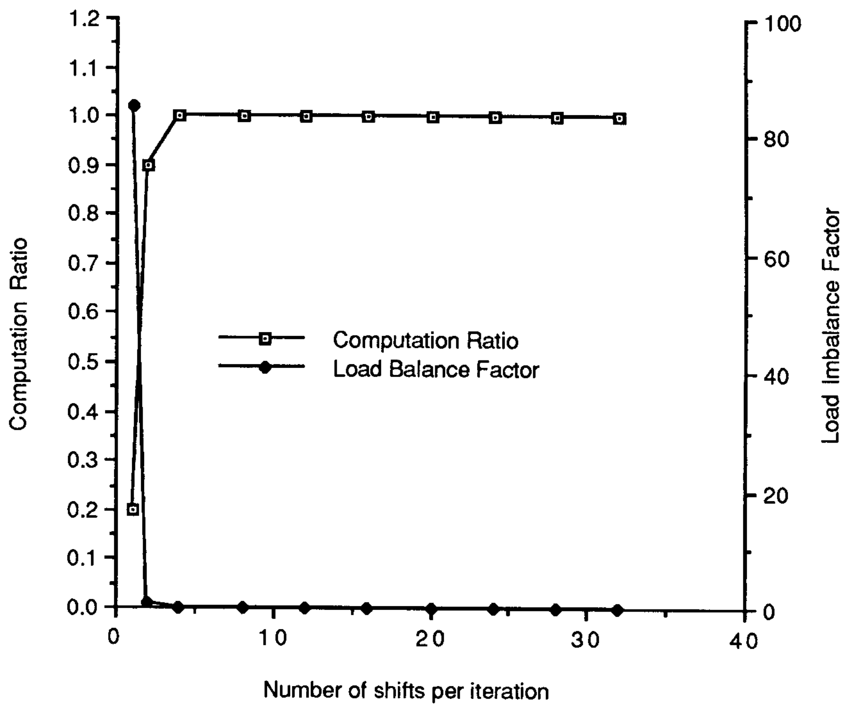

Figure 6.8. C_r and ℓ of the SHIFT algorithm versus s.

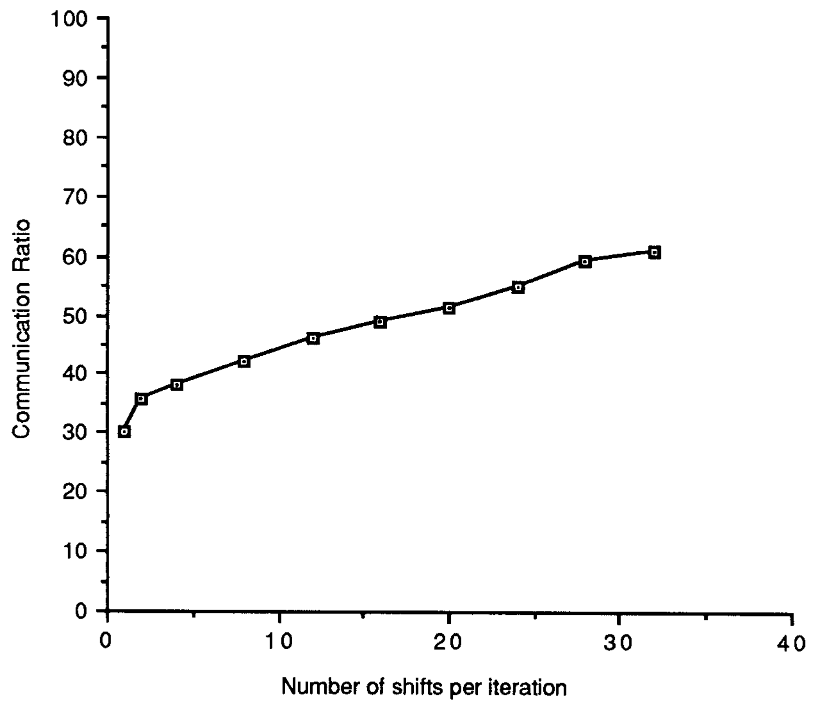

Figure 6.9. The communication ratio of the SHIFT algorithm versus s.

overhead increases, but not significantly to cause the performance to degrade. However, when the number of shifts is increased beyond 4, the load balance and the computation ratio further improve, but the speedup of the algorithm decreases indicating that the overall performance of the algorithm degrades. This is due to the communication-overhead incurred by the algorithm which keeps increasing as the number of shifts is increased.

Therefore, while increasing the number of shifts improves the performance of the algorithm initially by reducing computation-overhead and improving its load balance, communication-overhead becomes large as the number of shifts is further increased and causes the performance to degrade again. Consequently, the tradeoff caused by the number of shifts, that has been described in section 5.3, between computation-overhead and load imbalance on one hand, and communication-overhead on the other can be seen in the overall performance of the SHIFT algorithm. A number of shifts exists that causes the overall performance of the algorithm to become optimal. This number has been obtained

Application Problem	# of shifts
8-puzzle	3
0–1 integer programming	4
Traveling salesman	4
integer programming	3

Table 6.2. Observed optimal number of shifts.

experimentally, and is shown in table 6.2 for the four applications. This number of shifts is observed to be unaffected by the number of processors used when the number of processors is larger than 4.

The number of shifts shown is larger in all cases than the single shift suggested by [WaMa84] for the operation of MANIP. This discrepancy can be caused by two factors. The first is the different application used in the simulation of MANIP (the vertex covering problem). This problem can have special characteristics that causes the number of shifts needed to obtain a nearly complete distribution to be smaller than those observed above for the four applications. The second is the fact the simulation used a combined best-first/depth-first search strategy to reduce the amount of memory needed for the simulation. The search strategy is best-first until memory is exhausted, at which time it is changed to depth-first. This change from a search strategy that is sensitive to the values of h to a one which is less sensitive can also cause the number of shifts needed to achieve a nearly complete distribution of subproblems to become smaller.

Furthermore, the effect of the number of shifts on the performance of the SHIFT algorithm is consistent with that on the performance of MANIP only in that a small number of shifts is necessary to achieve good performance [WaMa84]. The performance of MANIP is obtained by simulation and does not take into consideration the overhead of communication incurred in shifting subproblems. The overall performance of MANIP

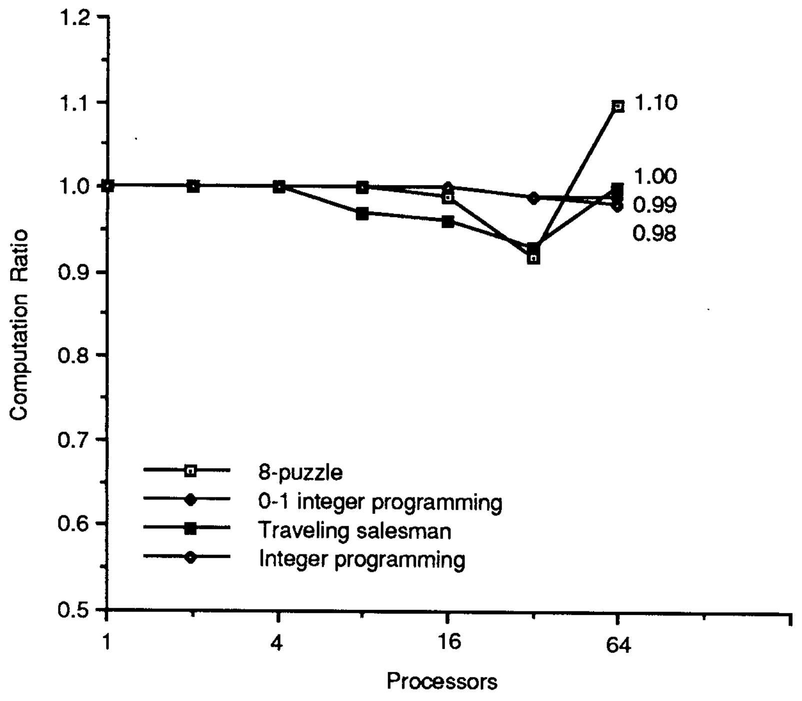

Figure 6.10. The computation ratio of the SHIFT algorithm.

is presented only in terms of the total number of iterations, and not in terms of the total execution time. This does not reflect the increase in the execution time of each iteration that can occur when the number of shifts is increased, and therefore, does not account for the existence of an optimal number of shifts for the operation of MANIP.

The experimental performance results presented for the SHIFT algorithm in the remainder of this section are measured using the above observed optimal number of shifts per iteration for each application. The number of shifts used is $P - 1$ when the number of processors is less than 4. Therefore, these results represent the best performance that can be obtained by the SHIFT algorithm for each application.

The computation ratio of the algorithm is shown in figure 6.10. The load imbalance factor and the communication ratio are shown in figures 6.11 and 6.12 respectively. The speedup of the algorithm indicating its overall performance is shown in figure 6.13. The figures reflect the various aspects of the performance of the algorithm for the values of P

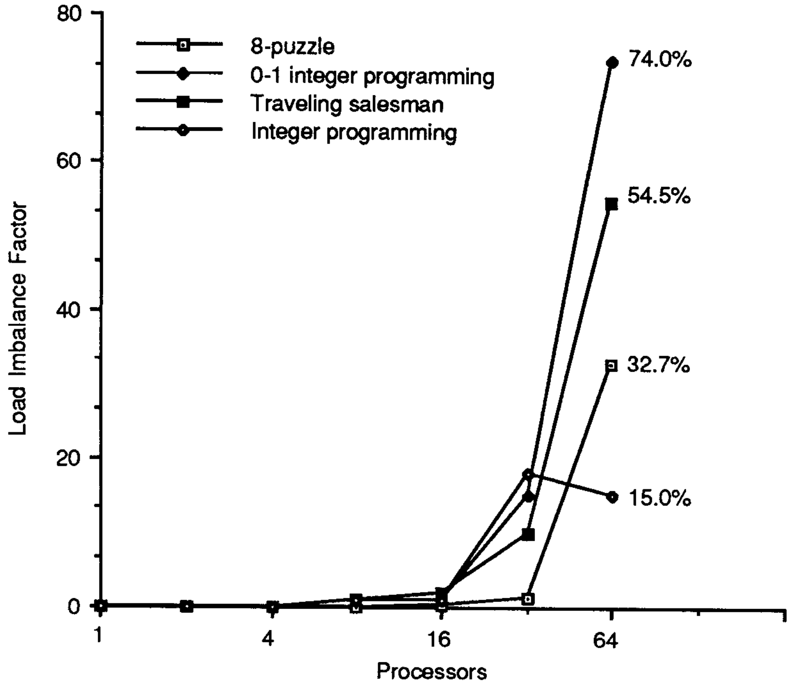

Figure 6.11. The load imbalance factor of the SHIFT algorithm.

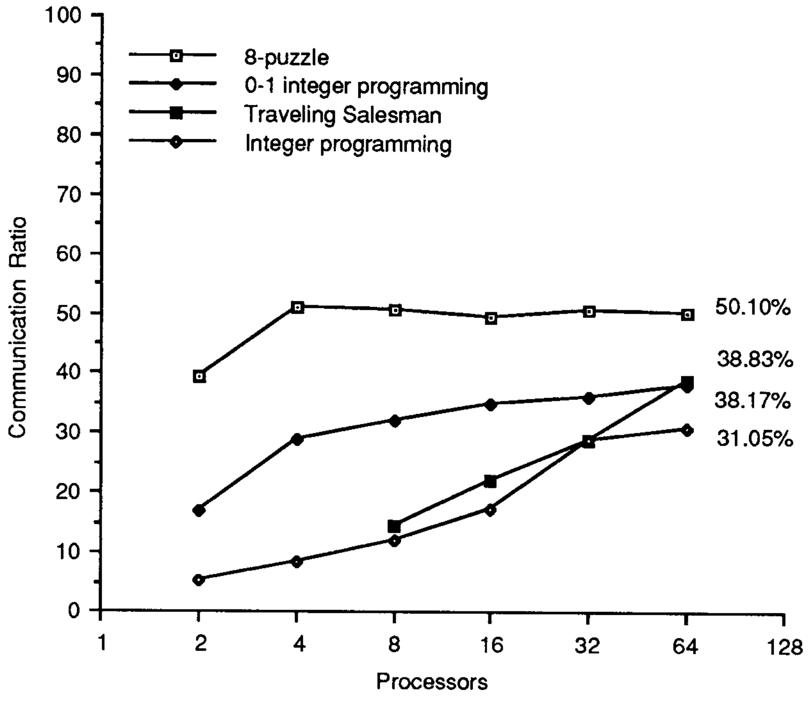

Figure 6.12. The communication ratio of the SHIFT algorithm.

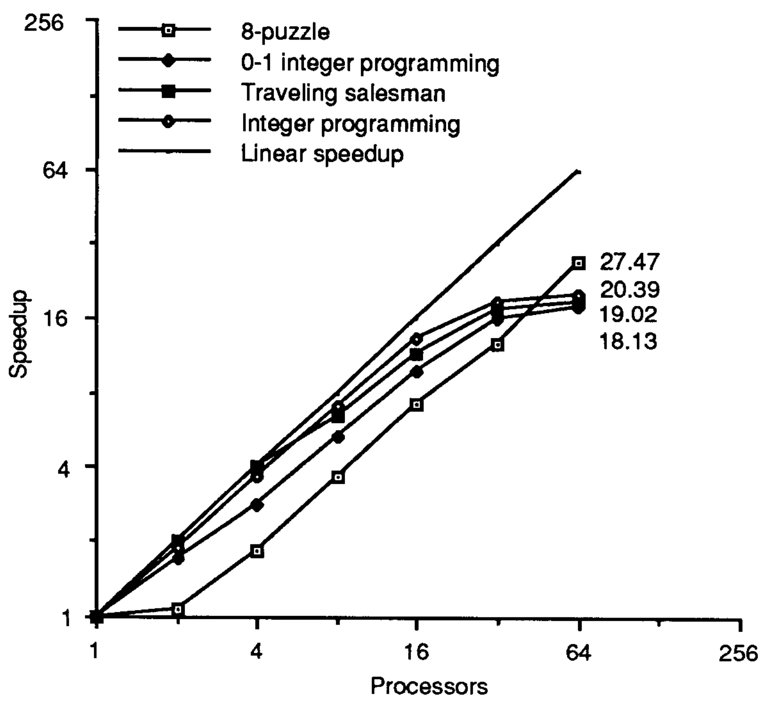

Figure 6.13. The speedup of the SHIFT algorithm.

used.

The computation ratio indicates that no considerable computation-overhead is incurred by the algorithm when the number of processors is small. However, when the number of processors is increased, the amount of computation-overhead increases. This is a result of the number of shifts used by the algorithm, which at a small number of processors are sufficient to achieve almost a complete distribution of subproblems. However, as the number of processors increases, the small number of shifts becomes insufficient to achieve a close to complete distribution. This number of shifts, however, is the one that achieves the best overall performance as indicated above.

The load imbalance factor of the algorithm indicates a similar behavior to that of the computation ratio. A well balanced load is maintained by the algorithm when the number of processors is small, but more load imbalance is introduced as the number of processors is increased. This is also due to the small number of shifts which has a similar effect on the load imbalance as it has on the computation ratio. The load imbalance becomes significant when 64 processors is used.

The communication ratio indicates that the amount of communication-overhead is small when the number of processors is small, and increases as the number of processors is increased. This is expected since the time spent by a processor in communication and synchronization increases as the number of processors is increased (as described in section 5.3). This is particularly clear in the case of the integer programming problem, which shows a large amount of communication-overhead despite its large granularity. This is attributed to the large variance in the amounts of time taken to expand subproblems in this application, which causes processors to spend considerable amounts of time idle due to synchronization.

It is important to note again, that although the SHIFT algorithm incurs the amounts of overhead shown above due to the small number of shifts used in each iteration, this number of shifts causes the overall performance of the algorithm to be its best due to the balance between the amounts of computation-overhead and communication-overhead incurred.

The speedup of the algorithm for the four applications shows a sublinear performance that increases as the number of processors is increased. The gain in performance decreases, however, as the number of processors in increased. This can be attributed to the overhead that increases as the number of processors is increased as described earlier.

The effect of the applications granularity on performance can be seen from the speedup of the algorithm. With one exception, the 8-puzzle problem which has the finest granularity exhibits the poorest overall performance. Again, with one exception, the speedup of the algorithm increases as the granularity of the application increases. This is consistent with the model of section 4.3.5, which indicates that the extent by which communication-overhead affects performance is limited by the granularity. The coarser the granularity of computation is, the smaller becomes the effect of communication-overhead on overall performance. The effect of the granularity becomes less apparent as the number of processors increase. This is because of the increase in the computation-overhead, which when the granularity is coarser becomes more significant in affecting the performance.

The exception to the general effect of granularity on performance can be observed on the speedup curve for the 8-puzzle problem, where the performance of the algorithm at 64 processors shows an improvement which is not consistent with the performance of the algorithm for smaller values of P, nor with the performance of the algorithm for the other three applications. The number of 8-puzzle subproblems examined by the parallel algorithm at 64 processors is actually less than the number of subproblems expanded by the sequential one. This reduction in the amount of computation performed by the parallel algorithm improves its performance. This phenomena is known as an *acceleration anomaly*. The effect of this anomaly can also be seen from the computation ratio of the algorithm, which indicates a value of C_r that is larger then 1 at 64 processors for the 8-puzzle problem, implying that the number of subproblems expanded by the parallel algorithm is less than the number of subproblems expanded by the sequential one. The phenomena of acceleration anomalies in general and an explanation for why it occurs in the case of the SHIFT algorithm for the 8-puzzle problem in particular are described in Chapter 7.

A similar behavior can be seen in the computation-ratio of the algorithm for the traveling salesman problem. The value of the computation-ratio of the algorithm for that application at 64 processors is equal to 1.0, and is not consistent with the its values for other values of P. Furthermore, although the test problems for that application indicate lack of parallelism at 64 processors, the value of C_r for that application is 1.0, which may suggest the non-existence of any computation-overhead. This is attributed, however, to an acceleration anomaly that causes the number of subproblems that are expanded by the SHIFT algorithm to be roughly equal to the number of subproblems expanded by the sequential one, and makes $C_r = 1.0$. The effect of this anomaly on the overall performance of the SHIFT algorithm is not as evident as its effect for the 8-puzzle problem. This is attributed to the smaller *extent* of the anomaly in the traveling salesman problem, as will be explained in Chapter 7.

6.4 The DL Algorithm

The performance of the DL algorithm is presented in this section. The overall performance, the factors that affect it, and the effect of the granularity of computation on that performance are discussed. The computation ratio, load balance factor, communication ratio and speedup of the algorithm are shown in figures 6.14, 6.15, 6.16, and 6.17 respectively.

The computation ratio of the algorithm indicates that the computation-overhead incurred by the algorithm increases as the number of processors increases. The load imbalance of the algorithm also increases as the number of processors increases as indicated by the load imbalance factor.

The amount of communication-overhead incurred by the algorithm also increases with the number of processors as evident by the communication ratio. The amount of this communication overhead is small compared with the amounts incurred by the CL and SHIFT algorithms as will be illustrated in the following section.

The speedup of the DL algorithm indicates that the overall performance of the algorithm is nearly linear up to 16 processors, after which the speedup degrades. This is attributed to load imbalance, communication-overhead, and computation overhead, which all affect the overall performance.

The performance of the traveling salesman problem at 64 processors shows more degradation than the other three applications. This is due to the lack of parallelism,

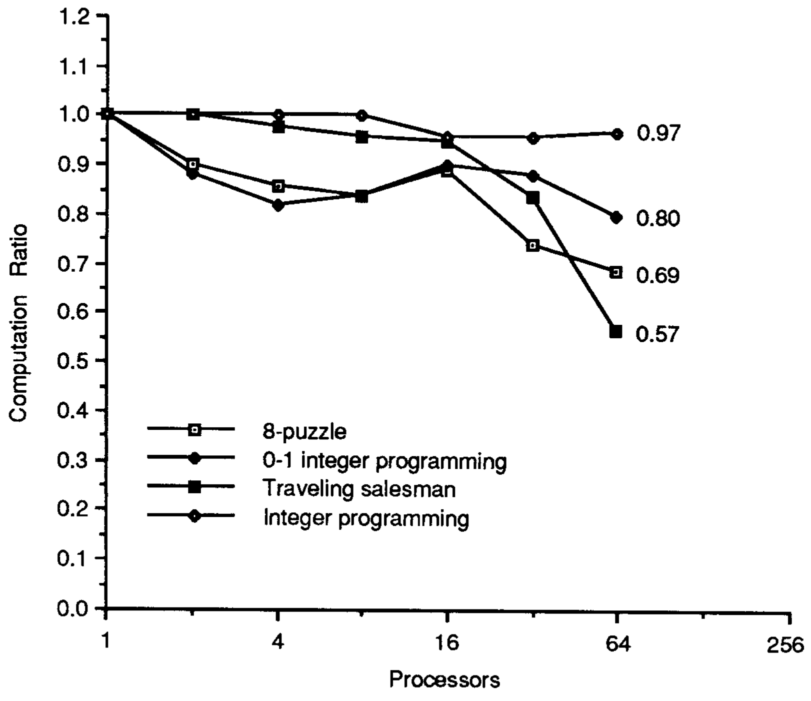

Figure 6.14. The computation ratio of the DL algorithm.

noted in section 6.1, which causes the performance of the DL algorithm to degrade as in the case of the SHIFT algorithm.

The processors in the DL algorithm operate in repeated iterations, as has been described in section 5.4. A processors responds to requests for subproblems during the load-balance component of its iteration. This is performed on the NCUBE/ten using the message passing run-time system VERTEX (described in Appendix A). The sender processor cannot send another request until a response to the one currently pending is received. Since the receiving processor operates asynchronously with the sender, it is possible for the receiving processor to be in the middle of its compute component of its iteration, and therefore not be able to respond the request as soon as it is received. This is also mandated by VERTEX which does not allow the arrival of a message to interrupt an application program. The program must check for the arrival of messages at intervals in order to determine if a message exits. Therefore a processor in the middle of its compute component

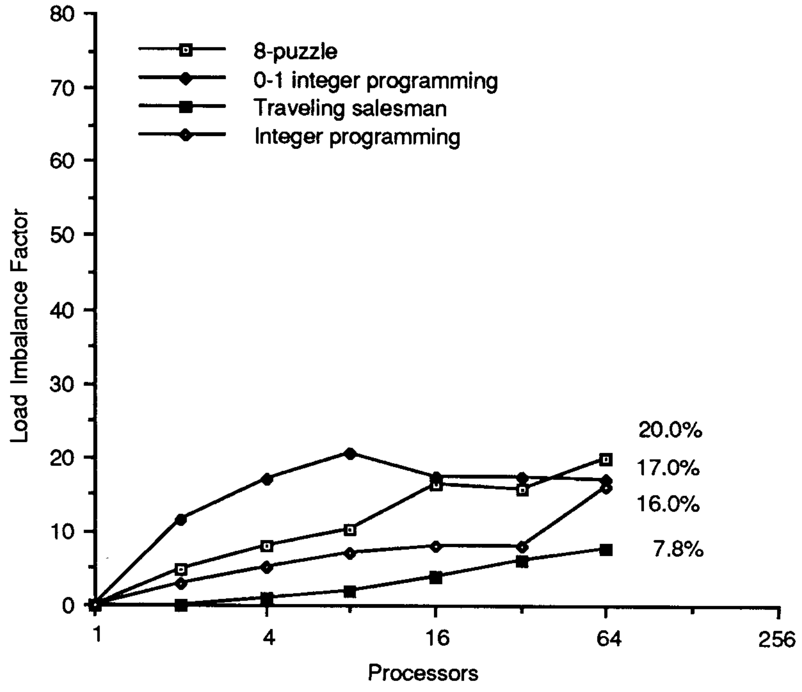

Figure 6.15. The load imbalance factor of the DL algorithm.

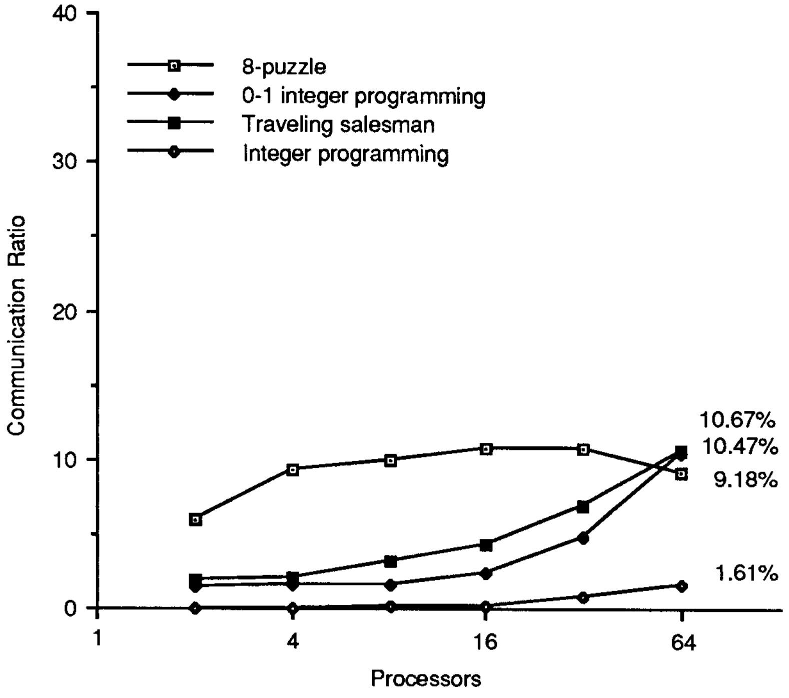

Figure 6.16. The communication ratio of the DL algorithm.

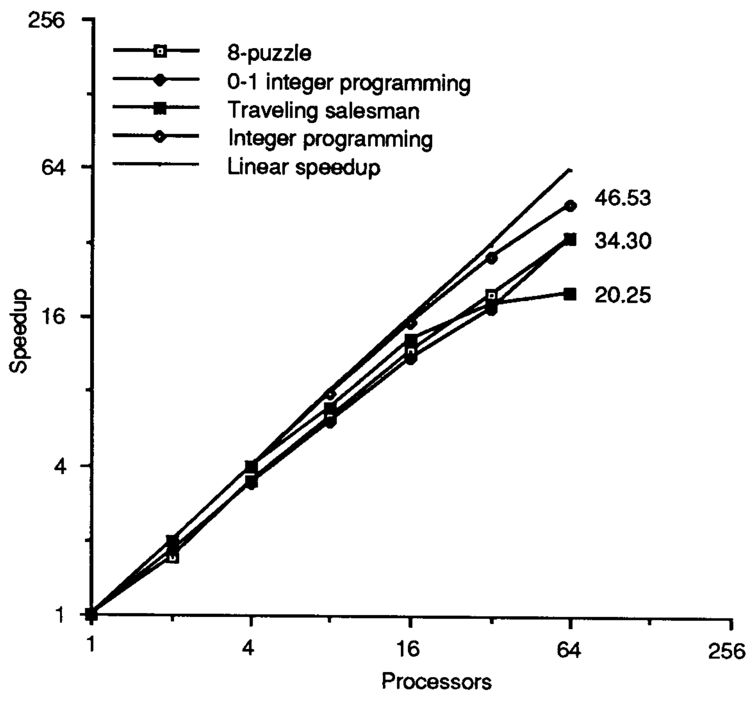

Figure 6.17. The speedup of the DL algorithm.

is unable to detect the arrival of a request for subproblems, and therefore, a response to that request is delayed until the load balance component. This can cause inefficiencies due to increased computation-overhead and due to increased idle time. A processor that is waiting for a response to its request either can be expanding subproblems, which may be non-essential, or can be idle performing no computations.

The effect of the scheme of operation on performance is demonstrated using the integer programming problem, which has the largest granularity among the four applications. The computations performed on a subproblem in the case of the integer programming problem is sliced into smaller components. At the end of each component, the presence of requests for subproblems is checked for, and if requests exist, they are handled at that point of time. The effect of this computation slicing on the performance is shown in tables 6.3 below. The table gives the improvement in the speedup of the DL algorithm with slicing relative to its speedup without the slicing. An improvement of about 10% can be achieved.

P	%improvement
2	0.0%
4	0.0%
8	3.5%
16	9.9%
32	7.3%
64	4.8%

Table 6.3. The improvement in the speedup of the DL algorithm due to slicing.

6.5 Comparison of Performance

In this section the performance of the three algorithms is compared in order to demonstrate that the performance of the DL algorithm is better than the performance of the CL and SHIFT algorithms, and to illustrate the effect of the computation-communication tradeoff on performance.

The speedups of the three parallel algorithms for the 8-puzzle problem is shown in figure 6.18. The speedups indicate that the overall performance of the DL algorithm is consistently better than the overall performance of the two other algorithms, which is a result of the reduction in communication-overhead of the algorithm, albeit at the expense of higher computation-overhead and more load imbalance, as can be seen from figures 6.20, 6.19, and 6.21. The load imbalance factors of the algorithm indicate that more load imbalance is incurred by the DL algorithm than the other two. Similarly, more computation-overhead is incurred by the DL algorithm as can be seen from the computation ratio of the three algorithms. However, the amount of communication-overhead incurred is less than that incurred by the CL and SHIFT algorithms. This makes the overall performance of the DL algorithm better than that of the first two, as indicated by the speedups.

The performance of the three algorithms for the 0–1 integer programming problem

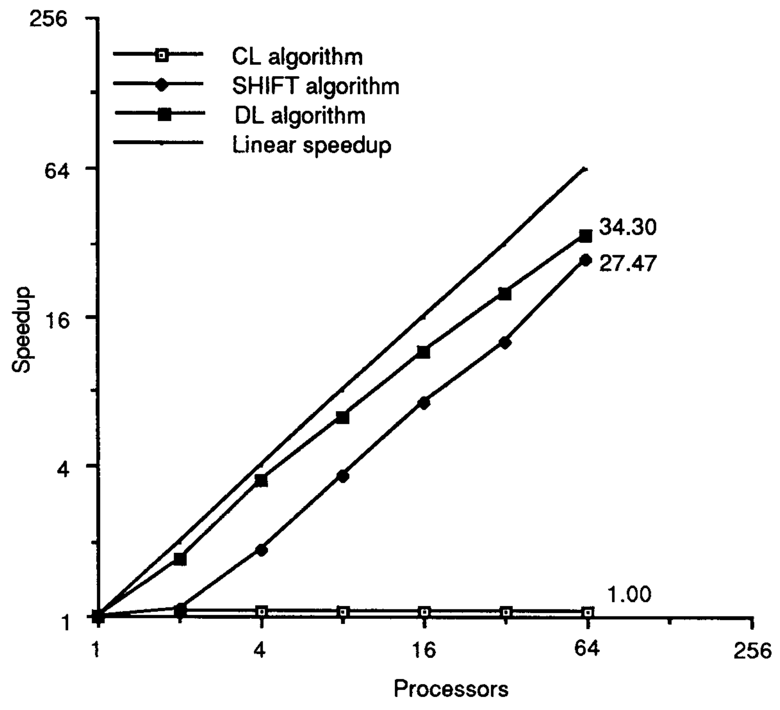

Figure 6.18. The speedups for the 8-puzzle problem.

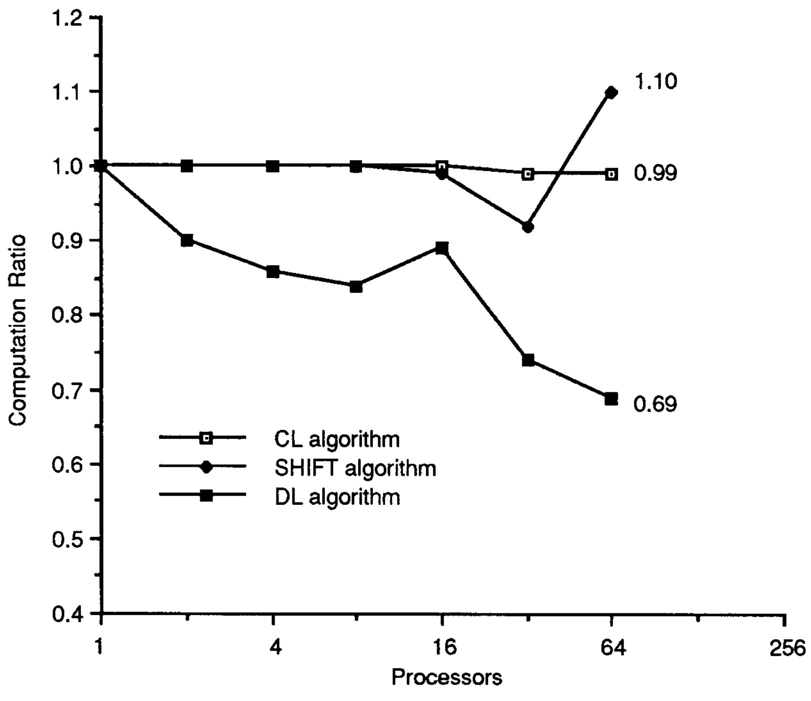

Figure 6.19. The computation ratio for the 8-puzzle problem.

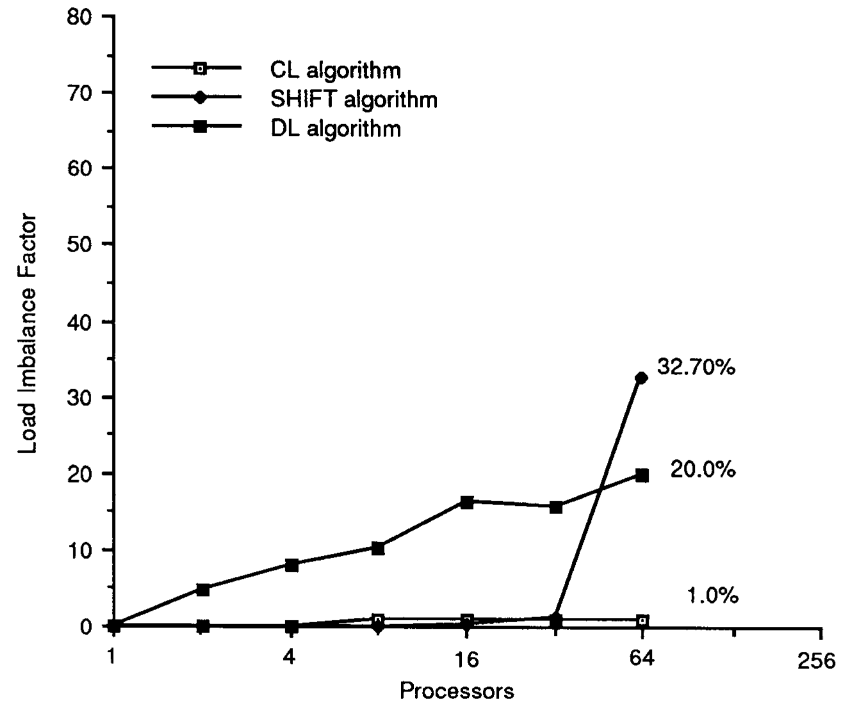

Figure 6.20. The load imbalance factor for the 8-puzzle problem.

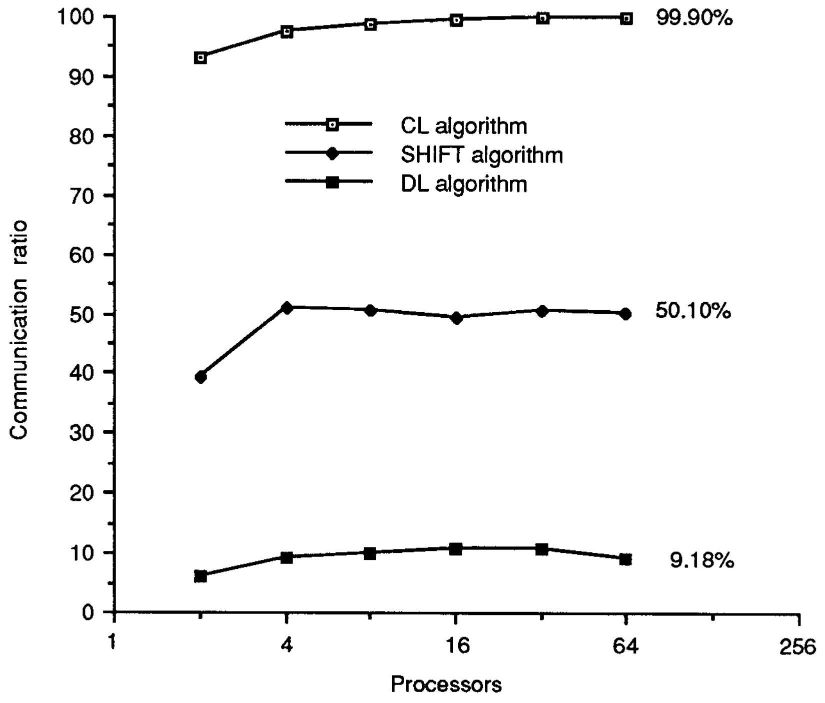

Figure 6.21. The communication ratio for 8-puzzle problem.

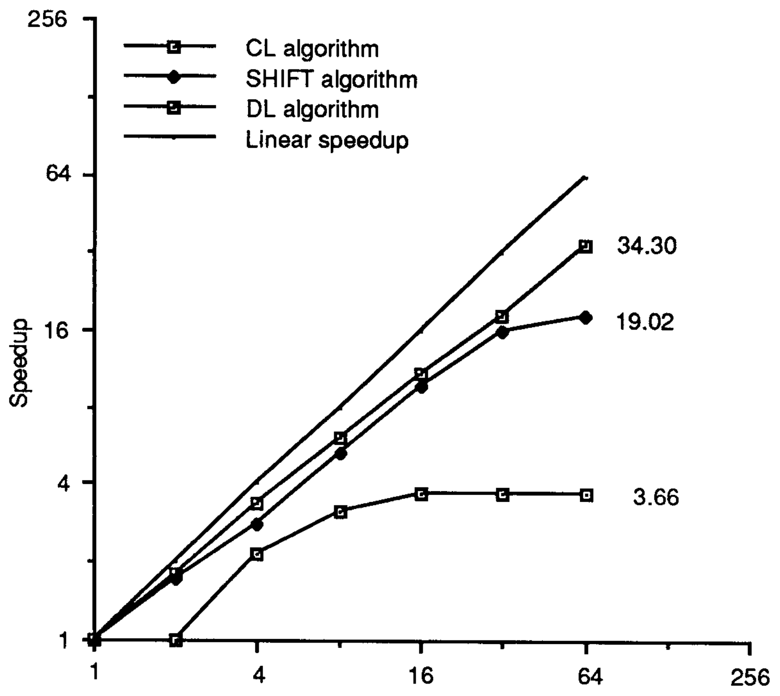

Figure 6.22. The speedups for the 0–1 integer programming problem.

shows a similar behavior. The speedup of the three algorithms for that application is shown in figure 6.22. The performance of the DL algorithm is consistently higher than the performance of the CL and SHIFT algorithms. This is due to the same factors discussed above for the 8-puzzle problem. This can be seen from the computation ratios, communication ratios, and load imbalance factors of the three algorithms, which are not shown compared to each other in this section, but are shown in the previous sections for the individual algorithms. They display similar behavior as the ones shown above for computation ratio, communication ratio and load imbalance factor of the 8-puzzle problem.

The performance of the three algorithms for the traveling salesman problem problem is shown in figure 6.23. The figures lead to similar conclusions as those of the previous two applications. The DL algorithm displays a better performance than the other two algorithms. The effect of lack of parallelism of the problem can be seen on the performance of the SHIFT and DL algorithm. The performance of both algorithms is significantly

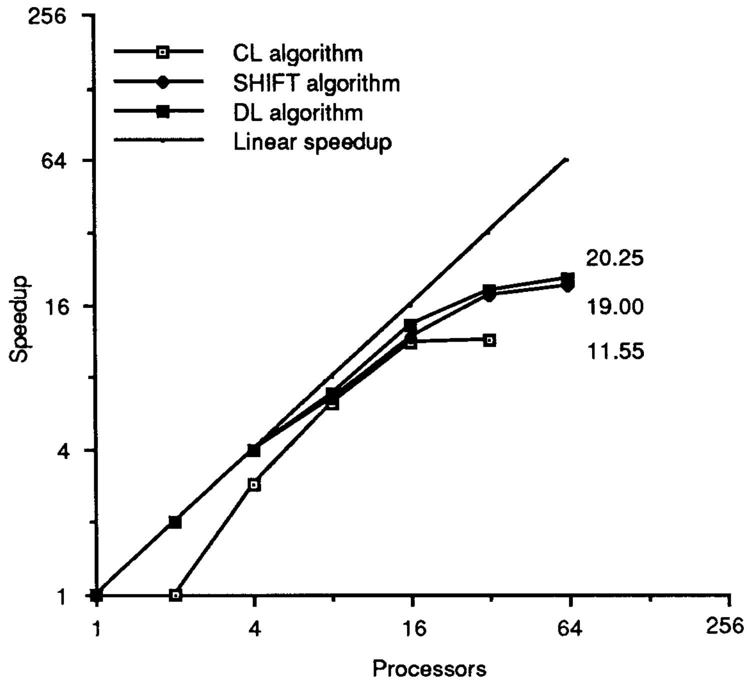

Figure 6.23. The speedup of the traveling salesman problem.

degraded by this lack of parallelism and the performance of the DL algorithm is only slightly better than that of the SHIFT algorithm.

The performance of the three algorithms for the integer programming problem is shown in figure 6.24. The DL algorithm displays a better performance than the first two up to 32 processors. This again, is attributed to the same factors as in the case of the first three applications, and the computation ratio, communication ratio and load balance factors are shown for the algorithms in the respective sections in which their performance is discussed.

However the CL algorithm displays a better speedup at 64 processors than the SHIFT and DL algorithm. This is attributed to the large amount of computation-overhead incurred by the two algorithms (as can be seen from the computation ratio in figure 6.14) coupled with the coarse granularity of the integer programming problem.

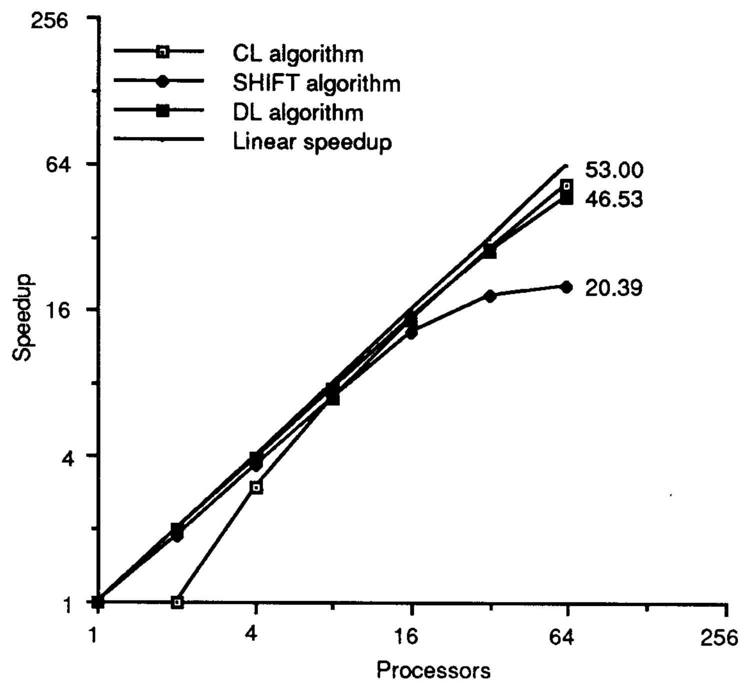

Figure 6.24. The speedup of the integer programming problem.

Although this seems to be an advantage for the CL algorithm over the DL algorithm, consideration must be given to the limitations of the CL algorithm that contribute to this result. The CL algorithm maintains all the list of active subproblems on a single processor, the master. The size of the memory on a single NCUBE/ten processor allows the solution of problems only of moderate size (the number of subproblems on the list of active subproblems is on the order of 2000). For more practical problems, a larger number of subproblems is generated, implying that the CL algorithm cannot be used, and an algorithm that uses a distributed strategy must be employed.

6.6 Summary of Results

The performance of the Logical Model algorithm has been measured using simulation, and has been used to reflect the effect of lack of parallelism in the test problems used on performance. The results indicated that lack of parallelism has a non-significant effect on the performance of the 8-puzzle problem and the 0–1 integer programming problem. However, slight lack of parallelism affects the performance of the traveling salesman problem and the integer programming problem.

The performance of the CL algorithm indicated that it is possible to obtain a speedup that is close to linear only when the granularity of computation of the algorithm is very large. In the case of medium to small granularity, the speedup is close to linear only for a small number of processors, but shows no gain in performance beyond 16 to 32 processors. This is mainly due to the large amount of communication-overhead that becomes significant as the contention for the master increases when the number of processors is increased. However, the amount of computation-overhead is minimal as a global list of active subproblems is maintained.

The performance of the SHIFT algorithm showed significant improvements in performance compared to the CL algorithm. This is due to a reduction in the amount of communication-overhead. However, the amount of this overhead is still significant due to the communication of subproblems and the synchronization of the processors. The algorithm incurs small amounts of computation-overhead.

The performance of the DL algorithms showed further improvements over the performance of the CL and SHIFT algorithms. This is attributed to the asynchronous strategy used by the algorithm which further reduces the amount of communication-overhead. This is done, however, at the expense of higher computation-overhead and more load imbalance compared to the CL and SHIFT algorithms. The algorithm, hence, reflects a better computation-communication tradeoff that leads to a better overall performance.

The performance of the three algorithms on systems with larger numbers of processors can be predicted from their performance which has been described in the previous sections. The CL algorithm, whose performance is mainly affected by communication-overhead, can only exhibit more diminishing gains in performance as the number of processors is increased. This is due to the increasing contention for the master, which can be seen from figure 6.4. The amount of communication-overhead incurred by the algorithm due to that contention is expected to continue to increase when the number of processors is increased beyond 64.

The performance of the SHIFT algorithm, which is less affected by communication-overhead, is also affected by computation-overhead and imbalance-overhead. The amounts of these two types of overhead increase as the number of processors are increased, as can be seen from figures 6.10 and 6.11, when the number of processors is increased beyond 64. The combined effect of these two types of overhead can cause the gains that can be obtained from the algorithm to be diminishing. In fact, the diminishing gain in performance of the SHIFT algorithm can be seen from its speedup (figure 6.13) as the number of processors approaches 64. Hence, more diminishing gains can be expected when the number of processors is increased significantly beyond 64 processors.

The performance of the DL algorithm is the least affected by communication-overhead among the three algorithms. Although the amount of computation-overhead and load imbalance increase when the number of processors is increased, their effect on performance is small which causes the performance of the algorithm to scale to a large number of processors, as described above. However, when the number of processor is significantly increased, the amount of computation-overhead and imbalance-overhead cause the gains in the performance of the algorithm to diminish as the advantage of reducing the amount of communication-overhead is overweighed by the disadvantage of large amounts of computation-overhead and imbalance-overhead.

The performance of the three algorithms for larger numbers of processors, however, can be severly degraded due to lack of parallelism in the problems solved when the size of the a problem is not increased when the number of processors is increased. Therefore, for systems with much larger number of processors, the sizes of the problems must also be increased to avoid performance loss by lack of parallelism. In system with larger numbers of processors, the size of memory per processor can be larger, which allows the solution of large problems.

CHAPTER 7

ACCELERATION ANOMALIES

An *acceleration anomaly* occurs in the execution of a parallel BB algorithm when one or more of the essential subproblems expanded by the sequential algorithm are not expanded by the parallel algorithm. An acceleration anomaly can cause the speedup of the parallel algorithm to be *superlinear*, i.e., greater than P when P processors are used to execute the algorithm.

Acceleration anomalies have been studied by researchers in the past using the framework described by the LM algorithm. Acceleration anomalies have been studied by Lai and Sahni [LaSa83], by Lai and Sprague [LaSp85, LaSp86], by Li [Li85], and by Li and Wah [LiWa84a, LiWa84b, LiWa86]. However, acceleration anomalies in BB algorithms with best-first search have not been studied in the context of the framework described by the DM algorithm. In this chapter the occurrence of anomalies in parallel BB algorithms and the anomalous behavior of the speedup curve of the SHIFT algorithm for the 8-puzzle problem are explained.

An acceleration anomaly can occur during the execution of a parallel BB algorithm when the selection heuristic function h is inconsistent with the lower bound function g [Li85, LiWa86]. The heuristic selection function h is said to be *consistent* with the lower bound function g if

$$h(P_i) < h(P_j) \Longrightarrow g(P_i) < g(P_j) \qquad \forall P_i,\, P_j \in \mathcal{A}, \tag{7.1}$$

otherwise, h is said to be *inconsistent* with g. It has been shown by [Li85, LiWa86] that for an acceleration anomaly to occur it is necessary for h to be inconsistent with g. This condition has been shown by [Trie86] to hold in the case where the processors operate asynchronously. Therefore, an acceleration anomaly cannot occur if h is consistent with g and may only occur if h is inconsistent with g.

The selection heuristic function h defined by equation 3.4 can often be inconsistent with the lower bound function g. (Recall that $h(P_i) = (g(P_i), p(P_i))$, i.e., subproblems with smaller lower bounds are selected first, and if there is a tie, then subproblems that are deeper and to the left of the tree are selected first). This is because in most applications, the lower bound function is not one-to-one, and many subproblems that have equal lower bounds are expanded by the sequential algorithm before the optimal solution is discovered. The values of h for these subproblems are not equal and less than the value of h for the subproblem that generates the feasible solution, yet their lower bound values are all the same, which makes h inconsistent with g.

This possible inconsistency of h with g is illustrated in figure 7.1, which shows a hypothetical BB tree generated by the sequential algorithm. There are six essential subproblems expanded by the algorithm before the optimal solution is discovered (subproblems 1, 2, 4, 6, 5, and 8). However, both subproblems 5 and 6 have equal lower bounds (that are also equal to the value of the optimal solution in this example). Although the value of h for subproblem 6 is less than the value of h for subproblem 5, the two subproblems have equal lower bounds, which makes h inconsistent with g in that example.

A subproblem whose value of h is inconsistent with its lower bound, and whose lower bound is equal to the value of the optimal solution becomes a candidate for deletion by the lower bound elimination test should the optimal solution (or any other feasible solution whose value is equal to that of the optimal solution) be discovered before the subproblem is expanded. This is illustrated using the example of figure 7.2. The subproblem P_i is

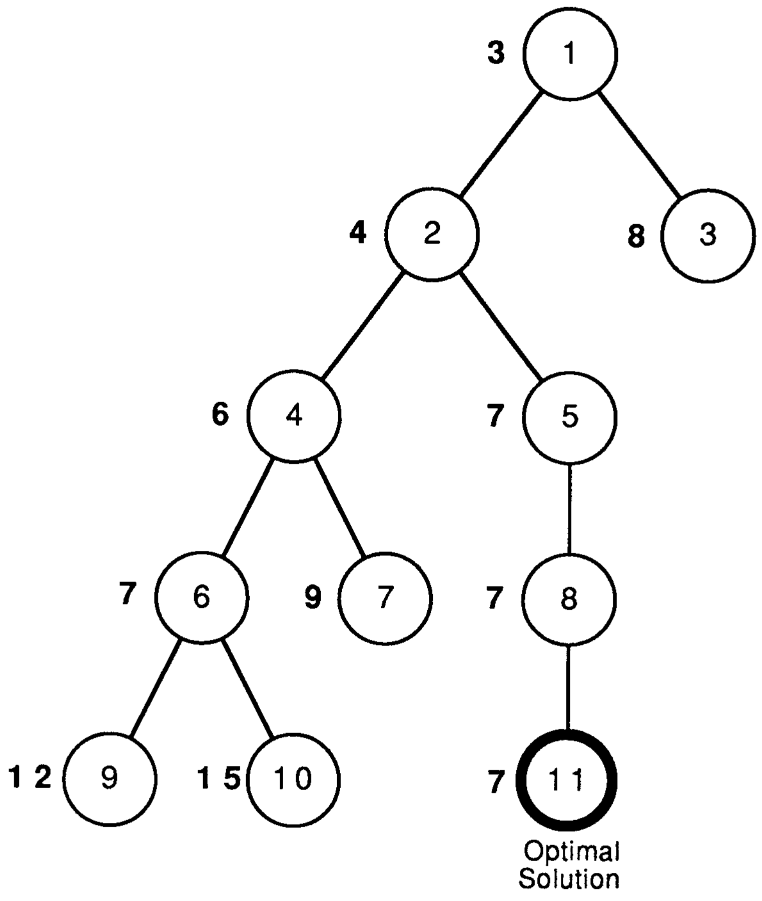

Figure 7.1. Example showing the possible inconsistency of h with g.

the subproblem whose expansion does not lead to a feasible solution, but whose lower bound $g(P_i)$ is equal to the value of the optimal solution g_{opt}. The subproblem P_j is the subproblem whose expansion generates the optimal solution P_{opt}. An acceleration anomaly occurs if the subproblem P_j is expanded before the subproblem P_i during the execution of a parallel BB algorithm. The optimal solution P_{opt} is generated and the incumbent is updated. Consequently, all subproblems whose lower bounds are greater than or equal to that of the new incumbent are deleted. Those include the subproblem P_i. Consequently, the essential subproblem P_i and any other essential subproblems in the subtree rooted at P_i are not expanded by the parallel algorithm, which leads to an acceleration anomaly.

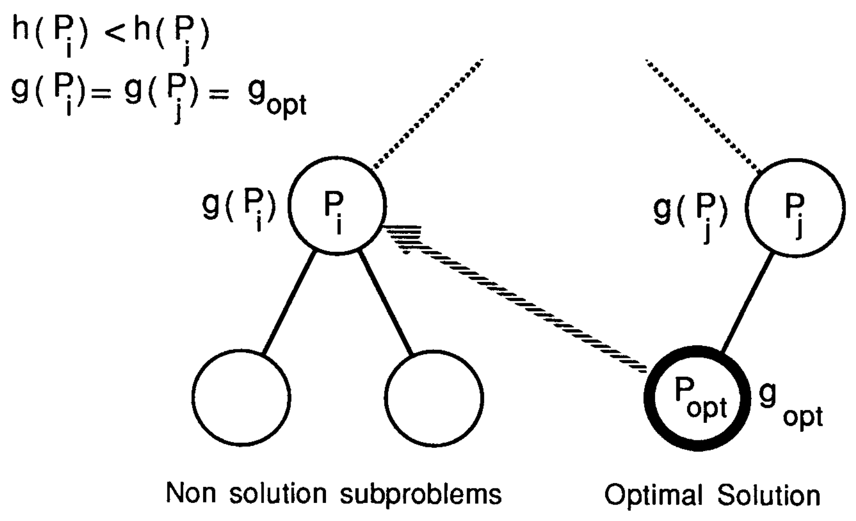

Figure 7.2. The occurrence of an anomaly.

(The anomaly can also occur when subproblems have lower bounds that are equal to the value of a suboptimal solution that is discovered during the search. However, we restrict the discussion below to the case where the solution is the optimal one).

The subproblem P_j, whose expansion causes the anomaly, can be a non-essential subproblem. In this case, P_j is not the subproblem that generates the feasible solution in the sequential algorithm, but is generated and expanded by the parallel algorithm as it incurs computation-overhead. It is possible for a non-essential subproblem to lead to the optimal solution since multiple optimal solutions can exist, and only one be discovered by the sequential algorithm. The acceleration anomaly occurs when this subproblem is expanded before P_i, as described above. This is illustrated by figure 7.3 which shows a hypothetical BB tree. It can be seen that h is inconsistent with g since subproblem 4 has a value of h that is smaller than that of subproblem 3, yet both subproblems have the same value of g. The sequential algorithm expands subproblem 1 into subproblems 2 and 3. The algorithm then expands subproblem 2 into subproblems 4 and 5. The algorithm expands subproblem 4 into subproblems 7 and 8. Finally, the sequential algorithm expands subproblem 7 to generate the optimal solution, which eliminates subproblems 3, 5 and 8, and terminates the algorithm. Therefore, subproblems 1, 2, 4 and 7 are the essential subproblems. A parallel BB algorithm employing two processors expands subproblem 1 into subproblems 2 and 3. Subsequently, subproblems 2 and 3 are expanded by the two processors, and the optimal solution is generated by expanding subproblem 3; subproblems 4 and 5 are deleted by the lower bound test, and the algorithm is terminated. Therefore, essential subproblems 4 and 7 are not expanded by the parallel algorithm, resulting in an acceleration anomaly. The expansion of the non-essential subproblem 3 caused that anomaly.

The expansion of non-essential subproblems that generate feasible solution occurs as the parallel BB incurs computation-overhead due to the factors described in Chapter 4. The LM algorithm, whose framework has been used to study acceleration anomalies, incurs computation-overhead due to lack of parallelism. However, a parallel BB algorithm on a DMM incurs computation-overhead due to other factors as well. Hence, the likehood that a parallel BB algorithm will incur an anomaly due to the expansion of a non-essential subproblems is higher than that of the LM algorithm.

The subproblem P_j, whose expansion causes the anomaly, can also be an essential subproblem. Therefore, P_j is the subproblem that generates the optimal solution in the case of the sequential algorithm. Consequently, for the acceleration anomaly to occur, P_j must coexist with P_i, and a change in the order of expansion of subproblems from that of the sequential one must occur in order for P_j to be selected before P_i and, hence, cause the anomaly. The change in the order of expansion of subproblems can happen due to the action of the load-balance component which leads to the selection of a set of subproblems other than the set $\mathcal{S}$ selected by the LM algorithm. The load imbalance incurred by the algorithm can also cause subproblems to be expanded in a different order.

We define the set S_e to be the set of subproblems whose expansion by the sequential

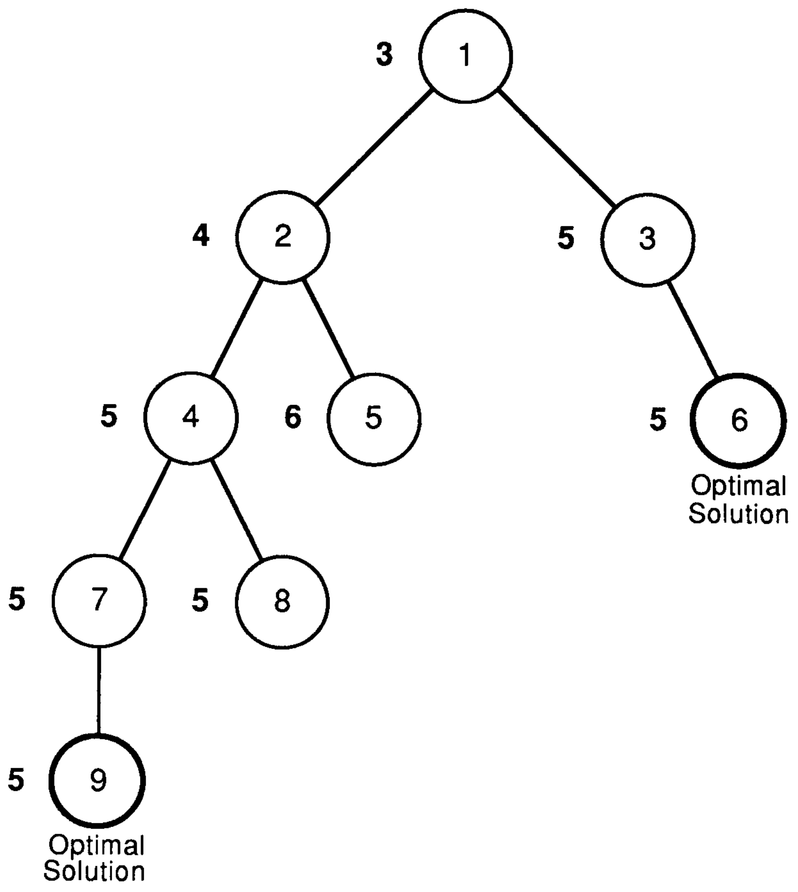

Figure 7.3. Example illustrating an acceleration anomaly.

algorithm does not lead to the optimal solution, but whose lower bound values are equal to the value of the optimal solution. The subproblems that belong to this set are all candidates for deletion in the event of an acceleration anomaly as described above. We define the *extent* of the acceleration anomaly to be the number of essential subproblems that are deleted by the lower bound elimination test due to the anomaly. The maximum extent of the anomaly is $|S_e|$ since only subproblems belonging to the set S_e can be deleted in case of an anomaly. However, the extent of the anomaly is generally less than $|S_e|$. The structure of the BB tree can cause some subproblems belonging to S_e be expanded before the optimal solution is discovered, and hence, before the anomaly can occur. This can be seen in the example of figure 7.1. The parallel algorithm must expand subproblem 5 before the subproblem leading to the optimal solution can be expanded. Furthermore, the change in order necessary to cause an anomaly, as described above, may only happen after a number of the subproblems that belong to S_e have been expanded, which also makes the extent of the anomaly less than $|S_e|$.

The existence of an acceleration anomaly in the execution of the parallel BB algorithm does not necessarily imply that the speedup of the algorithm is superlinear. This is due to other factors that degrade the performance of the parallel BB algorithm, and have been described earlier in Chapter 4. The first factor is computation-overhead. It is possible for the parallel algorithm to experience an acceleration anomaly, yet, due to computation-overhead, have the total number of subproblems expanded by the parallel algorithm be larger than the number of subproblems expanded by the sequential algorithm, hence preventing the speedup from becoming superlinear. In other, words, computation-overhead can make the total number of subproblems expanded by the parallel algorithm larger than the number of subproblems expanded by the sequential one, although the number of essential subproblems expanded by the parallel algorithm is less than the number of essential subproblems expanded by the sequential algorithm. The second factor is communication-overhead. It is possible for the parallel algorithm to incur large amounts of communication-overhead whose effect in degrading the performance of the parallel algorithm overweighs any gains in performance due to the reduction in the number of subproblems expanded by the parallel algorithm, also preventing the speedup from becoming superliner. The third factor is imbalance-overhead, which has a similar effect on overall performance to that of communication-overhead; the speedup of the algorithm becomes sublinear due to imbalance-overhead although the number of subproblems expanded by the parallel algorithm is less than the number of subproblems expanded by the sequential one.

However, the existence of an acceleration anomaly can be seen as an unexpected improvement in the overall performance of the parallel algorithm, as that seen in the speedup curve of the SHIFT algorithm for the 8-puzzle problem (see figure 6.13). The deletion of a number of essential subproblems in the case of the parallel algorithm that is caused by the anomaly reduces the overall number of subproblems expanded by the parallel algorithm compared to the number of subproblems that would have expanded had the anomaly not occurred. This in turn reduces the effect of computation-overhead on the overall performance and causes the speedup to improve. The extent of the anomaly determines the amount of this improvement in the speedup; the larger the extent of the anomaly, the more visible the effect of the anomaly is on overall performance. Indeed, if the extent of the anomaly is large enough, it is possible for the speedup to be superlinear, in spite of other factors that degrade performance.

It is possible to explain the anomalous behavior of the speedup curve of the SHIFT algorithm for the 8-puzzle problem based on the understanding of how acceleration anomalies occur, and based on some characteristics of the applications. The ratio of $|S_e|$ to the total number of subproblems expanded by the sequential algorithm is shown in table 7.1 for each of the four applications (averaged only for the test problems that showed the anomaly). The table indicates that this ratio is negligible in the case of the integer programming problem and the 0–1 integer programming problem. Hence, the possible extent of any acceleration anomaly is small and is unlikely to affect the overall performance of a parallel algorithm. The ratio is larger for the traveling salesman problem and for the 8-puzzle problem, as can be seen from the table. Therefore, the possible extent of an acceleration anomaly in these two applications is large. The larger ratio for the 8-puzzle problem indicates that the performance of this application problem is more likely to be affected by an acceleration anomaly than the traveling salesman problem. Hence, while acceleration anomalies occur in the case of the traveling salesman problem, as evident from

Application Problem	Ratio
8-puzzle	32%
0–1 integer programming	2%
Traveling salesman	20%
Integer programming	0%

Table 7.1. The ratio of $|S_e|$ to the total number of subproblems.

the computation ratio curves of the SHIFT algorithm for that application (see figure 6.10), the effect of these anomalies is significant only in the case of the 8-puzzle problem due to the large value of $|S_e|$.

However, it is not sufficient for $|S_e|$ to be large in order for an acceleration anomaly to happen; considerable expansion of non-essential subproblems and a change in the order of expansion of subproblems must also be present if the anomaly is to occur. The load imbalance factor of the three parallel algorithms for the 8-puzzle problem is shown in figure 6.20. The load imbalance factors of the three algorithms is relatively small for all values of P except for the SHIFT algorithm at 64 processors; the load imbalance factors indicate significant load imbalance. This load imbalance for the SHIFT algorithm coupled with the large ratio of $|S_e|$ for the 8-puzzle problem explain why the acceleration anomaly occurs for the 8-puzzle problem in the SHIFT algorithm but not in the other two algorithms and for the other three applications. The number of non-essential subproblems expanded by the algorithm, and the possible change in the order of expanding subproblems both caused by the load imbalance enhances the chances for an acceleration anomaly to occur in the case of the SHIFT algorithm compared to the other two algorithms.

The above explanation for the occurrence of acceleration anomalies in the SHIFT algorithm for the 8-puzzle problem is based on the observed behavior of the algorithms. However, it is important to note that the occurrence of anomalies is influenced by many other factors such as the structure of the BB tree, the nature of the generation of optimal solutions from non-essential subproblems, as well as the nature of the communication of information among the processors. The effect of these factors is not investigated in this study, but can be pursued in future research.

CHAPTER 8

CONCLUSIONS

8.1 Summary and Contributions

The BB algorithm is a heuristic search algorithm used to solve many problems in science and engineering that have no direct methods of solution or only inefficient ones. Although considerably more efficient than exhaustive search, the BB algorithm is a computationally intensive algorithm that can benefit from the high performance potential of DMMs. However, the non-deterministic and irregular nature of the data set generated by the algorithm, as well as the need for global information make the mapping of this algorithm onto multiprocessors without shared memory a challenge. This study has investigated the mapping of the BB algorithm onto DMMs and has shown that good performance gains are possible even when a large number of processors is employed, and when inherent parallelism exists.

The factors that affect the performance of a parallel BB algorithm on DMMs have been identified in this study. These factors give rise to three types of overhead that degrade performance: computation-overhead, imbalance-overhead, and communication-overhead. Computation-overhead reflects the expansion of subproblems that are not expanded by the sequential algorithm. Imbalance-overhead reflects idle time spent the processors due to load imbalance. Communication-overhead reflects the cost of inter-processor communications. The performance of a parallel BB algorithm is affected not only by the three types of overhead, but is also affected by a tradeoff among them. This tradeoff, which is referred to as the computation-communication tradeoff, has been effectively used to obtain good performance gains, particularly when the number of processors is large.

The limitations of analytical modeling and algorithm analysis techniques in predicting the performance of parallel BB algorithms on DMMs suggested the use of experimental approach. The extent to which the three types of overhead and the computation-communication tradeoff affect the performance of a parallel BB algorithm for actual applications has been demonstrated experimently using three parallel algorithms for BB that solve four applications. The three algorithms have been implemented on a hypercube multiprocessor to obtain experimental results. The third algorithm (the DL algorithm) also demonstrates the performance of a new load balancing strategy that, in general, leads to superior scalable performance compared to the strategies used by the first two algorithms.

The major contributions of this study are:

- The factors that affect the performance of parallel BB algorithms on DMMs have been described.
- The effects of these factors and a possible tradeoff between their effects on performance for actual applications have been demonstrated.
- The effective use of the computation-communication tradeoff embodied in the DL algorithm through its load balancing strategy is shown to lead to an overall performance that is generally better than previously reported approaches.
- Acceleration anomalies have been shown to occur in the execution of a parallel BB algorithm for one of the applications implemented. An explanation for the occurrence of these anomalies has been provided.

8.2 Future Research

A number of research directions can be explored that extend the scope and results of this thesis. They include:

- Further characterization of the computation-communication tradeoff, in particular the effect of other problems characteristics, such as the structure of the BB tree, on this tradeoff.

- Design of parallel algorithms that exploit other levels of parallelism in the sequential BB algorithm, in particular combining subproblem-level parallelism with algorithm-level parallelism for application problems with high granularity.

- Investigating the possible superlinear behavior of BB algorithms in particular, and the larger class of heuristic search algorithms in general. In some of the experiments conducted in this study, evidence of such anomalous behavior has been observed. While the results do not explicitly reflect superlinear speedups, they indicate the possibility of such speedups occurring for larger problems and at larger numbers of processors.

- Investigating the mapping of BB algorithms on shared memory multiprocessors. The presense of shared memory can considerably reduce the amount of computation-overhead since access to a globally shared list and a globally shared incumbent is possible. However, the contention for this shared data is likely to introduce an overhead that is similar in nature to communication-overhead present in DMMs. This overhead can potentially degrade the performance of the parallel algorithm, and a distributed strategy in which multiple lists exists in shared memory may be feasible.

- Examining the mapping of other classes of search algorithms on parallel processors. Examples include AND/OR tree search, alpha-beta search and Iterative Deepening A*.

APPENDICES

APPENDIX A

The NCUBE/ten

This appendix briefly describes the architecture and the programming of the NCUBE/ten multiprocessor. A more detailed description of the multiprocessor and its use can be found in [NCUB85, HMSC86, HMSP86]. This appendix also describes the embedding of rings into the hypercube topology, and the use of minimal spanning trees to perform the broadcast of feasible solutions.

The NCUBE/ten is a commercial hypercube multiprocessor system that is made by the NCUBE Corporation. The system can host up to 1024 processing nodes connected as a 10-dimensional hypercube array. The system used in our experiments had 64 processing nodes connected as a 6-dimensional hypercube array.

A processing node consists of a custom 32-bit processor chip and 512 K-bytes of high speed memory. The processor chip is a general purpose processor which is capable of executing non-floating point instructions at about 2 MIPS, single precision floating point instructions at about 0.5 MFLOPS and double precision floating point instructions at about 0.3 MFLOPS. The processor has a two-address instruction set that is similar to that of the VAX-11 series. The instruction set has special instructions that facilitate message communication.

Communication with other nodes in the hypercube array is done asynchronously through 22 bit-serial links, which are paired into 11 bidirectional channels. Ten of these bidirectional channels are used to form the 10-dimensional hypercube array. The eleventh channel provides a channel to I/O processors, as will be described below. The channels operate at 7.5 MHz with parity check, which results in a data transfer rate of about 750 K-byte per second in each direction. Each channel has two 32-bit write-only registers associated with it. One is the address register which contains the location, in the node memory, of the first byte in the message. The other is the count register which indicates the number of bytes left to send or receive in the message. Send or receive operations are initiated by the processor by writing the address of the first byte in the message to the address register and the size, in bytes, of the message to the count register (a non-zero count actually triggers the DMA action). The processor then continues with its operations while the DMA channels complete the communication operation. Interrupts are used to signal the processor when the channel is ready for a new transfer.

The hypercube array is connected via the I/O channels described above to as many as 8 I/O processors that provide an interface between the array and the external world. The I/O structure allows data transfers with the cube array over separate bit-serial links to and from each node (the eleventh channel). These links can support a total data rate as high as 90 M-bytes per second into or out of the cube array. At least one of the I/O processors must be a host processor, and there can be as many as eight. The host processor supports user terminals, disk drives, and a number of other peripherals. The host processor which also runs the operating system, is used for program development.

Programs on the NCUBE can be developed in host and node assembly language or in one of two high level languages: FORTRAN 77 and C. All our implementations were coded using the C programming language.

The C programming language is augmented by a number of library calls to facilitate message passing, which is accomplished through the node operating system VERTEX. VERTEX is a small nucleus that resides in each node, and basically provides communication between the nodes via a set of send and receive functions that facilitates message transfer between any two nodes in the hypercube array. The send function has the following general form:

nwrite(*length, message, dest, type, status, error*)

where *length* is the length of the outgoing message (the length of a message can be up to 64 K-byte), *message* is the name of the buffer that contains the message, *dest* is the logical number of the node that is to receive the message, *type* is the type of the message, an attribute that is used to distinguish messages, *status* indicates when the message has left the buffer *message* to the VERTEX buffer area and *error* is an error code. The receive function has a similar general form:

nread(*length, message, source, type, status, error*).

nread looks for the first message from *source* of type *type* and copies it into the buffer *message*. It is possible to specify do not care conditions for *source* and *type* to receive a message regardless of it source or type.

Common topologies such as rings, grids, trees, and pyramids can be embedded efficiently into the hypercube topology. (see [Buzz88] for a brief description of the embedding of some common topologies). In the following two sections the embedding of rings and of spanning trees is described. An embedded ring is used in the SHIFT algorithm to accomplish the shifting of subproblems among processors; and an embedded spanning tree is used to broadcast a feasible solution from where it is discovered to all other processors in the hypercube array.

A.1 Embedding of Rings

The embedding of rings in the hypercube topology can be easily described using Gray codes. A Gray code $G(x)$ is a one-to-one mapping between integers such that for

i	$G(i)$
000	000
001	001
010	011
011	010
100	110
101	111
110	101
111	100

Table A.1. A Gray code for three bit integers

any two consecutive integers in the domain, the binary representation of their values in the range differs in exactly one bit position. It is assumed that the domain is finite and that the smallest and largest integers in the domain are consecutive. The inverse Gray code function is denoted by $F(x)$. An example of a Gray code for three bit integers is shown in Table A.1.

It is possible to have many distinct Gray codes. The one shown in the Table A.1 is obtained using the following rule: let j be the logical right shift of an integer i by one bit position; $G(i)$ is then the exclusive OR of i and j.

An N-processor ring consists of N processors connected in such a way that each processor has one predecessor and one successor. This is shown in figure A.1(a). The predecessor of a processor i is given by $G(F(i) - 1)$, and the successor of a processor i is given by $G(F(x)+1)$. The embedding of the 8-processor ring in a 3-dimensional hypercube is shown in figure A.1(b). The darkened links are those used to form the ring.

A.2 Incumbent Broadcast

In both the SHIFT and the DL algorithms, a feasible solution is broadcast from the processor where it is discovered to all other processors in the hypercube array. An

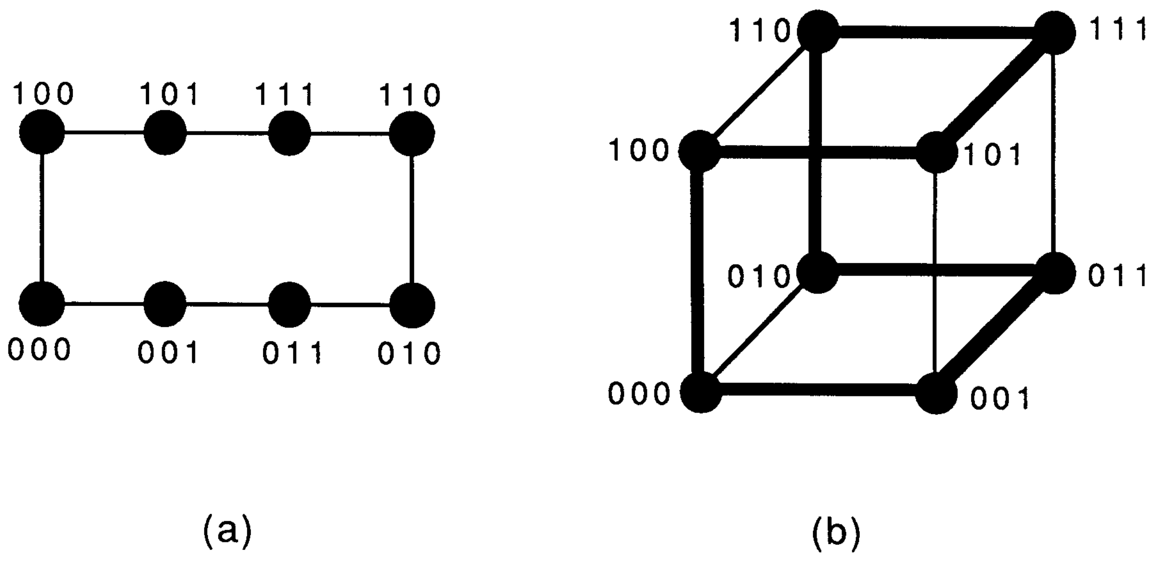

Figure A.1. Embedding of a ring in a 3-dimensional hypercube.

efficient strategy to accomplish this broadcast is based on the embedding of a minimal spanning tree in the hypercube, rooted at the source of the broadcast (i.e., the processor that discovered the feasible solution). The root processor sends the feasible solution to all of its neighbors, who in turn send it to their neighbors who have not received the solution. In n steps (where n is the dimension of the hypercube), all processors receive the broadcast feasible solution. Therefore, the use of the minimal spanning tree to perform the broadcast takes $O(n)$ steps. This compares favorably to the $O(2^n)$ steps it would take if an embedded ring was used to perform the broadcast.

The embedding of a spanning tree in a 3-dimensional hypercube is illustrated in figure A.2. The spanning tree is shown embedded in the hypercube in figure A.2(a), and is shown rearranged according to the steps of the broadcast in figure A.2(b); in both cases processor 0 is assumed to be the root of the broadcast. In the first step of the broadcast, processor 0 sends the feasible solution to processors 1, 2, and 4. In the second step of the broadcast, processor 1 send the feasible solution to processors 3 and 5; processor 2 sends the solution to processor 6. In the third and final step of the broadcast, processor 3

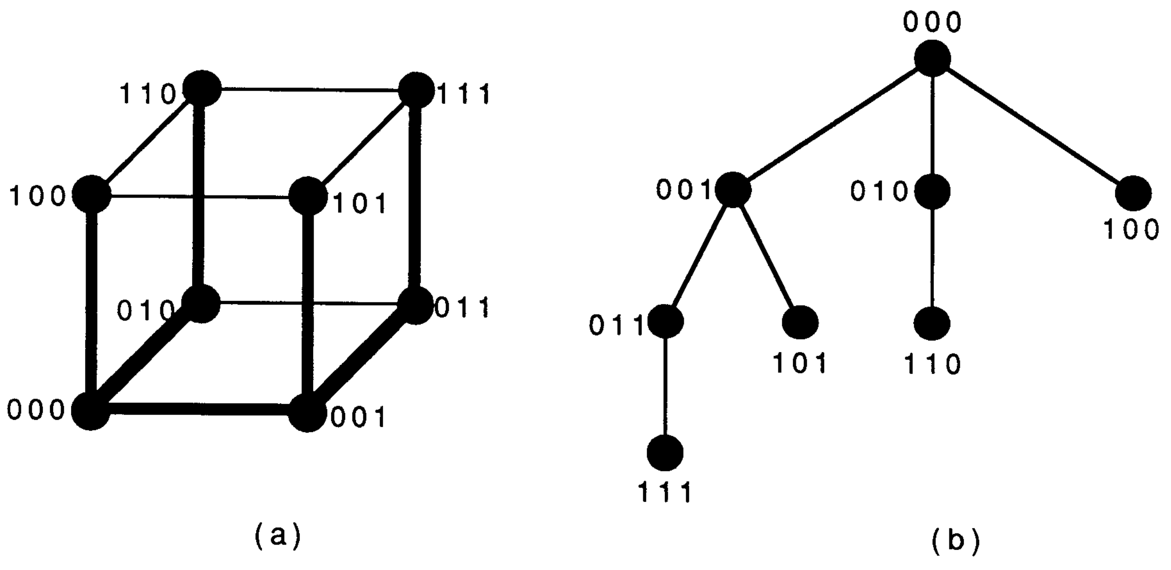

Figure A.2. Embedding of a spanning tree in a 3-dimensional hypercube.

sends the feasible solution to processor 7. The pseudocode of the implementation of the broadcast using the C programming language is shown in figure A.3.

```
broadcast(root,nodeid,dim,data)
{
        int i,nid,ptwo,to;

        nid=nodeid^root;
        ptwo=1;
        for (i=0 ; i < dim ; ++i) {
            if (ptwo > nid) {
               to=(nid+ptwo)^root;
               send(data,to);
            }
            ptwo=2*ptwo;
        }
}
```

Figure A.3. The implementation of broadcast using a spanning tree.

APPENDIX B

The Test Problems

A number of test problems have been used to obtain performance results for the parallel algorithms on the NCUBE/ten multiprocessor. In this appendix these test problems are described.

The test problems are instances of the four applications: the traveling salesman problem, the 8-puzzle problem, the integer programming problem, and the 0–1 integer programming problem. The test problems for each application have been selected from among standard benchmark problems for that application to obtain realistic performance. However, since benchmark problems are not available for all applications, some of the test problems have been randomly generated as will be described.

It is important to note that the size of a test problem has been selected such that the test problem can be solved on a single NCUBE/ten processor and at the same time generate the largest possible number of subproblems, subject to the memory limits of the single processor. The size of a test problem is not increased as the number of processors is increased. This is done in order to obtain true speedups for each application.

B.1 The Traveling Salesman Test Problems

A test problem for an N-city traveling salesman problem consists of an $N \times N$ matrix which represents the cost of traveling between pairs of cities. The number of cities (i.e., the value of N) is 30. The cost of traveling between any two cities is randomly generated using a uniform number generator between 1 and 99. The cost of traveling from city i to city j is not necessarily the same as the cost of traveling from city j to city i, which makes the test problems asymmetric. Twelve test problems were used generating a maximum of 1200 subproblems. It was possible to solve some problems with N ranging from 35 to 40 on a single processor, but those were not used as test problems since only a small number of them were possible to obtain.

B.2 The 8-puzzle Test Problems

A test problem for the 8-puzzle consists of two parts: an initial state and a goal state. The state of the 8-puzzle is represented as a 3x3 matrix whose elements correspond to the tile positions on the 8-puzzle board. The value of each element in the matrix represents the number of each tile on the board. The number 0 is used to represent the empty slot on the board.

The goal state in our test problems is always the same and is given by:

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 0 \end{bmatrix}$$

The initial state for a test problem is generated from the goal state using the following simple procedure. Two elements from the goal matrix are selected at random. The two elements are then exchanged. This is repeated on the resulting matrix a large number of times to obtain an initial state.

The solution space of the 8-puzzle problem is divided into two separate halves [Nils80]. That is, an initial state from one half of the space can never yield to a goal state in the other half. In this case, the 8-puzzle problem is said to be not solvable. The above procedure for generating the initial state for a test problem does not guarantee that the 8-puzzle test problem is solvable. The procedure given in [Scho67] is used to insure that a test problem is solvable. The procedure gives a necessary and sufficient test to decide on the solvability of a pair of initial and goal states. This procedure is employed in order to insure the solvability of the test problems.

The number of test problems used for the 8-puzzle is five. The problems generated 9000 subproblems on the average.

B.3 The Integer Programming Test Problems

The integer programming test problems are selected from a standard benchmark for integer programming problems suggested by Halidi [TrWo69]. These problems are referred to as IBM-0 to IBM-9. The problems are considered simple yet difficult to solve [TrWo69]. The size of some of the test problems is too small for the purpose of the experiments, generating only a few subproblems to discover the optimal solution. Therefore, only one problem (IBM-9) was used from the complete set of problems, generating about 1800 subproblems. The number of variables is 15 and the number of constraints is 35. A complete description of the problems and their solutions can be found in [TrWo69].

B.4 The 0–1 Integer Programming Test Problems

The same set of benchmark problems used for the integer programming problem are used for the 0–1 integer programming problem. The integer programming problems are converted to 0–1 integer programming problems using the method described in [WuCo80]. Two binary variables were used to represent each integer variable. Only four of the benchmark problems generated a sufficient number of subproblem, and were used (IBM-1, IBM-2, IBM-3 and IBM-9). The average number of subproblems generated is 2500.

APPENDIX C

IMPLEMENTATION DETAILS

There are two major components to the implementation: the *Control Program (CP)* and the *User Program (UP)*. The UP provides the definition of the specific problem to be solved. It basically defines the procedure by which new subproblems are generated from a subproblem. Therefore, the UP is a sequential program which does not express any parallelism. The CP provides the parallel implementation of a generic BB program which can solve any problem specified by the user program. This approach facilitates experimentation with different applications. The UP can be written for a number of problems with ease since it represents only sequential code. Different UP's can be then used with the same CP to implement different problems [FiMa85, FiMa87]. However, the approach can be less efficient than that in which the implementation is specific to a particular problem only.

The two components of the implementation interface through a defined set of procedures. The two components and their interface are described below for a sequential BB algorithm.

C.1 The Control Program

The CP program starts by initializing its variables and creating the lists: a list of active subproblems and a child list. The list of active subproblems is implemented as a Heap since that data structure is more efficient for maintaining large priority queues in the case of BB algorithms (see [YuWa83] for a discussion of various data structures for implementing the list of active subproblems, including the Heap). The key used to insert and remove subproblems from the Heap can be user defined. This allows the implementation of various search strategies. The child list is implemented as a linked list since the number of child nodes generated when expanding a subproblem is relatively small.

The initial problem is then read from the user through the user defined function **read_problem**. The initial subproblem is created using that problem and is inserted on the active subproblems list.

The first subproblem on the list of active subproblems is removed. A call is made to the user defined function **expand**, which returns the children of the subproblem on the child list. The CP performs the termination procedure, checks for the feasibility, and performs the lower bound test for each of the subproblems on that list and inserts those that pass the tests on the list of active subproblems. This process is repeated until the list of active subproblems is empty, at which point the algorithm terminates. The user defined function **print_solution** is used to print the solution stored in the incumbent.

The representation of a subproblem consists of the following:

1. The subproblem ID.
2. The subproblem level number.
3. The subproblem path number.
4. The lower bound of the subproblem.
5. A flag to indicate when the subproblem represents a feasible solution.
6. A flag to indicate when a problem cannot lead to a feasible solution.
7. A data structure which is specific to a particular problem which represents the partial solution represented by the subproblem.

A special subproblem, referred to as the *constant* subproblem, is also used. This subproblem is similar to the regular subproblems described above but is used to store the description of the part of the subproblem that does not change when a subproblem is expanded. That is, the data structures that are constant through out the computation, but are still needed for the computation of lower bounds and partial solutions. Examples of such structures include the goal state in the 8-puzzle problem, the cost matrix in the traveling salesman problem, and the constraint matrix in the 0–1 integer programming problem. This serves two purposes. First, it saves memory space since constant parts of the structure are not duplicated in each subproblem. Second, the constant subproblem is also broadcasted to all processors at the beginning of the program, therefore, saving communication time during the algorithm.

C.2 The User Program

The user program implements the function needed to decompose a subproblem into smaller subproblems, determine feasibility of new subproblems, and determine whether a subproblem can lead to a feasible solution or not. The UP interfaces to the CP through a set of defined procedures and termination. These procedures are:

1. **print_solution**: prints the final solution.
2. **print_subproblem**: prints a subproblem.
3. **read_problem**: reads the initial problem description on the host.
4. **expand**: takes a subproblem, branches it into smaller subproblems and returns the resulting subproblems on the child list. This procedure also sets the appropriate flags in the subproblem to indicate if it is a feasible solution, or if it is can lead to one.

BIBLIOGRAPHY

BIBLIOGRAPHY

[AbMu88] T. S. Abdelrahman and T. N. Mudge, "Parallel branch and bound algorithms on hypercube multiprocessors," *Proc. Third Conference on Hypercube Concurrent Computers and Applications,* pp. 1492–1499, 1988.

[AgJa86] D. P. Agrawal and V. K. Janakiram, "Evaluating the performance of multicomputer configurations," *Computer,* vol. 19, no. 5, pp. 23–37, 1986.

[Amet85] Ametek Corporation, *Hypernet System 14/n,* 1985.

[Amet88] Ametek Corporation, *Series 2010 Brochures and Application Software Reference Material,* 1988.

[AnCh86] S. Anderson and M. C. Chen, "Parallel branch-and-bound algorithms on the hypercube," *Proc. Second Conference on Hypercube Multiprocessors,* pp. 309–317, 1986.

[Batc80] K. E. Batcher, "Architecture of a Massively Parallel Processor," *Proc. 7th Annual International Symposium on Computer Architecture,* pp. 168–174, 1980.

[BCLR86] S. Bhatt, F. Chung, T. Leighton and A. Rosenberg, "Optimal simulation of tree machines," *Proc. Foundations of Computer Science Conference,* Toronto, 1986.

[Boeh85] R. L. Boehning, *A Parallel Branch-and-Bound Algorithm for Integer Linear Programming Models,* PhD Dissertation, University of Missouri–Rolla, 1985.

[BrKa88] B. Breas and P. Karger, "Placement, assignment and floorplanning," in *Physical Design Automation of VLSI Systems,* B. Breas and M. Lorenzetti, eds., The Benjamin/Cummings Publishing Company, Menlo Park, California, 1988.

[BrKT87] A. Bruin, A. Kan and H. Trienekens, "A simulation tool for the performance evaluation of parallel branch and bound algorithms," Report 8720/A, Econometric Institute, Erasmus University, The Netherlands.

[Buzz88] G. D. Buzzard, *High Performance Communications for Hypercube Multiprocessors,* PhD Dissertation, The University of Michigan, 1988.

[ChSa86] T. F. Chan and Y. Saad, "Multigrid algorithms on the hypercube multiprocessor," *IEEE Transactions on Computers,* vol. C-35, no. 11, pp. 969–977, 1986.

[Dant63] G. B. Dantzig, *Linear Programming and Extensions,* Princeton University Press, Princeton, N.J., 1963.

[DeFS87] D. DeWitt, R. Finkel and M. Solomon, "The Crystal multicomputer: design and implementation experience," *IEEE Transactions on Software Engineering,* vol. SE-13, no. 8, pp. 953–966, 1987.

[EfRa66] M. A. Efroymson and T. C. Ray, "A branch-and-bound algorithm for plant location," *Operations Research,* vol. 14, pp. 361–368, 1966.

[Felt88] E. W. Felten, "Best-first branch and bound on a hypercube," *Proc. Third Conference on Hypercube Concurrent Computers and Applications,* pp. 1500–1504, 1988.

[FiMa85] R. Finkel and U. Manber, "DIB – A distributed implementation of backtracking," *Proc. Conference on Distributed Computing,* pp. 446–452, 1985.

[FiMa87] R. Finkel and U. Manber, "DIB – A distributed implementation of backtracking," *ACM Transactions on Programming Languages and Systems,* vol. 9, no. 2, pp. 235–256, 1987.

[Flyn66] M. J. Flynn, "Some computer organizations and their effectiveness," *IEEE Transactions on Computers,,* vol. C-21, no. 9, pp. 948–960, 1966.

[FJLO88] G. Fox, M. Johnson, G. Lyzenga, S. Otto, J. Solomon and D. Walker, *Solving Problems on Concurrent Processors, Volume I: General Techniques and Regular Problems,* Prentice Hall, Englewood Cliffs, N.J., 1988.

[GaJo79] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP–Completeness,* W. H. Freeman and Company, New York, N.Y., 1979.

[GeSS87] E. Gehringer, D. Siewiorek and Z. Segall *Parallel Processing: The* Cm* *Experience,* Digital Press, 1987.

[GeMa72] A. M. Geoffrion and R. E. Marsten, "Integer programming algorithms: A framework and state-of-the-art survey," *Management Science,* vol. 18, no. 9, pp. 465–491, 1972.

[GuMB88] J. Gustafson, G. Montry and R. Benner, "Development of parallel methods for a 1024-processor hypercube," *SIAM Journal on Scientific and Statistical Computing,* vol. 9, no. 4, pp. 1–32, 1988.

[HwBr84] K. Hwang and F. Briggs, *Computer Architecture and Parallel Processing,* McGraw-Hill, New York, 1984.

[Hill85] W. D. Hillis, *The Connection Machine,* MIT Press, Cambridge, Mass., 1985.

[HiSt86] W. D. Hillis and G. Steele, "Data parallel algorithms," *Communications of the ACM,* vol. 29, no. 12, 1986.

[HMSC86] J. P. Hayes, et al., "Architecture of a hypercube supercomputer," *Proc. International Conference on Parallel Processing,* 1986.

[HMSP86] J. P. Hayes, et al., "A microprocessor–based hypercube supercomputer," *IEEE MICRO,* vol. 6, no. 5, pp. 6–17, 1986.

[Ibar76a] T. Ibaraki, "Computational efficiency of approximate branch-and-bound algorithms," *Mathematics of Operations Research,* vol. 1, no. 3, pp. 287–298, 1976.

[Ibar76b] T. Ibaraki, "Theoretical comparisons for search strategies in branch-and-bound algorithms," *International Journal of Computer and Information Sciences,* vol. 5, no. 4, pp. 315–344, 1976.

[Ibar77] T. Ibaraki, "The power of dominance relations in branch-and-bound algorithms," *Journal of the ACM,* vol. 24, no. 2, pp. 264–279, 1977.

[Ibar78a] T. Ibaraki, "Branch-and-bound procedures and state-space representation of combinatorial optimization problems," *Information and Control,* vol. 36, no. 1, pp. 1–27, 1978.

[ImFY79] M. Imai, T. Fukumura and Y. Yoshida, "A parallelized branch and bound algorithm: Implementation and efficiency," *Systems, Computers and Controls,* vol. 10, no. 3, pp. 62–70, 1979.

[InKo77] G. Ingargiola and J. Korsh, "A general algorithm for one dimensional knapsack problems," *Operations Research,* vol. 25, no. 5, pp. 752–759, 1977.

[Inte85] Intel Corporation, *The iPSC Data Sheet,* Beaverton, Ore., 1985.

[Inte86] Intel Corporation, *The iPSC-VX Data Sheet,* Beaverton, Ore., 1986.

[Karp87] A. H. Karp, "Programming for parallelism," *Computer*, vol. 20, no. 5, pp. 43-57, 1987.

[KaNa85] H. Kasahara and S. Narita, "Parallel processing of robot-arm control computations on a multiprocessor system," *IEEE Journal of Robotics and Automation,* vol. RA-1, pp. 104–113, June 1985

[KuRR88] V. Kumar, V. N. Rao and R. Ramesh, "Parallel depth-first search on the ring architecture," AI Lab TR-AI88-68, Computer Science Department, University of Texas at Austin, March 1988.

[KuKa83] V. Kumar and L. Kanal, "A general branch-and-bound formulation for understanding and synthesizing AND/OR tree search procedures," *Artificial Intelligence,* vol. 21, pp. 179–198, 1983.

[LaSa83] T. H. Lai and S. Sahni, "Anomalies in parallel branch-and-bound algorithms," *Proc. International Conference on Parallel Processing,* pp. 183–190, 1983.

[LaSp85] T. H. Lai and A. Sprague, "Performance of parallel branch-and-bound algorithms," *Proc. International Conference on Parallel Processing,* pp. 194–201, 1985.

[LaSp86] T. H. Lai and A. Sprague, "A note on anomalies in parallel branch-and-bound algorithms with one-to-one bounding functions," *Information Processing Letters,* vol. 23, pp. 119–122, 1986.

[LaWo66] E. L. Lawler and D. W. Wood, "Branch-and-bound methods: a survey," *Operations Research,* vol. 14, pp. 699–719, 1966.

[Lens76] J. Lenstra, "Sequencing by enumerative methods," Mathematisch Centrum, Amsterdam, Math. Cen. Tract 69, The Netherlands, 1976.

[Li85] G. J. Li, *Parallel Processing of Combinatorial Search Problems,* PhD Dissertation, Purdue University, 1985.

[LiWa84a] G. J. Li and B. W. Wah, "Computational efficiency of parallel approximate branch-and-bound algorithms," Technical Report TR-EE-84-6, School of Electrical Engineering, Purdue University, March 1984.

[LiWa84b] G. J. Li and B. W. Wah, "Computational efficiency of parallel approximate branch-and-bound algorithms," *Proc. International Conference on Parallel Processing,* pp. 473–480, 1984.

[LiWa86] G. J. Li and B. W. Wah, "Coping with anomalies in parallel branch-and-bound algorithms," *IEEE Transactions on Computers,* vol. C-35, no. 6, pp. 568–573, 1986.

[LMSK63] J.D.C. Little, K. G. Murty, D. W. Sweeney and C. Karel, "An algorithm for the traveling salesman problem," *Operations Research,* vol. 11, no. 6, pp. 972–989, 1963.

[MaSe87] R. T. Marz and W. D. Seward, "Performance evaluation of parallel branch-and-bound search with the Intel iPSC hypercube computer," *Proc. International Conference on Supercomputing and First World Supercomputer Exhibition,* 1987.

[MaTM88] R. Ma, F. Tsung and M. Ma, "A dynamic load balancer for a parallel branch and bound algorithm," *Proc. Third Conference on Hypercube Concurrent Computers and Applications,* pp. 1505–1513, 1988.

[Moha83] J. Mohan, "Experience with two parallel programs solving the traveling salesman problem," *Proc. International Conference on Parallel Processing,* pp. 191–193, 1983.

[NCUB85] NCUBE Corp., *NCUBE Handbook,* version 0.6, Beaverton, Ore., Dec. 1985.

[Nils80] N. J. Nilsson, *Principles of Artificial Intelligence,* Palo Alto, CA: Tioga, 1980.

[PaWo88] R. Paragas and D. Wooster, "Branch and bound algorithms on a hypercube," *Proc. Third Conference on Hypercube Concurrent Computers and Applications,* pp. 1514–1519, 1988.

[Pear84] J. Pearl, *Heuristics: Intelligent Search Strategies for Computer Problem Solving,* Reading, MA: Addison-Wesley, 1984.

[Poly88] C. D. Polychronopoulos, *Parallel Programming and Compilers,* Kluwer Academic Publishers, Norwell, Mass., 1988.

[Pott85] J. L. Potter, *The Massively Parallel Processor,* MIT Press, Cambridge, Mass., 1985.

[PTLP85] J. C. Peterson, J. O. Tuazon, D. Liberman and M. Pniel, "The Mark III hypercube-ensemble concurrent computer," *Proc. International Conference on Parallel Processing,* pp. 71–73, 1985.

[Quin87] M. J. Quinn, *Designing Efficient Algorithms for Parallel Computers,* McGraw-Hill, New York, N.Y, 1987.

[QuDe85] M. J. Quinn and N. Deo, "An upper bound for the speedup of parallel branch-and-bound algorithms," *Proc. of the 3rd Conference on Foundations of Software Technology and Theoretical Computer Science,* Bangalore, India, pp. 488–504, 1985.

[QuDe86] M. J. Quinn and N. Deo, "An upper bound for the speedup of best-first branch-and-bound algorithms," *BIT,* vol. 26, no. 1, pp. 35–43, 1986.

[Quin86] M. J. Quinn, "Implementing best-first branch-and-bound algorithms on hypercube multicomputers," Technical report PCL–86–02, Department of Computer Science, University of New Hampshire, Durham, New Hampshire 03824.

[Rayn88] M. Raynal, *Distributed Algorithms and Protocols,* John Wiley and Sons, New York, N.Y., 1988.

[ScGB88] K. Schwan, J. Gawkowski and B. Blake, "Process and load migration for a parallel branch and bound algorithm on a hypercube multicomputer," *Proc. Third Conference on Hypercube Concurrent Computers and Applications,* pp. 1520–1530, 1988.

[Scho67] P. D. A. Schofield, "Complete solution of the eight puzzle," in *Machine Intelligence 1,* N. Collins and D. Michie, eds., American Elsevier Publ. Co., New York 1967.

[Smit78] B. J. Smith "A pipelined shared resource MIMD computer," *Proc. International Conference on Parallel Processing,* pp. 6–8, 1978.

[Smit84] D. R. Smith "Random trees and the analysis of branch and bound procedures," *Journal of the ACM,* vol. 32, no. 1, 1984.

[Stou86] Q. Stout, "Hypercubes and pyramids," in *Pyramidal Systems for Image Processing and Computer Vision,* V. Cantoni and S. Leviadi, eds., NATO ASI Series ARW, Springer-Verlag, 1986.

[Taha75] H. A. Taha, *Integer Programming: Theory, Applications and Computations,* Academic Press, New York, N.Y., 1975.

[TrWo69] C. A. Trauth and R. E. Woolsey, "Integer programming: a study in Computational efficiency," *Management Science,* vol. 15, no. 9, pp. 481–493, 1969.

[TPPL85] J. O. Tuazon, J. C. Peterson, M. Pniel and D. Liberman, "CALTECH/JPL Mark II hypercube concurrent processor," *Proc. International Conference on Parallel Processing,* pp. 666–671, 1985.

[Trie86] H. Trienekens, "Parallel branch and bound on an MIMD system," Report 8640/A, Econometric Institute, Erasmus University, The Netherlands.

[WaLY84] B. W. Wah, G. J. Li and C. F. Yu, "The status of MANIP–A multicomputer architecture for solving combinatorial extremum-search problems," *Proc. 11th Annual International Symposium on Computer Architecture,* pp. 56–63, 1984.

[WaLY85] B. W. Wah, G. Li and C. F. Yu, "Multiprocessing of combinatorial search problems," *Computer,* vol. 18, no. 6, pp. 93–108, 1985.

[WaMa84] B. W. Wah and Y. W. Eva Ma, "MANIP—A multicomputer architecture for solving combinatorial extremum-search problems," *IEEE Transactions on Computers,* vol. C-33, no. 5, pp. 377–390, 1984.

[WaYu82] B. W. Wah and C. F. Yu, "Probabilistic modeling of branch-and-bound algorithms," *Proc. COMPSAC,* pp. 647–653, 1982.

[WaYu85] B. W. Wah and C. F. Yu, "Stochastic modeling of branch-and-bound algorithms with best-first search," *IEEE Transactions on Software Engineering,* vol. SE-11, no. 9, pp. 922–934, 1985.

[WiKC88] S. Wimer, I. Koren and I. Cederbaum, "Optimal aspect ratios of building blocks in VLSI," *Proc. 25th ACM/IEEE Design Automation Conference,* pp. 66–72, 1988.

[WuCo80] N. Wu and R. Coppins, *Linear Programming and Extensions,* McGraw-Hill, New York, N.Y., pp. 420–426, 1982.

[YuWa83] C. F. Yu and B. W. Wah, "Virtual memory support for branch-and-bound algorithms," *Proc. COMPSAC,* pp. 618–626, 1983.

[YuWa84] C. F. Yu and B. W. Wah, "Efficient branch-and-bound algorithms on a two-level memory system," *Proc. COMPSAC,* pp. 583–593, 1984.

**RULES COVERING USE OF MANUSCRIPT THESES
IN THE UNIVERSITY OF MICHIGAN GRADUATE LIBRARY
AND THE HORACE H. RACKHAM SCHOOL OF GRADUATE STUDIES**

Manuscript copies of theses submitted for the doctoral degree and deposited in The University of Michigan Library and in the Graduate School are open for inspection, but are to be used only with due regard to the rights of the authors. For this reason it is necessary to require that a manuscript thesis be read within the Library or the Graduate School. If the thesis is borrowed by another library, the same rules should be observed. Bibliographical references may be noted, but passages may be copied only with the permission of the authors, and proper credit must be given in subsequent written or published work. Extensive copying or publication of the thesis in whole or in part must have the consent of the author as well as the Dean of the Graduate School.

This thesis by ______________________________ . has been used by the following persons, whose signatures attest their acceptance of the above restrictions.

A Library which borrows this thesis for use by its readers is expected to secure the signature of each user.

NAME AND ADDRESS DATE

6625/0981