PARALLEL PROCESSING OF COMBINATORIAL SEARCH PROBLEMS

A Thesis
Submitted to the Faculty

of

Purdue University

by

Guo-Jie Li

In Partial Fulfillment of the
Requirements for the Degree

of

Doctor of Philosophy

December 1985

This is dedicated
to the memory of my dad

ACKNOWLEDGMENTS

The author would like to express the most sincere gratitude and appreciation to his major Professor Benjamin W. Wah, for his invaluable guidance, support, and inspiration during the entire course of this research. He is also grateful to the other advisory committee members, Professor Kai Hwang, Professor Jose A. B. Fortes, and Professor Susanne Hambrusch, for their guidance and comments. The author expresses his deepest sorrow to the late Professor King-Sun Fu, who served on his doctoral advisory committee.

The author would like to gratefully acknowledge the support of the Purdue Research Foundation through a David Ross Fellowship, the National Science Foundation through grants ECS80-16580, ECS 81-05968, and DMC85-19649, and CIDMAC, a research unit of Purdue University, sponsored by Purdue University, Cincinnati Milicron Corporation, Control Data Corporation, Cummins Engine Company, Ransburg Corporation, and TRW.

TABLE OF CONTENTS

TABLE OF CONTENTS

LIST OF FIGURES

ABSTRACT

Li, Guo-Jie. Ph.D., Purdue University. December 1985. Parallel Processing of Combinatorial Search Problems. Major Professor: Benjamin W. Wah.

The search for solutions in a combinatorially large problem space is a major problem in artificial intelligence and operations research. Parallel processing of combinatorial searches has become a key issue in designing new generation computer systems. The research gives a theoretical foundation of parallel processing of various combinatorial searches upon which the architectures are based. In this thesis parallel processing of searching AND trees (graphs), OR trees (graphs), and AND/OR trees (graphs) are investigated, and different functional requirements of the architecture are identified.

Some of the difficulties in buiding parallel computers for searching arise from the inability to predict the performance of the resulting systems. One important issue in implementing AND-tree searches is to determine the granularity of parallelism. In this thesis, the optimal granularity of AND-tree searches is found and analyzed. Another important result of this research is in finding the bounds of performance of parallel OR-tree searches and a variety of conditions to cope with anomalies of parallel OR-tree searches that involve approximations and dominance tests. In contrast to previous results, our theoretical analysis and simulations show that a near-linear speedup can be achieved with respect to a large number of processors.

Logic programming, one of the foundations of new generation computers, can be represented as searching AND/OR trees. In this research, an optimal search strategy that minimizes the expected overhead of searching AND/OR trees is found. An efficient heuristic search strategy for evaluating logic programs, which can be implemented on a multiprocessor architecture (MANIP-2), is proposed.

Dynamic programming problems, a class of problems that can be formulated in multiple ways and solved by different architectures, are used to illustrate the results obtained on graph and tree searches. Dynamic programming formulations are classified into four types and various parallel processing schemes for implementing different formulations of dynamic programming problems are presented. In particular, efficient systolic arrays for solving monadic-serial dynamic programming problems are developed.

CHAPTER I
INTRODUCTION

Basic concepts of parallel processing of combinatorial searches are introduced in this chapter. We explain the elements of a search algorithm, such as heuristic guiding and pruning, and characterize combinatorial searches as AND/OR-graph searches. Two special cases of AND/OR-graph search, the AND-graph and OR-graph searches, are dealt with separately. The power, limitations, and issues of parallel processing of combinatorial searches, and the corresponding requirements of computer architecture are discussed. To highlight the key ideas of this thesis, the objectives and significance of this research are also included in this chapter.

1.1 The Ubiquity of Search

A wide class of problems arising in artificial intelligence, operations research, decision making, and various scientific and engineering fields are to find one or more, optimal (suboptimal) or feasible solutions in a combinatorial large problem space. The only technique available to solve many problems of this kind is to enumerate some or all elements of the problem space until the solution is found. This universal method is referred to as a *combinatorial search*, or in short, a search. In this section, applications using search algorithms are exemplified and the search strategies are characterized from the viewpoint of implementations by computers.

1.1.1 Problem Solving and Search

In well-developed science and engineering, simple yet quantitatively predictive model have been constructed by using the language of mathematics, e.g., Maxwell's equations for electromagnetism. Solving this kind of problems is to implement deterministic numerical algorithms. However, for many other important problems, the domain specific information is not enough to construct a rigorous mathematical model, and combinatorial search becomes inevitable. We can consider search to imply a complement of knowledge. More knowledge means less search. We will concentrate on problems in artificial intelligence and operations research in which combinatorial search is prevailing.

In artificial intelligence, the terms "problem solving" and "search" refer to the large body of core ideas that deal with deduction, inference, planning, commonsense reasoning, theorem proving, and related processes [BaF81]. Most cognitive tasks can be cast as explorations through a rather enormous search space. General ideas on search have been applied to natural-language understanding, robotics, computer vision, game playing, expert systems, and others [EHL80,Bra84,BaB82,New79,HWL83]. In operations research, combinatorial search is an essential tool of solving a variety of discrete optimization problems.

Generally speaking, combinatorial search problems can be classified into two types. The first type is decision problems or satisfiability problems that decide whether at least one solution exists and satisfies a given set of constraints. No distinction is made between the feasible solutions, and the aim of the search is to reach any of the feasible solutions. Theorem-proving, expert systems, and most problem-solving programs in artificial intelligence belong to this class [SIK75]. The properties of decision problems and the corresponding search strategies are discussed in Chapter 5 of this thesis. The second type is combinatorial extremum-search or optimization problems that are characterized by an objective function to be minimized or maximized and a set of constraints to be satisfied. The aim of the search is to discover a solution bearing the maximum (or minimum) value and

satisfying the constraints. Practical problems such as finding the shortest path, planning, finding the shortest tour of a traveling salesman, warehouse location, job-shop scheduling, packing a knapsack, vertex cover, and integer programming belong to this class.

1.1.2 Elements of a Search Algorithm

Many problems requiring combinatorial search appear intractable, in the sense that every known solution method experiences a combinatorial explosion in its worse-case running time as a function of the input size. For instance, the search space of a chess game can be as large as 10^{120} positions. Exhaustive search is usually impractical and prohibitively expensive for solving large search problems, especially when the problem is NP-hard [GaJ79].

One approach to solving these hard problems is to design specialized efficient algorithms tailored to the characteristics of each problem. Most of the proposed polynomial-time approximation algorithms with guaranteed bounds belong to this type [Gra66,GaJ76,Sah77]. Some researchers have addressed the probabilistic approach that guarantee to obtain optimal or near-optimal solutions for almost all problem instances [Kar76]. Others have studied polynomial-time algorithms that optimize the average performance [Smi84]. Experience has revealed that efficient algorithms for slightly different problems are usually different and cannot be generalized. Moreover, good algorithms are generally difficult to design. This approach is not taken in this thesis.

On the other hand, research has been conducted on designing a few unified methods for a wide variety of problems [KoS74,Iba76b,Sek81]. Among the most general techniques are divide-and-conquer, branch-and-bound (B&B), dynamic programming, and other graph-search strategies such as A^*, AO^*, and SSS^*. All these general approaches involve search algorithms. The search space for these unified methods is often excessive large. Studies on improving the search efficiency is, thus, of considerable importance. There are several complementary approaches to reduce the amount of time and space to find a solution. One important way is to choose an appropriate representation of the problem to be solved, that is, to recast the problem such that the size of the search space is reduced. In artificial intelligence, *state-space* and *problem-reduction* are two major representation schemes.

A state-space representation of a problem employs two kinds of entities: *states*, which are data structure giving "snapshots" of the condition of the problem at each stage of its solution, and *operators*, which are means for transforming the problem from one state into another. The knapsack problem of packing n objects into a knapsack is used to illustrate the state-space representation. Object i has a weight w_i, and the knapsack has a capacity M. If Object i is placed into the knapsack, then a profit p_i is earned. The objective is to fill the knapsack such that the profit is maximized. The corresponding state-space graph is depicted in Figure 1.1. In this graph, each node denotes a state, and branches represent the transformation of states. For example, node B represents the state that object 1 has been placed into the knapsack and object 2 is being considered. Two alternative operations applied to this state are possible: (a) Object 2 is included in the knapsack, and (b) Object 2 is not included. In case (a), the state is transformed into state C, otherwise, it is transformed into state D.

In the problem-reduction approach, the principle data structures are problem descriptions or *goals*. An initial problem is solved by a sequence of transformations that ultimately change it into a set of subproblems whose solutions are immediate. A problem may be transformed into several subproblems, all of which must be solved in order to solve the former. This expansion process can be presented by an AND-node in the graphical representation. In addition, several different operators (transformations) may be applicable to a single problem, and it suffices to solve the subproblem produced by any one of operator applications. This process can be represented as an OR-node in the graphical representation. Problem-reduction can be regarded as searching an AND/OR-graph.

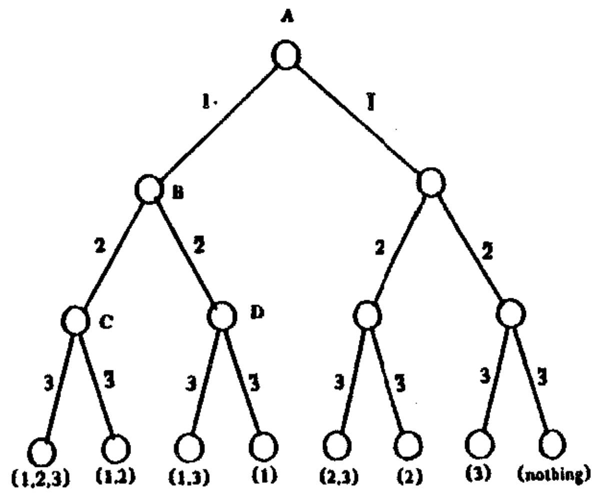


Figure 1.1 The state-space graph for the knapsack problem

Figure 1.2 shows a AND/OR graph in which problem A is transformed into three subproblems B, C, and D and will be solved if any one of subproblems B, C, and D is solved. Problem B is in turn reduced to subproblems E and F, and solving both E and F implies solving B. Note that a state-space graph can be understood as an AND/OR-graph containing only OR-nodes, each state of which corresponds to the problem of getting from this state to a goal state. A goal state of the state-space becomes the primitive problem of getting from the goal state to itself. In this thesis, we adopt the AND/OR-graph as a unified representation scheme and define the AND/OR-graph according to its nonterminal nodes without distinguishing between states and subproblems.

We represent a search problem as an acyclic AND/OR- graph.[1] The representation is characterized by a root node with no edge entering it, and one or more terminal nodes with no edge exiting it. In a search graph, there may be one or more edges entering a node except the root. An edge in a search graph represents an assignment of value to an unassigned parameter. For example, in the 0/1 knapsack problem shown in Figure 1.1, the unassigned parameters are the set of objects that have not been considered.

The non-terminal nodes in a search graph can be classified as AND-nodes and OR-nodes. An *AND-node* represents a (sub)problem that is solved only if all its children have been solved. An example of an AND-node is one that adds the solutions from all subtrees expanded from this node. In Figure 1.2, nodes B and C are AND-nodes. In contrast, an *OR-node* represents a (sub)problem that is solved if any one of its children can be solved.[2] In Figure 1.2, node A is an OR-node. The expansion of a 0/1 knapsack problem by choosing an object to be included or excluded corresponds to a transformation of the problem from one state to another until the goal state is found. In this sense, the resulting tree contains only OR-

[1] An acyclic graph is one without cycles. More generally, some graphs with cycles can be searched. However, this is beyond the scope of this thesis.
[2] In this thesis the definitions of AND-and OR-nodes are taken from [MaM73]. The roles of the AND-and OR-nodes in an AND/OR-tree are reversed in Nilsson's definitions [Nil80]

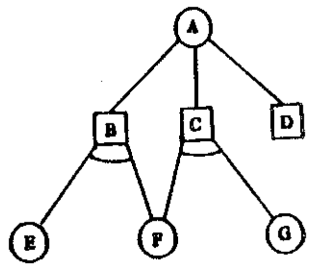


Figure1.2 An AND/OR graph

nodes.

To facilitate the design of multiprocessing systems for solving a search problem, the problem is transformed into one of the following paradigms according to the functions of the nodes:

(i) *AND/OR-graph*: The non-terminal nodes are either AND- or OR-nodes.

(ii) *AND-graph*: This is a special case of an AND/OR-graph in which all non-terminal nodes are AND-nodes. Searching an AND-graphs is implemented deterministically and is quite different from searching an AND/OR-graph. We investigate AND-graph searches first since some results of analysis can be applied to AND/OR-graph searches.

(iii) *OR-graph*: This is another special case of an AND/OR-graph in which all non-terminal nodes are OR-nodes. An OR-graph search is essentially a state-space approach.

A search tree is a special case of a search graph. In a search tree, each node except the root has exactly one edge entering it. In fact, a search graph can be coverted into a search tree by duplicating common nodes. Likewise, a tree-search procedure can be converted into graph-search procedure by modifying the action performed each time a node is generated such that only nodes that have not been created before are added to the graph. Searching trees is easier than searching graphs, and most practical problem-solving procedures are based on tree searches. For example, game playing and logic programs can be represented as AND/OR-trees; *divide-and-conquer* algorithms are AND-tree searches; conventional *branch-and-bound* algorithms are viewed as OR-tree searches.

A problem can be represented in multiple forms. For example, the knapsack problem may either be represented in an OR-tree and solved by a branch-and-bound algorithm [HoS78], or formulated in dynamic programming and solved by an acyclic AND/OR-graph search. As another example, the search of the extrema from a set of numbers can be solved by either a divide-and-conquer (AND-tree) algorithm [AHU74] or a decision-tree (OR-tree) search. In general, the search

procedures for various representations are equivalent in the sense that they generate the same solution(s). Kumar and Kanal have shown that various heuristic search procedures for state-space representations (e.g. A^*, SSS^* [Nil80]), AND/OR-graph searches (e.g. AO^* [Nil80]), and game-tree searches (e.g. α-β [KnM75]) are equivalent to branch-and-bound searches with dominance tests [KuK83].

The efficiency of solving a given problem depends on the representation. Although efficient search procedures for some problems have been established, the general question of deciding the representation that leads to an efficient search is still open for many problems, especially when multiprocessing is concerned. We will study the variant architectures required by multiple problem representations in Chapter 6.

A second way to improve the search efficiency is to guide the search by heuristic information. To avoid combinatorial explosion, it is often necessary to compromise between processing efficiency and optimality of the resulting solutions. A *heuristic search* is a method that utilizes additional information about the properties of the specific problem domain to be solved beyond the information built into the data structures. A node is *active* if its solution value has not been found; otherwise, it is said to be *terminated*. In a serial algorithm, the set of active nodes are maintained in a single list. A heuristic value defined by a heuristic function is computed for each node. The active node with the minimum heuristic value is always expanded first. A search is called *depth-first search* if the negation of the level number is used as the heuristic function. In this case the nodes in the active list are expanded in a last-in /first-out order. A search is called *breadth-first search* if the level number is used as the heuristic function. In this case the nodes in the active list are expanded in a first-in/first-out order. The information built into the data structures is considered as syntactic knowledge, while other problem-domain information is semantic knowledge. Depth-first and breadth-first searches that use syntactic knowledge only are regarded as special cases of heuristic searches.

Heuristic search of logic programming problems is discussed in Chapter 5.

The last way to reduce the search space is to prune unnecessary expansions by elimination rules. The most important elimination rule is a *dominance test*. When a subproblem P_i dominates another subproblem P_j, it implies that the subtree rooted at P_i contains a solution with a value no more (or less) than the minimum (or maximum) solution value of the subtree rooted at P_j. As an example, suppose that the assignment for a subset of the objects to be packed into a knapsack in the 0/1 knapsack problem has been determined. For two assignments, P_1 and P_2, on this subset of objects, if the total profit of the objects assigned to the knapsack for P_1 is more than that of P_2, and the total weight of the objects assigned in P_1 is less than that of P_2, then the best solution expanded from P_1 will be better than the best solution expanded from P_2. In this case, P_1 dominates P_2. A special case of dominance tests is the *lower-bound tests* that are used in branch-and-bound algorithms to solve minimization problems. Suppose that a terminal node with value v has already been found. Then all active nodes with lower bounds greater than v can be terminated since they would not lead to better solutions. The minimum of the solution values obtained at any time can be conveniently kept in a single location called the *incumbent*. The properties and parallel implementation of dominance tests will be discussed in Chapter 3. α-β pruning on AND/OR trees will be discussed in Chapter 5.

1.2 Parallel Processing of Combinatorial Search

Within the past two decades, significant advances have occurred in computer architecture and microelectronics. New ways to construct the architecture of computers enable computations to be processed in parallel, leading to large improvement in machine performance. Due to the huge amount of computations in combinatorial searches, parallelism or ultra-parallelism are required. In this section the issues of parallel processing of combinatorial searches are discussed.

1.2.1 The Power and Limitation of Parallel Search

Search problems can be evaluated in parallel by exploring simultaneously several nodes sprung from the single root node provided that they are independent. The potential for parallelism is usually large because there are a great number of subproblems that are independent of each other, especially for OR-tree searches. In applications in which the computational complexity is exponential, the search algorithms are characterized by relatively large computational overheads as compared to the input-output overheads. In this case, parallelism using Very-Large-Scale-Integrated (VLSI) technology is suitable.

It is important to note that parallel processing is not an approach to circumvent the difficulty of combinatorial explosion. The limitation of parallel processing is an important issue to be considered before designing parallel search schemes. The problem should be *polynomially solvable by a serial computer*. In other words, parallel processing is applicable when the problem is solvable in polynomial time, or when the problem is NP-hard, but is solvable in polynomial time on the average [Smi84] or with probability one, i.e., almost always solvable [Kar76], or the problem is heuristically solvable in polynomial time (such as computer game playing and expert systems). It is not practical to use parallel processing to solve a problem with an exponential complexity on the average because an exponential number of processors must be used to solve the problem in polynomial time.

Multiprocessing is generally used to improve the computational efficiency of solving a given problem, and *not to extend the solvable problem space of the problem.* Suppose the best serial algorithm for the problem has a polynomial time complexity of n^k, where n is the problem size and k is a constant ($\geqslant 1$). Then in the time it takes the serial algorithm to solve the problem of size n, a parallel algorithm with n processors can only solve the problem of size $n^{(1+1/k)}$ assuming a linear speedup. For k=3, this is $n^{1.33}$. Similarly, for a serial algorithm with an exponential time complexity of k^n, $k>1$, the solvable problem size using n processors is $n+\log_k n$.

1.2.2 Issues of Parallel Search

Combinatorial searches are quite different from conventional deterministic numerical computations. Some important issues on parallel searches are listed here.

(a) *Predication of performance*: One of the major difficulties associated with combinatorial searches is the inability to predict either the efficiency of a given algorithm or the performance difference between different approaches without actually executing the search program. This is due to the nondeterminism of combinatorial search algorithms. The estimation of performance of combinatorial searches is, clearly, an important issue, especially when multiple processors are used.

(b) *Linear Scaling*: When a large number of processors are used, it is expected that the speed of computation will increase in direct proportion to the number of computing elements without any reprogramming. This linear scaling rule, usually, does not hold for parallel searches even though the communication time is ignored. Further, it is possible that using more processors might degrade the performance. Attention must be paid to coping with the anomalous behaviors when designing multiprocessor systems for parallel searches.

(c) *Granularity of parallelism*: In parallel searches, determining the minimum size of a subproblem that will be evaluated by a single processor is an important issue. If the granularity is large, then the processors can be loosely coupled; otherwise, tight coupling as in systolic arrays may be necessary. The proper granularity depends on the capacity of the communication network and the complexity of the search problems.

(d) *Parallel selection*: In parallel heuristic searches, multiple subproblems with the smallest heuristic values must be selected from the active list(s). Whether all active subproblems are maintained in a very large heap or subproblems are selected in multiple active lists through a selection network, the selection overhead is high.

Notice that for some problems, selection is necessary to assure the correctness of the solution; whereas in some other cases, the order of evaluation is insignificant. For performing parallel searches, a cost-effective parallel selection strategy is required.

(e) *Communication of pruning information*: When two subtrees (subgraphs) are searched in parallel, more work than necessary might be performed if the pruning information of one processor is unavailable to other processors. The extra work that must be carried out due to a lack of pruning information is called the *information-deficiency overhead*. Pruning information can be exchanged by messages or through a common memory. This increased communication overhead needed for passing this information is called the *information-transfer overhead*. In general, a tradeoff exists between the information-deficiency and information-transfer overheads. If better pruning is obtained by increasing the information-transfer overhead, then the information-deficiency overhead will decrease. A good parallel search algorithm should consider these tradeoffs and reduce the run-time overheads by proper assignments of tasks to processors. This tradeoff is especially critical in parallel dominance tests.

(f) *Interconnection structure*: To take advantage of VLSI technologies, a regular structure of interconnections of the architecture is desired. The mapping of a regular tree or graph onto regular structures such as systolic arrays is straightforward, but an irregular tree or graph might have to be transformed into a regular one before an efficient implementation can be found. The transformation of an irregular tree or graph into a regular one is an interesting research topic.

1.2.3 Computer Architectural Requirements

Efficient architectures to evaluate various search algorithms are different. It is difficult to map search algorithms to general purpose architectures, since they have different architectural requirements. One of our objectives in this thesis is, therefore, *to obtain the functional requirements of various search algorithms*. Some of the general functional requirements of the necessary computer architecture is described in this section. In subsequent chapters, we will investigate the problem-dependent functional requirement.

Combinatorial search is extensively applied in artificial intelligence problems that require symbolic processing operations such as comparison, selection, sorting, matching, and logic set operations. In a higher level, these applications may require the processing of nonnumerical data such as sentences, speech, graphical data, and images. Efficient computers designed for these applications should possess hardware for symbolic processing functions.

Most combinatorial search algorithms are nondeterministic, that is, it is impossible to plan in advance the procedures to execute and to terminate with the available information. Therefore, dynamic allocation of computational resources is essential in the architectures. Further, an efficient broadcast bus or interconnection network is needed to disseminate information for the scheduler. Owing to this nondeterminism, the detection of dependencies that control the level of parallelism is more difficult for combinatorial searches. Load balancing is also essential because there may be a great disparity in the workload of different processors. Moreover, efficient garbage collection is required because of the dynamically allocated storage.

Another important functional requirement for a parallel search system is a large degree of parallelism and a large memory space due to the huge amount of data and computations. To prevent the bottleneck of a centralized controller, intelligence in such a system should be decentralized. To guide the search by heuristics and to prune unnecessary expansions, local information is often inadequate, and the architecture should be capable of distributing global information efficiently. A secondary storage is indispensable in such a system. Due to difference in locality of access of search algorithms, the staging of data between the main and secondary storage is an important problem to consider [YuW84].

The architecture should also support changes in granularity and data structures when the problem size is changed.

1.3 Objectives of This Thesis

This primary purpose of this thesis is *to give the theoretical foundation of parallel processing of various combinatorial searches upon which the architecture is based*. The emphasis of this thesis is in analyzing the computational efficiency and in showing the reasonableness of the proposed parallel search scheme. That is, we are interested to find the functional requirement of various combinatorial search problems and not to describe the details of the underlying architecture.

Since efficient architectures to evaluate various search algorithms differ, the functional requirements are studied with respect to the paradigm of classifying search problems, namely, AND-tree (graph), OR-tree (graph) and AND/OR-tree (graph). The results derived for one search strategy may shed light on understanding other search strategies. Based on these requirements, a general-purpose architecture can be assessed as whether it is suitable for a given search algorithm, and the most efficient way of mapping the algorithm can be developed. Special-purpose architectures can also be developed from the functional requirements. In what follows, we list the objectives of this thesis with respect to different search paradigms.

For a parallel AND-tree search, we will find the optimal granularity of parallelism, that is, determine the number of processors that achieves the optimal processor utilization or the maximal computational efficiency. The optimal granularity is problem-dependent when an OR-tree or AND/OR graph is searched due to the nondeterminism in the problem.

For an OR-tree (graph) search, we concentrate on understanding the impacts of various problem parameters of search on the computational efficiency and developing a theory for predicating the performance. This is done by deriving the bounds of computational time to perform parallel best-first, depth-first, and breadth-first searches, respectively. Conditions for coping with anomalies of parallel search, and the appropriate parallel search strategy under which a near-linear speedup will hold for a considerable number of processor will be determined.

Satisfiability problems such as evaluating a logic program is generally solved by depth-first search. In this thesis, we will prove an optimal AND/OR-tree search strategy in the sense of the minimizing average search effort and provide an efficient parallel best-first AND/OR-tree search heuristic for solving logic programs.

Finally, to show the dependence between architecture and problem representation, we will model dynamic programming problems as searching an AND-tree, folded OR-tree, folded AND/OR-tree, and AND/OR-graph. Various parallel schemes according to the problem formulation will be proposed. The method for transforming irregular AND/OR-graphs into regular ones will also be discussed.

We do not attempt to enumerate all possible cases in this thesis, but will illustrate the different approaches generously with examples. It is hoped that these guidelines and examples can aid the designers to select the appropriate parallel processing system in solving combinatorial search problems.

1.4 Significance of This Research

Combinatorial search is a fundamental of artificial intelligence and operations research. After a good heuristic algorithm is developed, searching is inevitable, and parallel processing of combinatorial search becomes a key issue in designing new generation computer systems. A lot of interesting projects are in progress today on designing highly parallel architecture for combinatorial searches. Examples include the Parallel Inference Engine (PIE) in the core of the Japanese Fifth Generation Computer Systems project [GTM84], DADO designed at Columbia University [StM84], and ALICE developed at the Imperial College [CFR85]. However, results in this direction are premature and cannot be termed successful.

Some of the difficulties in bulding parallel search systems arise from an inability to predict the performance a priori. Before a design decision is made, a set of questions should be answered on the potential parallelism, the ideal problem representation, the optimal search strategy, the number of processors, the optimal granularity, and the predicted speedup. It is more important to initially understand the applicability of the architecture to a class of problems than to strive for high performance in a prototype implementation. Thus, to know that a 100-processor system gives a 50-fold increase in performance over a single processor is more important than knowing the maximum instruction rate with a prototype. The computer architects should be guided by well tested theory before implementing a prototype. To build a 100-processor system to test the theory is not as important as to predict the performance (accurately) with any number of processors.

We advocate the theoretical approach in this thesis. Although we do not try to undermine the importance of prototype implementation, some simplifying assumptions may have been made in the analysis, and the results have to be verified in a realistic prototype. The significance of this thesis lies in the theoretical foundations of parallel processing of combinatorial searches. Near-linear speedup can be guaranteed with a considerable number of processors as long as certain conditions are met. The conditions discovered in this thesis to cope with anomalies will assure a continuous increase in the performance when more processors are used. Moreover, the functional requirements derived for different kinds of problem formulations will help designers to map search algorithms onto general-purpose computer systems or to design new computer architectures for parallel searches.

In contrast to Japanese Fifth Generation Computer System in which the depth-first search is used, we emphasize on the use of heuristic information to reduce the search complexity. A optimal search strategy for evaluating an AND/OR tree is proved and a practical heuristic search method for solving logic programs is proposed in this thesis. This approach is interesting because it integrates the research results in both artificial intelligence and parallel computer architecture. One of the future directions to lead to better search algorithms lies in the use of domain knowledge to cut down unnecessary enumerations. We hope that this research will contribute to the design of new generation computer system.

CHAPTER II
PARALLEL PROCESSING OF AND-TREE SEARCHES

As discussed in Chapter One, an AND-graph is a special case of an AND/OR-graph in which all nonterminal nodes are AND-nodes. Searching an AND-graph is deterministic, and the search space is fixed. In many practical problems, such as finding the maximum (or minimum), merge-sort, and quiksort, a (sub)problem is divided into smaller *distinct* subproblems. This kind of problems can be represented as AND-tree searches. An AND-graph can be converted into an equivalent AND-tree by duplicating shared child nodes. In what follows, we will only discuss parallel AND-tree searches.

2.1 Previous Work on Parallel Divide-and-Conquer Algorithms

A well-known AND-tree search strategy to solve many combinatorial search problems is the divide-and-conquer algorithm. It partitions a problem into smaller distinct subproblems, finds solutions for the subproblems, and combines the solutions into a solution for the original problem. The procedure is applied recursively until the subproblems are so small that they can be solved directly.

Divide-and-conquer algorithms naturally suggest implementation on parallel machines due to the independence of subproblems. Studies conducted on parallel computers for executing divide-and-conquer algorithms can be classified into three types. First, multiprocessors that are connected in the form of a tree, especially a binary tree, can be used to exploit the potential parallelism of divide-and-conquer algorithms [HaS79,Bro80,Pet81]. A tree machine has a simple, planar wiring scheme that is suitable for VLSI implementation [BeK79]. It also permits fast access to any processor in the structure and has a constant external connection function. The mapping of a divide-and-conquer algorithm on this architecture is easy due to the resemblance between the interconnection and the logical structure of the algorithm. However, to arrange the processors into a tree physically has certain disadvantages. For instance, the root processor is often a bottleneck for problems such as merge sorting, and the fixed tree structure is not flexible enough.

A second approach is the *virtual tree machine* [BuH84]. This consists of a number of processors with private memory connected by an interconnection network, such as the binary n-cube, and a suitable algorithm to decide when and where each subproblem should be solved. The hierarchy of process communications in divide-and-conquer algorithms allows them to be mapped easily onto this architecture. This approach may circumvent the drawback of the physical tree network by allowing each processor to execute programs originally designated to a number of different processors. Care should be taken to keep the communication times within reasonable limits. Martin has described how an arbitrary binary tree of computations may be mapped onto a finite network of machines [Mar79].

The third approach is a variation of the above approaches that uses a common memory. All processors are connected to the memory by a common bus [HoZ83]. Since data must be shared in the execution, the memory or bus may become a bottleneck. Multi-module memory augmented by caches has been proposed to reduce the memory and bus contentions.

Divide-and-conquer algorithms are common in programming. A top-down synthesis of divide-and-conquer algorithms has been presented for the derivation of algorithms from a formal specification of a problem and has been implemented in a program-synthesis system called CYPRESS [Smi85].

2.2 Model of Parallel AND-Tree Searches

To analyze the optimal granularity of parallel AND-tree searches, a synchronous model for parallel computation is described below. An AND-tree search can be viewed as a process with two phases, the top-down decomposition of (sub)problem and the bottom-up composition of resulted subproblems. In practice, these phases may be interleaved but it helps to analyze them separately. We now show that the assumption of separating the two phases is reasonable.

Figure 2.1 shows an example of an AND-tree search. Suppose that the decomposition and composition phases are interleaved, for instance, the sequence of evaluation is T_1, T_1', T_2, and T_2', and the corresponding times needed are t_1, t_1', t_2, and t_2'. Then the total time from splitting nodes B and C to returning the results to nodes B' and C' is the sum of t_i and t_i', $i = 1, 2$, because all these computations are independent. Without loss of computational efficiency, we can reorder the computation in the sequence of T_1, T_2, T_1', and T_2. This means that in the analysis we can assume that the composition phase begins until the problem to be solved is fully decomposed, i.e., all terminals of the AND-tree are available. Further, the composition and decomposition are dual processes. The results of the efficiency analysis for one phase can be extended to another phase easily. Therefore, we will analyze the optimal granularity of parallel AND-tree searches only for the composition phase in subsequent discussion.

In the composition phase, a node or(or subproblem) is said to be *active* if the result to this subproblem has been obtained but its parent subproblem has not been solved yet. A node is said to be *executable* if all its children are active. After a computational step, some (or all) executable nodes will become active nodes. If one or more sibling nodes of an active node is not active, then they cannot be combined until all the sibling nodes are active. In this case, this node is called a *waited-active* node. As shown in Figure 2.2, in Step 3 of an AND-tree search, nodes B, D, and F are executable nodes, all shaded node are active nodes, and node E is a waited-active node. In Step 4, only one node, C, is executable and Nodes B and D become

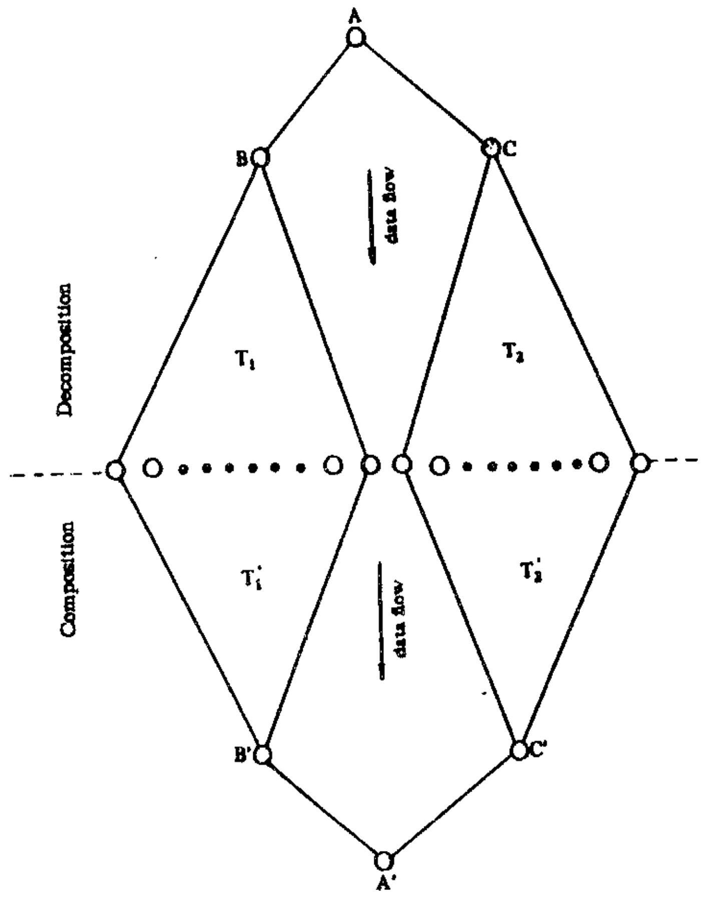


Figure 2.1 Decomposition and composition phases in an AND-tree evaluation

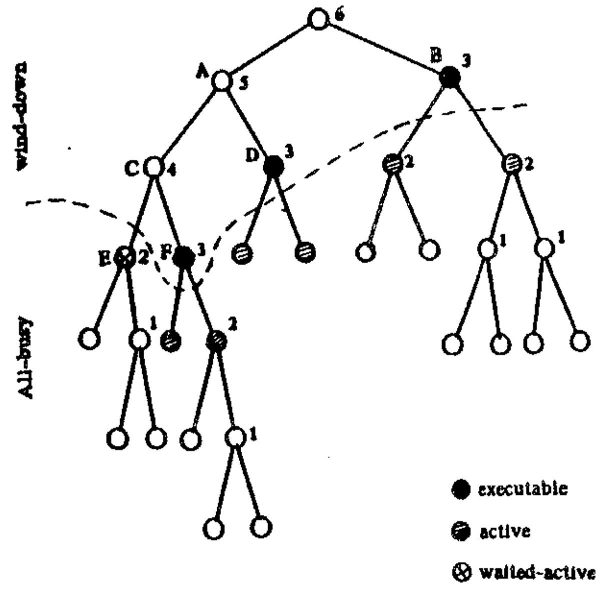


Figure 2.2 All-busy and wind-down subphases in the composition phase of an AND-tree evaluation

waited-active nodes because nodes A and C are inactive.

In parallel AND-tree searches, a composition phase can be divided into two subphases: *all-busy* and *wind-down*. During the all-busy period, all the processors are kept busy until the number of executable nodes is less than the number of processors. In the wind-down period, the results are combined together, and some processors would be idle because of insufficient number of executable nodes. For example, if four processors are used to evaluate the binary AND-tree in Figure 2.2, one of the possible boundary between the all-busy and wind-down phases is indicated by the dashed line.

The following proposition shows an important property of AND-tree searches with respect to the executable nodes.

Proposition 2.1: During the composition phase of searching an AND-tree, the number of executable nodes decreases monotonously.

Proof: Suppose that there are n active nodes and p executable nodes in Step i. This means that among the n active nodes, m nodes ($m \leq n$) are immediate descendants of the p executable nodes. The other (n-m) nodes are waited-active nodes. If the number of processors used is larger than or equal to p, then all executable nodes can be fired and become active; otherwise, some executable nodes remain. In the next step, any new executable node must have at least one immediate descendant that was an executable node and evaluated in the last step. A node whose immediate descendants were all waited-active nodes in the last step cannot become executable in this step; otherwise it would have been executable in the last step. This implies that this node is not a "new" executable node, a contradiction! The number of executable node in Step (i+1), as a result, cannot be greater than p.

□

If AND-trees to be searched are complete or the time needed to evaluate each node is a constant and the completion time is to be minimized, then nodes no less than the number of processors used should be evaluated as long as the number of executable nodes are larger than or equal to the number of processors. This means

that the all-busy phase can follow the decomposition phase if the number of terminals of the AND-tree is not less than the number of processors. According to Proposition 2.1, once the wind-down phase begins, it cannot become the all-busy phase again. That is, the two phases are separated. In the following section, we will discuss the optimal granularity to maximize processor utilization or processor-time efficiency with respect to the problems for which the separation of the all-busy and wind-down phases are valid. For general AND-tree searches, dynamic scheduling is needed to assign subproblems to processors, and the optimal granularity is problem-dependent.

2.3 Optimal Granularity of Parallel AND-Tree Searches

The functional requirement for evaluating an AND-tree is an interconnected conglomerate of processors. One important issue is to determine the *granularity of parallelism*. This is the minimum size of a subproblem that will be evaluated by a single processor to achieve the optimal performance. The criteria generally used are the processor utilization, kT^2, or AT^2, where k is the number of processors, T is the computational time, and A is the area of a VLSI implementation.

2.3.1 Optimal Processor Utilization

The complexity of divide-and-conquer algorithms in an SIMD model and the conditions that assure the optimal processor utilization have been studied [HoZ83,TaL83]. We now investigate the asymptotic utilization.

In the composition phase, the processor utilization depends on the ratio of the amount of times spent in the all-busy and wind-down phases. The time complexity of searching a binary AND-tree of N leaves, T(N), can be formulated in the following recursive equation:

$$T(N) = \begin{cases} D(N) + 2\,T\left\lceil \frac{N}{2} \right\rceil + C(N) & N>1 \\ \Theta(1) & N=1 \end{cases} \qquad (2.1)$$

where D(N) and C(N) are, respectively, the complexities of the decomposition and composition computations and Θ indicates the set of functions of the same order. In fact, C(N) reflects the time needed to solve the problem of size N when the results of subproblems of size N/2 are obtained. The granularity that results in the optimal processor utilization is related to the complexity of D(N) and C(N). In the problem of finding the sum or the maximum of N numbers, $D(N)+C(N) = \Theta(1)$, and using $N/(\log_2 N)$ processors will achieve the maximum processor utilization [Kuc77]. In sorting N numbers, $D(N)+C(N) = \Theta(N)$, and $\log_2 N$ processors should be used to maximize the processor utilization. Note that the higher the orders of complexity of D(N) and C(N) are, the larger the granularity is. When the orders of complexity of D(N) and C(N) are high, the times spent in the decomposition and wind-down phases are dominating the time in the all-busy phase, and the gain in the all-busy phase with a small granularity is negligible. In other words, a small granularity will result in under-utilization of processors.

The following proposition shows the asymptotic utilization, PU, when different number of processors are used in parallel AND-tree search algorithms.

Proposition 2.2: Let PU(k,N) be the processor utilization in the composition phase of performing a parallel binary AND-tree search algorithm with k processors on an AND tree of height $\left\lceil \log_2 N \right\rceil$ and N terminal nodes. Assuming that each processor evaluates any nonterminal node in the binary AND-tree in a constant time of $\Theta(1)$. By defining $c_\infty = \lim_{N\to\infty} \frac{k(N)}{N/\log_2 N}$, the asymptotic processor utilization is

$$\lim_{N\to\infty} PU(k,N) \begin{cases} = 0 & \text{if } c_\infty \to \infty \\ = \frac{1}{1+c_\infty} & \text{if } 0 < c_\infty < \infty \\ = 1 & \text{if } c_\infty = 0 \end{cases} \qquad (2.2)$$

Proof: The number of nodes evaluated in the first iteration of the wind-down phase is at least k/2 and at most k-1. Hence, the total number of nodes evaluated in the

wind-down phase is at least k-1 and at most 2k-3. As the total number of nonterminal nodes is N-1, the total number of the nonterminal nodes evaluated in the computation phase is at least (N-1)-(2k-3) = N+2-2k, and at most (N-1)-(k-1) = N-k. I_a, the number of iterations in the computation phase, is bounded by:

$$\frac{N+2}{k} - 2 \leqslant I_a \leqslant N/k - 1 \qquad (2.3)$$

According to the bound of tree-height reduction, I_w, the number of iterations in the wind-down phase, is bounded by:

$$\log_2(2k-3) \leqslant I_w \leqslant \log_2(k-1) \qquad (2.4)$$

By the definition of PU, we have

$$\lim_{N\to\infty} PU(k,N) = \lim_{N\to\infty} \frac{N-1}{k\,(I_a+I_w)} \qquad (2.5)$$

We now discuss three separate cases.

(a) $c_\infty \to \infty$: From Eq's (2.3)-(2.5), we get

$$\lim_{N\to\infty} PU(k,N) \leqslant \lim_{N\to\infty} \frac{N-1}{k\left[\frac{N+2}{k} - 2 + \log_2(2k-3)\right]} \qquad (2.6)$$

$$\leqslant \lim_{N\to\infty} \frac{N-1}{N + k(\log_2(2k-3)-2)} \leqslant \frac{1}{1 + c_\infty \lim_{N\to\infty} \frac{\log_2((2k-3)-2)}{\log_2 N}}$$

Since $c_\infty \to \infty$, we can assume that $k > N/\log_2 N$ for $N > N_0$ and $\log_2((2k-3)-2) > \log_2 k > \log_2 N - \log_2\log_2 N$. This means that $\lim_{N\to\infty} \frac{\log_2((2k-3)-2)}{\log_2 N} > 1$ implying that $PU(k,N) \leqslant 0$. As PU(k,N) must be nonnegative, the first case of Eq. (2.2) is proved.

(b) $c_\infty = 0$: From Eq's (2.3)-(2.5), we get

$$\lim_{N\to\infty} PU(k,N) \geqslant \lim_{N\to\infty} \frac{N-1}{k\left[\frac{N}{k} - 1 + \log_2(k-1)\right]} \qquad (2.7)$$

$$\geqslant \lim_{N\to\infty} \frac{N-1}{N + k(\log_2(k-1) - 1)}$$

Since $c_\infty \to \infty$ we can assume that $k < N/\log_2 N$ for $N > N_0$, and $\log_2((k-1)-1) < \log_2 k < \log_2 N - \log_2\log_2 N$. This means that $\lim_{N\to\infty} \frac{\log_2((k-1)-1)}{\log_2 N} < 1$ implying that $PU(k,N) \geqslant 1$. As $PU(k,N) \leqslant 1$, the third case of Eq. (2.2) is proved.

(c) $0 < c_\infty < \infty$: From the definition of c_∞, we have $\lim_{N\to\infty} k = \lim_{N\to\infty} \frac{c_\infty N}{\log_2 N}$ and

$$\lim_{N\to\infty} \frac{\log_2 k}{\log_2 N} = \lim_{N\to\infty} \frac{\log_2 c_\infty + \log_2 N - \log_2\log_2 N}{\log_2 N} = 1 \qquad (2.8)$$

From Eq's (2.6)-(2.8), it is easy to see that

$$\frac{1}{1+c_\infty} \leqslant \frac{1}{1 + c_\infty \lim_{N\to\infty} \frac{\log_2((k-1)-1)}{\log_2 N}} \leqslant PU(k,N) \qquad (2.9)$$

$$\leqslant \frac{1}{1 + c_\infty \lim_{N\to\infty} \frac{\log_2((2k-3)-2)}{\log_2 N}} \leqslant \frac{1}{1+c_\infty}$$

This proves the second case of Eq. (2.2). □

As an example in applying Eq. (2.2), suppose that there are $\sqrt{N}$ processors. It is easy to show that $c_\infty = \lim_{N\to\infty} \frac{\sqrt{N}*\log_2 N}{N} = 0$. Hence $PU(k,N) = 1$. Note that the above proposition holds in the case in which each node is evaluated in $\Theta(1)$ time, as the constant complexity does not affect the limits in Eq's (2.6) and (2.7).

2.3.2 Optimal Processor-Time Efficiency

Since PU increases monotonically with decreasing number of processors, it is not an adequate measure for the effects of parallel processing. Another appropriate measure is the kT^2 criterion, which considers both PU and computational time. The following theorem proves the lower-bound kT^2 complexity of parallel binary AND-tree-search algorithms. This lower bound is attained when $k(N)$ is $\Theta(N/(\log_2 N))$ and the time required to evaluate any node is $\Theta(1)$, i.e., $C(N)$ has a complexity of $\Theta(1)$.

Theorem 2.1: Suppose that in the composition phase of a parallel binary AND-tree search, N inputs (terminals) are combined by $k(N)$ processors in time $T(N)$ and that each processor evaluates any node in the binary AND-tree in a constant T_0 time units. Then $k(N)T^2(N) \geqslant \Theta(N \cdot \log_2 N)T_0^2$, and equality holds when $k(N) = \Theta(N/\log_2 N)$.

Proof: Since inputs are represented by the terminal nodes, and there are N terminal nodes in the binary AND-tree, the number of computational steps or the number of nonterminals is $N-1$. According to our model, in the composition phase, the all-busy and wind-down phases are separated. During the all-busy phase, all processors are kept busy until the executable nodes are less than the number of processors. In the all-busy phase, there are $(N-1)-(k(N)-1) = N-k(N)$ nodes to be evaluated, and at least $(N/k(N) - 1) \cdot T_0$ time units are needed. In the wind-down phase, some processors would be idle. According to the data dependence, at least $\log_2 k(N) \cdot T_0$ time units are required in this phase. Let $T_k(N)$ be the time needed in the composition phase of evaluating an AND-tree with N terminal nodes by k processors. Therefore, the following lower bound of time complexity holds.

$$T_k(N) \geqslant \left[\frac{N}{k(N)} - 1 + \log_2 k(N)\right] \cdot T_0 \qquad (2.10)$$

where $1 \leqslant k(N) \leqslant N$. For simplicity, the constant term in Eq. (2.10) can be ignored without affecting the validity of the following proof. The kT^2 lower bound is derived as

$$k(N)T_k^2(N) \geq \left[\frac{N^2}{k(N)} + 2N\cdot\log_2 k(N) + k(N)\cdot\log_2^2 k(N)\right]\cdot T_0^2 \qquad (2.11)$$

To find the order-of-magnitude minimum of Eq. (2.11), it is necessary to compare the following three cases. When $k(N) = \Theta(N/\log_2 N)$, $k(N)T^2(N) = \Theta((N\cdot\log_2 N)T_0^2)$. In contrast, when $k(N) < \Theta(N/\log_2 N)$, the first term on the right-hand side of Eq. (2.11) is

$$\frac{N^2}{k(N)} > \Theta(N\cdot\log_2 N) \qquad (2.12)$$

When $k(N) > \Theta(N/\log_2 N)$, the third term on the right-hand side of Eq. (2.11) is

$$k(N)\log_2^2 k(N) > \Theta(N\cdot\log_2 N) \qquad (2.13)$$

since $\log_2^2 k(N) \geq \Theta(\log_2^2 N)$. The above analysis shows that the kT^2 complexity is $\Omega((N\cdot\log_2 N)T_0^2)$, and that $\Theta(N/\log_2 N)$ is the optimal granularity to achieve this lower bound. □

To investigate the relationship between k and kT^2, the exact time required to search a binary AND-tree using k processors is derived here. The total time required is

$$T(N) = \left\lceil\frac{N-1}{k}\right\rceil\cdot T_0 + \left\lceil\log_2\left\lceil N + k - 1 - k\cdot\left\lceil\frac{N-1}{k}\right\rceil\right\rceil\right\rceil\cdot T_0 \qquad (2.14)$$

The numerical evaluations of Eq. (2.14) for N=4096 is shown in Figure 2.3, in which kT^2 is minimum when 431 or 465 processors are used. The simulation results for different values of N verify that the optimal granularity is close to $N/\log_2 N$.

In general, C(N), the complexity of merging two subproblems, may be higher then $\Theta(1)$. In this case, the time needed to evaluate a node depends on the level in which the node is located. A complicated scheduling strategy, therefore, is required for an asynchronous computer system. Moreover, the all-busy and wind-down

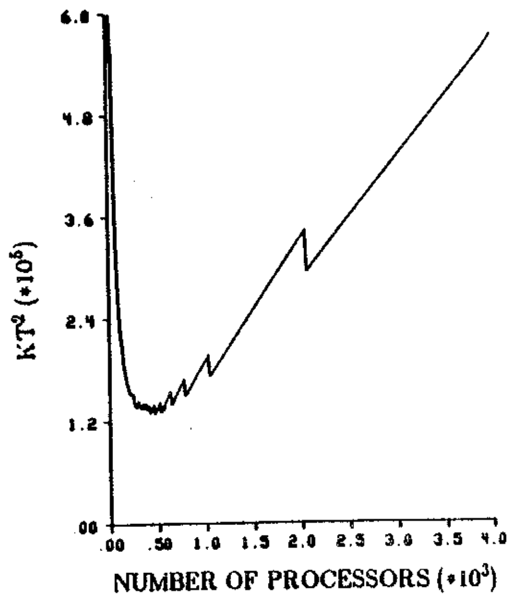


Figure 2.3 Simulation results of finding the optimal granularity of parallel divide-and-conquer algorithms (N = 4096)

phases might be interleaved. For simplicity, in the following theorem and the subsequent efficiency analysis of AND-tree searches, we assume that a synchronous model is adopted and the AND-tree to be searched is complete.

From the property of minmax computations, suppose that $x(k)$ and $y(k)$ are, respectively, monotonically increasing and decreasing functions of k, then the minimum of the function $\max(x(k), y(k))$ will be achieved when k equals a value such that $x(k) = y(k)$. This will be used used as a basis in proving the following theorem.

Theorem 2.2: For searching a complete binary AND-tree of N terminals using k processors, the order of magnitude of $kT_k^2(N)$ achieves the minimum if $\Theta(T_a(N,k)) = \Theta(T_w(N,k))$, where N and k are powers of two.

Proof: Let $C(N)$ be time need to merge two subproblems of size N/2 in a complete binary AND-tree. In view of the fact that N and k are powers of two, $T_a(N,k)$ and $T_w(N,k)$ can be derived as follows. As shown in Figure 2.4, if merging N terminal nodes needs $(C(2)\cdot N/2)/k$ units of time, then $(C(4)\cdot N/4)/k$ units of time are required to merge the intermediate results obtained. The last iteration in the all-busy subphase takes $C(N/k)$ units of time since there are only k nodes in this level, and each subproblem (node) has a size of N/k. Therefore,

$$T_a(N,k) = \frac{N}{k} \sum_{i=1}^{\log_2 \frac{N}{k}} \frac{C(2^i)}{2^i} \tag{2.15}$$

It is clear from Eq. (2.15) that $T_k(N)$ is monotonically decreasing with respect to k as both N/k and the number of terms in the summation are decreased when k is increased. To account for the time required in the wind-down subphase, we consider the time in each iteration from Level $(\log_2 k-1)$ to Level 0. The

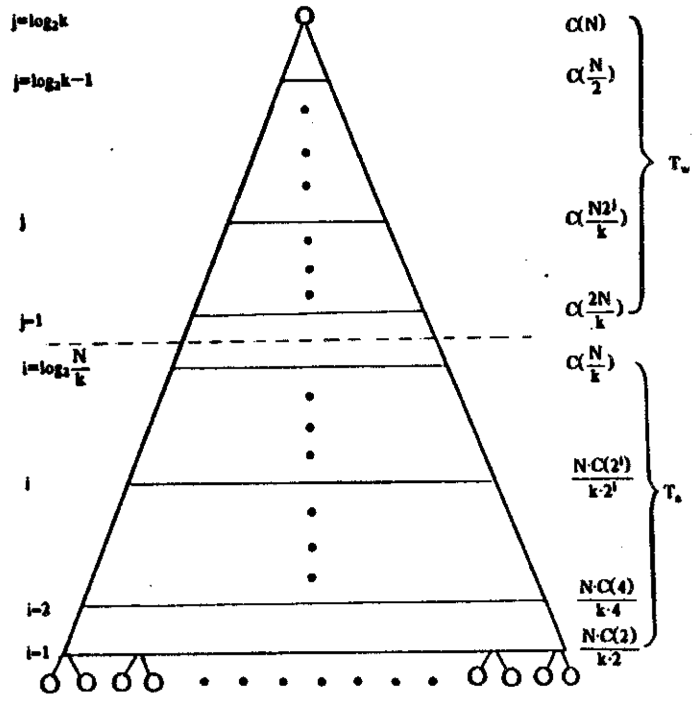


Figure 2.4 The proof of Theorem 2.2

corresponding times are shown in Figure 2.4.

$$T_w(N,k) = \sum_{i=1}^{\log_2 k} C\left(\frac{N \cdot 2^i}{k}\right) \qquad (2.16)$$

To see that $T_w(N,k)$ is monotonically increasing with respect to increasing k, Eq. (2.16) can be converted into the following equation:

$$T_w(N,k) = \sum_{i=1}^{\log_2 k} C\left(\frac{N}{2^{i-1}}\right) \qquad (2.17)$$

In Eq. (2.17), the times are accounted from the root to Level ($\log_2 k - 1$). When k is increased, so is the number of terms in the summation of Eq. (2.17), and consequently, $T_w(N,k)$ is increased. Note that

$$\Theta(T_k^2(N)) = \Theta((T_s(N,k) + T_w(N,k))^2) = \max(\Theta(T_s^2(N,k)), \Theta(T_w^2(N,k)))$$

and

$$\min(\Theta(kT_k^2(N))) = \min(\max(\Theta(kT_s^2(N,k)), \Theta(kT_w^2(N,k)))) \qquad (2.18)$$

It is easy to see that $kT_w^2(N,k)$ is increasing monotonically with respect to k. To show that $kT_s^2(N,k)$ is monotonically decreasing with respect to k, we compare $kT_s^2(N,k)$ with $k'T_s^2(N,k')$, where k' is an integer and $k' > k$.

$$kT_s^2(N,k) - k'T_s^2(N,k') = k\left(\frac{N}{k}\sum_{i=1}^{\log_2 \frac{N}{k}} \frac{C(2^i)}{2^i}\right)^2 - k'\left(\frac{N}{k'}\sum_{i=1}^{\log_2 \frac{N}{k'}} \frac{C(2^i)}{2^i}\right)^2$$

$$= \frac{N^2}{k}\left(\sum_{i=1}^{\log_2 \frac{N}{k}} \frac{C(2^i)}{2^i}\right)^2 - \frac{N^2}{k'}\left(\sum_{i=1}^{\log_2 \frac{N}{k'}} \frac{C(2^i)}{2^i}\right)^2 > 0$$

In terms of the property of minmax value of two monotonic functions, the minimum of $\Theta(kT_k^2(N,k))$ will be achieved when $\Theta(T_s(N,k)) = \Theta(T_w(N,k))$. □

According to Theorem 2.2, the optimal granularities with respect to various C(N) can be found. This will be illustrated in the following corollaries.

Corollary 2.1: For searching a complete binary AND-tree of N terminals, in case

when $C(N) = \Theta(\log_2^s N)$, $s \geq 0$, the optimal granularity for the composition phase is $\Theta\left(\frac{N}{\log_2^{s+1} N}\right)$.

Proof: Let $C(N) = c \log_2^s N$, where c is a constant. From Eq. (2.15) we obtain

$$T_s(N) = \frac{c \cdot N}{k}\sum_{i=1}^{\log_2 \frac{N}{k}} \frac{i}{2^i} \qquad (2.19)$$

$$\frac{1}{2}T_s(N) = \frac{c \cdot N}{k}\sum_{i=1}^{\log_2 \frac{N}{k}} \frac{i}{2^{i+1}} \qquad (2.20)$$

Subtracting Eq. (2.20) from Eq. (2.19) results in

$$\frac{1}{2}T_s(N) = \frac{c \cdot N}{k}\left(\left(\sum_{i=1}^{\log_2 \frac{N}{k}} \frac{1}{2^i}\right) - \frac{k(\log_2 N - \log_2 k)}{2N}\right)$$

$$T_s(N) = c\left(\frac{2N}{k} - 2 - \log_2 N + \log_2 k\right)$$

In Theorem 2.1, we have shown that the optimal granularity is $\Theta(N/\log N)$ if $C(N) = \Theta(1)$. In case when $C(N) = \Theta(\log_2^s N)$, the optimal granularity, $\Theta(k)$, should be less than $\Theta(N)$. Hence,

$$\Theta(T_s(N)) = \Theta\left(\frac{N}{k}\right)$$

From Eq. (2.16) and $C\left(\frac{N\,2^i}{k}\right) = C\left(\log_2 \frac{N}{k} + i\right)^s$, we get

$$T_w(N) = c \cdot \sum_{i=1}^{\log_2 k}\left(\log_2 \frac{N}{k} + i\right)^s$$

$$\Theta(T_w(N)) = \Theta\left(\sum_{i=1}^{\log_2 k}\left(\sum_{j=0}^{s} \log_2^{s-j}\left(\frac{N}{k}\right) i^j\right)\right) = \Theta\left(\sum_{i=1}^{\log_2 k} \log_2^s \frac{N}{k}\right) + \Theta\left(\sum_{i=1}^{\log_2 k} i^s\right)$$

From a result derived by Knuth [Knu73],

$$\Theta(T_w(N)) = \Theta\left(\log_2^s \frac{N}{k} \cdot \log_2 k\right) + \Theta(\log_2^{s+1} k)$$

From Theorem 2.2, we can conclude that when $C(N) = \Theta(\log_2^s N)$, $s \geq 0$, $kT_k^2(N)$ achieves the minimum when

$$\Theta\left(\frac{N}{k}\right) = \Theta\left(\log_2^s \frac{N}{k} \cdot \log_2 k\right) + \Theta(\log_2^{s+1} k) \tag{2.21}$$

Only $\Theta(k) = \Theta\left(\frac{N}{\log_2^{s+1} N}\right)$ satisfies Eq. (2.21). That is, the optimal granularity is $\Theta\left(\frac{N}{\log_2^{s+1} N}\right)$ in this case. □

Corollary 2.2: For searching a complete binary AND-tree of N terminals, in case when $C(N) = \Theta(N^r \log_2^s N)$, $0 < r < 1$, $s \geq 0$, the optimal granularity for the composition phase is $\Theta\left(\frac{N^{1-r}}{\log_2^s N}\right)$

Proof: In substituting $c \cdot (2^{ir} i^s)$ for $C(2^i)$ in Eq.(2.15), where c is a constant, we obtain

$$T_s(N) = \frac{c \cdot N}{k} \sum_{i=1}^{\log_2 \frac{N}{k}} \frac{2^{ir} i^s}{2^i} = \frac{\cdot N}{k} \sum_{i=1}^{\log_2 \frac{N}{k}} (2^{i(r-1)} i^s) \tag{2.22}$$

Since $r < 1$, we have

$$\lim_{i \to \infty} \frac{(2^{(i+1)(r-1)} (i+1)^s)}{(2^{i(r-1)} i^s)} = \lim_{i \to \infty} 2^{r-1} < 1$$

This implies that the series $\sum_{i=1}^{\log_2 \frac{N}{k}} (2^{i(r-1)} i^s)$ converges to a constant, and hence

$$\Theta\left(\sum_{i=1}^{\log_2 \frac{N}{k}} 2^{i(r-1)} i^s\right) = \Theta(1)$$

$$\Theta(T_s(N)) = \Theta\left(\frac{N}{k}\right) \tag{2.23}$$

From Eq. (2.17), we get

$$T_w(N) = c\left(\sum_{i=1}^{\log_2 k} \left(\frac{N}{2^{i-1}}\right)^r (\log_2 N - (i-1))^s\right)$$

Since $\sum_{i=1}^{\log_2 k} \frac{1}{2^{(i-1)r}}$ and $\sum_{i=1}^{\log_2 k} \frac{(i-1)^s}{2^{(i-1)r}}$ converge to constants, we get

$$\Theta(T_w(N)) = \Theta(N^r \log_2^s N) \tag{2.24}$$

From Theorem 2.2, Eq's (2.23), and (2.24), we conclude that when $C(N) = \Theta(N^r \log_2^s(N)$, $0 < r < 1$, the optimal granularity is $\Theta\left(\frac{N^{1-r}}{\log_2^s(N)}\right)$. □

Corollary 2.3: For searching a complete binary AND-tree of N terminals, and $C(N) = \Theta(N \cdot \log_2^s N)$, $s \geq 0$, the optimal granularity for the composition phase is $\Theta(\log_2 N)$.

Proof: $C(N) = \Theta(N \cdot \log_2^s N)$ implies that $C(N) = c \cdot N \cdot \log_2^s N$, where c is a constant. From Eq. (2.15) we obtain [Knu73]

$$T_s(N) = \frac{c \cdot N}{k} \sum_{i=1}^{\log_2 \frac{N}{k}} i^s$$

$$\Theta(T_s(N)) = \Theta\left(\frac{N}{k} \log_2^{s+1} N\right) \tag{2.25}$$

From Eq. (2.16) and a result derived by Knuth [Knu73],

$$T_w(N) = c\left(\sum_{i=1}^{\log_2 k} \frac{N \cdot 2^i}{k} (\log_2 \frac{N}{k} + i)^s\right)$$

$$\Theta(T_w(N)) = \Theta\left(\frac{N}{k} \log_2^s \frac{N}{k} \sum_{i=1}^{\log_2 k} 2^i\right) + \Theta\left(\frac{N}{k} \sum_{i=1}^{\log_2 k} 2^i i^s\right)$$

$$= \Theta\left(N\log_2^s\frac{N}{k}\right) + \Theta(N\log_2^s k) \qquad (2.26)$$

From Theorem 2.2, we can conclude that when $C(N) = \Theta(N\cdot\log_2^s N)$, $s \geqslant 0$, $kT_k^2(N)$ achieves the minimum when

$$\Theta\left(\frac{N}{k}\log_2^{s+1}\frac{N}{k}\right) = \Theta\left(N\log_2^s\frac{N}{k}\right) + \Theta(N\log_2^s k) \qquad (2.27)$$

Only $\Theta(k) = \Theta(\log_2 N)$ satisfies Eq. (2.27). That is, the optimal granularity is $\Theta(\log_2 N)$ when $C(N) = \Theta(N\log_2^s N)$. □

Corollary 2.4: For searching a complete binary AND-tree of N terminals, the optimal granularity for the composition phase is $\Theta(1)$, when $C(N) = \Theta(N^p)$, $p > 1$.

Proof: Similar to the proofs of the above corollaries, we can derive $T_s(N)$ and $T_w(N)$ as:

$$\Theta(T_s(N) = \Theta\left(\frac{N^p}{k^p}\right)$$

$$\Theta(T_w(N) = \Theta(N^p)$$

To make $\Theta\left(\frac{N^p}{k^p}\right)$ equal to $\Theta(N^p)$, $\Theta(k)$ must be $\Theta(1)$. This means that the optimal granularity is $\Theta(1)$ when $C(N) = \Theta(N^p)$ and $p > 1$. □

The above corollaries give the optimal granularities in the sense minimizing $\Theta(kT^2)$ under our synchronous model. To find the optimal granularity precisely, we need to solve the equation below, in which k is viewed as a continuous variable. Simulations have verified that this relaxation is reasonable.

$$\frac{\delta kT_k^2}{\delta k} = T_k^2 + 2kT_k\frac{\delta T_k}{\delta k} = 0$$

namely,

$$T_k + 2k\frac{\delta T_k}{\delta k} = 0 \qquad (2.28)$$

Eq. (2.28) may not have a closed-form solution, but it is easy to evaluate the optimal solution numerically. For instance, in a merge sort problem, $C(N)=\Theta(N)$, and

$$T_k(N) = (\log_2 N - \log_2 k)\frac{N}{k} + 2N\left(1 - \frac{1}{k}\right) \qquad (2.29)$$

where the first term on the R.H.S of Eq. (2.28) is T_s and the second term is T_w. Eq. (2.28) can be verified in the two extreme cases. If $k=1$, i.e, a single processor is used, no wind-down phase exists ($T_w=0$), and $T_1(N) = N\cdot\log_2 N$. In contract, if $k=N$, all evaluations are carried out in the wind-down phase ($T_s=0$). In this case,

$$T_N(N) = 2 + 4 + 8 + \cdots + N = 2N-2$$

Substituting Eq. (2.29) into Eq. (2.28) yields

$$k + \log_2 k = \log_2 N \qquad (2.30)$$

The solution of Eq. (2.30) is close to $k = \log_2 N$. This illustrates that the exact optimal granularity will be near to $\log_2 N$.

2.4 Architectural Requirements

The functional requirement for evaluating an AND tree is an interconnected conglomerate of processors. The proper granularity of parallelism depends largely on the complexity of D(N)+C(N). In the last section, an assumption that D(N) is equal to zero was made. In general, the sum of D(N) and C(N) should be considered. All results obtained in the last section are valid provided that C(N) is replaced by D(N)+C(N). A lot of researchers have studied the relationship among the granularity, communication overhead, and flexibility in load balancing

[KLT84,Mag85]. Among the previous studies, a lager granularity was chosen to reduce the communication overhead [KLT84]; while a small granularity was chosen to balance the workload more evenly and to achieve a high speedup. Little attention has been paid to the limitation of parallel AND-tree searches due to the complexity of $D(N)+C(N)$.

According to our efficiency analysis of parallel AND-tree searches, if the complexity of $D(N)+C(N)$ is $\Theta(N^p)$, $p > 1$, and a large number of processors are used, then the processor-time efficiency, kT^2, must be poor regardless of the capacity of the interconnection network and the load balancing policy. In this case, the time needed to evaluate a subproblem will be increased quickly during the reduction process, and the root of the tree is an obvious bottleneck. Parallel divide-and-conquer algorithms will not be suitable for solving this kind of problems. A more cost-effective computer architecture is a sequential processor or a multiprocessor with few processing units and a shared memory [HoZ83].

In contrast, if $D(N)+C(N)$ is $\Theta(1)$, then the time needed to evaluate any subproblem is bound by a constant, and the root will not be a bottleneck. For example, when a logic program is evaluated, each subproblem returns only TRUE or FALSE, (sometimes a pointer needs to be returned). In this case, a fine-grain architecture is appropriate and a large speedup will be obtained by using a large number of processors. Tree-structured computer architectures [HaS79,Bro80,Mag85] and virtual-tree computers [BuH84] may be good candidates for this kind of applications. Carlson has proved that tree-like computational problems of the form $y = x_0 + x_1 + \cdots + x_{n-1}$, where + is an associative operation such as addition, multiplication, boolean AND, and boolean OR, can be implemented by $\Theta(N/\log N)$ processors connected by a perfect-shuffle or hypercube network in $\Theta(\log N)$ parallel time [Car84]. In fact, $C(N) = \Theta(1)$ for this kind of computations and the optimal granularity was found, though not with respect to kT^2.

In case when $C(N) = \Theta(N)$ or $\Theta(\log^s N)$, $s \geq 0$, the time needed to evaluate a subproblem is increased slowly during the reduction process. A medium-grain

architecture will be more cost-effective. For example, to sort about 4000 elements by a parallel merge-sort algorithm, using 12 processors will be a good choice. The optimal granularities and architectural requirements with respect to various $C(N)$ are summaried in Table 2.1.

In parallel processing of AND-tree searches, the ideal situation is a single initial scheduling decision that sends equal-sized grains to all processing units. It is seldom possible to make such a priori decision. Instead, in many applications, the problems to be solved are usually irregularly-structured, and the workloads may be data dependent. An important functional requirement for parallel AND-tree searches is the ability to dynamically distribute workload in the architecture. For a computer architecture with a small granularity, an efficient interconnection network is required to transfer data and control information. In a loosely coupled computer system with a coarse grain, an effective load balancing mechanism is also needed.

Table 2.1 Optimal granularity of parallelism in AND-tree searches (Order-of-magnitude minimum of kT^2)

Complex. of $C(N)$	Optimal granu.	Arch. Requirements
$\Theta(\log_2^s N)$ $s \geq 0$	$\Theta\left(\frac{N}{\log_2^{s+1} N}\right)$	A very large number of processors; tree or other efficient interconnection
$\Theta(N^r \log_2^s N)$ $0 < r < 1$ $s \geq 0$	$\Theta\left(\frac{N^{1-r}}{\log_2^s N}\right)$	A large number of processors; tree or other efficient interconnection
$\Theta(N \log_2^s N)$ $s \geq 0$	$\Theta(\log_2 N)$	A small number of processors; loosely coupled; simple interconnection
$\Theta(N^p)$ $P > 1$	$O(1)$	Single or few processors; shared memory

CHAPTER III
PARALLEL PROCESSING OF OR-GRAPH SEARCHES

In this chapter we investigate strategies for OR-graph searches. In particular, we develop a general heuristic function for guiding the search, study dominance tests for pruning, propose a computer architecture for implementing branch-and-bound algorithms with dominance tests, and find performance bounds of parallel OR-tree searches. The conditions to cope with anomalies of parallel OR-tree searches will be derived in the next chapter.

3.1 OR-Graph Search Versus OR-Tree Search

An OR-graph is an acyclic graph in which all nonterminal nodes are *OR-nodes*. An OR-node represents a (sub)problem that is solved if any one of its descendents (or decomposed) subproblems can be solved. An OR-tree is a special case of an OR-graph, in which, each node except the root has exactly one parent node, that is, exactly one edge entering it. The conventional branch-and-bound algorithm without dominance tests is a typical OR-tree search algorithm.

Two techniques, dominance tests and equivalence tests, can be used to convert OR-tree searches into OR-graph searches. If dominance tests are applied, a set of nodes are compared with each other, and only one node that is not dominated is active. In other words, all nodes that are connected by the dominance relation can be represented as a single node. As a result, several nodes, can share a common descendent node. With dominance tests, a branch-and-bound (B&B) algorithm can be viewed as an OR-graph search algorithm. An equivalence relation is a special case of the dominance relation. If only one solution is sought, all equivalent nodes except one are eliminated; that is, all equivalent subproblems are denoted by a single node. The equivalence relation is illustrated by the knapsack problem. A subproblem (1,2) representing that Objects 1 and 2 are included in the knapsack, and another subproblem (2,1) representing that Objects 2 and 1 are included are equivalent. Since the order of selections is not important in the knapsack problem, it is only necessary to continue the search from one of the equivalent subproblems.

In general, implementing an OR-tree search is simple and requires little bookkeeping. However, if a transformation is applied to transform the OR-graph to an OR-tree, then it may result in the same node being generated in different paths and so being processed more than once. Treating the search process as a graph search rather than a tree search reduces the amount of redundancy. However, an additional effort may be needed each time a node is generated to see if it has been generated before. Whether this effort is justified depends on the particular problem under consideration. If it is very likely that either the same node will be generated in several different ways or the dominance relation is strong (the dominance relation exists among many nodes), then it is worthwhile to use a graph-search procedure. In Section 3.5.2, we address the problem on parallel dominance tests that corresponds to OR-graph searches. Since dominance tests are systematically used in solving dynamic programming problems, they will be discussed in more details in Chapter 6.

3.2 Basic Rules of an OR-Graph Search

Many theoretical properties on OR-graph search algorithms, especially OR-tree search algorithms, have been developed by Nilsson [Nil80], Kohler and Steiglitz [KoS74], and Ibaraki [Iba76a,Iba76b,IbaA77a]. In this section, a summary of these properties that will be used in the following sections are given. Conventional B&B

algorithms will be extended to include (a) a generalized heuristic function as the selection rule and (b) approximate lower-bound and dominance tests as general elimination rules. Some new properties on heuristic functions and dominance tests will also be developed.

The way in which a *minimization problem* P_0 is repeatedly decomposed into smaller subproblems can be represented as a finite rooted acyclic graph, $B = (P, E)$, where P is a set of disjunctive subproblems, and E is a set of edges. The root of the acyclic graph is P_0. If a subproblem P_{i_j} is obtained from P_i by decomposition, then $(P_i, P_{i_j}) \in E$. Note that during decomposition an edge exists only between a pair of nodes that have the parent-children relation. The decomposition of a problem will result in the construction of a tree called the *underlying tree* that must be divided into levels. An acyclic graph is formed by introducing dominance tests. The *level number* of a node in the underlying tree is the number of edges leading from the root to this node (the root is at level 0). Let $f(P_i)$ be the value of the best solution obtained by evaluating all the subproblems decomposable from P_i, let P_{i_j} be the j'th subproblem decomposable from P_i, and let k_i be the number of such subproblems (i.e., $k_i = |\{(P_i,x): (P_i,x) \in E\}|$). Then f satisfies:

$$f(P_i) = \min_{j=1,\ldots,k_i} \{f(P_{i_j})\} \qquad (3.1)$$

Each subproblem is characterized by a lower-bound value that is computed from a lower-bound function g. Let T be the set of all feasible solutions. The lower-bound function satisfies the following properties:

(a) $g(P_i) \leq f(P_i)$ for $P_i \in P$ (g is a lower—bound estimate of f) (3.2)

(b) $g(P_i) = f(P_i)$ for $P_i \in T$ (g is exact when P_i is feasible) (3.3)

(c) $g(P_i) \leq g(P_{i_j})$ for $(P_i,P_{i_j}) \in E$ (lower bounds always increase) (3.4)

OR-Graph search algorithms can be characterized by four constituents: a branching rule, a selection rule, an elimination rule and a termination condition.

The first two rules are used to decompose problems into simpler subproblems and to appropriately order the search. The last two rules are used to to eliminate generated subproblems that are not better than the ones already known. Appropriately ordering the search and restricting the region searched are the key ideas behind branch-and-bound algorithms.

3.2.1 Selection Rule Using a General Heuristic Function

The active subproblems are selected based on certain criterion. If the list of active subproblems is maintained in a first-in/first-out order, the algorithm is called a *breadth-first search*. If the list is maintained in a last-in/first-out order, the algorithm is called a *depth-first search*. Lastly, if the list is maintained in increasing order of lower bounds, the search algorithm is called a *best-first search*. Ibaraki mapped these searches into a general form called *heuristic searches* [Iba76b]. In a general heuristic search[3], a heuristic function is defined to govern the order in which subproblems are selected and decomposed. The algorithm always decomposes the subproblem with the minimum heuristic value. In a best-first search, the lower-bound values define the order of expansion. Therefore, the lower-bound function can be taken as the heuristic function. In a breadth-first search, subproblems with the minimum level numbers are expanded first. The level number can, thus, be taken as the heuristic function. Lastly, in a depth-first search, subproblems with the maximum level numbers are expanded first. The negation of the level number can be taken as the heuristic function. If U is the current list of active subproblems in the process of expansion and h is the heuristic function, then the search function for a serial branch-and-bound algorithm is:

$$s_s(U) = \{P_i \mid h(P_i) = \min_{P_j \in U} h(P_j)\} \qquad (3.5)$$

[3] Differing from the conventional definition of heuristic search in artificial intelligence, the definition of general heuristic search in this thesis is taken from Ibaraki [Iba76b]. Depth-first and breadth-first searches are considered as special cases of heuristic searches.

However, searches based on Eq. (3.5) are potentially anomalous when parallel expansions are allowed. An anomaly implies that the speedup is not linear with respect to the number of processors. For instance, the performance of using multiple processors may be worse than that of a single processor. Anomalies are resulted partly from the ambiguity in the heuristic function. For example, in the serial depth-first search, the subproblems are maintained in a last-in-first-out list, and the subproblem with the maximum level number is expanded first. When multiple subproblems have identical level numbers, the node chosen for expansion depends on the order of insertion into the stack. The level number alone does not give adequate information to uniquely select a node for expansion. The same situation happens in a best-first search when some nodes have identical lower bounds. Anomalous behavior of parallel OR-tree searches will be discussed in detail in the next chapter.

To resolve the ambiguity of the selection of subproblems, distinct heuristic values must be defined for the nodes. In this thesis, a path number is proposed to uniquely identify a node. The *path number* of a node in a tree is a sequence of d+1 integers representing the path from the root to this node, where d is the maximum number of levels of the tree. The path number $E = e_0e_1e_2...e_d$ is defined recursively as follows. The root P_0 exists at Level 0 and has a path number of $E_0 = 000...0$. A node P_{i_j} on Level ℓ, which is the j'th child (counting from the left) of P_i with path number $E_i = e_0e_1...e_{\ell-1}000...$, has path number $E_{i_j} = e_0e_1...e_{\ell-1}j00....$ As an example, the path numbers of all nodes in the tree of Figure 3.1 are shown next to the nodes.

To compare path numbers, the relations '>' and '=' must be defined. A path number $E_x = e_1^1e_2^1...$ is less than another path number $E_y = e_1^2e_2^2...$ ($E_x < E_y$) if there exists $0 \leqslant j \leqslant d$ such that $e_i^1 = e_i^2$, $0 \leqslant i < j$, and $e_j^1 < e_j^2$. The path numbers are equal if $e_i^1 = e_i^2$ for $0 \leqslant i \leqslant d$. For example, the path number 01000 is less than 01010. According to our definition of path numbers, nodes can have equal path numbers if they have the ancestor-descendant relationship. Since these nodes

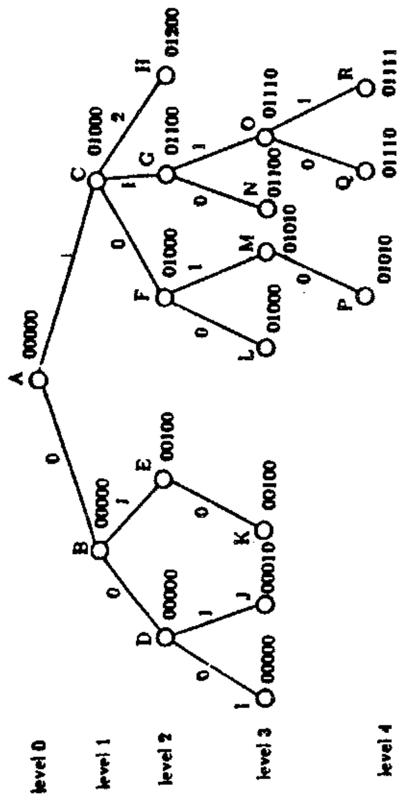


Figure 3.1 The path numbers of a tree

never coexist simultaneously in the active list of subproblems of a B&B algorithm, the subproblems in the active list always have distinct path numbers.

The path number is now included in the general heuristic function. The primary key is still the lower-bound value or the level number. The secondary or ternary key is the path number and is used to break ties in the primary key.

$$h(P_i) = \begin{cases} \text{(level number, path number)} & \text{breadth-first search} \\ \text{(path number)} & \text{depth-first search} \\ \text{(lower bound, level number, path number)} & \\ \text{or (lower bound, path number)} & \text{best-first search} \end{cases} \quad (3.6)$$

where the level number, path number, and lower bound are defined for P_i. For a best-first search, nodes with identical lower bounds can be searched in a breadth-first or depth-first fashion.

A heuristic function is said to be *monotone* when the following inequality is satisfied:

$$h(P_i)<h(P_j) \quad \text{if } P_j \text{ is a descendant of } P_i \quad (3.7)$$

Ibaraki has proved that, for any heuristic function, there exist an equivalent monotone heuristic function [Iba76b].

A heuristic function is said to be *unambiguous* when:

$$h(P_i)\neq h(P_j) \quad \text{if } P_i\neq P_j \quad P_i, P_j \in P \quad (3.8)$$

The property of unambiguity is very important in predicating the performance of parallel OR-tree searches. In general, unambiguous heuristic functions are not restricted to the use of path numbers. Any tie-breaking rule can be adopted as long as Eq. (3.8) is satisfied. Unambiguous heuristic functions other than Eq. (3.6) can also be designed. For example, the lower bound can be used as the secondary key and the path number as the ternary key in a breadth-first search.

3.2.2 Lower-Bound Elimination and Termination Rules

A lower bound is calculated for a subproblem when it is created. If a subproblem is a feasible solution with the best objective-function value so far, then the solution value becomes the *incumbent* z. The incumbent represents the best solution obtained so far in the expansion process. In minimization problems, if the lower bound of a subproblem exceeds the value of the incumbent, then this subproblem can be pruned because it will not lead to an optimal solution. The decomposition process continues until all the subproblems are either expanded or pruned.

During the expansion process, P_i is terminated if

$$g(P_i) \geqslant z \qquad (3.9)$$

Let L denotes the lower-bound cutoff test, that is, $P_j \, L \, P_i$ means that P_j is a feasible solution and that $f(P_j) \leqslant g(P_i)$. For simplicity, only the search for a single optimal solution is considered in this thesis. The computational efficiency for searching all optimal solutions can be investigated similarly.

The above lower-bound test for obtaining an exact optimal solution can be relaxed to obtain a suboptimal solution with guaranteed accuracy [LaW66]. Suppose it were decided that a deviation of 10% from the optimum was tolerable. If a feasible solution of 150 is obtained, all subproblems with lower bounds of 136.4 (or 150/(1+0.1)) or more can be terminated since they cannot lead to a solution that deviates by more than 10% from 150. This technique significantly reduces the amount of intermediate storage and the time needed to arrive at a suboptimal solution. Define an *allowance function* $\epsilon(z)$: $R \rightarrow R$ (set of reals) such that P_i is terminated if

$$g(P_i) \geqslant z - \epsilon(z). \qquad (3.10)$$

The final incumbent value z_F obtained by the modified lower-bound test deviates from the optimal-solution value z_O by [Iba76a]

$$z_F - \epsilon(z_F) \leqslant z_O \leqslant z_F \qquad (3.11)$$

Examples of often used allowance functions are

$$\epsilon(z) = \epsilon \geqslant 0 \quad \text{(absolute error deviation) and} \qquad (3.12)$$

$$\epsilon(z) = \frac{z\epsilon}{1+\epsilon}, \ \text{i.e.,} \ \frac{z-z_O}{z_O} \leqslant \epsilon, \ \ \epsilon \geqslant 0, z \geqslant 0 \quad \text{(relative error deviation).} \qquad (3.13)$$

Properties of these allowance functions are similar. In this thesis, the function for relative error deviation is assumed.

3.2.3 Dominance Elimination Rules

Dominance tests are powerful elimination rules that are systematically applied in decision problems and in dynamic programming to reduce the complexity of enumeration [BeD62,MoM76]. For some problems that are solvable in polynomial time, the branch-and-bound approach with lower-bound tests is not competitive with existing algorithms. For example, the shortest-path and the two-stage flowshop scheduling problems require exponential time using lower-bound tests only, but can be solved in linear time on the average with dominance tests [Iba77b]. Some of the well-known dominance relations are defined for the knapsack, the n-job two-machine mean-finishing-time flowshop [CMM67], and the n-job one-machine scheduling with deadlines [Sah76] problems. Dominance tests are rarely used when the problem is inherently intractable.

A dominance relation is a binary relation such that $P_i \, D \, P_j$ (or $(P_i,P_j) \in D$) implies P_i dominates P_j [KoS74,Iba76a]. This means that the subtree rooted at P_i contains a solution node with a value no more than the minimum solution value of the subtree rooted at P_j. Thus if P_i and P_j are generated and $P_i \, D \, P_j$, then P_j can be terminated. A dominance relation satisfies the following conditions [Iba77b]:

(a) $P_i \, D \, P_j$ implies that $f(P_i) \leqslant f(P_j)$ and that P_i is not a proper descendant of P_j; (3.14)

(b) D is a partial—ordering (reflexive, antisymmetric and transitive); (3.15)

(c) $P_i \, D \, P_j$ and $P_i \neq P_j$ imply that for any proper descendant $P_{j'}$ of P_j, there exists a descendant $P_{i'}$ (including P_i) of P_i such that $P_{i'} \, D \, P_{j'}$. (3.16)

A dominance relation is a partial order, that is, for a pair of nodes P_i and P_j, it is possible that neither $P_i \, D \, P_j$ nor $P_j \, D \, P_i$ holds. In this case, P_i and P_j are said to be *incomparable*. If a dominance relation is weak, then most of the nodes are incomparable. A node P_i is said to be a *current dominating node* if it has been generated and has not been dominated so far. All current dominating nodes are incomparable to each other.

To use dominance tests, only the set of current dominating nodes (denoted by N_d) has to be stored. In general, the set of active nodes is not sufficient to determine N_d because $P_i \, D \, P_j$ does not imply that there must exist a proper descendant $P_{i'}$ of P_i such that $P_{i'} \, D \, P_j$ (Eq. (3.16)). This means that some or all of the generated nodes have to be available to carry out the dominance tests. The size of N_d is exponentially large for NP-hard problems. Suppose on the contrary that this size is a polynomial function of problem size, $w(n)$, that the degree of the branch-and-bound tree is bounded by m, and that the height of the tree is a polynomial function of n, $\theta(n)$. Since the set of active nodes is a subset of N_d, there exists at most $w(n)$ active nodes at any time. There are at most $m \times w(n)$ nodes after branching, and $w(n)$ active nodes remain after $O(m^2 w^2(n))$ dominance tests. For $\theta(n)$ levels of the tree, the total computation time is $O(m^2 w^2(n) \theta(n))$, which contradicts the NP-hardness of the problem. This fact implies that dominance tests are not space-effective for intractable problems. Indeed, dominance test is a strategy that saves the computational time by using more memory space to store information which can prune unnecessary expansions. This time-space tradeoff should be considered when dominance tests are applied.

3.2.4 Branch-and-Bound Algorithm

The operations of a branch-and-bound algorithm are summarized below:

Serial Branch-and-Bound Algorithm (Single Solution)

```
g = lower-bound function;     ε = allowance function;
D = dominance relation;
s_s = serial selection function;   z = incumbent value;
N_d = set of current dominating nodes;
U = set of active subproblems.
(1) (Initialize): $z \leftarrow \infty$; $U \leftarrow \{P_0\}$; $N_d \leftarrow \{P_0\}$;
(2) (Select): if $U = \emptyset$, then go to Step 8 else let $P_i \leftarrow s_s(U)$, $U \leftarrow U - \{P_i\}$; if
    $g(P_i) \geq z - ε(z)$, then go to Step 2;
(3) (Decompose): Generate sons $P_{i_1}, P_{i_2}, ..., P_{i_{k_i}}$ of $P_i$; $N_d \leftarrow N_d \cup \{P_{i_1}, ..., P_{i_{k_i}}\}$; U
    $\leftarrow U \cup \{P_{i_1}, ..., P_{i_{k_i}}\}$
(4) (Feasibility test): For all $j \in \{1, ..., k_i\}$, if $P_{i_j}$ is a feasible solution,
    then $U \leftarrow U - \{P_{i_j}\}$ and $z \leftarrow \min[z, f(P_{i_j})]$;
(5) (Lower-bound test): For all $j \in \{1, ..., k_i\}$, if $g(P_{i_j}) \geq z - ε(z)$, then $U \leftarrow U - \{P_{i_j}\}$;
(6) (Dominance test): For all $j \in \{1, ..., k_i\}$, if there exists $P_k (\neq P_{i_j}) \in N_d$ such that
    $P_k \, D \, P_{i_j}$, then $U \leftarrow U - \{P_{i_j}\}$; if $P_{i_j} \, D \, P_k$ then $N_d \leftarrow N_d - \{P_k\}$.
(7) (Terminate): Go to Step 2;
(8) (Halt): $f(P_0)$ satisfies $z - ε(z) \leq f(P_0) \leq z$.
```

In the above algorithm, if one or both of the elimination rules (Steps 5, 6) are not used, then the corresponding steps are skipped. Further, eliminations are done after branching instead of after selection as in Ibaraki's algorithm [Iba76a]. This reduces the memory space required for storing the active subproblems.

To illustrate the algorithm, consider the *knapsack problem* [GaJ79]. Given n objects with positive profits and weights, the objective is to pack objects into a knapsack of fixed capacity such that the total profit is maximized. It is assumed that each object is allocated as an unit. Since this is a maximization problem, the upper bound u is calculated instead. Furthermore, the bounding tests in Eq's (3.9) or (3.10) become $u(P_i) \leq z$ or $u(P_i) \leq z+\epsilon(z)$. A good upper bound can be computed by arranging the objects in decreasing profit-to-weight ratios and allocating as many objects as possible into the knapsack while allowing fractional allocation of the last object packed. A branch-and-bound tree with depth-first search on a knapsack problem with three objects is shown in Figure 3.2. The number inside each node is the order in which the nodes are examined. The number on each edge represents the object included or excluded. z and ub are the updated values at the end of each iteration. Nodes 3 and 5 are infeasible because the knapsack capacity is exceeded. Node 8 is dominated by node 4 because, for the same set of objects {1, 2}, the total profit (−11) is larger and the total weight (−4) is smaller for node 4. Node 11 is terminated by the upper-bound test because $u(P_{11})<z$. In this example, the allowance function used is $\epsilon=0$. If a relative error function, Eq. (3.13), with $\epsilon=0.2$ is used, then node 9 will also be terminated by the upper-bound test. A solution with value 11 that deviates by no more 20% from the optimal value of 12 is obtained.

3.3 Parallel Branch-and-Bound Algorithm

Branch-and-bound algorithms have inherent parallelism. Each of the four rules of serial branch-and-bound algorithms can be implemented by parallel processing.

(a) *Parallel selection of subproblems*: In the parallel case, a set of subproblems less than or equal in size to the number of processors have to be selected for decomposition in each iteration. The selection problem is especially critical under a best-first search because a set of subproblems with the minimum lower bounds must be selected. The selection function of Eq. (3.5) becomes

OBJECT	PROFIT	WEIGHT
1	11	4
2	10	5
3	12	6

Knapsack weight capacity = 9

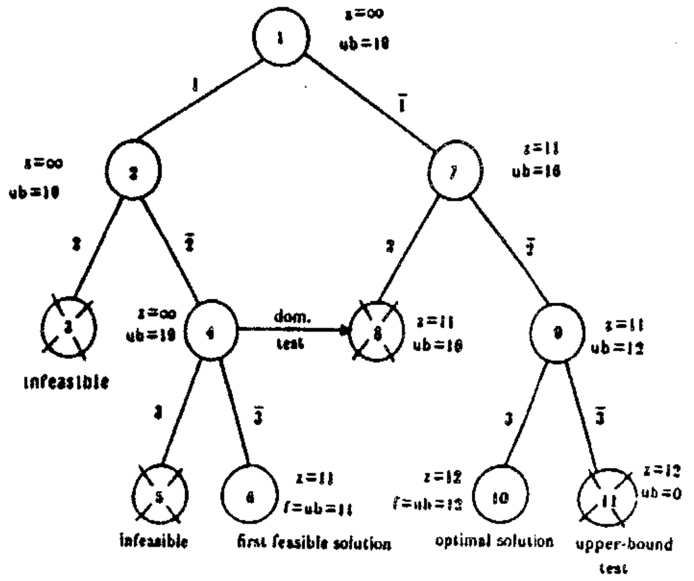


Figure 3.2 An Example of branch-and-bound algorithm using the knapsack problem

$$s_p(U) = \begin{cases} \{P_{i_1}, \ldots, P_{i_k}\} \\ \text{where } h(P_i) < h(P_j),\ P_i, P_j \in U \\ i \in \{i_1, \ldots, i_k\},\ j \notin \{i_1, \ldots, i_k\} & \text{if } |U| > k \\ U & \text{if } |U| \leq k \end{cases} \qquad (3.17)$$

where k is the number of processors. This returns the set of k subproblems with the minimum heuristic values from U.

(b) *Parallel branch*: The subproblems assigned to the processors can be decomposed in parallel. In order for the processors to be well utilized, the number of active subproblems should be greater than k in each iteration.

(c) *Parallel termination test*: Multiple infeasible nodes can be eliminated in each iteration. Further, multiple feasible solutions may be generated, and the incumbent has to be updated in parallel.

(d) *Parallel elimination test*: The lower-bound test (Eq's (3.9) or (3.10)) can be sped up by comparing lower bounds of multiple subproblems with the incumbent. The elimination of subproblems by dominance tests can also be carried out in parallel. However, the bounding function and the dominance test are problem-dependent, and software implementation is more flexible.

3.4 Bounds of Computational Efficiency of an OR-Tree Search

To predict the number of processors needed to assure a near-linear speedup, we need to derive the bounds of computational performance of an OR-tree search. The results in this section indicate the relationship among the number of iterations required in a parallel search, the number of processors used, and the complexity of the problem to be solved.

In presenting the performance bounds in this thesis, a synchronous model is assumed, that is, all processors must finish an iteration before proceeding to the next iteration together. The performance is difficult to evaluate if the search algorithm is evaluated asynchronously in each processor. The performance results

for synchronous models form a lower bound to that of asynchronous models.

In this section, Theorems 3.2, 3.3 and 3.4 show the bounds in performance for a parallel best-first, depth-first and breadth-first search, respectively. The proofs of these theorems require the following definitions. A node, P_i, expanded by a serial OR-tree search, is said to be an *essential node*, otherwise it is called a non-essential node. The speedup of a parallel OR-tree search depends on the number of essential nodes selected in each iteration. An iteration is said to be a *perfect iteration* if the number of essential nodes selected in an iteration is equal to the number of processors, otherwise it is said to be an *imperfect iteration*. The performance is not affected by keeping the active subproblems in a single or multiple subproblem lists. We denote $T_b(k,0)$, $T_d(k,0)$, and $T_r(k,0)$ as the number of iterations required by using k ($k \geq 1$) processors in a best-first, depth-first, and breadth-first search, respectively. The subscripts b, d, or r will be omitted when more than one search strategy is concerned. Before proving the theorems, we need to explain an important concept on the basic node, which is a key to studying the performance.

3.4.1 Basic Nodes and Its Properties

A *basic node* is the node with the smallest heuristic value in each iteration. Basic nodes have the following property.

Lemma 3.1: Let P_i be a basic node, then for any node P_j such that $h(P_j)<h(P_i)$, P_j must be either expanded or terminated when P_i is expanded.

Proof: Suppose that in the current active list, U, $P_i \in U$ is a basic node. Assume that there exists a node P_j such that $h(P_j)<h(P_i)$ and that P_j has not been expanded or terminated when P_i is expanded. Since P_i has the minimum heuristic value among the active nodes in U, P_j must not be active at that time. That is, P_j is a descendant of some node P_k, $P_k \in U$, and $h(P_i)<h(P_k)$. By Eq. (3.8), $h(P_i)<h(P_k)<h(P_j)$, which contradicts the assumption that $h(P_j)<h(P_i)$. □

Let $\Phi^k, k \geq 1$, be the set of nodes expanded in the OR-tree using k processors. In fact, Φ^1 is a set of essential nodes. In what follows, we show that any heuristic search with an unambiguous heuristic function can guarantee at least one node in Φ^1 to be expanded in each iteration of the parallel search, when an exact optimal solution is sought and dominance tests are inactive.

Theorem 3.1: Let $\epsilon=0$, i.e., an exact optimal solution is sought, and D=I, i.e., dominance tests are inactive. For any parallel heuristic search with a heuristic function satisfying Eq's (3.7) and (3.8), all basic nodes are essential nodes, i.e., all basic nodes belong to Φ^1.

Proof: The proof is by contradiction. Suppose that there exists a basic node P_{i_1} in the parallel search such that $P_{i_1} \notin \Phi^1$ (see Figure 3.3). This means that either P_{i_1} or its ancestor is terminated by a lower-bound test in the serial case. Hence there must exist a feasible solution $P_{i_2} \in \Phi^1$ such that $f(P_{i_2}) \leq g(P_{i_1})$ and that P_{i_2} has not been obtained when P_{i_1} is expanded in the parallel case. It implies that a proper ancestor $P_{i_3} \in \Phi^1$ of P_{i_2} exists in the serial case such that $h(P_{i_3}) < h(P_{i_1})$, and that P_{i_2} is obtained before P_{i_1} and terminates P_{i_1}. Since P_{i_1} is a basic node in the parallel search, $h(P_{i_3}) < h(P_{i_1})$, and P_{i_3} has not been expanded when P_{i_1} is expanded in the parallel case, P_{i_3} must be terminated according to Lemma 3.1. For the parallel search, there must exist a feasible solution $P_{i_4} \in \Phi^k$ such that $f(P_{i_4}) \leq g(P_{i_3})$, and that P_{i_4} has not been obtained when P_{i_3} is expanded in the serial case. Two cases are possible.

First, P_{i_4} is not generated when P_{i_3} is expanded in the serial case, i.e., a proper ancestor $P_{i_5} \in \Phi^k$ of P_{i_4} exists when P_{i_3} is active and $h(P_{i_5}) > h(P_{i_3})$. According to the properties of lower-bound functions (Eq's (3.2), (3.3), and (3.4)), we have $f(P_{i_4}) \leq g(P_{i_3}) \leq f(P_{i_3}) \leq f(P_{i_2}) \leq g(P_{i_1})$. Moreover, in the parallel case, P_{i_4} should be obtained before P_{i_1} is expanded (otherwise, P_{i_3} would not be terminated by P_{i_4}). Hence P_{i_1} has to be terminated by P_{i_4} in the parallel algorithm, which contradicts

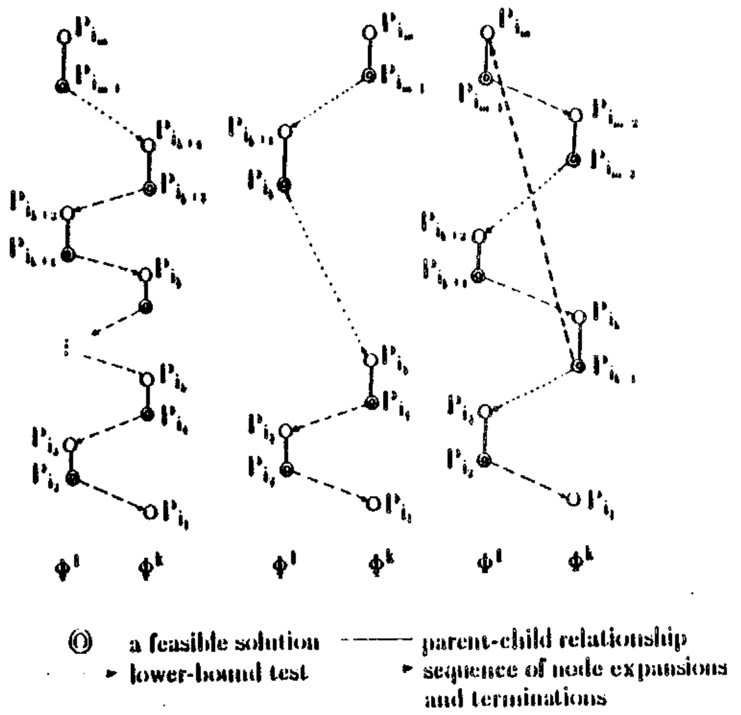

Φ^1
Φ^k
a feasible solution
lower-bound test
parent-child relationship
sequence of node expansions and terminations

Figure 3.3 The proof of Theorem 3.1

the assumption that $P_{l_1} \in \Phi^k$.

Second, $h(P_{l_3})<h(P_{l_1})$, and P_{l_2}, as well as its descendant P_{l_3}, have been terminated in the serial case and not in the parallel case. We can then apply the above argument again to P_{l_3} and eventually obtain a sequence of nodes $P_{l_1}, P_{l_2}, ..., P_{l_m}$ as depicted in Figure 3.3, in which P_{l_m} is not terminated by any lower-bound test. There are three possibilities.

(a) The first node P_{l_m} occurs in the serial case (Figure 3.3(a)). Since $P_{l_{m-1}}$ is a feasible solution, we have: $g(P_{l_m}) \leqslant f(P_{l_{m-1}}) \leqslant g(P_{l_{m-2}}) \leqslant f(P_{l_{m-2}}) \leqslant g(P_{l_{m-3}}) \leqslant \cdots \leqslant f(P_{l_1}) \leqslant g(P_{l_1})$. Further, since $h(P_{l_{k+2}})<h(P_{l_k})$ (otherwise, P_{l_k} could not have been terminated by $P_{l_{k+1}}$ in the serial case) and since $h(P_{l_{k+4}})<h(P_{l_{k+2}})$ (by the same argument as $h(P_{l_3})<h(P_{l_1})$), we have $h(P_{l_{k+4}})<h(P_{l_k})$. Repeating this, we get $h(P_{l_m})<h(P_{l_1})$. By Lemma 3.1, P_{l_m} must be expanded in the parallel case and terminates P_{l_1}, which contradicts that $P_{l_1} \in \Phi^k$.

(b) The first node P_{l_m} occurs in the parallel case (Figure 3.3(b)). Similar to the argument for P_{l_2} discussed before, we can explain that $P_{l_{m-1}}$ has been obtained when P_{l_1} is selected. Therefore, P_{l_1} must be terminated in the parallel case, which contradicts that $P_{l_1} \in \Phi^k$.

(c) There is a cycle of cutoffs such that $P_{l_{m-1}}$ L $P_{l_{m-2}}$, $P_{l_{m-3}}$ L $P_{l_{m-4}}$, ..., $P_{l_{k+1}}$ L P_{l_k}, and $P_{l_{k-1}}$ L P_{l_m}, where L denotes a lower-bound-cutoff test (Figure 3.3(c)). By transitivity, we have $f(P_{l_m}) \leqslant f(P_{l_{m-1}}) \leqslant f(P_{l_{k-1}}) \leqslant f(P_{l_m})$, which implies that $f(P_{l_m})=f(P_{l_{m-1}})=f(P_{l_{k-1}})$. The heuristic value of all nodes of the cycle are less than $h(P_{l_1})$, so a feasible solution has been obtained before P_{l_1} is selected. Thus P_{l_1} must be terminated in the parallel case, which contradicts that $P_{l_1} \in \Phi^k$ □

3.4.2 Parallel Best-First Searches

The performance of a parallel best-first search is described in Theorem 3.2. The proof of Theorem 3.2 requires the result in the following Lemma.

Lemma 3.2: In a parallel OR-tree search, all essential nodes must be expanded if the following property is satisfied: P_i L P_j and P_k is the parent of P_i implies that $h(P_k) < h(P_j)$.

Proof: Suppose that an essential node P_1 is eliminated by node P_2, and that P_3 is the parent of P_2. By assumption, $h(P_3)$ must be less than $h(P_1)$. Since in a serial search, the nodes are expanded according to the heuristic values, hence node P_3 should be selected before P_1, and P_1 must be eliminated in the serial case. A contradiction! □

Theorem 3.2: Suppose $\epsilon=0$, D=I. Let the root be in level 0 and h be the maximum number of levels in an OR-tree. If the heuristic function is defined as $h(P_i)$ = (lower bound, level number, path number), then the following bounds hold for the parallel best-first search with k processors,

$$\frac{T_b(1,0)-1}{k} + 1 \leqslant T_b(k,0) \leqslant \frac{T_b(1,0)}{k} + \frac{k-1}{k}h \tag{3.18}$$

Proof: All the required iterations to find an optimal solution can be classified into either perfect or imperfect. Let $h_{min}(x)$ be the minimum level in which some active essential nodes reside in the x'th iteration. For levels less than $h_{min}(x)$, all active nodes are non-essential. We show that iteration x can be imperfect only if all the essential nodes in $h_{min}(x)$ are selected for expansion. Suppose that iteration x is an imperfect iteration and that some essential nodes, say P_i, in $h_{min}(x)$ are not selected for expansion, then this contradicts with the selection rule of a best-first search, since in this case at least one node whose heuristic value is greater than $h(P_i)$ is selected. (Note that in a best-first search, all non-essential nodes have heuristic values greater than that of essential nodes according to the given definition of $h(P_i)$). Thus after an imperfect iteration x is carried out, $h_{min}(x)$ must be increased by at least one, that is, $h_{min}(x+1) \geq h_{min}(x)+1$.

As $h_{min}(x+1)$ must be less than the maximum number of levels, h, and the root is defined as level 0, there can be at most h imperfect iterations. From our assumptions that nodes with lower bound equal to the optimal solution are searched in a breadth-first search fashion, that $\epsilon = 0$, and that $D = I$, it is true that once all essential nodes are expanded, the optimal solution is found. Recall that in each iteration at least one essential node must be selected according to Theorem 1. The upper bound of $T_b(k,0)$ satisfies:

$$T_b(k,0) \leq \left\lceil \frac{T_b(1,0)-h}{k} \right\rceil + h \leq \frac{T_b(1,0)}{k} + \frac{k-1}{k}h$$

To find the lower bound of $T_b(k,0)$, we show that all essential nodes must be expanded in the parallel case. Owing to $\epsilon = 0$ and D=I, all nodes whose lower bounds are less than the optimal solution value have to be expanded. If there exists multiple nodes whose lower bounds are equal to the optimal solution, then they are searched in a breadth-first search fashion. Note that breadth-first searches have the property that $P_i L P_j$ and P_k is the parent of P_i imply that $h(P_k) < h(P_j)$. Based on the property of the best-first search mentioned above and Lemma 3.2, we know that all essential nodes have to be expanded in a parallel best-first search. On the other hand, the first iteration during which the root node is expanded is imperfect for $k > 1$. Hence,

$$T_b(k,0) \geq \frac{T_b(1,0)-1}{k} + 1 \qquad \square$$

In case that a feasible solution must be located in levels greater than or equal to h', then h' iterations are necessary, and $T_b(k,0) \geq \max(h', (T_b(1,0)-1)/k - 1)$. In this thesis, we assume that $(T_b(1,0)-1)/k - 1 > h'$. If the OR-tree is regular, i.e., each node has m branches, then we can derive a tighter lower bound of $T_b(k,0)$ as

$$T_b(k,0) = \frac{T_b(1,0)}{k} + \left\lceil \log_m k \right\rceil - \frac{m^{\left\lceil \log_m k \right\rceil} - 1}{k(m-1)}$$

This is true because in a regular tree, the number of selected nodes in the i'th iteration is at most m^{i-1} and the first $\left\lceil \log_m k \right\rceil$ iterations must be imperfect.

Notice that if we know the maximal level, θ_{max}, at which at least one essential node exists, i.e. there is no essential node in levels larger than θ_{max}, the upper bound can be tighted by substituting θ_{max} for h in Eq. (3.18). In practice, the information on θ_{max} is usually not available in advance[4].

The assumptions in Theorem 3.2 that $\epsilon = 0$, D=I and $h(P_i)$ = (lower bound, level number, path number) uniquely define the number of essential nodes expanded. If dominance tests are active, some nodes eliminated by dominance test in the serial case may be expanded in the parallel case. The number of imperfect iteration, as a result, may be greater than h. On the other hand, if $\epsilon > 0$ or the heuristic function $h(P_i)$ = (lower bound, path number), then some essential nodes may not be expanded in the parallel case. In both cases, the lower bound of $T_b(k,\epsilon)$ will be much less than that in Eq. (3.18).

Eq. (3.18) shows that a near-linear speedup may be obtained when a large number of processors are used as long as $T_b(1,0)/k$ is much larger than h. As an example, if h=50, $T_b(1,0)=10^6$ (for a typical traveling-salesman problem), and k=1000, then $T_b(1000,0) \leq 1049$. This means that a near-linear speedup can be attained with one thousand processors.

From Theorem 3.2, it is also easy to determinate the maximum number of processors that guarantee a near-linear speedup. Assume that the speedup required is $T_b(1,0)/T_b(k,0) \geq (\eta k), 0<\eta<1$ and $\eta \approx 1$. From Eq. (3.18),

$$T_b(k,0) \leq \frac{T_b(1,0)}{k} + \frac{k-1}{k}h \leq \frac{T_b(1,0)}{\eta k}$$

This results in:

[4] Our paper including the result similar to theorem 3.2 has been published in the *Proceedings of the 1984 International Conference on Parallel Processing* [LiW84a]. Lai and Sprague's paper, in which the major result is almost the same as theorem 3.2 except for the assumption that the maximal level, θ_{max}, is known, was published in the *Proceedings of the 1985 International Conference on Parallel Processing* [LaS85a] and the *IEEE Transaction Computers*, Oct. 1985 [LaS85b].

$$k \leq \frac{1-\eta}{\eta h} T_b(1,0) + 1$$

For instance, if η=0.9, h=50, and $T_b(1,0)=10^6$, then $k \leq 2223$. That is, a minimum of 0.9k speedup is obtained if 2223 or less processors are used.

3.4.3 Parallel Depth-First Searches

The performance of a parallel depth-first search can be analyzed in terms of the generalized heuristic function. The following theorem demonstrates that the bounds on $T_d(k,0)$ are looser than the corresponding bounds on $T_b(k,0)$.

Theorem 3.3: For a parallel depth-first search with k processors, ϵ=0, D=I, a generalized heuristic function of $h(P_i)$ = (path number), and $g(P_i) \neq f^*$ if P_i is not an optimal-solution node, then

$$\frac{T_b(1,0)-1}{k} + 1 \leq T_d(k,0) \leq \frac{T_d(1,0)}{k} + \frac{k-1}{k}(c+1)h \qquad (3.19)$$

where c is the number of the distinct incumbents obtained during the serial search.

Proof: The sequence of iterations obtained during a serial depth-first search can be divided into c+1 subsequences according to the c distinct monotonically decreasing incumbents obtained. Let the c feasible solutions and their parents be denoted as $F_1, F_2 \ldots F_c$, and $P_1, P_2 \ldots P_c$ and assume that $F_1, F_2 \ldots F_c$ are obtained in the i_1'th, i_2'th ... i_c'th iterations respectively. Then, iterations from 1 to i_1 belong to first subsequence, iterations from i_1+1 i_2 belong to second subsequence, and so on.

To begin with, let us consider the first subsequence of iterations. Suppose that the j'th iteration $(1 \leq j \leq i_1)$ is imperfect, only two cases are possible: either less than k nodes are selected due to the absence of enough active subproblems, or some nodes that are eliminated in the serial depth-first search are expanded in the parallel case. Note that in the latter case the path numbers of these nodes must be greater than $h(P_1)$. This is true because the initial incumbents that are either given to be infinite or yielded by the same heuristic algorithm for the serial and parallel

cases are identical, and if the path numbers of all expanded nodes in an iteration are less than $h(P_1)$, then this iteration is necessarily a perfect iteration. Let $h_{min}(x)$ be the minimum level in which some active nodes whose path numbers are less than $h(P_1)$ reside in the x'th iteration. Similar to the proof of Theorem 3.2, we have

$$h_{min}(x+1) \geq h_{min}(x) + 1$$

Consequently, after at most h imperfect iterations, the first feasible solution, F_1, that is better than the initial incumbent must be found.

Analogous to the above argument, we can prove that in the i'th $(1 < i \leq c)$ subsequence of iterations, at most h imperfect iterations will be encountered before a better feasible solution nodes is obtained. During the last subsequence of iterations, since the optimal solution has been generated, all imperfect iterations occur only if less than k nodes have been selected in an iteration. In other words, an imperfect iteration implies that all active nodes at that time will be expanded and only their descendents can be active in the next iteration. Hence, no active node remains after at most h imperfect iterations in the last subsequence. The previous analysis shows that at most (c+1) * h imperfect iterations can appear in a parallel depth-first search. Since at least one node in each parallel iteration belongs to Φ^1, the upper bound of $T_d(k,0)$ can be derived as:

$$T_d(k,0) \leq \frac{T_d(1,0)-(c+1)\cdot h}{k} + (c+1)\cdot h$$

To derive the lower bound, it is observed that all nodes whose lower bounds are less than the optimal solution i.e., all essential nodes in a serial best-first search, must be expanded in a parallel depth-first search. Since no nodes have lower bounds equal to optimal solution, it is easy to prove the lower bound of $T_d(k,0)$ in Eq. (3.19). □

If all feasible solutions are located in the maximum level of the OR-tree, then the condition in Lemma 3.2 is satisfied. This implies that all essential nodes must be expanded in a parallel depth-first search. Hence, we get a tighter lower bound of $T_d(k,0)$:

$$\frac{T_d(1,0)-1}{k}+1 \leqslant T_d(k,0) \qquad (3.20)$$

Theorem 3.3 predicts the performance of parallel depth-first searches. The comparison of best-first with depth-first searches with respect to speedup and anomalous behavior will be discussed in the next chapter.

3.4.4 Parallel Breadth-First Searches

The following theorem presents the bounds on $T_r(k,\epsilon)$.

Theorem 3.4: For a parallel breadth-first search with k processors, D=I, and satisfying the generalized heuristic functions in Eq's. (3.7) and (3.8), then

$$\frac{T_r(1,\epsilon)-1}{k}+1 \leqslant T_r(k,\epsilon) \leqslant \frac{T_r(1,\epsilon)}{k}+\frac{k-1}{k}(c+h) \qquad (3.21)$$

Proof: For a breadth-first search, regardless of whether approximate lower-bound tests are applied or not, there are two possible cases in which imperfect iterations may exist. First, when the number of active subproblems is less than k, an imperfect iteration occurs. These happen at most h times from our discussion in Theorems 3.2, and 3.3. Second, imperfect iterations may occur if there exist some nodes that are eliminated in the serial case by a feasible solution but are expanded simultaneously with the ascendant of the feasible solution in the parallel case. This is possible because in our parallel OR-tree algorithm, the feasibility test is performed after the selected nodes are expanded. The upper bound of $T_r(k,\epsilon)$ is, thereby, obtained as follows:

$$T_r(k,\epsilon) \leqslant \frac{T_r(1,\epsilon)}{k}+\frac{k-1}{k}(c+h)$$

On the other hand, in (approximate) breadth-first searches, all essential nodes have to be expanded in the parallel case (Lemma 3.2). This argument results in the lower bound of $T_r(k,\epsilon)$

$$\frac{T_r(1,\epsilon)-1}{k}+1 \leqslant T_r(k,\epsilon) \quad \square$$

Clearly, the upper bound of $T_r(k,0)$ is looser than that of $T_b(k,0)$ if $c>1$.

The performance bounds of parallel OR-tree searches derived in this section provide a foundation to determine the number of processors needed to assure a near-linear speedup. Choosing a proper number of processors to configure a parallel computer system is one of the first issues in designing a cost-effective computer architecture. Once the number of processors has been determined, we need to consider the connection of these processors. This will be discussed in the next two sections.

3.5 Computer Architecture for Parallel OR-Tree Search

A number of computer architectures have been proposed to support parallel OR-tree searches. In this section, we discuss MINIP, a multiprocessor for parallel best-first searches with (possibly) parallel dominance tests.

3.5.1 Previous Work on Parallel OR-Tree Searches

Several studies have been done on parallel processing of branch-and-bound algorithms with lower-bound tests. Desai used implicit enumeration to find the optimal solution of NP-hard problems [Des78]. The system dedicates one processor to each subproblem, and this processor reports to its parent processor when the evaluation is completed. Implicit enumeration is time-consuming and wasteful for large problems.

Imai, et al. studied parallel branch-and-bound algorithms with a depth-first search in a multiprocessor with shared memory [IFY79]. Depth-first search is used owing to memory limitations. Similarly, EL-Dessouki and Huen studied the same algorithm on a general-purpose network architecture with slow inter-processor communications [EIH80]. Recently, some computer architectures are being developed to support OR-parallelism of logic programming. Examples include PIE, a parallel inference engine [Mot84], and Bagof, an OR-Parallel token machine [CiH84]. In all of these projects, a depth-first search strategy is adopted.

It is well-known that a depth-first search is not effective in minimizing the execution time because a serial best-first search expands the minimum number of nodes when all lower bounds are distinct [LaW66]. However, the best-first search is space-consuming because all active subproblems must be stored as intermediate data in the computer. Currently, with the availability of VLSI technology, larger and inexpensive memories, and faster communication media, reducing the execution time becomes an important problem.

3.5.2 MANIP—A Multiprocessor for Parallel Best-First Search Without Dominance Tests

The architecture of MANIP is shown in Figure 3.4 [WaM84,WLY84]. It consists of five major components: selection and redistribution network, secondary storage, processors, global data register, and subproblem memory controllers.

The selection network selects subproblems with the minimum lower bounds for expansion in each iteration, and connects the memory controllers for load balancing. Excess subproblems that cannot be stored in the memory controllers are stored in the secondary storage. The memory controllers manage the local list of subproblems, maintain the secondary storage, and communicate with other controllers through the selection and redistribution network. The processors are general-purpose computers for partitioning subproblems and evaluating lower bounds. The global data register is a register accessible to all memory controllers and contains the value of the incumbent. To avoid contention during updates, this register can be implemented by a broadcast bus or a sequential associative memory. In the latter case, the minimum is found when the values of the feasible solutions

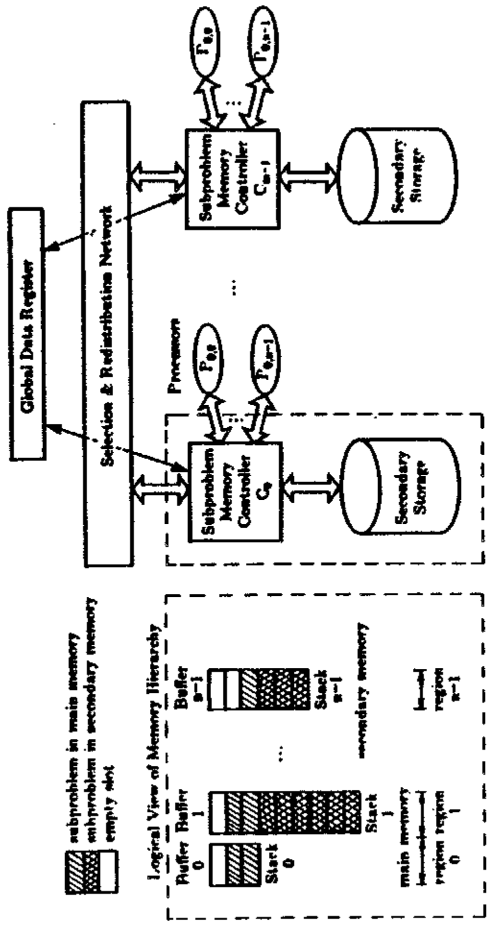


Figure 3.4 Architecture of MANIP—a multiprocessor for parallel best-first search [WLY85]

are shifted out bit-serially and synchronously from all processors.

Two difficult issues must be solved in a parallel best-first search. First, the k subproblems with the smallest lower bounds must be selected from the N active subproblems in the system. Selection by software requires a time overhead of $O(N)$ in each iteration. A practical multi-stage selection network for selecting k elements from N elements requires $O(\log_2 N \cdot \log_2 k)$ time complexity and $O(N \cdot \log_2^2 k)$ hardware complexity [WaC84]. A single-stage selection network may also be used. One or more subproblems with the minimum lower bounds in each processor are sent to the neighboring processors and inserted into their local lists. A maximum of $(k-1)$ shift-and-insert operations are needed to ensure that each processor has one of the k subproblems with the smallest lower bounds [WaM84]. Assuming that insertion is implemented in software, the time overhead in each iteration is $O(k \cdot \log_2 N)$. In all these cases, selection represents a significant overhead of the system.

In addition to the high overhead of selection, it is known that the selection rule is based on a fallible lower-bound heuristic. These suggest that it may be more efficient not to follow the selection rule strictly. A *No-Wait Policy* is proposed here. Instead of waiting for one of the k subproblems with the smallest lower bounds, each processor should expand the 'most promising' subproblem in its local memory and initiate a fetch of the 'most promising' subproblem from its neighbors. In this case, the 'most promising' subproblem is the one with the minimum lower bound.

When the k 'most promising' subproblems are randomly distributed among the processors, the average fraction of processors containing one or more of the 'most promising' subproblems is at least 0.63 [WaM84], resulting in a speedup proportional to 0.63k. However, as expansion proceeds, the distribution may become non-random, and an interconnection network is needed to randomize the distributions and balance the workload in the system. Experimental results on vertex-cover and knapsack problems have shown that the number of subproblems

expanded increases by about 10% when the above scheme is used in place of a complete selection. The performance is almost as good as that of a complete selection when the processors expand subproblems synchronously and perform one shift-and-insert operation for each subproblem expanded. The shift-and-insert operation can be overlapped with subproblem expansions and supported by a unidirectional ring network.

A second issue in implementing a best-first search lies in the management of the large memory space required. The multiprocessing model used to study this problem comprises of a CPU, a main memory, a slower secondary memory, and a secondary-memory controller. The expected completion time of the B&B algorithm on this model is taken as the performance measure.

A direct implementation involving an ordered list of pointers to the subproblems results in a poor locality of access because the subproblems are not ordered by lower bounds in the virtual space. A better alternative is a special virtual memory that tailors its control strategies according to the locality of access [YuW83]. However, this approach is inflexible as the parameters of the control strategies are problem dependent. The inadequacies of these approaches are again due to the strict adherence to the selection rule. The No-Wait Policy may also be applied here. This has resulted in the design of a modified B&B algorithm [YuW84].

In this modified algorithm, the range of possible lower bounds is partitioned into b disjoint regions (Figure 3.4). The subproblems in each region are maintained in a separate list. The top portion of each list resides in the main memory while the rest resides in the secondary memory. Due to the high overhead of secondary-storage accesses, subproblems in a list are expanded in a depth-first manner. To implement the No-Wait Policy, the *modified selection rule* chooses the subproblem in the main memory with the smallest lower bound for expansion. Since subproblems within a list are not sorted, the *lower-bound elimination rule* has to be modified. Suppose that the current incumbent lies in the range of List 0 , then all lists with indices greater than θ are eliminated. Subproblems in List θ with lower bounds greater than the incumbent are eliminated only when they are moved to the main memory during the expansion of List θ. As a result, it is necessary to carry out the lower-bound test on each selected subproblem before it is expanded.

The modified algorithm is identical to a depth-first search when one list is used, and a best-first search when infinitely many lists are used. In general, as the number of lists increases, the number of subproblems expanded decreases while the overhead of the secondary-memory accesses increases. The number of lists should be chosen to maximize the overlap between computations and secondary-memory accesses. This, in turn, depends upon the accuracy of the lower-bound function and the access times of the main and secondary memories. The accuracy of the lower-bound function is problem dependent and can be estimated from sample problems of the same type.

Experimental results on integer-programming and vertex-cover problems verify that the modified B&B algorithm is very useful. For vertex-cover problems, the lower-bound function is very accurate, so a depth-first search results in the best performance. For integer-programming problems, the lower-bound function is less accurate. As a result, more stacks (two to three) are needed for the best performance. The improvement in paging overhead over a direct implementation of the best-first search can exceed one hundred times.

Experience on MANIP and other previous studies has shown that the functional requirements for efficient evaluation of B&B algorithms with lower-bound tests only are a loosely coupled interconnection of processors with load balancing capability, and a method of concurrent update and broadcast of the incumbent.

3.5.3 Parallel Dominance Tests

When general dominance tests are used, it is necessary to keep the set of *current dominating nodes* (denoted by N_d) in memory. These are nodes that have been generated but have not been dominated so far. For the special case in which only lower-bound tests are used, N_d contains only one undominated solution node—the incumbent. In general, N_d could be larger than the set of active nodes. A newly generated node, P_i, has to be compared with all nodes in N_d to see if P_i or any node in N_d is dominated.

N_d can be stored in a bank of global data registers, if its size is small. However, centralized comparisons are inefficient when the size of N_d is large. N_d should then be partitioned into k subsets, $N_d^0, \ldots, N_d^{k-1}$, and distributed among the local memories of the k processors. A subproblem, $P_{i,j}$, generated in Processor i is first compared with N_d^i, and any subproblems in N_d^i dominated by $P_{i,j}$ are removed. If $P_{i,j}$ is not dominated by a subproblem in N_d^i, then it is sent to a neighboring processor, and the process repeats. $P_{i,j}$ will eventually return to Processor i if it has not been dominated by any node in N_d and will be inserted into N_d^i.

A difficult problem on balancing the dominance tests in the parallel processors has to be solved. Since the current dominating nodes are distributed in the k memory controllers randomly, it is possible that a large number of the current dominating nodes are stored in a particular memory controller while there are only few current dominating nodes in another memory controller. For a synchronous system, the memory controllers that have few current dominating nodes must wait for other processors when dominance tests are executed. One way to balance the workload is to shift the current dominating nodes before the parallel dominance tests begin. The following theorem shows an effective load balancing strategy.

Theorem 3.5: Let k and n_i, $0 \leq i < k$, be the number of memory controllers and current dominating nodes in controller i, respectively, and n be the average of all n_i's. Let the memory controllers be connected by a unidirectional ring. Suppose that a shift strategy is defined such that each controller shifts $(n_i - n)$ current dominating nodes into its neighbor in each iteration if $n_i > n$. Then regardless of how the current dominating nodes are distributed, each memory controller contains n current dominating nodes after at most k-1 iterations.

Proof: To simplify the representation, we use the urn model to analyze the shift process. The k controllers in the system are represented as k urns that contain a total of k·n marbles which stand for the current dominating nodes. An urn is said to be an *L, or E, or G urn* if it contains less than, or equal to, or greater than n marbles respectively. The number of urns in each class is varied as the marbles (subproblems) are shifted. Eventually all urns become urns of type-E. When Urn i is a type-G urn, Urn (i+1) is a type-L urn, and Urn i shifts $(n_i - n)$ marbles into Urn (i+1), then it is possible that both Urns i and (i+1) become urns of type-E. In this case, we say that Urn (i+1) becomes a type-E urn later than Urn i.

Without loss of generality, we assume that Urn 0 is the last urn to become a type-E urn. It is easy to see that Urn 0 cannot become a type-G urn during the shift process, that is, the number of marbles in Urn 0 is always less than n until all urns contain n marbles. Suppose the Urn 0 becomes a type-G urn during the shift process. It implies that at least one urn becomes a type-E urn later than Urn 0, which contradicts the assumption that Urn 0 is the last urn to becomes a type-E urn.

We now show that after each shift at least one type-E urn is generated, and that the number of generated type-E nodes always increase as the marbles are shifted. When all urns are not of type-E, there exists at least one type-G urn. Let Urn j be a G-type urn and that all urns between Urn 0 and Urn j are not of type-G. These urns must be type-E because Urn 0 cannot change into a type-G urn. Suppose that an Urn k (0<k<j) is a type-L urn, it cannot become a type-E urn.

Therefore, after a shift operation, Urn j must become an urn of type-E, and Urn j remaims type-E hereafter. Figure 3.5 shows snapshots of the load balancing process. For instance, in the third iteration (row 3), Urn 3 is an urn of type-G and all urns between Urn 0 and Urn 3 are urns of type-E. Urn 3 must become an urn of type-E after this iteration. The above argument can be applied (k-1) times. This means that after at most (k-1) shifts, all urns become urns of type-E, and all memory controllers contains the same number of dominating nodes after at most (k-1) iterations. □

The above theorem is restricted to the case when n is an integer. If n is not an integer, then the shift strategy has to be modified. If the fractional part of n is less than or equal to 0.5, then $\lfloor n_i - n \rfloor$ marbles are shifted in each iteration if $n_i > n$; otherwise $\lceil n_i - n \rceil$ marbles are shifted in each iteration. This strategy ensures that the difference in the numbers of marbles in any two urns are less than k/2 in the worse case. More accurate load balancing can be achieved by more complicated shift control. To find the average number of subproblems in a processor, a global controller is required to collect the information on workload of each processor.

To implement load balancing operations overlapped with branching operations, Step 3 and 7 of the B&B algorithm discussed in Section 3.2.4 should be adapted as follows.

(3)' (Decompose and load balance): Generate a child P_{i_j} of P_i; $N_d \leftarrow N_d \bigcup P_{i_j}$;

$U \leftarrow U \bigcup P_{i_j}$. Perform load balancing by shifting the current dominating nodes.

The two operations are carried out in overlap with each other.

(7)' (Terminate): If $j < k_i$, then go to Step 3'; otherwise go to Step 2.

If the number of current dominating nodes is very large, i.e., the dominance relation is weak and most nodes are incomparable, then the above parallel dominance-test scheme might be impractical due to the large amount of data communications. However, a lot of the known dominance relations exists for nodes in the same level of a branch-and-bound tree. For example, in the knapsack

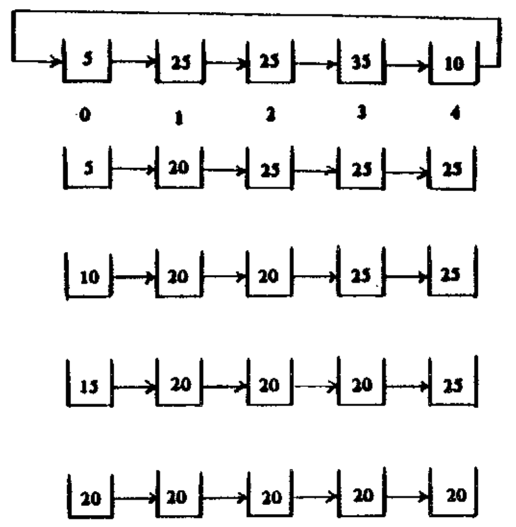


Figure 3.5 An example of load balancing in a ring network
(average = 20)

problem, the dominance relation is defined over the same subset of objects. In the n-job two-machine mean-completion time flow-shop problem [CON67] and the n-job one-machine scheduling problem with deadlines [SAH76], the dominance relations exist between subproblems with the same subset of jobs. In the problem of eight queens, the dominance relations exist only for subproblems with the same number of queens put on the chess board. In these cases the dominance tests can be implemented by comparing newly generated nodes with the current dominating nodes in the same level of the search tree. This will reduce the overhead for dominance tests substantially.

The functional requirements for implementing parallel dominance tests depend on the size of N_d and the structure of the dominance relation. When $|N_d|$ is small, unstructured dominance tests, in which a dominance relation can exist between any pair of nodes, can be implemented by broadcast busses or global registers. For structured dominance tests such as some dynamic-programming problems, it may be possible to partition the search tree and localize the dominance tests. System architectures with regular connections might be suitable for this kind of problems and will be investigated in Chapter 6. On the other hand, when $|N_d|$ is large, it is necessary to partition N_d into subsets and to perform the dominance tests in parallel. This results in tight coupling of the processors because the transfer of newly generated nodes between processors has to be synchronized and overlapped with computations. Load balancing should be considered when multiple lists of current dominance nodes are used.

3.6 Summary

In this chapter, we have derived the performance bounds of parallel best-first, depth-first, and breadth-first OR-tree searches, respectively. These bounds provide the theoretical foundation to determine the number of processors to assure a near-linear speedup. It is found that for best-first searches, the speedup is related to the problem complexity, which is reflected by $T_b(1,0)/h$. To guarantee a near-linear speedup, the number of processors must be much less than $T_b(1,0)$. For depth-first and breadth-first searches, the speedups are related to the number of incumbents obtained during the searches in addition to the problem complexity. Since the performance bounds on best-first searches are tighter than that on depth-first and breadth-first searches, the range within which the near-linear speedup is maintained is usually larger for best-first searches.

It has been found that, for parallel branch-and-bound algorithms without dominance tests, a loosely coupled architecture, such as a system connected by a ring network is cost-effective [WaM84]. This is true because the controllers do not have to be connected in order to evaluate a parallel branch-and-bound algorithm in the minimum expected completion time. However, if dominance tests are included in parallel OR-tree searches, loose coupling may be unsuitable. The corresponding functional requirements depend on the size of the current dominating nodes, N_d, When $|N_d|$ is small, dominance tests can be implemented by broadcast busses or global registers. On the other hand, when $|N_d|$ is large, it is necessary to partition N_d into subsets and to perform the dominance tests in parallel. This results in tight coupling of the processors. The functional requirements of parallel OR-tree searches are summaried in Table 3.1. When multiple lists of current dominanting nodes are used, load balancing is an important issue. An efficient load balancing strategy for multiprocessors with a ring network has been presented.

Table 3.1 Functional requirements of parallel OR-tree searches

Algorithm	Func. requirements		Tasks
Without dominance tests	Loosely coupled; broadcast capability		Balance load; share incumbents
With dominance tests	Small N_d	Broadcast busses or global registers	Balance load; share dominating nodes
	Large N_d	Tight coupling; syncronous control	Balance load; transfer new generated nodes

CHAPTER IV

COPING WITH ANOMALIES IN PARALLEL OR-TREE SEARCH

Up to now, efficiency analysis of parallel OR-tree search have been little studied. When comparing the efficiency of searching an OR-tree using k_2 processors against that of k_1 processors ($k_2 > k_1 \geq 1$), a k_2/k_1-fold speedup (ratio of the execution time in the two cases) is expected. However, simulations have shown that the speedup for parallel OR-tree searches with respect to k_2 and k_1 processors can be (a) less than one—called a *detrimental anomaly* [IFY79,LaS84,Moh83]; (b) greater than k_2/k_1— called an *acceleration anomaly* [IMA79,LAI84]; or (c) between one and k_2/k_1— called a *deceleration anomaly* [IFY79,WaMH81,LaS84,Moh83]. Similar anomalous behavior have been reported by others. For instance, the achievable speedup for AND/OR-tree searches is limited by a constant (5 to 6) independent of the number of processors used (parallel-aspiration search) or $\sqrt{k}$ with k processors (tree-splitting algorithm) [Fin82]. So far, all known results of parallel tree searches showed that a near-linear speedup holds only for a small number of processors.

4.1 Model of Efficiency Analysis

For simplicity, the case of searching a single optimal (or suboptimal) solution is discussed in this chapter. The case in which all solutions are sought can be analyzed similarly. The parallel computational model used here consists of a set of ~~processors connected to a shared memory~~. It is assumed that the processors operate synchronously in executing the steps of the parallel OR-tree search algorithm. In

each iteration of the algorithm, multiple subproblems are selected and decomposed. The newly generated subproblems are tested for feasibility (and the incumbent updated if necessary), eliminated by (exact or approximate) lower-bound tests and dominance tests, and inserted into the active list(s) if not eliminated. In this model, eliminations are performed after branching instead of after selection as in Ibaraki's algorithm [Iba76a].

The shared memory in the proposed computational model may seem to be a bottleneck of the system. However, this model can be transformed into a second model in which all processors have a private memory and are connected by a ring network. In Section 3.5.2, we have discussed that the efficiency of the parallel OR-tree search algorithm in the transformed model is very close to that of the original model, except that very little interprocessor communication and interference are involved [WaM84,WLY84]. Moreover, communications and computations can be overlapped. Since subproblems are decomposed synchronously and the bulk of the overhead is on branching operations, the number of iterations, which is the number of times that subproblems are decomposed in each processor, is an adequate measure in both the serial and parallel models. The *speedup* is thus measured by the ratio of the number of iterations with respect to the different number of processors used. Once the optimal solution is found, the time to drain the remaining subproblems from the list(s) is not accounted for. This is reasonable because the predominant overhead in an OR-tree search is in the branching operations.

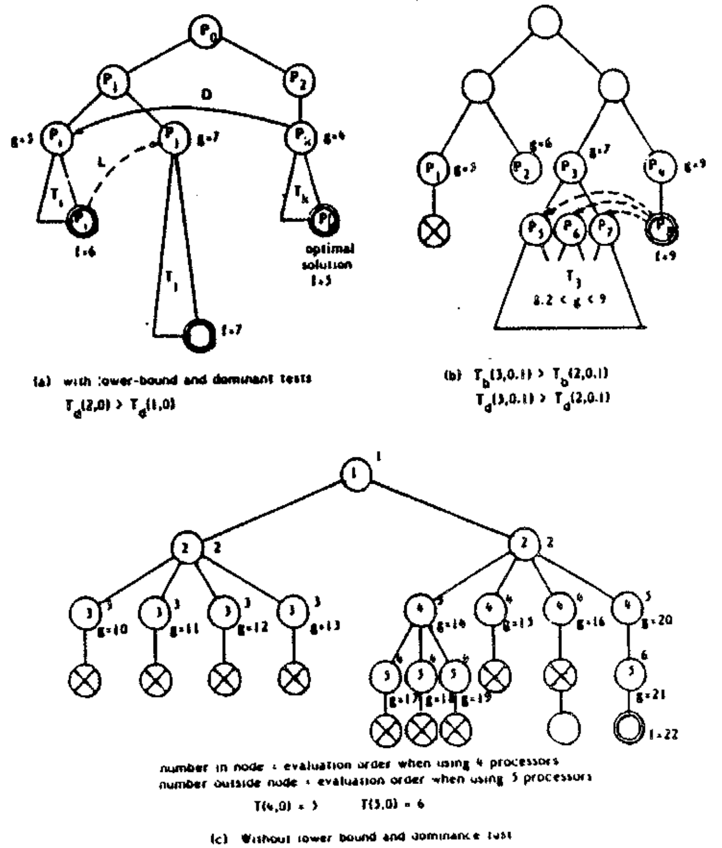
The objective of this chapter is to study conditions to cope with the anomalous behavior of parallel tree search algorithms under approximations and parallel processing. Anomalies are studied with respect to the same search strategy. In general, anomalies should be studied with respect to the best serial algorithm and the best parallel algorithm (with possibly a different search strategy than that of the serial algorithm). However, conditions to resolve these anomalies would be problem dependent and may result in a large number of cases that cannot be enumerated. Since the conditions to resolve anomalies between n_1 and n_2 processors, $1 \leq n_1 < n_2$, are different from those between serial and parallel processing, we discuss the different conditions separately. These results on resolving anomalies are useful for designers to understand the existence of anomalies and to modify existing algorithms to prevent detrimental anomalies and enhance acceleration anomalies.

4.2 Anomalies of Parallel OR-Tree Search

In this section, some anomalies on parallel OR-tree searches are illustrated. Let $T(k,\epsilon)$ denote the number of iterations required for expanding an OR-tree, where k is the total number of processors used, and ϵ is the allowance function specifying the allowable deviation of a suboptimal value from the exact optimal value. In this chapter, except Section 4.4.5, a single list of subproblems is assumed.

For the OR-tree in Figure 4.1(a), the order of nodes expanded using a depth-first search on a single processor is P_0, P_1, P_i, all the nodes in T_i that result in a feasible solution $P_{i'}$ with value 6, P_k, and all the nodes in T_k that result in the optimal solution of $f(P_{k'})$=5. P_j is terminated by the lower-bound test using $P_{i'}$. In contrast, when two processors are used, P_1 and P_2 are expanded concurrently. After the expansion, P_i is terminated as a result of dominance by P_k. Since T_i is terminated, T_j is expanded next. If subtree T_j is large, the combined time of expanding T_i and T_k using one processor can be smaller than the combined time of expanding T_j and T_k using two processors.

Figure 4.1(b) illustrates a detrimental anomaly under a depth-first or best-first search with approximations. When two processors are used, the optimal solution, $f(P_8)$, is found, in the fourth iteration. Assuming that the lower bounds of P_5, P_6, and P_7 are greater than 8.2 and $\epsilon = 0.1$, all nodes in the subtree T_3 will be eliminated, since $9/(1+\epsilon)<8.2$. However, when using three processors, P_3 is expanded in the third iteration and P_5, P_6, and P_7 will be selected in the next



(a) with lower-bound and dominant tests
$T_d(2,0) > T_d(1,0)$
optimal solution f=5
g=5 g=7 g=6 f=6 f=7
D
L
(b) $T_b(3,0.1) > T_b(2,0.1)$
$T_d(3,0.1) > T_d(2,0.1)$
$8.2 < g < 9$
g=5 g=6 g=7 g=9 f=9
number in node : evaluation order when using 4 processors
number outside node : evaluation order when using 5 processors
$T(4,0) = 5$ $T(5,0) = 6$
(c) Without lower bound and dominance test
g=10 g=11 g=12 g=13 g=14 g=15 g=16 g=20 g=21 g=17 g=18 g=19 f=22

Figure 4.1 Examples of detrimental anomalies

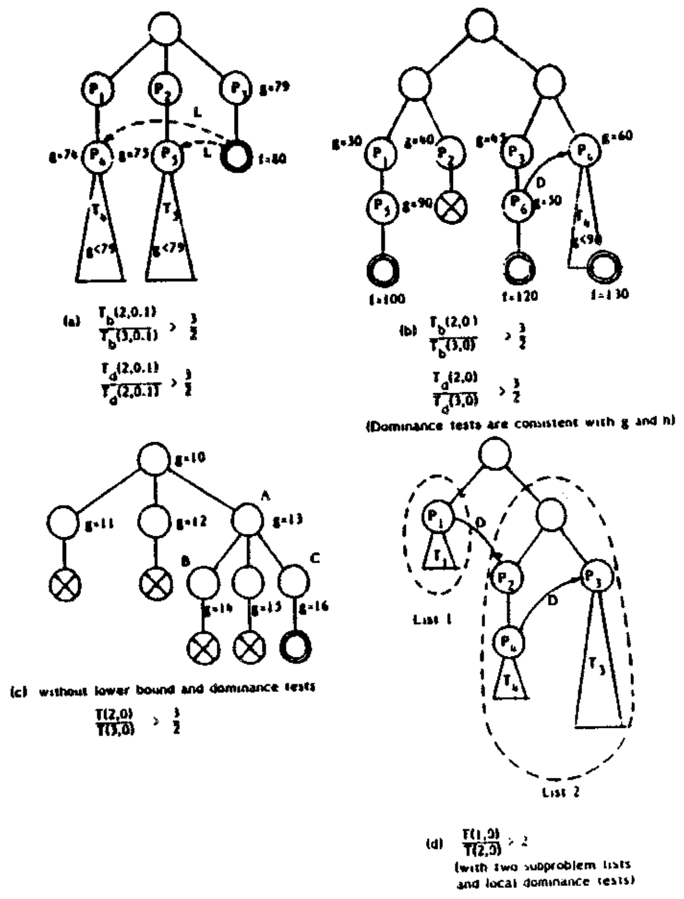
iteration, assuming that the lower bounds of all descendent nodes of P_3 are less than $g(P_4)$, 9. If the subtree T_3 is large, $T(2,\epsilon) < T(3,\epsilon)$ will occur.

An example of an acceleration anomaly with approximation under a depth-first or best-first search is shown in Figure 4.2(a). When using three processors, the solution is obtained in the second iteration, and P_4 and P_5 are eliminated; while if two processors are used, subtrees T_4 and T_5 have to be expanded. The ratio $T(2,\epsilon)/T(3,\epsilon)$ will be much larger than 3/2 if the subtrees T_4 and T_5 are very large.

Figure 4.2(b) illustrates another example of acceleration anomaly with dominance tests under a best-first search. When three processors are used, P_4 will be dominated by P_6, and only four iterations are required to complete the search. In contrast, when using two processors, this dominance relation is not active because P_4 has been expanded before P_6 is generated and no dominance relation exists between P_6 and the descendents of P_4 (which is quite possible). Therefore, subtree T_4 needs to be searched and $T(2,0)/T(3,0) > 3/2$.

Anomalies may occur even when both the lower-bound and dominance tests are not active. Figure 4.1(c) shows a detrimental anomaly under both depth-first and best-first searches. The example in Figure 4.2(c) confirms that acceleration anomaly may occur regardless of the search strategy. Figure 4.2(d) shows an acceleration anomaly due to local dominance tests and will be described in Section 4.4.5.

Many anomalous examples can be created for various combinations of search strategies, allowance and elimination rules. However, the important consideration here is not in knowing that anomalies exist, but in understanding why these anomalies occur. It is desirable to find the sufficient conditions to ensure that $T(k_2,\epsilon) \leq T(k_1,\epsilon)$ as well as the necessary conditions for $T(k_1,\epsilon)/T(k_2,\epsilon) \geq k_2/k_1$. The necessary conditions to eliminate detrimental anomalies are not evaluated because they are problem dependent. A condition necessary for avoiding detrimental anomalies depends on the sequence of nodes expanded and the size of the resulting subtrees. There are many possible combinations, and it is difficult to



P1
P2
P3
g=79
L
g=74
P4
g=73
P5
f=80
T4
T5
g<79
(a) $\frac{T_b(2,0.1)}{T_b(3,0.1)} > \frac{3}{2}$
$\frac{T_d(2,0.1)}{T_d(2,0.1)} > \frac{3}{2}$
g=30
g=40
g=45
g=60
P1
P2
P3
P4
P5
P6
g=90
g=50
D
T4
g<90
f=100
f=120
f=130
(b) $\frac{T_b(2,0)}{T_b(3,0)} > \frac{3}{2}$
$\frac{T_d(2,0)}{T_d(3,0)} > \frac{3}{2}$
(Dominance tests are consistent with g and h)
g=10
g=11
g=12
g=13
A
B
C
g=14
g=15
g=16
(c) without lower bound and dominance tests
$\frac{T(2,0)}{T(3,0)} > \frac{3}{2}$
D
T1
List 1
List 2
T3
T4
(d) $\frac{T(1,0)}{T(2,0)} > 2$
(with two subproblem lists and local dominance tests)

Figure 4.2 Examples of acceleration anomalies

enumerate them for a given problem. Furthermore, the necessary conditions developed for one problem cannot be generalized to other problems. These are also the reasons for not evaluating the sufficient conditions to preserve acceleration anomalies.

4.3 Coping With Serial-to-Parallel Anomalies

In this section, we study the conditions to cope with serial-to-parallel anomalies first, The conditions to cope with general anomalies between using k_1 and k_2 processors will be discussed in next section.

One of the motivations of addressing this problem is to improve Lai and Sahni's results on anomalies of parallel B&B algorithms [LaS84]. First, Lai and Sahni have made an implicit assumption that all nonterminal nodes of the B&B tree have identical lower bounds, and hence the nonterminal nodes can be expanded in any order. Second, Lai and Sahni have only considered best-first searches and finding optimal solutions. However, our theoretical analysis and simulations have shown that anomalies are infrequent when optimal solutions are sought using best-first searches, while they are frequent in approximate B&B algorithms with depth-first searches [LiW84a]. We will prove conditions that can avoid detrimental anomalies even when approximations are allowed. Since the anomalous behavior depends on the search strategies, we will investigate the anomalies with respect to the various search strategies separately. Lastly, Lai and Sahni have claimed that a near-linear speedup for parallel B&B algorithms with best-first search holds only for a "small" number of processors. On the contrary, we have shown that a near-linear speedup may hold for a large number of processors, and that the maximum number of processors to attain a near-linear speedup can be predicted [LiW84b].

4.3.1 Sufficient Condition to Eliminate Detrimental Anomalies

(a) Finding an Exact Optimal solution

In this section, we show that when the optimal solution is sought, $T(k,\epsilon) \leqslant T(1,\epsilon)$ holds if the heuristic function is monotone and unambiguous, and the dominance relation satisfies some consistency requirements on the lower-bounds and heuristic functions. At first, we explain some concepts that are important for the following proof. When lower-bound and dominance tests are used together, it is important to note that dominance tests are not transitive with the lower-bound tests. That is, $P_i \, L \, P_j$ and $P_j \, D \, P_k$ do not imply $P_i \, L \, P_k$. Similarly, $P_i \, D \, P_j$ and $P_j \, L \, P_k$ do not imply $P_i \, D \, P_k$. In both cases, only $f(P_i) \leqslant f(P_k)$ can be deduced. Recall that $P_j \, D \, P_k$ implies $f(P_j) \leqslant f(P_k)$. The converse is not true because some nodes are incomparable; otherwise, the number of active nodes can always be reduced to one.

To combine the dominance and lower-bound tests, conditions are defined for a special class of dominance relations. A dominance relation D is said to *consistent* with the heuristic function h if $P_i \, D \, P_j$ implies that $h(P_i) < h(P_j)$ for all $P_i, P_j \in \mathbf{P}$. A dominance relation D is said to *consistent* with the lower-bound function if $P_i \, D \, P_j$ implies that $g(P_i) \leqslant g(P_j)$ for all $P_i, P_j \in \mathbf{P}$. To show that $T(k,0) \leqslant T(1,0)$, it is necessary to prove: (a) that at least one node that belongs to Φ^1 is expanded in each iteration of the parallel search; and (b) that once all the nodes in Φ^1 are expanded or terminated, the parallel heuristic search must terminate.

Theorem 4.1: $T(k,0) \leqslant T(1,0)$ holds for heuristic searches that satisfy Eq's (3.7) and (3.8) and the following conditions: (a) Dominance relation D is consistent with lower-bound function g, and (b) Dominance relation D is consistent with h.

Proof: see Appendix I. □

The detrimental anomaly illustrated in Figure 4.1(a) for a depth-first search is caused by the inconsistency of D with h. The requirement on the consistency of the dominance relation with respect to g is satisfied in many practical problems, such as the shortest-path problem [Iba77a], the traveling-salesman problem

[MoM76], the n-job two-machine mean-finishing-time flowshop problem [CMM67], and the n-job one-machine scheduling problem with deadlines [Sha76]. However, the requirement on the consistency of the dominance relation with h may not be satisfied. For instance, the dominance relation for the 0-1 knapsack problem is consistent with the upper-bound function (instead of the lower-bound function for maximization problem). However, the conventional definition of dominance relations for the 0-1 knapsack problem may result in detrimental anomalies, since when the profit and weight of P_i is the same as the corresponding profit and weight of P_j, it is possible that either $P_i D P_j$ or $P_j D P_i$. Suppose that $h(P_i) < h(P_j)$ and that $P_j D P_i$ is allowed, then D is inconsistent with h. Detrimental anomalies can be avoided if we redefine the dominance relation as follows. For P_i and P_j defined on a given subset of objects, $P_i D P_j$ if (1) the profit of P_i is larger than that of P_j and the total weight of P_i is less than that of P_j; or (2) the profit and weight of P_i are equal to the corresponding profit and weight of P_j and the path number of P_i is less than that of P_j. The above example demonstrates that Theorem 4.1 is a useful guide to defining dominance relations that can eliminate detrimental anomalies.

In the special case of D=I, Theorem 4.1 shows that a monotone and unambiguous heuristic function guarantees that $T(k,0) \leq T(1,0)$.

(b) Finding an Approximate Solution

When approximate OR-tree search algorithms are considered, Theorems 4.1 are no longer valid. The reason for the anomaly is that the lower-bound tests under approximation, L, are not transitive. That is, $P_i L P_j$ and $P_j L P_k$ do not imply $P_i L P_k$, since $f(P_i)/(1+\epsilon) \leq g(P_j)$ and $f(P_j)/(1+\epsilon) \leq g(P_k)$ imply $f(P_i)/(1+\epsilon)^2 \leq g(P_k)$ rather than $f(P_i)/(1+\epsilon) \leq g(P_k)$. Somewhat surprisingly, it is possible that Φ^1 and Φ^k are almost disjoint, and most of the nodes in Φ^1 are not expanded in the parallel case. However, for best-first and breadth-first searches, detrimental anomalies can be avoided under the conditions as stated in following theorem. Since approximation tests do not affect the dominance tests, the conditions with respect to dominance tests for avoiding detrimental anomalies are the same as those in Theorem 4.1. In the proofs of Theorems 4.2 and 4.3, the part related to the dominance tests will be omitted.

Theorem 4.2: $T(k,\epsilon) \leq T(1,\epsilon)$, $\epsilon>0$, holds for best-first and breadth-first searches when: (a) the heuristic function, h, satisfies Eq's (3.7) and (3.8); and (b) the dominance relation, D, is consistent with h and g.

Proof : The key idea of this proof is to show that a detrimental anomaly cannot occur although transitivity of lower-bound tests is not valid here. For a best-first search, if P_{i_3} is a basic node in the parallel case and $P_{i_2} L P_{i_1}$ is true in the serial case (see Figure 3.3), then it is true that $g(P_{i_3}) \leq g(P_{i_1})$ since P_{i_3} must be selected before P_{i_1} and the selection is based on the lower bounds. Suppose that $P_{i_4} L P_{i_3}$ is true in the parallel case, we have $f(P_{i_4})/(1+\epsilon) \leq g(P_{i_3}) < g(P_{i_1})$, and P_{i_1} must be eliminated in the parallel case. This is a contradiction!

In serial and parallel breadth-first searches, the order of occurrence of the feasible solutions are the same even though $\epsilon > 0$. Hence, the basic node in each iteration of the parallel case must be an essential node because it is not eliminated by feasible solutions found before selecting it in the serial case.

The above arguments proved that the basic node in each iteration belong to Φ^1 in approximate best-first and breadth-first searches. In both cases, once all the essential nodes are expanded, the suboptimal solution is necessarily found and all remaining nodes have to be terminated. This completes the proof. □

For a depth-first search, the conditions of Theorem 4.2 are not sufficient. Theorem 4.3 below presents the corresponding sufficient conditions for approximate depth-first searches.

Lemma 4.1 In a parallel depth-first search using the path numbers as heuristic values, if node P_i at level l is expanded, then all nodes whose heuristic value is less than $h(P_i)$ and whose level number is less than or equal to l must be either expanded or terminated.

Proof: Suppose that there exists an active node P_j located in the j'th level, $j \leq l$, and that $h(P_j) < h(P_i)$ when P_i is expanded. Suppose that P_k is an ancestor node of P_j located in level k ($k<l$), and that P_k is active when P_i is selected. From the defined selection rule and Eq. (3.7), P_k and P_i have to be expanded simultaneously since $h(P_k) \leqslant h(P_i)$. Let P_m be the first common ancestor node of P_i and P_j. From the definition of path numbers, all ancestor nodes of P_j have path numbers smaller than that of ancestor nodes of P_i until P_m is encountered. Therefore, the parent of P_k must be expanded simultaneously with the parent node of P_i for the same reason stated above. Carrying out this process repeatly will yield a conclusion that node P_m will be expanded simultaneously with its descendent node. This, obviously, is impossible. □

The above lemma shows that in the active subproblem list the level numbers of the active nodes are non-increasing as the heuristic values increase. It also shows that if P_i lies at the bottom level of the OR-tree and a node P_j, which is not at the bottom level, is eliminated by P_i, then $h(P_j)$ must be greater than $h(P_i)$.

Theorem 4.3 For a depth-first search with approximate lower-bound tests, in addition to Conditions (a) and (b) of Theorem 4.2, all feasible solutions must be located at the bottom level of the OR-tree to prevent detrimental anomalies.

Proof: Suppose that a basic node P_1 is eliminated by a feasible solution P_2 in the serial case and not in the parallel case. This means that a feasible solution P_4 exists in the parallel case, and the parent node P_3 of P_2 must be eliminated by P_4. Since all feasible solutions lie at the bottom level of the OR-tree, $P_4 L P_3$ implies that $h(P_4)$ and $h(P_3)$ are less than $h(P_3)$ according to Lemma 3.1, where P_3 is the parent of P_4. For the same reason, another feasible solution P_6 must eliminate P_5 in the serial case. Applying this argument repeatly will contradict the fact that the number of feasible solutions are finite and that the feasible solution with the smallest path number has to occur in both the serial and parallel cases. □

From another point of view, the reasonableness of Theorem 4.3 can be visualized as follows. For serial and parallel OR-tree searches, the order of

occurrence of the feasible solutions is the same regardless of ϵ and number of processors used. Like breadth-first searches (proof of Theorem 4.2), this guarantees that $T_d(k,\epsilon) \leqslant T_d(1,\epsilon)$.

4.3.2 Necessary Conditions to Allow Acceleration Anomalies

In this section, the necessary conditions for $T(k,0) < T(1,0)/k$ are developed. One of these conditions is based on the complement of the special class of dominance relations defined in Section 4.3.1. A dominance relation, D, is said to be *inconsistent* with h if there exist two nodes P_i and P_j such that $P_i D P_j$ and $h(P_i) > h(P_j)$. Another condition is based on the complete consistency of heuristic functions. A heuristic function h is said to be *consistent (completely consistent)* with the lower-bound function g if $h(P_i) < h(P_j)$ implies that $g(P_i) \leqslant g(P_j)$ ($g(P_i) < g(P_j)$) for all $P_i, P_j \in P$. A heuristic function, h, is said to be *not completely consistent* with g if there exist two nodes P_i and P_j such that $h(P_i) > h(P_j)$ and $g(P_i) \leqslant g(P_j)$. Note that if $g(P_i)=g(P_j)$ is allowed, then the heuristic function for a best-first search is consistent, but not completely consistent, with the lower-bound function.

Theorem 4.4: The necessary condition for $T(k,0) < T(1,0)/k$ is either (a) that the heuristic function is not completely consistent with g, or (b) that the dominance relation is inconsistent with h.

Proof: (See Appendix 2) □

According to Theorems 4.1 and 4.4 we can conclude that if the dominance tests are active then both detrimental and acceleration anomalies exist when a breadth-first or depth-first search is used. It is important to note that the conditions in Theorem 4.4 are not necessary when approximate solutions are sought, i.e., acceleration anomalies may occur even though h is completely consistent with g and the dominance tests are consistent with h. The corresponding necessary conditions are studied in Section 4.4.3.

4.4 Coping With General Anomalies

We have derived three theorems for the performance bounds with respect to different search strategies in Chapter 3. According to these theorems we can investigate the relative efficiency between using k_1 and k_2 processors. In this section, $1 < k_1 < k_2$ is assumed unless stated otherwise. First, we discuss the simple case in which $\epsilon = 0$ and D=I. Then strategies with approximate lower-bound and dominance tests will be studied.

4.4.1 Sufficient Conditions to Eliminate Detrimental Anomalies

In the following sections, we investigate some conditions to cope with anomalies with respect to k_1 and k_2 processors assuming that $\epsilon = 0$ and D=I.

First, we derive a sufficient condition to assure the monotonic increase in computational efficiency with respect to the number of processors.

Corollary 4.1: Suppose that a parallel best-first search satisfies the assumptions of Theorem 3.2, then $T_b(k_2,0) \leqslant T_b(k_1,0)$ when

$$T_b(1,0) \geqslant \frac{k_1(k_2-1)h - k_2(k_1-1)}{k_2-k_1}. \tag{4.1}$$

Proof: From Theorem 3.2, at least $\left|\frac{T_b(1,0)-1}{k_1} + 1\right|$ iterations are needed for k_1 processors and at most $\left|\frac{T_b(1,0)}{k_2} + \frac{k_2-1}{k_2} h\right|$ iterations are needed for k_2 processors to find an optimal solution in a given branch-and-bound tree. The sufficient condition for $T(k_2,0) < T(k_1,0)$ is:

$$\frac{T_b(1,0)}{k_2} + \frac{k_2-1}{k_2} h \leqslant \frac{T_b(1,0)-1}{k_1} + 1 \text{ or}$$

$$T(1,0) \geqslant \frac{k_1(k_2-1)\,h - k_2(k_1-1)}{k_2-k_1} \quad \square$$

For the example in Figure 4.1(c), $T_b(1,0)=15$, $k_1=4$, $k_2=5$, and $h=4$. Eq. (4.1) is not satisfied, and hence an increase in the number of processors from four to five may not ensure an increase in performance.

Corollary 4.1 shows that monotonic increase in performance depends on h and $T_b(1,0)$. h is a function of the problem size. However, $T_b(1,0)$ reflects the complexity of the problem to be solved and is unknown before the solution is found. It can be estimated in a similar way as in the analysis of Alpha-Beta pruning algorithms [KnM75, Bau78b, Pea82] which defines a branching factor α:

$$\alpha = \lim_{h\to\infty}[T_b(1,0)]^{\frac{1}{h}} \qquad \text{i.e., } \alpha^h \approx T_b(1,0)$$

The branching factor measures the average number of branches of an essential node and can be estimated statistically. For example, α is close to one for knapsack problems ($\alpha \leqslant 1.1$) if all the profits and weights are independent and are generated from uniform distributions.

Since $k_2^2 > k_1(k_2-1) \geqslant k_1(k_2-1)/(k_2-k_1)$, Eq. (4.1) is true if $T_b(1,0) \geqslant k_2^2 h$. In other words, there is always a monotonic increase in performance for all $1 \leqslant k_1 < k_2 \leqslant \sqrt{T_b(1,0)/h}$. For example, for h=50, $T_b(1,0)=10^6$, there will not be any risk of detrimental anomalies for any combinations of $1 \leqslant k_1 < k_2 \leqslant 141$. This example shows that if the complexity of the problem is high, i.e., the branch factor is large, then detrimental anomalies can be avoided in a wide range of the number of processor for parallel best-first searches.

Eq. (4.1) can be converted into a simpler form by ignoring the term $k_2(k_1-1)$ and defining $r_k = \frac{k_2}{k_1}$, that is,

$$\frac{T_b(1,0)}{h} > \frac{k_2-1}{r_k-1}$$

The term $\frac{T_b(1,0)}{h}$ can be viewed as the "average width" of the OR-tree that consists of essential nodes. Intuitively, detrimental anomalies can be prevented if the OR-tree searched is "wide" enough. On the other hand, for a given problem, the average width is fixed. If k_2 is not too large and k_2/k_1 is sufficiently large, then

$T_b(k_2,0)$ will be less than $T_b(k_1,0)$.

Similarly, for a depth-first search, the corresponding sufficient conditions can also be determined. However, the sufficient conditions are stronger than Eq. (4.1) due to the looser bounds of $T_d(k,0)$ as derived in Theorem 3.3.

Corollary 4.2: Let $r = T_d(1,0)/T_b(1,0)$. In a parallel depth-first search that satisfies the assumptions of Theorem 3.3, $T_d(k_2,0) \leq T_d(k_1,0)$ when:

$$\frac{k_2}{k_1} > r \text{ and } T_b(1,0) > \frac{k_1(k_2-1)(c+1)h-k_2(k_1-1)}{k_2-r\cdot k_1} \tag{4.2}$$

$$\text{or } T_d(1,0) > \frac{k_1(k_2-1)(c+1)h-k_2(k_1-1)}{k_2/r-k_1}$$

where c is the number of the distinct incumbents.

Proof: Similar to the proof of Corollary 4.1[5]. □

From Corollary 4.2, we can conclude that the existence of detrimental anomalies depends on $T_b(1,0)$, r $(T_b(1,0)/T_d(1,0))$, and c (the number of the distinct incumbents). If $r \approx 1$, c is small, and $T_b(1,0)$ is very large, then Eq. (4.2) may be satisfied. Our simulation results reveal that for some problems, such as the 0-1 knapsack and vertex-cover problems, $T_d(1,0)$ is close to $T_b(1,0)$, thereby $r \approx 1$. Moreover, if the feasible-solution values must be integers, c is often small. For this kind of problems, detrimental anomalies can be prevented for parallel depth-first searches when $T_b(1,0)$ is very large. It is a certainty that the range within which no detrimental anomalies occur for depth-first searches is smaller than that for best-first searches.

The following theorem shows that $T_f(k_2,\epsilon) \leq T_f(k_1,\epsilon)$ is true. Note that this is not a Corollary of Theorem 3.4.

[5] The proofs of Corollaries 4.2-4.8 are omitted due to the similarity with that of Corollary 4.1.

Theorem 4.5 In a parallel breadth-first search, if $D = I$ and the generalized heuristic function satisfies Eq's (3.7) and (3.8), then $T_f(k_2,\epsilon) \leq T_f(k_1,\epsilon)$.

Proof: Suppose that there are m imperfect iterations when k_2 processors are used. We divide the sequence of iterations into (m+1) subsequences, each of which ends in an imperfect iteration. Let $T_f^j(k_2,0)$ be the number of iterations from the first imperfect iteration (expanding root node) to the j'th one when k_2 processors are used. Further, when k_1 processors are used, let $T_f^j(k_1,0)$ be the number of iterations from the first imperfect iteration to one that includes at least one node belonging to the j'th imperfect iteration of using k_2 processors. We now show by induction on j that $T_f^j(k_2,0) \leq T_f^j(k_1,0)$. The basis, j=1, is trivial. Assuming that the induction hypothesis is true for j=n; consider the (n+1) subsequence of iterations. Note that all nodes expanded in this subsequence except for the last imperfect iteration have to be expanded when k_1 processors are used, since all these nodes belong to Φ^1. Suppose that there are a total of x $(x \geq 0)$ perfect iterations in the (n+1)'th subsequence when k_2 processors are used, then the total number of iterations in this subsequence when k_1 processors are used is greater than $\frac{k_2 x}{k_1}$, which is equal to or greater than (x+1). Invoking the inductive hypothesis, we obtain $T_f^{n+1}(k_2,0) \leq T_f^{n+1}(k_1,0)$. According to the induction above and considering that all iterations in the last subsequence are perfect, we conclude that $T_f(k_2,0) \leq T_f(k_1,0)$ under the given conditions. □

4.4.2 Necessary Conditions to Allow Acceleration Anomalies

From Theorem 3.2, we also can derive a necessary condition of acceleration anomalies with respect to k_1 and k_2 processors for parallel best-first searches.

Corollary 4.3: In a parallel best-first search that satisfies the assumptions of Theorem 3.2, $T_b(k_1,0)/T_b(k_2,0) > k_2/k_1$ only if

$$h > \frac{k_2-1}{k_1-1} \tag{4.3}$$

Note that this necessary condition cannot be obtained from the looser lower bound $T_b(k,0) \geqslant T_b(1,0)/k$ as discussed by Lai and Sahni [LaS84]. Figure 4.2(b) illustrate an acceleration anomaly, $T_b(2,0)/T_b(3,0) > 3/2$, which obviously satisfies the Eq. (4.3). Usually, if k_1 and k_2 are close to each other and h is large, then acceleration anomalies may occur in practice, even if the best-first search is adopted and all lower bounds are distinct. However, for best-first searches, $T_b(k_1,0)/T_b(k_2,0)$ cannot be much larger than k_2/k_1 due to the tight bound of $T_b(k,0)$.

For depth-first searches, the necessary condition for acceleration anomalies is given in the corollary below.

Corollary 4.4: In a parallel depth-first search that satisfies the assumptions of Theorem 3.3, $T_d(k_1,0)/T_d(k_2,0) > k_2/k_1$, $1 \leqslant k_1 \leqslant k_2$, is true only if

$$T_d(1,0) - T_b(1,0) > k_2 - (k_1-1)(c+1)h - 1 \tag{4.4}$$

Obviously, the necessary condition in Eq. (4.4) is readily satisfied and $T_d(k_1,0)/T_d(k_2,0)$ may be much greater than k_2/k_1.

Analogous to Corollaries 4.3 and 4.4, we can derive the the necessary condition from Theorem 3.4 for acceleration anomalies to exist in a breadth-first search:

$$c + h > \frac{k_2-1}{k_1-1} \tag{4.5}$$

This condition is also valid for breadth-first searches with approximate lower-bound test.

4.4.3 Coping With Anomalies under Approximation

It should be noted that Theorem 4.2 is no longer valid when comparing $T_b(k_1,\epsilon)$ and $T_b(k_2,\epsilon)$. Figure 4.1(b) gives a counterexample showing that, even if a best-first search is adopted and all lower bounds are distinct, $T_b(3, 0.1) > T_b(2, 0.1)$. This happens because a node with a large lower bound (not a basic node) may be expanded before nodes with smaller lower-bound in the parallel case, and nodes with smaller lower bounds may be eliminated by approximate lower-bound tests. A stronger sufficient condition is required in this case. To find the new sufficient condition, we need to update Theorem 3.2 first. Analogous to the proof of Theorem 3.2, the upper bound on $T_b(k,\epsilon)$ can be derived. We now try to refine the lower bound of $T_b(k,\epsilon)$. Let f_0 be the optimal-solution value and $MINT_b(\epsilon)$ be the minimum number of nodes to be expanded in the approximate OR-tree search algorithms. $MINT_b(\epsilon)$ is equal to the number of nodes whose lower bounds are less than $f_0/(1+\epsilon)$ because, in the best case, only these nodes will be expanded. Clearly, $MINT_b(\epsilon)$ depends on the distribution of the lower bounds. The point to be made here is that the assumption on lower-bound tests in Lemma 3.2 is not satisfied for approximate best-first searches. Essential nodes, consequently, might be eliminated in a parallel search. In other words, $T_b(1,\epsilon)$ may not achieve the minimum number, $MINT_b(\epsilon)$, since $T_b(k,\epsilon)$ is related to the sequence of feasible solution found, and it is possible for a feasible solution whose value is very close to the optimal-solution value to be found early in the parallel case. However, in the serial case, this feasible solution is found quite late or even not found if its parent has a lower bound larger than the optimal solution.

$MINT_b(\epsilon)$ can be obtained by estimating the distribution of the lower bounds. From simulations of the 0-1 knapsack and vertex-cover problems, it was observed that the distribution of the number of nodes with respect to the lower bounds is exponential. In this case, let $a^g \partial g$, $a > 1$, be the number of nodes whose lower bounds are between g and $(g + \partial g)$. It is easy to show that

$$T_b(1,0) = \int_{-\infty}^{f_0} a^g \partial g = a^{f_0}/\log_e a$$

$$\mathrm{MINT}_b(\epsilon) = \int_{-\infty}^{\frac{t_0}{1+\epsilon}} a^g \partial g = (T_b(1,0))^{\frac{1}{1+\epsilon}}$$

From the above analysis, we get

$$\frac{\mathrm{MINT}_b(\epsilon) - 1}{k} + 1 \leqslant T_b(k,\epsilon) \leqslant \frac{T_b(1,\epsilon)}{k} + \frac{k-1}{k} h \tag{4.6}$$

Similar to Corollaries 4.1 and 4.3, we can derive the corresponding sufficient as well as necessary conditions when $\epsilon > 0$.

Corollary 4.5: In a parallel best-first search that satisfies the assumptions of Theorem 3.2 with the exception that $\epsilon > 0$, $T_b(k_2,\epsilon) \leqslant T_b(k_1,\epsilon)$ when

$$k_2 \mathrm{MINT}_b(\epsilon) - k_1 T_b(1,\epsilon) \geqslant k_1(k_2-1)h - k_2(k_1-1) \tag{4.7}$$

If $T_b(1,\epsilon) = \mathrm{MINT}_b(\epsilon)$ and $\epsilon = 0$, then Eq. (4.7) is equivalent to Eq. (4.1). However, if $T_b(1,\epsilon)$ is much larger than $\mathrm{MINT}_b(\epsilon)$, then Eq. (4.7) is generally not satisfied in practice.

Corollary 4.6: In a parallel best-first search that satisfies the assumptions of Theorem 3.2 with the exception that $\epsilon > 0$, $T_b(k_1,0)/T_b(k_2,0) > k_2/k_1$ when

$$T_b(1,\epsilon) - \mathrm{MINT}_b(\epsilon) > k_2 - (k_1 - 1)h - 1 \tag{4.8}$$

Definitely, Eq. (4.8) is often satisfied. In the special case when $k_1=1$ and $k_2=k$, Eq. (4.8) becomes

$$T_b(1,\epsilon) - \mathrm{MINT}_b(\epsilon) > k - 1 \tag{4.9}$$

For depth-first search with approximate lower-bound tests, stronger conditions can be obtained as long as all feasible solutions are located at the bottom level of the search tree. This is true for some realistic problems such as integer programming.

Corollary 4.7: In a parallel depth-first search, in which $\epsilon > 0$, $D = 1$, the path number is used as a generalized heuristic function, and all feasible solutions are located at the bottom level of the OR-tree, then $T_d(k_2,\epsilon) \leqslant T_d(k_1,\epsilon)$ when

$$T_d(1,\epsilon) > \frac{k_1(k_2-1)(c+1)h - k_2(k_1-1)}{k_2 - k_1} \tag{4.10}$$

where c is the number of the distinct incumbents obtained during a serial search. The necessary condition to allow $T_d(k_1,\epsilon)/T_d(k_2,\epsilon) > k_2/k_1$ is $(c+1)h > (k_2-1)/(k_1-1)$

Corollary 4.7 is based on the following bounds of $T_d(k,\epsilon)$

$$\frac{T_d(1,\epsilon) - 1}{k} + 1 \leqslant T_d(k,\epsilon) \leqslant \frac{T_d(1,\epsilon)}{k} + \frac{k-1}{k} h$$

The bounds are valid because the assumptions of Lemma 3.2 are satisfied, and all essential nodes must be expanded in the parallel depth-first search. The above analysis reveals that for this kind of problems, detrimental anomalies will not occur in approximate depth-first searches provided k_1 and c are not too large.

If the feasible solutions are not restricted to the bottom level of the OR-tree, then the corresponding necessary condition can be derived by modifying Eq. (4.4). That is,

$$T_d(1,\epsilon) - \mathrm{MINT}_b(\epsilon) > k_2 - (k_1-1)(c+1)h - 1 \tag{4.11}$$

4.4.4 Coping With Anomalies Under Dominance Tests

When dominance tests are considered, the above corollaries are no longer valid. Figure 4.2(b) shows an example in which none of the necessary conditions discussed previously is met, but the acceleration anomaly $T_b(2,0) / T_b(3,0) > 3/2$ still occurs The reason for the anomaly is that $P_i D P_j$ does not imply the dominance of P_i over the descendents of P_j. For instance, in a 0/1 knapsack problem, $P_i D P_j$ only if the weight of P_i is less than or equal to that of P_j, and P_i and P_j are defined on the same subset of objects. However, for descendents of P_j, their weights may be greater than that of P_i, and hence the dominance relation may not exist. One way to alleviate the detrimental anomalies in this case is to update the dominance test as follows. If a newly generated node is not dominated, we need to test whether any of its ancestors is dominated by an active node (it is not necessary to check all expanded nodes). If so, the newly generated node can be eliminated. Although additional overhead may be incurred in the proposed domination test, the dominance relation may be applied more often, and more nodes may be pruned. When the number of active nodes is not very large, the new dominance-test procedure is acceptable, and detrimental anomalies can be reduced.

When dominance tests are applied with the approximate lower-bound tests, the sufficient conditions would be the conjunction of the corresponding ones in both cases. In general, there is no realistic sufficient condition to avoid detrimental anomalies. In contrast, the necessary conditions are the disjunction of the corresponding necessary conditions in both cases. Simulations have revealed that acceleration anomalies appear quite often.

Before leaving from this section, we must point out that the reasons for the anomalies discussed in Sections 4.3 and 4.4 are not exactly the same. Therefore, some conditions obtained in Section 4.3 cannot be derived directly from the corresponding condition in Section 4.4 by setting $k_1=1$. The sufficient conditions for the latter case are usually stronger, while the necessary condition are weaker.

4.4.5 Multiple Subproblem Lists

When there are multiple subproblem lists, one for each processor, a node is selected from each local list with the minimum heuristic value for decomposition. This node may not belong to the global set of active nodes with the minimum heuristic values. It is not difficult to maintain a global incumbent in a global data register and broadcast it to each processor. Hence when dominance tests are inactive ($D = I$), all the theorems and corollaries derived in Chapters 3 ad 4 are applicable, and the behavior of using multiple lists is analogous to that of a centralized list.

When dominance tests are active, these tests can be restricted to the local subproblem lists or can be performed globally. If global dominance tests are

applied, the behavior is similar to that of a centralized list. On the other hand, if the dominance tests are performed for subproblems within each local lists, it is possible that $P_i D P_j$ exists in the serial case and P_j is not terminated in the parallel case because P_i and P_j are stored in different processors. As a result, both detrimental and acceleration anomalies may occur. For instance, in Figure 4.2(d), acceleration anomaly may happen because P_2 is eliminated in the serial case and is not terminated in the parallel case. This means that T_3 is expanded in the serial case and pruned in the parallel case. Notice that this acceleration anomaly will not appear in case of a single subproblem list.

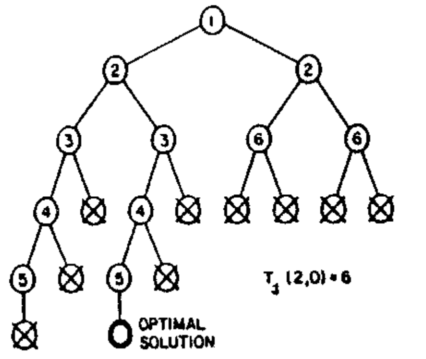
4.5 Comparison of Best-First, Depth-First, and Breadth-First Searches

In this section we answer the question on whether a parallel best-first search is the best search strategy as compared to a depth-first or breadth-first search when a constant number of processors are used. This fact has been established for serial searches. However, anomalies have been found in parallel searches. This is illustrated in Figure 4.3 with two processors. Six iterations are needed to complete the depth-first and breadth-first searches; whereas seven iterations are required for a best-first search. Anomalies usually occur when the total number of iterations is small. A sufficient condition to assure that $T_b(k,0)<T_r(k,0)$ is shown here.

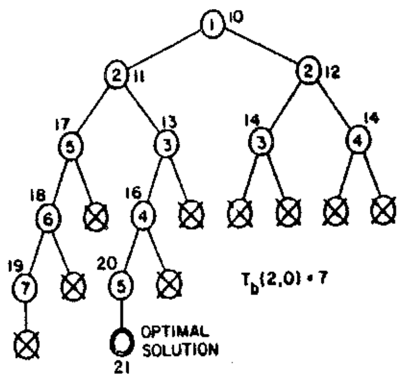
Corollary 4.8: In the parallel best-first and breadth-first searches that satisfy the assumptions of Theorems 3.2 and 3.4, $T_b(k,0) < T_r(k,0)$ when $T_r(1,0) - T_b(1,0) > (k-1)(h-1)$.

Corollary 4.8 says that if $T_r(1,0)$ is much larger than $T_b(1,0)$ and k and h are small enough, then a parallel best-first search often runs faster than a parallel breadth-first search.

Owing to the loose lower bound on $T_d(k,0)$, the sufficient condition to assure that $T_b(k,0) < T_d(k,0)$ cannot be derived from Eq's 3.18 and 3.19. We now show



(a) Parallel depth-first search with two processors (number inside each node is the selection order).



(b) Parallel best-first search with two processors (number inside each node is the selection order, number outside each node is the lower bound).

Figure 4.3 Anomaly in performance between depth-first and best-first searches

the corresponding condition in terms of the number of imperfect iterations.

Corollary 4.9: Let y_d be the number of imperfect iterations in a parallel depth-first search. If the assumptions in Theorems 3.2 and 3.3 are satisfied, then $T_b(k,0)<T_d(k,0)$ when $y_d>(k-1)h$.

Proof: Let m_{d_i} be the number of essential nodes selected in the i'th imperfect iteration of a parallel depth-first search. Similar to the proof of Theorem 3.2, we get

$$T_d(k,0) = \frac{T(1,0) - \sum_{i=1}^{y_d} m_{d_i}}{k} + y_d \geq \frac{T(1,0)-(k-1)y_d}{k} + y_d = \frac{T(1,0)}{k} + \frac{y_d}{k}$$

According to Eq. (3.18) we know that, in the worst case, $T_b(k,0) < T_d(k,0)$ if

$$\frac{T(1,0)}{k} + \frac{y_d}{k} > \frac{T(1,0)}{k} + \frac{(k-1)\,h}{k} \qquad \text{i.e.,}\quad y_d > (k-1)\,h \quad \square$$

In practice, the average number of essential nodes in an imperfect iteration is almost the same. Under this assumption, $T_b(k,0)<T_d(k,0)$ when $y_d>h$.

If all feasible solutions are located in the bottom level of the OR-tree, the lower bound on $T_d(k,0)$ has been derived as $(T_d(1,0)-1)/k+1$. In this case, $T_b(k,0) < T_d(k,0)$ will be assured if

$$T_d(1,0) - T_b(1,0) > (k-1)(h-1).$$

Actually, a considerable number of iterations are imperfect in a parallel depth-first search. Moreover, the number of imperfect iterations increases when the number of processors is increased. Hence a parallel best-first search usually requires less iterations than a parallel depth-first search for the same problem.

From the viewpoint of coping with anomalies, the bredath-first search is a conservative search strategy. There are neither detrimental anomalies nor serial-to-parallel acceleration anomalies when dominance tests are inactive. In contrast, a depth-first search is an adventurous search strategy. It may gain superlinear speedup but suffers from the risk of detrimental anomalies. The anomalous behavior of best-first searches depends in some degree on the policy of dealing with nodes having identical lower bounds. If a breadth-first strategy is adopted, then serial-to-parallel detrimental anomaly can be avoided even if an approximate solution is sought. In general, linear speedups can be achieved in a larger range of the number of processors for parallel best-first searches than for depth-first and breadth-first searches. In this sense, the best-first search is more robust for parallel processing.

On the other hand, since the best-first search requires the secondary memory to maintain a large number of active nodes, the total time, including time spent for data transfers between the main and secondary memories, to solve a problem should be taken as a measure of efficiency. From this point of view, the best-first search may not always be a good search strategy. Simulations have shown that the best OR-tree search strategy depends on the accuracy of the problem-dependent lower-bound function [WaY85]. Very inaccurate lower bounds are not useful in guiding the search, while very accurate lower bounds will prune most unnecessary expansions. In both cases, the number of subproblems expanded by depth-first and best-first searches will not differ greatly, and a depth-first search is better as it requires less memory space (proportional to the height of the search tree). When the accuracy of the lower-bound function is moderate, a best-first search gives a better performance. In this case, a good memory management system is necessary to support the memory space required.

In Figure 4.4, the average speedups of parallel best-first and depth-first searches for ten knapsack problems with thirty-five objects are compared. Note that for knapsack problems, the lower-bound function is accurate. The performance with respect to using the depth-first search, as a result, is close to that of using the best-first search. In the serial case, the average number of iterations for depth-first search, $T_d(1,0)$, is 15197, and the average of $T_b(1,0)$ is 15180, which are almost the same. The speedups in best-first searches are a little larger than that of the depth-first searches. When the number of processors are very large such

that, nodes in each level can be expanded simultaneously, then $T_d(k,0) = T_b(k,0) = h$, where h is the hight of the OR-tree. Therefore, the two curves on speedup will coincide eventuately.

Finally, we compare the space requirements between depth-first and best-first searches. In a serial search the space required by the best-first strategy is usually more than that required by the depth-first strategy. Somewhat surprisingly, the simulation results on 0-1 knapsack problems show that the space required by a parallel best-first strategy is not increased significantly (but may be reduced) until the number of processors is so large that the near-linear speedup cannot be kept. In contrast, the space required by the parallel depth-first search is almost proportional to the number of processors (Figure 4.4). Note that the space efficiency is problem-dependent. For vertex-cover problems, the space required by a parallel best-first search is not increased significantly no matter how many processors are used.

4.6 Summary

In this chapter, we have studied the anomalous behavior of parallel OR-tree searches. Anomalies are caused by a combination of the following reasons: (a) there are multiple solution nodes; (b) the heuristic function is ambiguous; and (c) the elimination rule is not consistent with the heuristic function; (d) the tree structure causes imperfect iterations when multiple processors are used, and (e) variability in the sequence of feasible solutions. The existence of a combination of these conditions causes the tree to be searched in a different order when different number of processors are used.

For OR-tree search algorithms with dominance tests, only a best-first search with the following conditions will guarantee that detrimental anomalies will not occur: (a) the heuristic function is unambiguous; (b) approximations are not allowed; (c) the dominance relation is consistent with the heuristic function; and

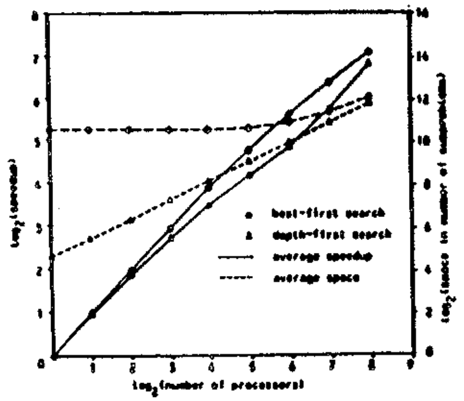


Figure 4.4 Average speedup and space requirements of parallel branch-and-bound algorithms for 10 knapsack problems with 35 objects

(d) the complexity of the problem to be solved is relatively high. Ambiguity in the heuristic function can be resolved by augmenting the original heuristic function with a tie-breaking rule (say, by level and left-right orientation). For most problems, dominance relations that are consistent with the heuristic function can be designed. Acceleration anomalies may occur in one of following cases: (a) a breadth-first or depth-first search is used; (b) a best-first search is used and some non-solution nodes have lower bounds equal to the optimal solution; (c) the dominance relation is inconsistent with the heuristic function; (d) multiple lists of subproblems are used; (e) a suboptimal solution is sought; or (f) when the problem size is large. We have precisely investigated various sufficient conditions to eliminate detrimental anomalies and necessary conditions to preserve acceleration anomalies with respect to the different search strategies. A summary of the results proved in Chapters 3 and 4 are shown in Table 4.1.

A best-first search is found to be a robust search strategy in the sense of the large range of number of processors within which the linear speedup is achived. The best OR-tree search strategy depends on the accuracy of the problem-dependent lower-bound function. Best-first search is more suitable for parallel processing when the accuracy of the lower-bound function is moderate; otherwise depth-first search is more cost-effective.

Table 4.1 Conditions to copy with anomalies of parallel OR-tree searches

Allow. func.	Domi. rela.	Search strat.	Suff. cond. (1 vs. k)	Nece. cond. (1 vs. k)	Suff. cond. (k_1 vs. k_2)	Nece. cond. (k_1 vs. k_2)
$\epsilon=0$	D=I	b	Eq.3.7,3.8	Th.4.4	Cor.4.1	Cor.4.3
		d		Eq.4.4	Cor.4.2	Cor.4.4
		br		no anom.	Th.4.5	Eq.4.5
	D≠I	all	Th.4.1	Th.4.4	anom.	exist
$\epsilon>0$	D=I	b	Eq. 3.7,3.8	Eq.4.9	Cor.4.5	Cor.4.6
		d	Th.4.3	Eq. 4.11	Cor. 4.7	Eq. 4.11
		br	Eq. 3.7,3.8	no anom.	Th.4.5	Eq.4.5
	D≠I	b	Th.4.2	exist	anom.	exist
		d	Th.4.3			
		br	Th.4.2			

b: best-first search
d: depth-first search
br: breadth-first search
anom.: when multiple subproblem lists and local dominance tests are used, sufficient conditions are impractical
exist: necessary conditions are too loose

CHAPTER V
PARALLEL PROCESSING OF AND/OR-GRAPH SEARCH

In this chapter, we address the parallel AND/OR-graph search, in particular, the heuristic AND/OR-tree search. To avoid unnecessary searching, an intelligent search strategy that guides the search by heuristic information is desirable. For decision problems, a heuristic function using the ratio of the success probability of a subgoal to the estimated overhead of evaluating the subgoal is found to be useful in guiding the AND/OR-tree search. We propose and analyze an optimal search strategy that minimizes the expected overhead. The optimal strategy is then applied to parallel processing of logic programs that are modeled as AND/OR-tree searches. An efficient heuristic search strategy for solving logic programs, which can be implemented on a multiprocessor architectures, is presented.

5.1 AND/OR-Tree and AND/OR-Graph Search

Searching an AND/OR-tree is more complex than that of an AND-tree or an OR-tree with lower-bound tests only. An AND/OR-tree is searched in two phases. The first phase is a top-down expansion as in searching an OR-tree, while the second phase is a bottom-up evaluation as in searching an AND-tree. Due to the existence of both AND- and OR-nodes, a good parallel search algorithm should combine the features of AND- and OR-tree searches. A good selection strategy must be developed owing to the OR-nodes. The granularity of parallelism, similar to that of parallel divide-and-conquer algorithms, is an important issue. Specific restrictions on a given problem, such as pruning rules, must be considered. These rules are usually more complicated, as more information is involved in the process.

In an AND/OR-tree search, some nodes may be generated more than once. This redundant effort can be avoided by searching the corresponding AND/OR-graph at the price of additional bookkeeping. AND/OR-graph searches are especially useful for dealing with some problems in artificial intelligence such as partially commutative production systems in which a give set of operations will produce the same result regardless of the order in which they are applied. However, cycles may be introduced in AND/OR-graph searches. A *cycle* is a path through the graph in which a given node appears more than once. Whenever there is a cycle, there can be paths of arbitrary length. Thus it may become more difficult to guarantee the termination of an AND/OR-graph algorithm. Mechanisms to eliminate cycles must be considered in AND/OR-graph-search algorithms.

In Chapter 3, we have discussed searches of OR-graphs for optimization problems. For decision problems, an AND/OR-tree-search algorithm can be converted to an AND/OR-graph-search algorithm by modifying the actions performed each time a node is generated as follows:

(1) Examine the set of nodes that have been created so far to see if the new node already exists.

(2) If not, simply add it to the graph as in the case of a tree.

(3) If it does already exists, then set the node being expanded to point to the already existing node as its successor. The newly generated node can be discarded

In general, a graph search is based on the sharing of nodes using pointers. Once a node is evaluated, its value can be stored. In any subsequent search for this node, the corresponding value will be immediately obtained. This is one of the major benefits of AND/OR-graph searches over AND/OR-tree searches.

The use of heuristic information to guide searches is a key issue for both AND/OR-tree and AND/OR-graph searches. In this chapter, we will study heuristic strategies for evaluating logic programs that are represented as AND/OR-

trees. The principle of the heuristic strategy can also be applied AND/OR-graph searches.

5.2 Previous Work of Parallel AND/OR-Tree Search

A lot of research efforts have been devoted to parallel AND/OR-Tre Parallel game tree search, especially parallel $\alpha-\beta$ pruning, and parall programming are major topics in this area.

5.2.1 Parallel Game-Tree Search

A two-person game between players MAX and MIN can be represented game tree in which the moves of MAX and MIN are put in alternate levels of tree. In the corresponding AND/OR-tree, board positions resulting from MA moves are represented by OR-nodes, while positions resulting from MIN's mov are represented by AND-nodes. Each player is trying to minimize the maximu gain of his opponent. The method used in common game-playing programs is t approximate the search of the entire game tree by examining a look-ahead tree of a fixed depth. For every terminal of the look-ahead tree, an estimate of the score of the discarded portion of the game tree is made by an evaluation function. All nonterminal MAX nodes take the maximum score of their children, while all nonterminal MIN nodes take the minimum score. This *minimax procedure* is used to find the best move for the MIN player represented as the root.

A well known technique to improve the pruning of a minimax search is the α-β *pruning* [BaF81]. This technique uses two parameters, α and β, to define the search window. α carries the low bound on the MAX nodes, while upper bound on the MIN nodes. α-β [illegible] search window is [illegible] distribution of values of the terminal nodes. The tree is said to have a *best-case ordering* if the first (or leftmost) branch from each node leads to the best value, and a *worst-case ordering* if the rightmost branch from each node leads to the best value.

A number of parallel game-tree-search techniques have been developed [MaC82]. In the parallel aspiration search, the α-β window is divided into non-overlapped sub-intervals, which are independently searched by multiple processors [Bau78a]. Baudet reported that the maximum expected speedup is around 5 or 6 regardless of the number of processors. The speedup is limited because at least $W^{\lceil D/2 \rceil} + W^{\lfloor D/2 \rfloor} - 1$ nodes must be evaluated for an uniform tree of depth D and constant width W, even when α and β are chosen to have the optimal minimax values [KnM75]. Further, acceleration anomalies may occur when the number of processors is small, say 2 or 3.

Finkel and Fishburn have proposed a tree-splitting algorithm that maps a look-ahead tree onto a processor tree with the same interconnection structure [FiF82]. The information-transfer overhead is small due to the close match between the communication requirements and the interconnection structure. However, this is a brute-force search algorithm, and pruning is not considered in process assignments. The speedup drops to $\sqrt{k}$ under the best-case ordering, where k is the number of processors.

In the *mandatory-work-first scheme* [ABD82], the minimum tree searched in a serial algorithm is searched in parallel during the first phase. The resulting α-β window is used to speed up the second phase, during which the rest of the tree is searched. Comparing this scheme with the tree-splitting scheme, it has better performance under the best-case ordering, but may be worse in the worst-case ordering. In the latter case, many nodes that are pruned in the tree-splitting scheme may be visited in the second phase.

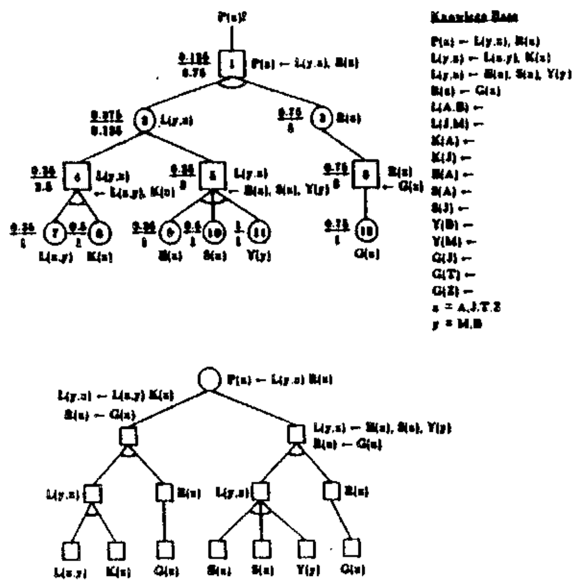
Another approach to speed up a game-tree search is to use a best-first search, such as the SSS* algorithm [Sto79]. SSS* is effective in searching a random or poorly ordered tree, but requires more space and is not significantly better than an α-β search on strongly ordered trees. Kumar and Kanal have shown that the SSS* algorithm can be interpreted as a B&B procedure and have presented two parallel implementations of SSS* [KuK84].

These approaches to parallel game-tree search have emphasized on the reduction of the information-transfer overhead, but have paid little attention to the information-deficiency overhead. In the remainder of this chapter, we will consider the information-deficiency overhead as illustrated by the parallel evaluation of logic programs.

5.2.2 Parallel Logic Programming

Logic programming is a programming methodology based on Horn-clause resolution [Kow79]. The efficiency of solving a logic programming problem depends strongly on the many ways of representing the logic program. One of the ways is to consider the evaluation The root is labeled by the initial problem to be queried; the OR nodes represent (sub)goals; and the AND-nodes represent clauses. All subgoals in the same body of a clause are children of an AND-node. A (sub)goal (OR-node) and its children display the nondeterministic choices of clauses with the same head. The terminal nodes denote clauses or subgoals that cannot be decomposed. Figure 5.1(a) shows an example of the AND/OR-tree representation of a logic program in which the OR-nodes are represented as circular nodes and the AND-nodes are represented as squared nodes. In Figure 5.1, the numbers outside each node are the ratio of success probability to expected search cost. In general, a logic program without any inference loop can be viewed as an acyclic AND/OR-graph. A finite AND/OR-tree is obtained from the AND/OR graph by duplicating common descendent nodes.

The AND/OR-tree in Figure 5.1(a) can be represented more clearly in Figure 5.1(b) as a high-level OR-tree involving the selection of all combinations of clauses, and multiple low-level AND trees representing the solution tree resulted



P: happy; L: like; R: rich; K: kind;
H: handsome; S: strong; Y: young;
G: good job; A: Ares; B: Betty;
J: John; M: Mary; T: Tom; Z: Zeus.

Figure 5.1 An example of a logic program represented as an AND/OR tree.

from clauses selected in the OR-tree. Parallel processing can be applied to evaluate the multiple solution trees in parallel (OR-parallelism), or can be applied to search a solution tree in parallel (AND-parallelism). Note that the number of edges in this representation could be much large than that of an AND/OR-tree representation.

A lot of research strivesfor the parallel execution of logic programs. Conery and Kibler have classified four kinds of parallelism of logic programs: AND-parallelism, OR-parallelism, stream-parallelism, and search parallelism, and have investigated AND-parallelism [CoK83]. Furukawa, et al. and Ciepielewski, et al. have discussed OR-parallelism [FuN82,CiH84], while Lindstrom, et al. have addressed stream-parallelism and pipelined PROLOG processors [LiP84].

AND-parallelism involves the simultaneous execution of subgoals in a clause. Since subgoals within a clause can share variables, the binding of variables of concurrently executing subgoals must be coordinated to avoid conflicts of a shared variable being bound to more than one value. AND-parallelism is limited by the measures to avoid conflicts. One approach to avoid conflicts is to annotate variables to indicate which subgoals can bind values to specific variables and which cannot [ClG83]. In particular, only one subgoal, called *producer*, is allowed to bind a value to a variable. Conery proposed a complex, non-annotated, process-structured system that dynamically monitors variables and continually develops data dependency networks to control the order of execution of subgoals, never allowing two potential producers for the same variable to execute in parallel [CoK83]. DeGroot described a method to obtain restricted AND-parallelism by compile-time creation of a parallel execution-graph expression for each program clause [DeG84].

In OR-parallelism, all subgoals are independent of each other, and consistency checks for shared variables needed in AND-parallelism are avoided [Kow79,Mot84]. However, an OR-tree representation is inefficient due to the large number of branches needed as compared to that of an AND/OR-tree. Given an initial query, ← A, B, with n ways of solving A and m ways of solving B, the OR-tree contains n×m branches, whereas an AND/OR-tree contains n+m. To improve the efficiency of an OR-tree search of logic programs, several models that modify pure OR-parallelism, such as introducing process bundles [YaN84] and bagof [CiH84], are proposed.

The search of logic programs is generally considered different from an AND/OR game-tree search for the following reasons. First, in contrast to combinatorial-extremum searches that find the best solution, solving a logic program corresponds to finding any (or all) solution(s) satisfying the given conditions, the implicative Horn clauses, and the consistent binding of variables for the subgoals. Second, the value of a node in the AND/OR-tree for a logic program is either TRUE (success) or FALSE (failure). The selection of a node for evaluation is usually based on a fixed order, and heuristic information to guide the search is not available. Third, a variable in a logic program can be bound to several values, and some subgoals may share a common variable. For example, in Figure 5.1, some subgoals share variable x, and x can be bound to any of Ares, John, Tom, and Zeus. For a particular variable in a subgoal, a subset of its possible values may be allowed. In contrast, the nodes in a game tree are independent. Lastly, pruning rules for evaluating the AND/OR-tree of a logic program are different from α-β pruning due to the binary values returned by the terminal nodes.

Several parallel models of logic programs and the corresponding multiprocessing architectures have been studied [FuN82,KKM83,Mot84, DeG84,CiH84,HaA84,StM84]. Nearly all these architectures were based on ad hoc search strategies and scheduling methods. In this chapter, we propose MANIP-2, a multicomputer architecture to evaluate logic programs. However, *the goal here is not in describing the details of an architecture, but in giving the theoretical foundation of the necessary search algorithm upon which the architecture is based.* The emphasis of this chapter is in showing the reasonableness of heuristic searching and the feasibility of an effective scheduling method.

5.3 Best-First Search of AND/OR-Trees

In most existing computer systems, decision problems such as logic programs are solved by brute force search of AND/OR-trees. Nevertheless, heuristic information is necessary to speed up AND/OR-tree search. In this section, we examine the heuristic information and prove a optimal search strategy of AND/OR-tree search.

5.3.1 Heuristic Information for Searching AND/OR-Tree

In a decision problem, the useful heuristic information to guide the search include the predicted success probability of a solution tree being found from a subgoal or clause, and the associated average cost of finding the solution tree.

The success probability of a node (or alternatively a branch) in an AND/OR-tree is an a priori probability that reflects the possibility of finding a solution tree over all unifications from this node. These probabilities are used to guide the search initially, and will be improved after more dynamic pruning information is obtained.

Assuming that a nonterminal node K has two immediate descendents, K_1 and K_2, let P(·) be the a priori success probability of a node. Then

$$P(K) = \begin{cases} P(K_1) \cdot P(K_2 \mid K_1) & (K \text{ is AND}) \\ P(K_1) + P(K_2) - P(K_1) \cdot P(K_2 \mid K_1) & (K \text{ is OR}) \end{cases} \tag{5.1}$$

Eq. (5.1) can be generalized to nodes with more than two descendents. The computation of the a priori conditional probabilities can be complex due to the shared variables.

The success probabilities can also be either assigned initially by the designer, or determinated by statistic collected during execution. In the latter case, no a prior probability is available before the program is executed, and all branches are assumed to have equal possibility of success. A deterministic search strategy, such as a depth-first search, has to be used initially.

The cost for searching a subtree depends on the structure and dependence of the subtrees, the query used, and the partial search results already obtained. One way is to define an average search cost based on the estimated probabilities of success. For node K with descendents K_1 and K_2 and assuming that K_1 is searched first, the average search cost is

$$C(K) = C(K_1) + \begin{cases} C(K_2 \mid K_1) \cdot P(K_1) & (K \text{ is AND}) \\ C(K_2 \mid K_1) \cdot (1 - P(K_1)) & (K \text{ is OR}) \end{cases} \tag{5.2}$$

Simulations have shown that the average search cost depends on the structure of the tree, but is quite insensitive to changes in the success probability. A complete binary AND/OR-tree with unitary search cost at the terminal nodes was assumed. The simulation results depicted in Figure 5.2 indicate the relationship between the success probabilities of the root of trees with height 12 and 16, respectively, and the associated average search cost. The average cost is the smallest when p is either zero or one, and is maximum when p is around 0.5. Moreover, the difference between the maximum and the minimum costs is relatively small, which reflects the insensitivity of the expected search cost with respect to the success probability.

5.3.2 Optimal Strategy of Searching AND/OR-Trees

Let P(x) and C(x) be the probability of success and the associated average search cost for node x. Define the criteria Φ_a and Φ_o for any node x as

$$\Phi_a(x) = \frac{P(x)}{C(x)} \quad (x \text{ is descendent of an OR-node}) \tag{5.3}$$

$$\Phi_o(x) = \frac{1 - P(x)}{C(x)} \quad (x \text{ is descendent of an AND-node}) \tag{5.4}$$

Simon and Kadane have studied the optimal OR-tree search and have proved that the search sequence $b = b_1, \ldots, b_n$ is optimal iff $\Phi_a(b_i) \geqslant \Phi_a(b_{i+1})$, where the b_is are descendents of an OR-node b with precedence relationships [SIK75]. Barnett has extended their results to optimal search from AND-nodes with the assumption that

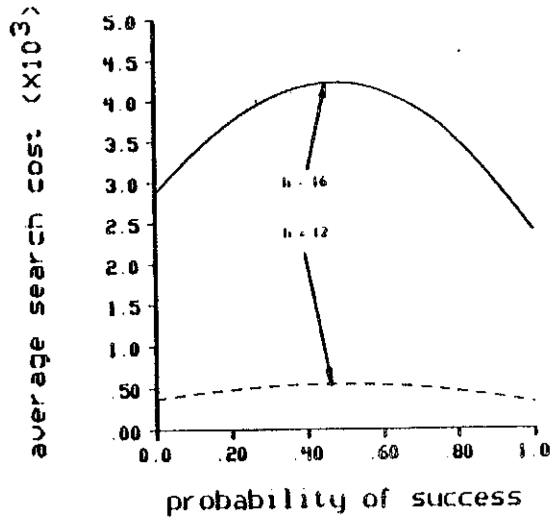


Figure 5.2 A plot of the search cost versus the success probability of the root.

all immediate descendents are independent [Bar83]. Garey has proved that the optimal sequence of performing a set of tasks until one of them fails or all tasks are fulfilled is in descending order of Φ_o [Gar73].

For a pure OR-tree or AND-tree search, the search order is well defined by a single heuristic function. However, when an AND/OR-tree is searched, there are two criteria, Φ_a and Φ_o, to order the AND and OR-nodes, respectively. Hence a complete order cannot be defined for all active nodes. To resolve this problem, we can decompose an AND/OR tree into a hierarchy of a single OR-tree, each terminal of which is an AND tree (Figure 5.1(b)). The following theorem relates the criteria Φ_a and Φ_o and defines an optimal search order for an AND/OR-tree with (possibly) dependent nodes.

Theorem 5.1: Suppose that an OR-node K has n immediate descendent AND nodes, $K_1, \ldots, K_n$, and that the AND-node K_i, $1 \leqslant i \leqslant n$, has i_m immediate descendent OR-nodes, $K_{i_1}, \ldots, k_{i_m}$. If $\Phi_a(K_i) \geqslant \Phi_a(K_{i+1})$, then the expected search cost C(K) is minimum when all descendents of K_i are searched before K_{i+1}.

Proof: Suppose that K_{i_1} is found to be TRUE, then the conjunction of the remaining subgoals of K_i, namely, $K_{i_2}, \ldots K_{i_m}$, forms a new AND-node, K_{i_2}'. The conditional probability of $K_{i_2}, \ldots, K_{i_m}$ being TRUE, given that K_{i_1} is TRUE is

$$P(K_{i_2}') = P(K_{i_2}, \ldots, K_{i_m} \mid K_{i_1}) = \frac{P(K_i)}{P(K_{i_1})} \tag{5.5}$$

To get the optimal strategy, we need to compare the probability-to-cost ratios in respect to node K_{i_2}' and K_{i+1}. Using Eq's (5.2) and (5.5),

$$\Phi_a(K_{i_2}') = \frac{P(K_{i_2}')}{C(K_{i_2}')} = \frac{P(K_i)/P(K_{i_1})}{C(K_{i_2}, \ldots, K_{i_m} \mid K_{i_1})} \tag{5.6}$$

$$\geqslant \frac{P(K_i)}{C(K_i)} \geqslant \frac{P(K_{i+1})}{C(K_{i+1})} = \Phi_a(K_{i+1})$$

Eq. (5.6) implies that the remaining subgoals $K_{i_2}, \ldots, K_{i_m}$ should be unified first

before K_{i+1}. If subgoal K_{i_2} is found to be TRUE, then the above proof can be applied again to show that the remaining subgoals of K_i should be unified before K_{i+1}. □

Theorem 5.1 shows that for decision problems the optimal strategy of searching an AND/OR tree is to select the most promising solution tree with the largest Φ_a value among all possible solutions trees, and to examine AND nodes in this solution tree in descending order of Φ_o values. The search is switched to the next best solution tree (with the next largest Φ_a value) if the first solution tree fails.

The key issue in performing the above optimal strategy is to find the most promising solution tree with the largest Φ_a value, and for the solution tree selected, the node with the largest Φ_a value. If h, the height of a complete AND/OR-tree, is taken as a measure of the problem size, then it is unlikely that a polynomial-time algorithm exists for finding the most promising solution tree. Let B_o and B_a be the numbers of branches of each OR- and AND-node, respectively. If $B_a = 1$, then there are $B_o^{h/2}$ possible solution trees, each of which consists of one node. To get the maximum Φ_a over all solution trees, at least $B_o^{h/2} - 1$ comparisons are needed. This is the lower-bound complexity for computing the largest Φ_a when $B_a = 1$. In general, if $B_a > 1$, then before Theorem 5.1 can be applied, the AND/OR-tree has to be transformed into a hierarchy of an OR-tree, each terminal of which is an AND tree. The number of terminal nodes in the transformed tree has a lower bound of $((B_o^{B_a})^{B_a^{h/2-1}})$. Hence to select the largest Φ_a and Φ_o values would require an exceedingly large amount of computational time. Another approach is to store these values associated with each node in the AND/OR-tree and to retrieve the decisions in real time. Unfortunately, this would require a large amount of storage space with a lower bound of the same complexity as stated above. Therefore, it is unrealistic to apply the optimal search strategy in respect to an AND/OR-tree search.

5.4 An Efficient Scheduling Algorithm for AND/OR-Tree Search

In most previous studies, parallelism is restricted to certain aspects in the search tree, such as searching the AND or OR-nodes in parallel. However, the nodes selected for parallel processing should be more flexible, and very few studies have addressed the problem of processor assignment to reduce the information-deficiency overhead. In this section, we present a scheduling algorithm that does not distinguish between AND and OR parallelism, but schedules the nodes to be searched according to their estimated probabilities of leading to a solution tree.

An efficient search method must involve pruning. Pruning is used to eliminate unnecessary expansions when it is determined that a solution tree cannot be found from a subgoal or clause. Two kinds of pruning exist in an AND/OR-tree search. In *AND-pruning*, if one successor of an AND-node is known to be FALSE, then all its remaining successors can be pruned. Likewise, if one solution is sought, then *OR-pruning* can be applied to prune other successors of an OR-node once one of its successors is known to be TRUE. In this chapter, it is assumed that one solution tree is sought, and hence both AND-pruning and OR-pruning can be applied.

Pruning and parallelism are conceptually illustrated in Figure 5.3(a). In a sequential depth-first search, if Node 1 fails, then Node 2 will be examined next; otherwise, Node 3 will be examined. Similarly, the traversal of Node 5 depends on the results of traversing Nodes 1, 2, 3, and 4. This dependence information can be represented in a *fail-token-flow graph*, G_f, as depicted in Figure 5.3(b) for the tree in Figure 5.3(a). A node (circle) in the graph will be active only if it receives a fail-token from an incident edge. When a terminal node is found to be FALSE, a fail-token is sent along the direction of the corresponding edge. The coordinator (shaded box) in the graph coordinates the activities of the connected blocks. When a fail-token is received from any incident edge of a coordinator, fail-tokens are sent to *all* directly connected nodes. At the same time, any node searched in the block directly connected to this coordinator can be terminated because it does not

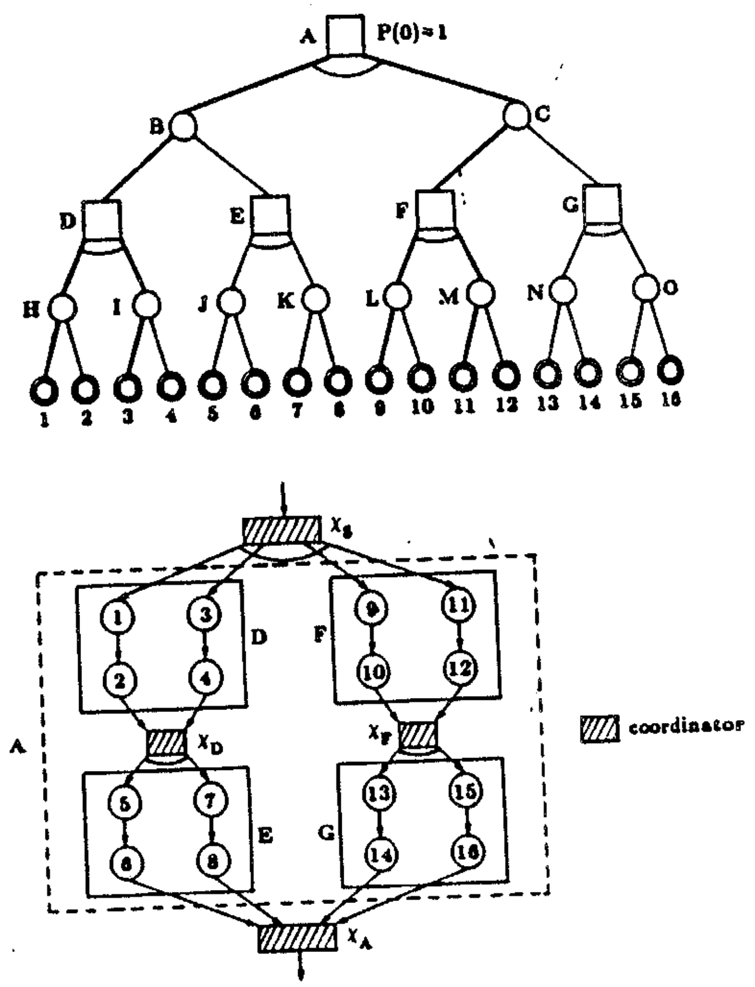


Figure 5.3 A binary AND/OR search tree with high success probability and the corresponding fail-token-flow graph, G_f.

belong to the solution tree. For example, when Node 1 is found to be FALSE, then a fail-token is sent to Node 2. If Node 2 is found to be FALSE, then a fail-token is sent to Coordinator χ_D. At this time, any node concurrently searched in Block D can be terminated. When a solution tree is found, there is one node in each column of G_f that returns TRUE. G_f can be used to represent pruning in AND-parallelism when the success probability is high and most of the terminal nodes are TRUE.

On the other hand, when the success probability of the goal is low, most of the terminal nodes are FALSE. The search for the inexistence of a solution tree in OR-parallelism can now be represented by the *success-token-flow graph*, G_s (Figure 5.4). G_s is the dual of G_f in the sense that a fail-token is replaced by a success-token, and the columns in G_f are transposed to become the rows in G_s. Referring to Figure 5.4(a), Nodes 1, 2, 5, and 6 are assigned to four processors. If any of Nodes 1, 2, 5, and 6 succeeds, then a success-token is generated, and the next connected node is assigned to an idle processor. Since most of the terminal nodes are FALSE, the search will be completed when a small number of nodes have been searched in each column of G_s.

The token-flow graph obtained for the root of an AND/OR-tree is modular and can be decomposed into modular token-flow subgraphs corresponding to all non-terminal nodes in the tree. If the probability of leading to a solution tree for a non-terminal node can be refined as the search progresses, the corresponding token-flow subgraph can be re-derived. An idle processor can be scheduled according to the token-flow subgraph derived for the root of the given subtree. In Section 5.6 we will discuss a multiprocessor architecture, MANIP-2, which is based on an intelligent search strategy and effective scheduling algorithm.

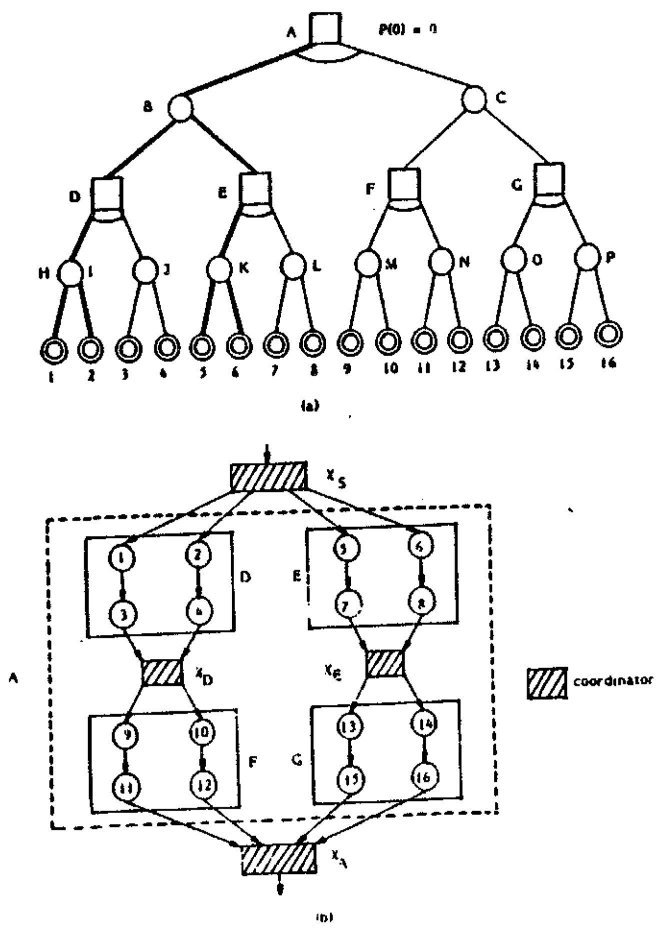


Figure 5.4 A binary AND/OR search tree with high failure probability and the corresponding fail-token-flow graph, G_s.

5.5 Evaluating Logic Programs as AND/OR-Tree Searches

In this section, we illustrate a heuristic AND/OR-tree search on the parallel processing of logic programs. Special requirements with respect to evaluating logic programming, such as shared variables and unification, will also be considered.

5.5.1 Heuristic Information of Logic Programs

Since the search space of a logic program is usually very large, an intelligent search strategy that guides the search is very important. Heuristic information to guide the search, such as the success probability of each subgoal or clause and the estimated overhead (or *cost*) of searching a subgoal or clause, remain an open problem. In this section, a heuristic search based on the information of the ratio of success probability to estimated cost is studied. This search strategy is aimed to minimize the expected search cost and the dynamic run-time overhead of evaluating logic programs.

First, we describe the evaluation of the success probability of each clause. For a terminal node with variable x, its success probability is m(x)/n(x), where n(x) is the total number of values that variable x can acquire, and m(x) is the number of values acquired by x in this terminal node. It is assumed that all values in the domain of a variable are equally likely to be assigned to a subgoal. When a subgoal shares more than one variable with other subgoals, and all variables are independent, its success probability can be computed as the product of the success probabilities of the variables. In general, the success probability of a node cannot be directly determinated by the success probabilities of its immediate descendents, since they may be correlated (the descendent nodes may contain shared clauses or subgoals that renders them dependent). It may have to be evaluated from actual information in the knowledge base. For example, in Figure 5.1(a), four values can be bound to x, and two can be bound to y. For the eight combinations of values of x and y, only two of them exist in the knowledge base. Hence, the success probability of L(x,y) is 0.25. The success probabilities can also be determinated by

statistics collected during execution. For instance, in Figure 5.1, the success probability of the clause L(y,x) ← L(x,y)·K(x) is the probability that both L(x,y) and K(x) succeed. However, L(x,y) and K(x) are dependent, and the success probability of L(y,x) would be difficult to compute. Statistics collected show that the success probability is 0.25. Other success probabilities in Figure 5.1 were computed by enumerations of all combinations of binding of variables.

To compute the heuristic value, the cost of evaluating a subgoal must be estimated. Since unification has a linear complexity [PaW78, MaM82], the search cost can be defined by the number of nodes traversed before a solution tree is found to exist or not exist. The cost for searching a subtree depends on the structure and dependence of the subtrees, the query used, and the partial search results already obtained. The average search cost of a subgoal is difficult to be formulated mathematically because it is related to the dependence of descendent subgoals (due to the shared clauses and subgoals). Moreover, the average search cost depends on the search strategy and the order that values are bound to variables, which in turn are driven by the average search costs. Hence the search costs would have to be initially estimated from statistic collected by a given search strategy. As better search costs are obtained, the search will become more efficient, and better estimates on the search costs can be obtained. For example, in Figure 5.1, the average search cost for L(y,x) was computed by averaging the search cost to verify the result of L(y,x) for all combinations of values of x and y. The search strategy used in computing the costs in Figure 5.1 assumed a left-to-right traversal of the descendents.

The expected search cost of a subgoal represents an average over all possible queries, all possible paths leading to this subgoal, and all possible combinations of values of variables for a given search strategy. However, when the path leading to a given subgoal is known, the minimum cost and the associated success probability of obtaining a solution tree are better measures to guide the search. Of course, this will result in an enormously large amount of stored information for each subgoal that renders the scheme impractical.

5.5.2 BAO – a Best-First AND/OR-Tree Search Algorithm for Solving Logic Programs

Owing to the intractable complexity of propagating the Φ_s and Φ_e values of all terminal nodes to the root in implementing the optimal AND/OR-tree search, an efficient top-down heuristic search is proposed here. As stated in Section 5.3.1, each node in the AND/OR tree can be assigned an estimated success probability and the associated expected search cost. These values, together with the information on the path leading from the root to this node, can be used to compute a heuristic value for the node. The search will be guided by the heuristic values.

A *solution tree* is a tree such that all nonterminals are AND-nodes and all terminals are OR-nodes. To minimize the search effort for a *solution tree*, it is necessary to first find one of the FALSE terminals in this tree, since the search can be terminated once this node is found. This method will be used to compute the heuristic values for AND-nodes. From the duality between AND and OR-nodes, a *failure tree* is a tree such that all nonterminals are OR-nodes and all terminals are AND-nodes. A failure tree is the dual of a solution tree, such that the entire AND/OR-tree is FALSE if all nodes in the failure tree are FALSE. To stop the search of a failure tree as early as possible, it is necessary to verify that one of the terminals is TRUE. This method will be used to compute the heuristic values for OR-nodes.

Referring to the partial binary AND/OR-tree in Figure 5.5, suppose that AND-node F is to be searched. For the goal to be TRUE, Nodes E and F must be TRUE. This is the information that can be extracted directly from the path leading from the root to F. The heuristic value of Node F will be the ratio of the probability of verifying Node A to be TRUE to the associated search cost. This can be computed as the ratio of the probability of success of Nodes E and F to the total average search cost of verifying that Nodes E and F are TRUE or any of them is

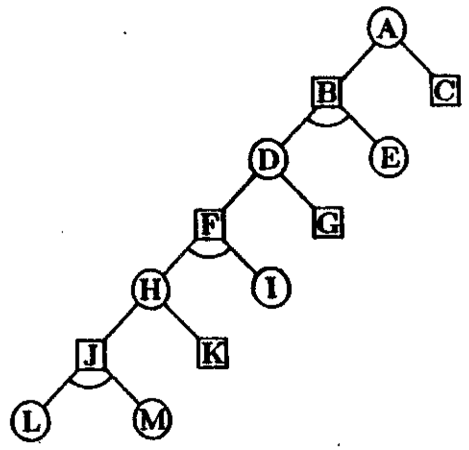


Figure 5.5 Computation of the heuristic values

FALSE. Note that Nodes E and F may be dependent. Other information, such as nodes searched from E, would be also crucial to computing the heuristic value for F. However, this information cannot be extracted directly from the path, and would require a high run-time overhead to maintain, hence will not be considered here. Similarly, the heuristic value of AND-node J is based on information about Nodes E, I, and J. In general, the heuristic value $\Phi_a(x)$ of an AND-node x is based on information extractable from the path leading from the root to x such that the goal can be verified to be TRUE. The information on the complete path from the root to any AND-node must be maintained with each AND-node.

To compute $\Phi_o(x)$ for an OR-node x, the information extractable from the path leading from the root to x such that the goal can be proved to be FALSE will be used. Complete information on the path from the root must also be maintained for each OR-node. Referring to Figure 5.5, in computing the heuristic value for OR-node H, the goal will be FALSE if Nodes C, G, and H are FALSE. Note that these nodes may be dependent.

The following heuristic-search algorithm, BAO, is a top-down best-first search algorithm that uses heuristic information about a particular set of solution trees for a given AND-node and a particular set of failure trees for a given OR-node. It is assumed that the values of Φ_a are available for each set of clauses with the same head (an OR-node) in the logic program. Similarly, the values of Φ_o for each subgoal in a clause (an AND-node) are also available. It is further assumed that the values bound to a variable are independent and in a fixed order, and that the values of Φ_o and Φ_a are applicable to all instantiations. The following procedure is applied iteratively by binding each variable to a constant value (or to a set of constant values that could be the domain of the variable) until the goal is proved to be TRUE or FALSE. There is an Active List containing tasks in decreasing order of heuristic values. Without loss of generality, assume that the goal node, S, is an OR-node.

BAO–Heuristic Search Algorithm for one solution tree:

(1) (*Initialization*): Initiate search from S. If S is known to be TRUE or FALSE, then stop. Otherwise, create a task for S with the information that S is an OR-node, and compute its heuristic value. The task is inserted into the Active List.

(2) (*Decomposition*): Select a task T from the Active List with the maximum heuristic value. If T is a ground node, then go to Step 3. If T contains a variable that is not bound, a constant value (or a set of constant values that could be the domain of the variable) is bound to the variable. Decompose the task (an AND or OR node) into its immediate descendent tasks. The new tasks, with their heuristic values, the information about the path from S, and the values bound to variables are inserted into the Active List in the proper order. Go to Step 2.

(3) (*Pruning*): For Task T under consideration, the following steps will be carried out based on values returned by the ground node.

- (a) The subgoal or clause (T_a) on the path from the root to T that is nearest to the root and becomes TRUE or FALSE is found.
- (b) The information on T_a is incorporated into all tasks in the Active List. Subtasks in some tasks can be removed based on T_a. If all subtasks within a task W are removed, then pruning will be carried out recursively on W.
- (c) The values successfully or unsuccessfully bound to variables are broadcast to all active tasks.
- (d) The heuristic values for all active tasks are updated. The active tasks are reordered if necessary.

(4) (*Termination*): If S is terminated, then return success with the bound values. If S cannot be bound to any new value, then return failure. Otherwise, go to Step 2.

There are several considerations when Algorithm BAO is implemented. First, a single list of active tasks is kept, and the AND and OR-nodes are not distinguished. Depending on the heuristic values and whether the node concerned is an AND or OR-node, the objective is to either prove that the goal is TRUE or prove that it is FALSE. Second, pruning performed in Step (3) requires a large overhead because the entire Active List has to be updated and reordered. However, the merits on the number of tasks eliminated and the better heuristic values generated are doubtful. To avoid this run-time overhead, pruning can be performed after a task is selected from the Active List (Step 2). In this case, all pruning information will be recorded in a common list. Of course, this may result in some unnecessary tasks in the Active List. Third, the computation of the heuristic values could be complex when all dependency of subgoals and variables are considered. In practice, some simplifying assumptions may be made in combining the heuristic values together. For example, a heuristic function to compute the success probability and cost of a conjunction of subgoals could be the product of the success probabilities and the sum of the associated costs of the subgoals. Lastly, the algorithm investigates many possible solution trees in parallel by switching from one to another based on the heuristic information obtained. This results in a large number of tasks in the Active List, which is a fundamental problem in many heuristic searches. To reduce the storage space required, static analysis can be performed to arrange the clauses with the same head and the subgoals in each clause in a proper order and use a depth-first strategy to search the AND/OR-tree. Of course, the order defined will be based on the average over all possible queries and all possible ways of reaching a particular subgoal.

The BAO algorithm is illustrated in Figure 5.6 by explaining snapshots of solving the logic program in Figure 5.1. The cost of each unification is assumed to be unity. The query is "Who is happy?" Node 1 is unified first. For simplicity, suppose that all successful bindings are equally likely, and that x is first bound to John. Let the updated heuristic values be Φ_a' and Φ_o', respectively. After decomposition, subgoals L(y,x) (Node 2) and R(x) (Node 3) with the corresponding

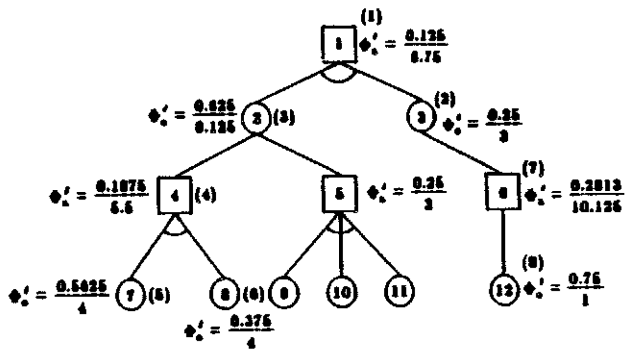


Figure 5.6 Illustration of the BAO procedure using the same example shown in Figure 5.1.

heuristic values are inserted into the Active List. Since $\Phi_o'(3) > \Phi_o'(2)$, Node 3 is selected, and Node 6 is created. In computing $\Phi_s'(6)$, the goal is expected to be TRUE, implying that Node 2 and 6 are TRUE. Hence the heuristic information of Node 2 must be included. The correlation between Nodes 2 and 6 may be complex. For simplicity, Nodes 2 and 6 are assumed to be independent. $P'(6)$ (resp. $C'(6)$), the new success probability (resp. new average cost) of Node 6, is the product (resp. summation) of the original success probabilities (resp. original average costs) of Nodes 2 and 6. $\Phi_s'(6)$ becomes 0.2813/10.125. As $\Phi_o'(2) > \Phi_s'(6)$, Node 2 is selected in the next iteration, and Nodes 4 and 5 are created. Intuitively, Node 6 is likely to be TRUE, but to terminate the search for the goal as soon as possible, the active node that may fail first should be investigated. In computing $\Phi_s'(4)$ and $\Phi_s'(5)$, the heuristic information on Node 3 must be included. For example, $\Phi_s'(4)$ is computed as $(0.75\times0.25)/(2.5+3)$. Suppose that variable y is first bound to Mary. Node 4 is next selected, as $\Phi_s'(4)$ is the largest among all active nodes. This means that the possible solution trees involving Node 4 is the most promising. Nodes 7 and 8 are inserted into the Active List in Iteration 4. Note that in computing $\Phi_o'(7)$ and $\Phi_o'(8)$, $\Phi_s(5)$ rather than $\Phi_s'(5)$ is used. The reason for this is that the goal is expected to be FALSE if Node 7 or 8 is selected, and the failure probability and cost of Node 5 are needed in this case, but the information on Node 3 (which is included in $\Phi_s'(5)$) does not affect the decision on Node 7 or 8. $\Phi_o(7)$ is computed as $(0.75\times0.75)/(1+3)$. Once Nodes 7 and 8 are instantiated in Iterations 5 and 6, respectively, Node 2 is known to be TRUE by values returned from the ground terms. Node 5 can thus be pruned by OR-pruning if a single solution tree is sought. In the last two iterations, Nodes 6 and 12 are unified, and the solution is "John is happy."

5.6 Parallel Computer Systems for AND/OR-Tree Search

In this section, we study the problems associated with the parallel processing of the AND/OR-tree representations of logic programs. By minor modifications of MANIP [WLY84], parallel heuristic search of logic programs can be carried out efficiently. The architecture of MANIP-2 is shown in Figure 5.7.

The first problem in parallel processing is the unification of shared variables. When AND-parallelism is involved, variable binding conflicts must be prevented. A lot of efforts have been devoted to solving this problem [CIG83,CoK83, DeG84,Sha83]. We do not attempt to propose a new method to overcome this problem. Instead, we assume one or more of the following conditions: (a) that the subgoals do not share variables; (b) that the shared variables are uninstantiated; (c) that a producer subgoal has bound one or more values to a shared variable and forwards them to many consumer subgoals. In any of these cases, processors can carry out tasks independently.

The second problem is the granularity of parallelism. If the granularity is small, then the information-transfer overhead will be high. In contrast, if the granularity is large, then the information-deficiency overhead will be high, and the degree of parallel processing may be small. The proper granularity depends on the capacity of the communication network, the number of processors, and the relative overhead between the top-down unification of variables and the bottom-up return of solution values. In searching a logic program, if the cost of a subgoal selected is smaller than the defined granularity, then the subgoal is considered as an indivisible operation and processed by a single processor.

Once the granularity is determined, the next problem is to determine the scheduling of the k parallel processors. Given a list of active tasks ordered by decreasing probability-to-cost ratios, the problem is to determine the number of processors to evaluate each task in parallel. To minimize the expected completion time, the problem can be formulated into a complex integer-programming optimization problem. Moreover, the processors have to be rescheduled again once

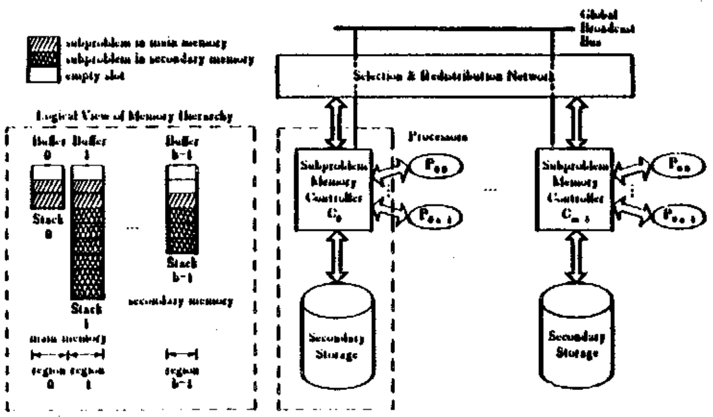

subproblem in main memory
subproblem in secondary memory
empty slot
Logical View of Memory Hierarchy
Buffer 0
Buffer 1
Buffer b-1
Stack 0
Stack 1
Stack b-1
main memory
secondary memory
region 0
region 1
region b-1
Global Broadcast Bus
Selection & Redistribution Network
Processors
Subproblem Memory Controller C_0
Subproblem Memory Controller C_{m-1}
$P_{0,0}$
$P_{0,k-1}$
Secondary Storage

Figure 5.7 The architecture of MANIP-2, a multiprocessor for parallel heuristic search of logic programs and AND/OR graphs.

a given solution tree is found to be TRUE or FALSE. Considering the facts that the scheduling algorithm is complex and that static scheduling is not feasible, we decide to assign a free processor to each task in the Active List. When a task has reached its minimum granularity, it will be processed by a single processor until completion.

Another problem is on resource sharing. For a set of processors evaluating a solution tree, it is necessary to search those subtrees with the largest failure-probability-to-cost ratios. Likewise, for sets of processors examining different solution trees, the possible solution trees with the largest success-probability-to-cost ratios must be evaluated. Further, load balancing must be carried out to keep all processors busy. The ring network in MANIP is adequate for load balancing the processors.

The parallel selection problem has been studied thoroughly in MANIP [WaM84] and discussed in Chapter 3. The no-wait policy can be applied here to schedule processors for evaluating subgoals in its local memory. Subgoals with large failure-probability-to-cost ratios (resp. large success-probability-to-cost ratios) and a suitable granularity can be sent to neighboring processors connected by the ring network when AND-parallelism (resp. OR-parallelism) is considered. Load balancing is, therefore, carried out automatically with the shuffle and selection of subgoals.

Yet another problem is on the communication of pruning information from one processor to another processor. When the result on a common subgoal is found, it must be communicated to other processors to stop the processing of a subset of the eliminated solution trees and allow the heuristic values of related tasks to be updated. Similarly, when either a solution tree is found or the goal is proved to be FALSE, all processors should stop further processing. These pruning information are more complicated than the incumbent in the parallel branch-and-bound algorithm implemented in MANIP. In this case, a bus is necessary to broadcast to all processors a subgoal or clause found to be TRUE or FALSE. To minimize the information broadcast, the subgoal or clause should correspond to the nonterminal node closest to the root in the AND/OR-tree. Other processors receiving this information must update all tasks in its Active List by reordering the tasks according to the new heuristic values computed and by eliminating tasks that cannot lead to a solution tree. The complete path from the root to each active task must be maintained to allow the pruning information to be incorporated. The overhead for the propagation of pruning information is extensive and may not be beneficial because the probability-to-cost ratios may be fallible. The no-wait policy is again applied here to continue the evaluation of tasks according to previously computed heuristic values. Information received on subgoals will be used to eliminate unnecessary work when the task is selected.

The last problem on the implementation of a heuristic search lies in the management of the large memory space required. In our study of MANIP, it was found that a direct implementation involving an ordered list of pointers to the subproblems results in a poor locality of access because the subproblems are not ordered by lower bounds in the secondary memory. A specially designed virtual memory that tailors its control strategies to the access behavior of the algorithm was found to be inflexible. The inadequacies of these approaches are due, again, to the strict adherence to the selection rule. A better solution is to use the no-wait policy to implement a modified heuristic search in each processor. In the modified heuristic search, the range of possible lower bounds (in this case, the range of probability-to-cost ratios) is partitioned into b disjoint regions (Figure 5.7). The subproblems in each region are maintained as a separate list. The top portion of each list resides in the main memory, and the rest resides in the secondary memory. Due to the high overhead of secondary-storage accesses, subproblems in a list are expanded in a depth-first manner. Only subproblems in the main memory are candidates for selection. The modified algorithm is identical to a depth-first search when one list is used, and is identical to a pure heuristic search when infinite lists are used. In general, as the number of lists increases, the number of

subproblems expanded decreases and the overhead of the secondary-memory accesses increases. The number of lists should be chosen to maximize the overlap between computations and secondary-memory accesses. Experience on branch-and-bound algorithms showed that two to three lists are adequate.

5.7 Summary

In this chapter, we have investigated the parallel processing of AND/OR-tree searches for decision problems. A logic program is assumed to be represented in the form of an AND/OR-tree. The results that we have obtained can be summarized as follows.

(1) OR-parallelism and AND-parallelism have been unified into OR-parallelism. The objective of an OR-tree search is to select a solution tree and to prove that the goal is TRUE; whereas in an AND-tree search, the objective is to find a set of subgoals to prove that the goal is FALSE. Both types of searches require only one of the correct descendents to be selected and can be considered as OR-tree searches. When the goal is likely to be TRUE, OR-parallelism should be used. In contrast, when the goal is likely to be FALSE, AND-parallelism should be used.

(2) Heuristic information using success probabilities and average overheads of evaluation have been defined to guide the search of logic programs. This information can be generated statically. However, it represents a prior information that does not take into account the query used, the dynamic pruning information obtained, and the values of variables being bound. It is useful to roughly differentiate between tasks that are likely to lead to solution trees and those that might not. Moreover, they define whether the goal is likely to be TRUE or FALSE.

(3) An optimal heuristic search strategy that minimizes the expected overhead of obtaining one solution tree is derived. The search is guided by the probability-to-cost ratios of subgoals.

(4) The architecture of MANIP, proposed earlier for the parallel evaluation of branch-and-bound algorithms, has been extended to implement a heuristic search of logic programs. Problems on selection and virtual-memory support have been considered. The strict adherence to the heuristic search is found to be unrewarding because the probability-to-cost ratios may be fallible, and the overheads of selecting tasks according to these ratios are high. These overheads include the update of the ratios when new pruning information is received, and the selection of tasks from other processors or the secondary memory according to these ratios.

CHAPTER VI
DYNAMIC PROGRAMMING: AN EXAMPLE OF VARIANT ARCHITECTURES REQUIRED BY MULTIPLE FORMULATIONS

Dynamic Programming (DP) is a powerful optimization methodology that is widely applicable to a large number of areas including optimal control, industrial engineering, economics and artificial intelligence [BeD62,BeB72,ClD83, Ney82,Whi69]. Many practical problems involving a sequence of interrelated decisions can be solved by DP efficiently. Bellman has characterized DP through the *Principle of Optimality*, which states that an optimal sequence of decision has the property that whatever the initial state and decision are, the remaining decisions must constitute an optimal decision sequence with regard to the state resulting from the first decision [BeD62]. Subsequently, numerous efforts have been devoted to the rigorous mathematical framework and effective evaluation of DP problems [KaH67,Iba73,Whi69].

In general, DP is an approach that yields a transformation of the problem into a more suitable form for optimization, but is not an algorithm for optimizing the objective function. Moreover, DP can be interpreted differently depending on the computational approach, and efficient implementations are based on the corresponding representations. In this sense, DP is a good example to show the variant architectures required by multiple formulations. The AND-tree, OR-graph, and AND/OR-graph search techniques discussed previously can be applied to solving DP problems.

Bellman, Dreyfus, White, and many others viewed DP as a multistage optimization technique, that is, reducing a single N-dimensional problem to a sequence of N one-dimensional problems [BeD62,Whi69]. The decisions that transform an initial state into a final state must be ordered in terms of stages, and functional equations relate state values in successive stages. The use of monotone sequential processes has been proved by Karp and Held to correspond naturally to DP [KaH7] and has been further developed by Ibaraki [Iba73] and Kumar [Kum84]. On the other hand, Gensi and Montanari have shown that formulating a DP problem in terms of polyadic functional equations is equivalent to searching for a minimum-cost solution tree in an AND/OR-graph with monotone cost function [GMM81]. DP can also be formulated as a special case of the branch-and-bound algorithm, which is a general top-down OR-tree search procedure with dominance tests [MoM76,Iba77,LiW84]. Lastly, nonserial DP has been shown to be optimal among all nonoverlapping comparison algorithms [BeB72,Ros82].

Although DP has long been recognized as a powerful approach to solving a wide spectrum of optimization problems, its applicability has been somewhat limited due to the large computational requirements. Recent advances in Very-Large-Scale Integration (VLSI) and multiprocessor technologies have provided feasible means of implementation. Casti, et al., have studied parallelism in DP [CRL73]. Guibas, Kung and Thompson have proposed a VLSI algorithm for solving the optimal parenthesization problem [GKT79]. Linear pipelines for DP have been described recently [VaR84]. Clarke and Dyer have designed a systolic array for curve and line detection in terms of nonserial DP [ClD83]. Wah, et al., have proposed parallel processing for branch-and-bound algorithms with dominance tests [WLY85]. However, these studies were directed towards the implementation of a few special cases of DP formulations.

In this chapter, we classify DP problems into monadic-serial, polyadic-serial, monadic-nonserial, and polyadic-nonserial. Potential parallelism and the corresponding systolic architectures are investigated for each class. Generally, a

problem can be expressed in different DP formulations, and the efficiency and costs of implementation must be compared.

DP problems can be solved as the search of an optimal path in a multistage graph or as the search for an optimal solution in an AND/OR-graph. We will adopt the graph search as a paradigm to illustrate the various approaches of DP. To take advantage of the regular and limited interconnections of systolic arrays, the graph should have a regular structure. For DP problems in serial formulations, the corresponding graph representations are serial; however, for nonserial problems, they must be converted into serial formulations before efficient implementations can be found. A type of regular graphs of special interest is the multistage graph in which nodes are decomposed into stages, and nodes in one stage are connected to nodes in adjacent stages only. From another point of view a multistage graph can be regarded as a folded OR-tree. Figure 6.1 depicts two examples of multistage graphs.

6.1 Classification of Dynamic-Programming Formulations

A DP formulation is represented in a recursive functional equation whose left-hand side identifies a function name and whose right-hand side is an expression involving the maximization (or minimization) of values of some cost functions. Note that the cost functions are neither restricted to be monadic nor additive; however, they must be monotone in order for the Principle of Optimality to hold. DP formulations are classified according to the form of the functional equations and the nature of recursion.

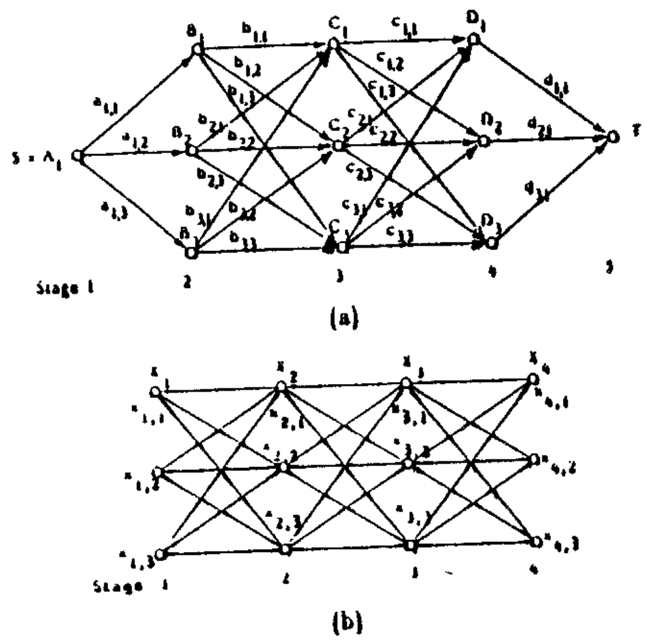


Figure 6.1 Examples of multistage graphs

6.1.1. Monadic versus Polyadic Formulations

A DP formulation is called *monadic* if its cost function involves only one recursive term, otherwise it is called *polyadic*. The distinction is illustrated by an example in finding the minimum-cost path in a multistage graph. For a multistage graph, let $c_{i,j}$ be the cost of an edge. The cost of a path from source, s, to sink, t, is the sum of costs on the edges of the path. Define $f_1(i)$ as the minimum cost of a path from i to t. Thus the cost of a path from i to t via a neighbor j is $c_{i,j}+f_1(j)$. To find $f_1(i)$, paths through all possible neighbors must be compared. Hence

$$f_1(i) = \min_j [c_{i,j} + f_1(j)] \tag{6.1}$$

This equation is termed a *forward functional equation*. Similarly, if $f_2(i)$ is defined as the minimum cost of a path from s to i, then the functional equation becomes

$$f_2(i) = \min_j [f_2(j) + c_{j,i}] \tag{6.2}$$

This equation is termed a *backward functional equation*. The formulations in Eq's (6.1) and (6.2) are *monadic* since each cost function involves one recursive term only.

Eq's (6.1) and (6.2) can be generalized to find the optimal path from any vertex i to any other vertex j. The functional equation is

$$f_3(i,j) = \min_{k \in V_m} [f_3(i,k) + f_3(k,j)] \tag{6.3}$$

where $f_3(i,j)$ is the minimum cost of a path traversing from i to j and passing through a node in Stage V_m. This cost function is *polyadic* because it involves more than one recursive term. Examples of this kind of problems include finding the optimal binary search tree and computing the minimum-cost order of multiplying a string of matrices.

For polyadic DP formulations, Bellman's Principle of Optimality must be generalized to include the statement that "all subsequences of an optimal policy are also optimal." For instance, according to Eq. (6.3), if it is found that the minimum-cost path from i to j passes through k, then the subpath from i to k of this optimal path must be optimal over all subpaths from i to k; so is the subpath from k to j.

6.1.2. Serial versus Nonserial Formulations

The distinction between serial and nonserial optimization problems is based on both the form of their objective functions and the nature of recursion. From the objective function, an optimization problem is said to be serial if all terms of its objective function share one variable with its predecessor term (except for the first term) and another one with its successor term (except for the last term); otherwise, it is said to be nonserial. The name "serial" refers to the interaction graph to represent the problem, in which vertices stand for variables, and an edge exist between two vertices if and only if two variables belong to a term of the objective function [BeB72]. It is obvious that a serial optimization problem has a corresponding interaction graph with a serial structure.

An example of a serial optimization problem is depicted in Figure 1(b). In this multistage graph, each stage, X_i, $1 \leqslant i \leqslant N=4$, stands for a discrete variable, and Node x_{i,j_i} stands for the j'th value taken by Variable X_i. Bold characters are used to denote vectors and matrices, and variables here can be considered as vectors of defined values. If the cost of edge $(x_{i,j_i}, x_{i+1,j_{i+1}})$ is $g_i(x_{i,j_i}, x_{i+1,j_{i+1}})$, then the minimum-cost path from any node in Stage 1 to any node in Stage N is

$$\min_{X} f(X) = \min_{X} \sum_{i=1}^{N-1} g_i(X_i, X_{i+1}) \tag{6.4}$$

where X is the set of discrete variables $\{X_1, \ldots, X_N\}$. In Eq. (6.4), every term of the objective function has two variables that only interact with variables in the neighboring terms. Therefore, Eq. (6.4) is a serial optimization problem.

Many practical DP problems can be represented in a serial formulation. For a traffic-control problem, X_i can be the possible times for the traffic light to be in State i, and the cost on an edge of the graph representation is the difference in

timing. For a circuit-design problem, X_i can be the possible voltages at Point i, and the cost of an edge of the graph representation may be the corresponding power dissipation. For a fluid-flow problem, X_i can be the possible pressure values in the i'th pump, and Function f may be the flow rate for a given pressure. For a scheduling problem, X_i can be the possible task service times for the i'th task, and the edge cost reflects the delay. Note that the optimal-path problem in multistage graphs is a special case of serial optimization problems.

In contrast, the objective function of a general *nonserial optimization problem* has the following form.

$$f(X) = \overset{N}{\underset{i=1}{\Phi}} g_i(X^i) \tag{6.5}$$

where $X = \{X_1, \ldots, X_N\}$ is a set of discrete variables, $X^i \subseteq X$, and Φ is a monotone function relating the g_is together. For example, the following equation is a nonserial optimization problem.

$$\min_{X} \{g_1(X_1, X_2, X_4) + g_2(X_3, X_4) + g_3(X_2, X_5)\}$$

where $X = \{X_1, \ldots X_5\}$.

From the viewpoint of recursion, a DP problem can be represented as a folded AND/OR-tree (or AND/OR-graph) in which the nodes are classified into levels or stages [MaM73]. If this AND/OR-graph has a serial structure such that arcs only exist between adjacent levels, then the corresponding DP problem has a serial formulation. For nonserial DP problems, the dependency between states is not restricted to successive stages, but may exist between states in arbitrary stages. In the corresponding AND/OR-graphs, the arcs are not restricted to successive levels, but may run between any two arbitrary levels.

As an example, consider the problem of finding the optimal order of multiplying a string of matrices. For simplicity, consider the evaluation of the product of four matrices.

$$M = M_1 \times M_2 \times M_3 \times M_4$$

where M_i, $1 \leq i \leq 4$, is a matrix with r_{i-1} rows and r_i columns. Let $m_{i,j}$ be the minimum cost of computing $M_i \times ... \times M_j$. Clearly,

$$m_{i,j} = \min_{i \leq k < j} (m_{i,k} + m_{k+1,j} + r_{i-1} \cdot r_k \cdot r_j) \tag{6.6}$$

The solution to be found is $m_{1,4}$. This formulation is polyadic-nonserial and can be represented as the search of an AND/OR-graph as shown in Figure 6.6, where the AND-nodes denote multiplications and the OR-nodes denote comparisons. In Figure 6.6, the topmost node represents the original problem of multiplying four matrices. This can be achieved in three ways: (1) $(M_1 \times M_2 \times M_3) \times M_4$; (2) $(M_1 \times M_2) \times (M_3 \times M_4)$; or (3) $M_1 \times (M_2 \times M_3 \times M_4)$. These three alternatives are represented by the three AND-nodes in the second level. Note that the first AND-node in the second level, which denotes $m_{1,3} \cdot m_{4,4}$, is connected to the node representing $m_{4,4}$ in the bottommost level. Similarly, the third node is connected to the node representing $m_{1,1}$ in the bottommost level. These arcs do not connect nodes in adjacent levels, hence the formulation in Eq. (6.6) is polyadic-nonserial.

We have classified DP problems in terms of their recursive functional equations and objective functions. Monadic and polyadic DP formulations are distinct approaches to representing various optimization problems, while serial and nonserial optimization problems are problems solvable by the corresponding DP formulations.

6.2 Solving Monadic-Serial DP Problem By Searching Folded OR-Tree

Monadic-serial DP problems can be conveniently solved as the multiplication of a string of matrices. In this section, three efficient systolic designs are presented. The proposed designs do not exploit all potential parallelism of solving a given problem, especially when the number of stages is large. Other parallel designs using different formulations may allow a higher degree of parallelism and will be discussed later.

6.2.1 Searching a Folded OR-Tree as Multiplying a String of Matrices

We have shown that the search for a solution of a problem in a monadic-serial DP formulation can be viewed as finding a path in a multistage graph. For the multistage graph in Figure 6.1(a) and from Eq. (6.2), $f(C_1)$, the minimum cost from C_1 to t, is:

$$f(C_1) = \min \{c_{1,1}+d_{1,1},\ c_{1,2}+d_{2,1},\ c_{1,3}+d_{3,1}\} \tag{6.7}$$

$f(C_2)$ and $f(C_3)$ are obtained similarly.

Eq. (6.7) is similar to an inner-product operation. If we define matrix multiplication in terms of a closed semi-ring (R, MIN, +, +∞, 0), in which 'MIN' corresponds to addition and '+' corresponds to multiplication in conventional matrix multiplications [AHU74], Eq. (6.7) becomes:

$$f(C) = C \cdot D = \begin{vmatrix} f(C_1) \\ f(C_2) \\ f(C_3) \end{vmatrix} = \begin{vmatrix} c_{1,1} & c_{1,2} & c_{1,3} \\ c_{2,1} & c_{2,2} & c_{2,3} \\ c_{3,1} & c_{3,2} & c_{3,3} \end{vmatrix} \begin{vmatrix} d_{1,1} \\ d_{2,1} \\ d_{3,1} \end{vmatrix} \tag{6.8}$$

Likewise, we have:

$$f(B) = B \cdot (C \cdot D)$$

$$f(A) = A \cdot (B \cdot (C \cdot D))$$

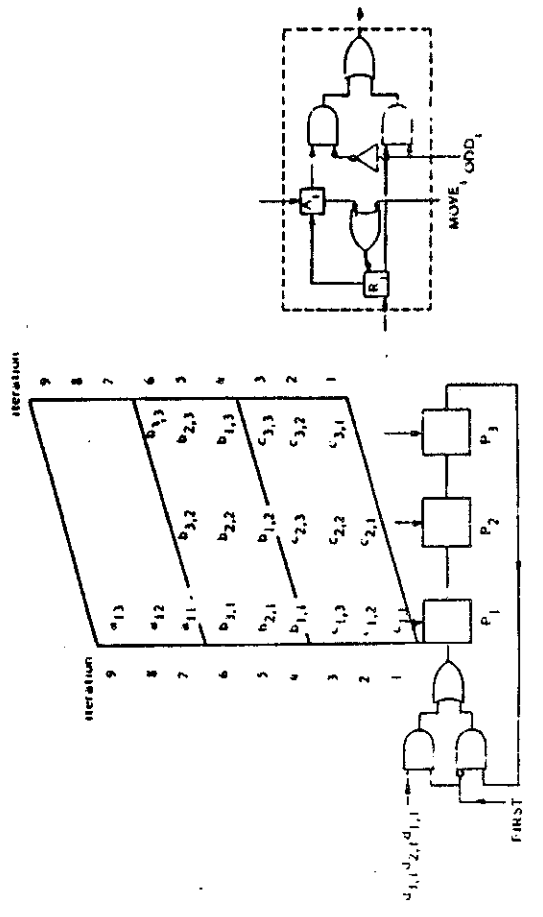
Thus solving the multistage-graph problem with a forward monadic DP formulation is equivalent to multiplying a string of matrices. The order of multiplications is reversed in backward monadic DP formulations.

For a multistage graph with N stages and m vertices in each stage, the computational complexity is (m^2N). For single-source, single-sink problems, the first and last matrices degenerate into row and column vectors, respectively.

6.2.2 Systolic-Array Implementation

Three linear systolic arrays for evaluating problems in monadic-serial DP formulations with parallel inputs are described in this section. The following scheme is based on a combination of two methods of multiplying a matrix with a vector, one of which was discussed by Kung [Kun79]. Figure 6.2(a) depicts a scheme for computing (A·(B·(C·D))) for the multistage graph in Figure 6.1(a). An *iteration* is defined as a shift-multiply-accumulate operation with respect to the time at which a row or column of the input matrix enters a given processor. Note that the same iteration number are carried out at different times in different processors. The iteration numbers are indicated in Figure 6.2(a). In the first three iterations, C·D is evaluated. The control signal FIRST is one; D, the input vector, is serially shifted into the systolic array; and the result vector, $\{f(C_i), i=1,2,3\}$, remains stationary. At the end of the third iteration, FIRST is set to zero. In the following three iterations, B·(C·D) is computed. Note that Matrix B is transposed, and the i'th column of Matrix B is fed into P_i. The input vector, $\{f(C_i), i=1, 2, 3\}$, remains stationary, while the result vector, $\{f(B_i), i=1, 2, 3\}$, is shifted. At the end of the sixth iteration, the output vector $\{f(B_i), i=1, 2, 3\}$ is formed. In the last three iterations, input vectors A and $\{f(B_i), i=1, 2, 3\}$ are shifted into P_1 to form the final result.

For the systolic array in Figure 6.2(a), the data shifted alternate between the input vector and the result vector every three iterations. This alternation can be controlled by the processor structure of P_i depicted in Figure 6.2(b). R_i is a register that stores an element of the input vector, and A_i is the accumulator that stores the temporary result of an element of the result vector. The data paths are controlled by control signals ODD_i and $MOVE_i$. When the number of matrix multiplications is odd, ODD_i is one, hence R_i is connected to the output, and the input vector is shifted along the pipeline. When the number of matrix multiplications is even, ODD_i is zero, A_i is connected to the output, and the result vector is shifted. At the end of a matrix multiplication, the result vector generated becomes the input vector in the next iteration and is moved by the control signal $MOVE_i$ from A_i to R_i.



iteration
a_{13} a_{12} a_{11} $b_{3,1}$ $b_{2,1}$ $b_{1,1}$ $c_{1,3}$ $c_{1,2}$ $c_{1,1}$
$b_{3,2}$ $b_{2,2}$ $b_{1,2}$ $c_{2,3}$ $c_{2,2}$ $c_{2,1}$
$b_{3,3}$ $b_{2,3}$ $b_{1,3}$ $c_{3,3}$ $c_{3,2}$ $c_{3,1}$
$d_{3,1} d_{2,1} d_{1,1}$
FIRST
P_1
P_2
P_3
R
A_i
$MOVE_i$
ODD_i
(a)
(b)

Figure 6.2 A pipelined version of systolic array for computing a string of matrix multiplications

Note that there is a one-cycle delay between switching the control signals for P_{i+1} and P_i.

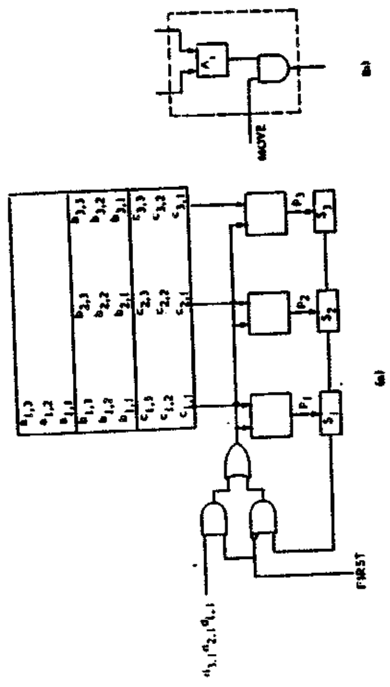
To search a multistage graph with (N+1) stages and m nodes in each intermediate stage (the first and last stages have one node each), it takes N·m iterations with m processors. There is no delay between feeding successive input matrices into the systolic array, and the processors are kept busy most of the time. In contrast, it takes $(N-2)m^2+m$ iterations to solve the problem with a single processor. Define PU, the *processor utilization*, as the ratio of the number of serial iterations to the product of the number of parallel iterations and the number of processors. PU for the above systolic array is:

$$PU = \frac{(N-2)m^2+m}{N \cdot m \cdot m} = \frac{N-2}{N} + \frac{1}{N \cdot m} \qquad (6.9)$$

When N and m are large, PU is very close to 1.

Although the proposed systolic array is designed for matrices in which each element is a single constant, it can be extended to many practical sequentially-controlled systems, such as Kalman filtering, inventory systems, and multistage production processes, in which each matrix element is a vector with many quantized values. In this case, the potential parallelism could be very large.

If broadcast is allowed, the above scheme can be simplified. In what follows, a linear systolic array with parallel inputs and broadcasting is described. Figure 6.3(a) depicts a scheme for computing (A·(B·(C·D))) for the multistage graph depicted in Figure 6.1(a). In this scheme, all input matrices are fed into the systolic array in the same format. In the first three iterations, C·D is evaluated. The control signal FIRST is one; D, the input vector, is broadcast to all PEs; and the intermediate results of $f(C_i)$, i=1,2,3, remain stationary. At the end of the third iteration, the result vector is gated into registers S_1, S_2, S_3, by the control signal MOVE (see Figure 6.3(b)), and FIRST is set to zero. Since FIRST is zero, $f(C_i)$, i = 1,2,3, are fed back and broadcast as new inputs. In the following three iterations, B·f(C) = B·(C·D) is computed. At the end of the sixth iteration, the output vector



$a_{1,3}$ $a_{1,2}$ $a_{1,1}$
$b_{1,3}$ $b_{1,2}$ $b_{1,1}$
$b_{2,3}$ $b_{2,2}$ $b_{2,1}$
$b_{3,3}$ $b_{3,2}$ $b_{3,1}$
$c_{1,3}$ $c_{1,2}$ $c_{1,1}$
$c_{2,3}$ $c_{2,2}$ $c_{2,1}$
$c_{3,3}$ $c_{3,2}$ $c_{3,1}$
$d_{3,1}d_{2,1}d_{1,1}$
P_1
P_2
P_3
S_1
S_2
S_3
FIRST
MOVE
A_i
(a)
(b)

Figure 6.3 A systolic array with broadcasts for computing a string of matrix multiplications.

$\{f(B_i), i=1, 2, 3\}$ is formed. In the last three iterations, input vectors A and $\{f(B_i), i=1, 2, 3\}$ enter PE P_1 to form the final result. PU for the simplified systolic array is the same as Eq.(6.9).

The degree of parallelism of the proposed scheme is restricted by the limited number of input/output ports in a VLSI chip and the fact that the ratio of the computational overhead to the input/output overhead is relatively low in matrix-vector multiplications. The input/output bottleneck is due to the large number of edge costs that must be fed into the systolic array. For the serial optimalization problems formulated by Eq. (6.4) and illustrated in Figure 6.1(b), the edge costs are expressed as functions of the nodes connected, and hence only the values of the nodes have to be input. This results in an order-of-magnitude reduction in the input overhead.

The search for an optimal assignment of X_is in Eq. (6.4) corresponds to the search for the shortest path in a multistage graph, where nodes in each stage represent values that can be assigned to a variable. An example graph with four variables, each of which can take on three quantized values, is shown in Figure 6.1(b). There are multiple sources and sinks, and all possible paths from any vertex in Stage 1 to any vertex in Stage N must be compared. Systolic processing is suitable when the number of quantized values in each stage is constant, and the f_is, the functions to compute edges costs, are independent of i.

To solve Eq. (6.4), the variables can be eliminated one by one. First, X_1 is considered. Since only one term, $f(X_1, X_2)$, is affected by X_1, it is sufficient to compute

$$h(X_2) = \min_{X_1} f(X_1, X_2) \tag{6.10}$$

In other words,

$$h(x_{2,j_2}) = \min_{x_{1,j_1} \in X_1} f(x_{1,j_1}, x_{2,j_2}) \qquad x_{2,j_2} \in X_2$$

The optimization problem then becomes

$$\min_{X} f(X) = \min_{X-\{X_1\}} \left| h(X_2) + \sum_{i=2}^{N-1} f(X_i, X_{i+1}) \right| \tag{6.11}$$

If $h(X_k)$ is defined as

$$h(X_k) = \min_{X_{k-1}} \{h(X_{k-1}) + f(X_{k-1}, X_k)\} \qquad 2 \leq k \leq N \tag{6.12}$$

or

$$h(x_{k,j_k}) = \min_{x_{k-1,j_{k-1}} \in X_{k-1}} \{h(x_{k-1,j_{k-1}}) + f(x_{k-1,j_{k-1}}, x_{k,j_k})\}$$

$$x_{k,j_k} \in X_k,\ 2 \leq k \leq N$$

then $h(x_{k,j_k})$ represents the shortest path from any vertex in Stage 1 to x_{k,j_k}. After eliminating k−1 variables, $X_1, \ldots, X_{k-1}$, the remaining optimization problem becomes

$$\min_{X} f(X) = \min_{X-\{X_1,\ldots,X_{k-1}\}} \left| h(X_k) + \sum_{i=k}^{N-1} f(X_i, X_{i+1}) \right| \tag{6.13}$$

Finally, we get $h(X_N)$, each element of which represents the shortest path from any vertex in Stage 1 to a node in stage N. The problem is solved by comparing the m elements of $h(X_N)$.

Figure 6.4 shows a systolic array with three PEs that performs the search of the graph in Figure 6.1(b). PE P_i consists of three registers, R_i, K_i, H_i, and three operation components, F_i, A_i, C_i. Input data pass through R_i in a pipelined fashion. Feedback data are maintained in K_i and H_i until new data replace them. The operation components, F_i, A_i, C_i, are used to compute function f, and perform additions and comparisons, respectively. For simplicity, function f is assumed to be independent of i, and hence the subscripts in F_i, A_i, and C_i will be dropped. The connections of the registers and operation components are shown in Figure 6.4(b).

The systolic array is initialized by zeroing all registers, H_is and K_is, and by sequentially loading input data in X_1, $x_{1,m}, \ldots, x_{1,1}$, to PEs $P_1, \ldots, P_m$. As the intermediate results are shifted out from P_3, the feedback controller feeds them back in

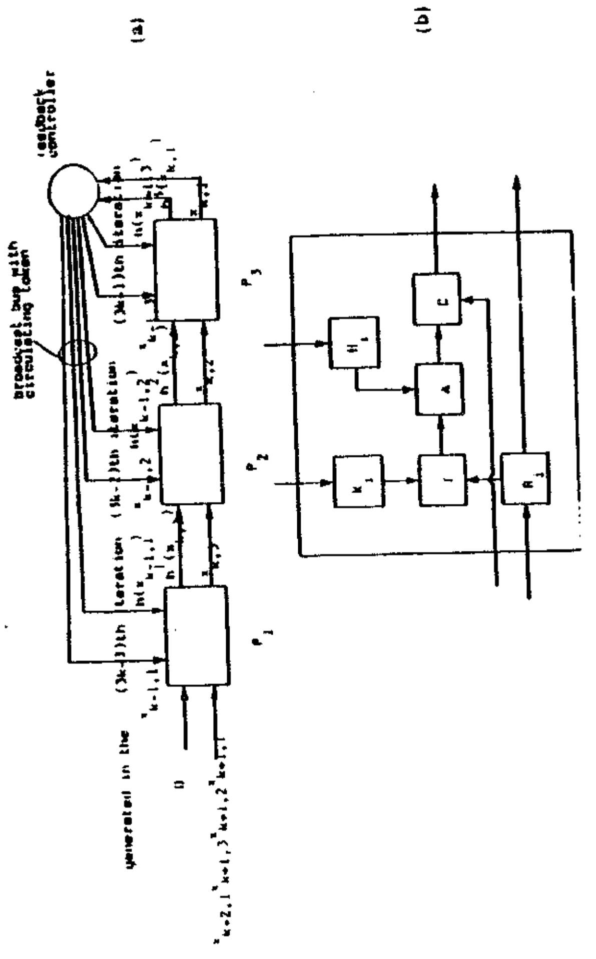


Figure 6.4 A systolic array with serial inputs and outputs to solve a monadic-serial DP problem.

a round-robin fashion. Referring to Figure 6.4(a), when $x_{2,1}$ enters P_1, $x_{1,1}$ and $h(x_{1,1})$ (equals 0) leave P_3 and are fed back to P_1 through the feedback controller. $f(x_{1,1}, x_{2,1})$ and $h^1(x_{2,1}) = \min(0, h(x_{1,1})+f(x_{1,1}, x_{2,1}))$ are then computed in P_1. In the next iteration, $x_{2,2}$ enters P_1, $x_{2,1}$ and $h^1(x_{2,1})$ are shifted to P_2, and $x_{1,2}$ and $h(x_{1,2})$ (equals 0) are fed back by the feedback controller to P_2. In P_2, $f(x_{1,2}, x_{2,1})$ and $h^2(x_{2,1}) = \min(h^1(x_{2,1}), h(x_{1,2})+f(x_{1,2}, x_{2,1}))$ are computed. In P_1, $f(x_{1,1}, x_{2,2})$ and $h^1(x_{2,2}) = \min(0, h(x_{1,1})+f(x_{1,1}, x_{2,2}))$ are computed. When $x_{2,1}$ and $h^2(x_{2,1})$ arrive at P_3, $h(x_{2,1}) = h^3(x_{2,1})$ is evaluated, and $x_{2,1}$ and $h(x_{2,1})$ are fed back to P_1 at the end of this iteration. Input data are continuously shifted into the pipeline, and the process is repeated. For the graph in Figure 6.1(b), the process is completed in fifteen iterations.

In general, to evaluate the optimal path for an N-stage graph, each with m quantized values, a pipeline with m PEs is needed. Between the $((k-1)m+1)$'st and $(k*m)$'th iterations, $2 \leqslant k \leqslant N$, $x_{k,1}, \ldots, x_{k,m}$ enter the R-pipeline; $x_{k-1,i}$ and $h(x_{k-1,i})$, $1 \leqslant i \leqslant m$, are fed back to registers K_i and H_i in the $((k-1)m+i)$'th iteration; and $h^m(x_{k,1}), h^{m-1}(x_{k,2}), \ldots, h^1(x_{k,m})$ are obtained at the end of the $(k*m)$'th iteration in $P_m, \ldots, P_1$. After $N*m$ iterations, $h^m(x_{N,1}), \ldots, h^1(x_{N,m})$ are obtained in $P_m, \ldots, P_1$, and the final solution is obtained in P_m by comparing $h(x_{N,1}), \ldots, h(x_{N,m})$. This is done by setting F=0 in the last m iterations and circulating the values of $h(x_{N,i})$, $1 \leqslant i \leqslant m$, through the pipeline. Therefore, the total computational time is $(N+1)m$ iterations, each of which includes the time for the computation of function f, one addition, and one comparison. PU for this scheme is $((N-1)m^2+m)/((N+1)m \cdot m) \approx 1$.

Although distinct feedback lines are shown in Figure 6.4(a), only one of the feedback lines is used in any iteration. Hence a single broadcast bus suffices, and the station to pick up the data from the bus is controlled by a circulating token.

If the optimal path in addition to the optimal cost value is desired, N path registers, each of which can store m indices, are needed in P_m. In the computation of $h^m(x_{k+1,i}) = \min_j \{h(x_{k,j})+f(x_{k,j}, x_{k+1,i})\}$, $1 \leqslant k \leqslant N-1$, index j, $1 \leqslant j \leqslant m$, of the edge $(x_{k,l}, x_{k+1,j})$ belonging to the optimal path from any vertex in Stage 1 to Vertex $x_{k+1,j}$ must be propagated in the pipeline and is known to P_m. Index l is stored in the l'th word of the k'th path register. The pointers stored in the path registers are used to trace the optimal path at the end of the computation.

6.3 Solving Polyadic-Serial DP Problems by Searching AND Trees

Recall that a serial optimization problem can be solved as the multiplication of a string of matrices. However, a problem expressed in a monadic-serial formulation does not exploit all the potential parallelism because the order of matrix multiplications is fixed. On the other hand, there is more flexibility for parallelism when the problem is formulated in a polyadic equation because the matrices can be multiplied recursively by a divide-and-conquer algorithm.

As mentioned in Chapter 2, one important issue in parallel divide-and-conquer algorithms is the *granularity of parallelism* [WLY85]. This is the minimum size of a subproblem that is evaluated by a processor[6] in order to achieve the optimal performance, as measured by either the PU or the AT^2 (or (area) × (computational time)) criteria. In Chapter 2 we have investigated the optimal granularity of the parallel AND-tree search. Note that when multiplying a string of matrices, the amount of computations to combine two nodes in any level of the corresponding AND-tree is independent of the problem size, i.e., $C(N) = \Theta(1)$. Consequently, the results obtained in Chapter 2 on the optimal granularity of parallel AND-tree searches can be applied here.

Consider the polyadic-serial DP formulation in Eq. (6.3) for the multistage-path problem in Figure 6.1(b).

$$f_3(s,t) = \min_{k \in \{v_{2,1}, v_{2,2}, v_{2,3}\}} \{f_3(s,k) + f_3(k,t)\} \qquad (6.14)$$

[6] Processors and systolic arrays are synonymous here.

where $f_3(i,j)$ is the cost of the optimal path from i to j, and k is a node in Stage 2 of the graph. In matrix notations, let $f_3(V_i,V_j)$ be a cost matrix, each element of which denotes the cost of the optimal path from a vertex in Stage i to a vertex in Stage j. It is easy to see, for an intermediate stage k between i and j, that:

$$f_3(V_i,V_j) = f_3(V_i,V_k) \cdot f_3(V_k,V_j) \qquad (6.15)$$

This formulation allows a string of matrix multiplications to be reduced to two smaller strings of matrix multiplications.

The fastest way to multiply N m-by-m matrices is to locate the matrices in the leaves of a complete binary tree of height $\lceil \log_2 N \rceil$. The N-stage graph problem can be solved in $\lceil m \lceil \log_2 N \rceil \rceil$ time units with $\lceil N/2 \rceil$ matrix-multiplication systolic arrays [LiW85]. The PU and AT^2 measures, however, are relatively low when (N) processor are used. Suppose that each processor (or systolic array) deals with the multiplication of a pair of matrices and the time to implement a matrix multiplication is taken as unity (assuming that all matrices have identical dimensions). To multiply a string of N matrices, KT and KT^2 are minimum when $\Theta(N/\log N)$ processors are used and N is sufficiently large.

So far, the matrices are assumed to have identical dimensions. When this is not true, the order in which the matrices are multiplied together has a significant effect on the total number of operations. Finding the optimal order of multiplying a string of matrices with different dimensions is itself a polyadic-nonserial DP problem, the so-called secondary optimization problem [BrE70,BeB69]. Guibas, Kung, and Thompson have proposed a systolic array to solve the optimal parenthesization problem, which can be used to compute the minimum-cost order of multiplying a string of matrices [GKT79]. Once the optimal order is found, the processors can be assigned to evaluate the matrix multiplications in the defined order and in an asynchronous fashion. In this sense, the tree of matrix multiplications can be treated as a dataflow graph.

6.4 Solving Polyadic-Serial DP Problems by Searching Folded AND/OR-Trees

In this section, we discuss the evaluation of polyadic-serial DP problems as AND/OR-graph searches. AND/OR-graphs are naturally obtained by representing the DP problem using a problem-reduction method. The mapping of a regular AND/OR-graph onto a systolic array is straightforward and will be illustrated in the next section.

Polyadic-serial problems are discussed with respect to the search of a multistage graph as formulated by Eq. (6.4). Suppose an (N+1)-stage graph, with stages from 0 to N and m nodes in each stage, is divided into p subgraphs, each of which contains N/p+1 consecutive stages. For simplity, assume that $N=p^{\theta}$, where θ is a non-negative integer. The minimum-cost path has to pass through one and only one vertex in Stage 0, N/p, ..., p N/p in the segmented graph. The cost of a path equals the sum of costs of the p subpaths. If all the m^2 subpaths from the m vertices in Stage $i\,N/p$ to the m vertices in Stage $(i+1)\,N/p$, $0 \leqslant i \leqslant p-1$, have been optimized, there are m^{p+1} possible combinations of subpaths from Stage 0 to Stage N that must be considered for the optimal path. Using a divide-and-conquer algorithm, each subgraph with N/p+1 stages is further divided into p smaller subgraphs. This partitioning process continues until each subgraph has one stage.

The partitioning process can be conveniently represented as an AND/OR graph, in which an AND-node corresponds to a subproblem sum, and an OR-node corresponds to alternative selections or comparisons. In this case, we have a regular AND/OR-graph of height $2\cdot\log_p N$, whose AND-nodes have p branches (p-arc nodes) and whose OR-nodes have m^{p-1} branches (m^{p-1}-arc nodes). Figure 6.5 shows an AND/OR-graph that represents the reduction of the multistage-graph problem with m=2 and p=2 from three stages to one stage. The four nodes at the top of the AND/OR-graph represents the four possible alternate paths in the reduced single-stage graph. The shortest path is obtained by a single comparison of these paths.

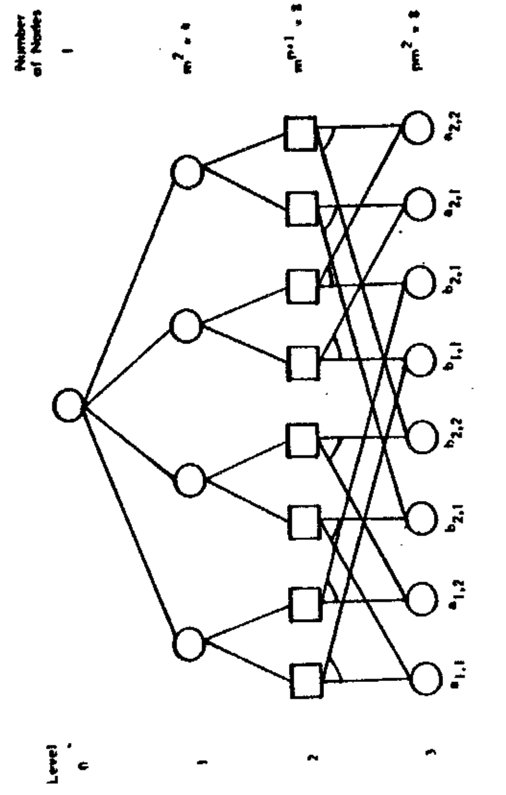


Figure 6.5 An AND/OR graph representation of the reduction in finding an optimal path in a 3-stage graph to a 1-stage graph.

6.5 Parallel Processing of Nonserial DP Problems

It has been shown that unrestricted nonserial optimization problems are NP-hard, but problems with a favorable pattern of term interactions may be solved efficiently [BeB72]. The key of DP is to break a complex optimization problem into a sequence of easier subproblems. In serial optimization problems, variables are shared by successive terms in the objective function and hence can be dealt with one by one. This serial structure allows efficient parallel processing, especially systolic processing. On the other hand, to implement nonserial DP problems by systolic processing, they may have to be transformed into the corresponding serial formulations before they are implemented. This transformation is possible if the nonserial problems have some special structures.

Generally speaking, there are two ways to convert a nonserial formulation into a serial one. One way is to combine several primary variables into a new variable. Another way is to transform an irregular AND/OR-graph into regular one by adding dummy nodes. The former one is suited for solving monadic-nonserial problems, and the latter one is usually applied to solving polyadic-nonserial problems. These methods are illustrated by some examples in the following sections.

6.5.1 Solving Monadic-Nonserial Problems

An approach to solve a monadic-nonserial problem with some structural properties is to first convert it into a monadic-serial problem, such as a multistage graph-search problem, and to map the serial formulation into systolic arrays.

For the nonserial formulation in Eq. (6.5), a multistage optimization procedure can be carried out separately for each variable V_i. Of course, this optimization must be performed on all values of the independent variables that "interact" with V_i. Let $V_{i_1}, \ldots, V_{i_k}$ be the variables that are related to V_i in one or more functional terms. The cost function can be written as:

$$f(v_1, \ldots, v_n) = \min_{v_k \in V_k} \left[h_1(v_i, v_{i_1}, \ldots, v_{i_k}) + h_2(v_1, \ldots, v_{i-1}, v_{i+1}, \ldots, v_n) \right] \qquad (6.18)$$

where h_2 is a function independent of V_i. By denoting $h_{1,opt}$ as $\min_{v_i \in V_i} h_1(v_i, v_{i_1}, \ldots, v_{i_k})$, the cost function can be rewritten as:

$$f(v_1, \ldots, v_n) = \min_{V-V_i} \left[h_{1,opt}(v_{i_1}, \ldots, v_{i_k}) + h_2(v_1, \ldots, v_{i-1}, v_{i+1}, \ldots, v_n) \right] \qquad (6.19)$$

A multistage optimization process is, therefore, a step-by-step elimination of all variables. The computational time and storage depend on the number of elements in the domain of function h_1. Eq. (6.19) can be treated as a monadic-serial form if the evaluation of $h_{1,opt}$ is done separately.

The method is illustrated by the following example. For instance, if $V = V_1 \bigcup \ldots \bigcup V_N$, and the objective function is:

$$f(V) = \min_{v_i \in V_i} \left[g_1(v_1, v_2, v_3) + g_2(v_2, v_3, v_4) + \ldots + g_{N-2}(v_{N-2}, v_{N-1}, v_N) \right] \qquad (6.20)$$

Let $h_1(v_2, v_3) = \min_{v_1 \in V_1} g_1(v_1, v_2, v_3)$, we have:

$$\min_V f(V) = \min_{V-V_1} \left[h_1(v_2, v_3) + \sum_{i=2}^{N-2} g_i(v_i, v_{i+1}, v_{i+2}) \right] \qquad (6.21)$$

If $h_k(v_{k+1}, v_{k+2})$ is defined as:

$$h_k(v_{k+1}, v_{k+2}) = \min_{v_k \in V_k} \{ h_{k-1}(v_k, v_{k+1}) + g_k(v_k, v_{k+1}, v_{k+2}) \} \qquad (6.22)$$

Eq. (6.22) represents the minimum of the summation of the first k terms of f(V). After eliminating k variables, $V_1, \ldots, V_k$, the remaining optimization problem becomes:

$$\min_V f(V) = \min_{V-(V_1 \bigcup \ldots \bigcup V_k)} \left[h_k(v_{k+1}, v_{k+2}) + \sum_{i=k+1}^{N-2} g_i(v_i, v_{i+1}, v_{i+2}) \right] \qquad (6.23)$$

The monadic DP procedure, thus, eliminates the variables in the order $V_1, \ldots, V_N$. If the variables V_2 and V_3 are treated as a single variable in a stage, and m_k, $1 \leqslant k \leqslant n$, quantized values are allowed for V_k, then there would be $m_2 \cdot m_3$ states in this stage, and $m_1 \cdot m_2 \cdot m_3$ steps are required to eliminate V_1, in which a step consists of a computation of function f, an addition, and a comparison operation. The process of eliminating the remaining variables is repeated until V_{N-1} and V_N remain. The optimal solution is obtained by comparing all values of $h_{N-2}(v_{N-1},v_N)$. The total number of steps required to compute Eq. (6.20) is:

$$\sum_{k=1}^{N-2}(m_k \cdot m_{k+1} \cdot m_{k+2}) + m_{N-1} \cdot m_N \qquad (6.24)$$

In short, the monadic-nonserial problem in Eq. (6.20) is solved from the following serial problem.

$$\min_{V} f(V) = \min_{V} \left[g_1'(V_1',V_2') + g_2'(V_2',V_3') + \ldots + g_{N-2}'(V_{N-2}',V_{N-1}') \right] \qquad (6.25)$$

where the new variable V_i' is combined from V_i and V_{i+1}. From this example, it is observed that more operations are needed for evaluating monadic-nonserial DP problems than that of monadic-serial DP problems. However, the potential parallelism is higher, and there is no increase in delay in processing the transformed problem. With additional control, the linear systolic array presented earlier can be applied to evaluate monadic-nonserial DP problems.

6.5.2 Solving Polyadic-Nonserial DP Problems by Searching AND/OR-Graphs

AND/OR-graphs can be sequentially searched in a breadth-first bottom-up fashion, which expands nodes by levels from the bottom up [Nil80]. Since an acyclic AND/OR-graph can be viewed as a folded tree, searching the AND/OR-graph can be accomplished by searching the corresponding AND/OR-tree. In a parallel AND/OR-tree search, the nodes in the tree are evaluated in parallel in a bottom-up fashion. The parallel architecture can be designed with a flexible interconnection, so a processor can be dynamically assigned when it is free, or can be designed with

a limited interconnection, so a static evaluation order is maintained for a given problem. A dataflow processor is an example of the first alternative. We will investigate the second alternative here.

Parallel AND/OR search of polyadic-serial DP problems is a special case of that of polyadic-nonserial DP problems. For nonserial DP problems, the dependency between states is not restricted to successive stages, but may exist between states in arbitrary stages. In the corresponding AND/OR-graphs, the arcs are not restricted to successive levels, but may run between any two arbitrary levels. It may be difficult to map an irregular AND/OR-graph to a systolic array with a regular interconnection structure. The nonserial AND/OR-graph may have to be transformed into a serial one before the mapping is done.

The strategy is illustrated by the problem of finding the optimal order of multiplying a string of matrices. For simplicity, consider the evaluation of the product of four matrices,

$$M = M_1 \times M_2 \times M_3 \times M_4$$

where M_i, $1 \leqslant i \leqslant 4$, is a matrix with r_{i-1} rows and r_i columns. Let $m_{i,j}$ be the minimum cost of computing $M_i \times \ldots \times M_j$. Clearly,

$$m_{i,j} = \min_{i \leqslant k < j} (m_{i,k} + m_{k+1,j} + r_{i-1} \cdot r_k \cdot r_j) \qquad (6.25)$$

The solution to be found is $m_{1,4}$. This problem can be represented as a search of an AND/OR-graph, where the AND-nodes denote additions and the OR-nodes denote comparisons (Figure 6.6)

The AND/OR-graph in Figure 6.6 can be mapped directly into six processors connected by multiple broadcast busses. Each processor evaluates an OR-node and its immediate descendent AND-node(s). The broadcast structure is necessary because a processor has to communicate with multiple processors and not its neighbors alone. Let $T_d(k)$, $1 \leqslant k \leqslant N$, be the time to find the optimal order of multiplying k matrices. Then,

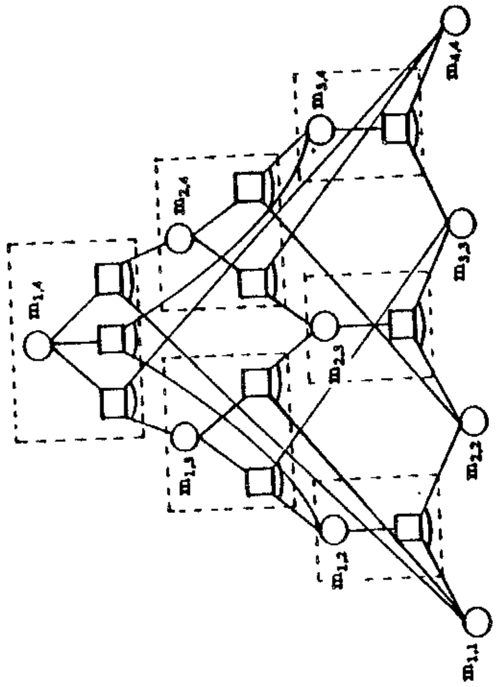


Figure 6.6 An AND/OR graph representation of finding the optimal order of multiplying a string of four matrices

$$T_d(k) = \begin{cases} T_d(\lfloor k/2 \rfloor) + \lfloor k/2 \rfloor & \text{if } k>1 \\ 1 & \text{if } k=1 \end{cases} \tag{6.26}$$

This is true because, once the subproblems of size $\lfloor k/2 \rfloor$ are completed, the results can be used as inputs to subproblems of size larger than $\lfloor k/2 \rfloor$. In the following step, only subproblems of size $\lfloor k/2 \rfloor+1$ can be completed, and the results will be available as inputs to subproblems of size larger than $\lfloor k/2 \rfloor+1$. Thus it takes $\lfloor k/2 \rfloor$ steps to solve a subproblem of size k. In each step, two additions and two comparisons are performed.

Proposition 6.1: The solution to Eq. (6.26) is $T_d(N) = N$.

Proof: The proposition can be proved by induction. □

Although the above scheme is fast, it requires a large number of broadcast busses and may be difficult to implement when the problem size is large. To overcome this, we can transform the nonserial problem into a serial one, that is, convert the general AND/OR-graph into a simpler graph in which all arcs connect nodes in successive levels. Suppose that an OR-node and its immediate parent are not located in adjacent levels, then the OR-node is connected to its parent via other intermediate nodes in adjacent levels. The additional connections are represented as dotted lines in Figure 6.7. This pipelined design is suitable for VLSI implementation because the interconnections can be mapped into a planar structure.

The computational time for the scheme in Figure 6.7 is analyzed here. Let $T_p(k)$, $1 \leqslant k \leqslant N$, be the time to find the optimal order of multiplying k matrices. Once a subproblem of size $\lfloor k/2 \rfloor$ is solved, it takes $\lfloor k/2 \rfloor$ time units to transfer the result into the processor that evaluates the subproblem of size k. Analogous to the explanation for Eq. (6.26), it takes $\lfloor k/2 \rfloor$ steps to solve the subproblem of size k after the results of subproblems of size $\lfloor k/2 \rfloor$ are available. Consequently,

$$T_p(k) = T_p(\lfloor k/2 \rfloor) + 2\lfloor k/2 \rfloor \tag{6.27}$$

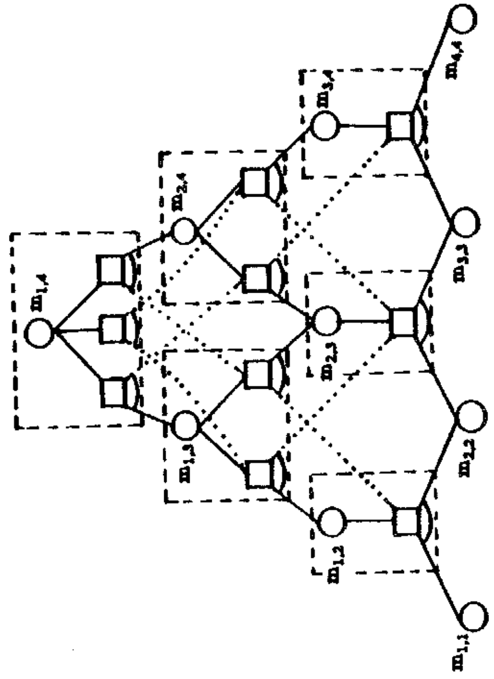


Figure 6.7 A structural AND/OR graph representation of finding the optimal order of multiplying a string of four matrices

Proposition 6.2: Suppose $T_p(1)=2$, then the solution to Eq. (6.27) is $T_p(N)=2N$.

Proof: The proposition can be proved by induction. □

A systolic array usually demands that all operands for an operation arrive at a processor simultaneously and that the computations are carried out in a pipelined fashion. Recall from Theorem 6.1 that the optimal branching factors for AND- and OR-nodes are two and m ($\geqslant 2$), respectively. Hence, it is necessary for two data items to arrive at an AND-node simultaneously, and that the OR-nodes are evaluated sequentially. Keeping the timing and Proposition 6.2 in mind, it is not difficult to design a systolic algorithm for this problem. In fact, the derived structure is the same as that proposed by Guibas, Kung and Thompson [GKT79].

The above example demonstrates the relationship between an AND/OR-graph representation of polyadic DP problems and the corresponding systolic design. In general, starting from an AND/OR-graph, a systolic array with planar interconnections can be designed by first serializing links that connect nodes not in adjacent levels in the AND/OR-graph, and by designing the appropriate control signals. As shown in the examples, the transformation may introduce additional delay and redundant hardware in the implementation.

6.6 Summary

Dynamic programming is a good example to show the variant architectures required by multiple formulations. In this chapter, DP formulations have been classified according to the objective functions and the structure of the corresponding AND/OR graphs. A given DP problem can usually be formulated in multiple ways, e.g., folded OR-tree (multistage graph), AND-tree, folded AND/OR-tree, and AND/OR-graph. Hence it is important to compare the alternative implementations. The applicability of systolic processing is most suitable when the formulation is serial.

Many sequential decision problems have serial formulations that can be considered as searching a multistage graph. If there are a large number of states and/or quantized values in each stage, then a monadic formulation is more appropriate, and the problem is efficiently solved as a serial string of matrix multiplications. On the other hand, if the number of stages is large, then the problem should be put into a polyadic formulation. The matrices are grouped into a binary tree and multiplied by a divide-and-conquer algorithm. We have found the AT^2 lower bound for multiplying a string of N m-by-m matrices, and have proved that dividing the string into $(N/\log_2 N)$ groups and multiplying each by a systolic array is optimal in the sense of achieving this lower bound.

When the formulation is nonserial, it may be necessary to transform the problem into a serial formulation before an efficient implementation can be found. A monadic-nonserial formulation can be transformed into a monadic-serial one by grouping state variables. A problem in a polyadic-nonserial formulation can be represented as the search of an optimal solution in an AND/OR graph, which can be transformed into an AND/OR graph for a serial problem by adding dummy nodes. The transformed AND/OR graph can be mapped directly into a planar systolic array by using appropriate control signals. The additional hardware and delay introduced is problem dependent. A summary of variant architectures required by multiple formulations of dynamic programming are shown in Table 6.1.

Table 6.1 Functional requirements of dynamic programming

Formulation		Problem characteritic	Suitable method	Functional requirements
Serial	Monadic	Many states or quantized values in each stage	Search folded OR-trees (as string of matrix multiplication)	Systolic proc.; feedback control
	Polyadic	Many stages	Search AND-trees or folded AND/OR-trees	Loose coupling for fine grain Tight coupling for coarse grain
Nonserial	Monadic	Variables can be eliminated one by one	Transform into monoadic-serial (representation by grouping variables)	Systolic proc. with additional control
	Polyadic	Unstructured problems	Search AND/OR-graphs; transform into serial AND/OR graphs	Dataflow proc.; Systolic proc. with additional control

CHAPTER VII
CONCLUSIONS AND FUTURE WORK

Combinatorial search is an essential technique in artificial intelligence and operations research. Due to the ubiquity of search and its difficulty in implementations, parallel processing of combinatorial search has become a key issue in designing new generation computer systems. Since search algorithms are usually nondeterministic and some subproblems might be pruned, conventional parallel architectures might not be suitable. In this thesis we have studied the computational efficiencies and architectural requirements of parallel combinatorial searches. These results are helpful to researchers and designers of new generation computer systems.

Combinatorial searches can be classified into AND-trees/graphs, OR-trees/graphs, and AND/OR-trees/graphs according to the function of the nodes included. AND-graphs and OR-graphs are special cases of AND/OR graphs. Since different kinds of combinatorial searches have different computational behavior, we have studied the three kinds of combinatorial searches separately.

Owing to the nondeterminism, it is difficult to accurately estimate the performance of parallel searches. but the bounds on performance can be evaluated. In Chapter 3 of this thesis, we have found the performance bounds of parallel OR-tree searches without dominance tests. For a best-first search, the search time depends on the height of the search tree, the problem complexity, and the number of processors used. For depth-first and breadth-first searches, the search time is also related to the number of distinct incumbents obtained during the search. In contrast to previous results, our theoretical analysis and simulations have shown

that a near-linear speedup can be achieved with respect to a large number of processors.

For a parallel OR-tree search, anomalies may occur frequently when approximation and dominance tests are applied. We have discovered that these anomalies are due to ambiguities of the heuristic function, inconsistencies between the selection and elimination rules, and variations in the sequence of feasible solutions obtained. Ambiguity in the heuristic function can be resolved by augmenting the original heuristic function with a tie-breaking rule (say by using path numbers). Consistency of dominance tests with the heuristic function can be satisfied by redefining dominance relations. Various conditions to cope with anomalies with respect to different search strategies have been investigated in this thesis.

Efficiencies (both in time and space) of best-first, depth-first, and breadth-first searches have been compared. A best-first search has been found to be a robust search strategy in the sense of avoiding detrimental anomalies and achieving a near-linear speedup within a large range of the number of processors. However, it depends on the accuracy of the problem-dependent lower-bound function, and is best applied in parallel processing when the accuracy of the lower-bound function is moderate. If the lower-bound function is very accurate or inaccurate, a depth-first search is more cost-effective.

For an AND-tree search using divide-and-conquer algorithms, the optimal granularity depends on the complexity of the problem to be solved. We have analyzed the relationship between the asymptotic processor utilization and the number of the processor used. The optimal granularity with respect to the KT^2 criterion has also been proved and simulated, where K and T are the number of processors and computational time, respectively. For problems with $\Theta(N)$ and $\Theta(N \log N)$ complexity, we have proved that using, respectively, $\Theta(N/\log N)$ and $\Theta(\log N)$ processors are optimal in the sense of achieving the lower bound of KT^2.

For decision problems solved by AND/OR tree searches, we have shown that OR-parallelism and AND-parallelism can be unified into OR-parallelism. When the goal is likely to be TRUE, OR-parallelism should be used. In contrast, when the goal is likely to be FALSE, AND-parallelism should be used. Heuristic information using success probabilities and average overheads of evaluation have been defined to guide the search of logic programs that are represented as AND/OR trees. These information can be generated statically and are useful to roughly differentiate between tasks that are likely to lead to solution trees and those that are not. An optimal heuristic search strategy that minimizes the expected overhead of obtaining one solution tree is derived. The search is guided by the probability-to-cost ratios of subgoals. The architecture of MANIP proposed for the parallel evaluation of branch-and-bound algorithms has been extended to implement a heuristic search of logic programs.

Parallel processing of dynamic programming has been investigated to show the variant architectures required by multiple formulations. Dynamic programming formulations have been classified according to the structures of the functional equations, the nature of recursions, and objective functions. A given problem can usually be formulated in multiple ways, hence it is important to compare the alternative implementations. The applicability of systolic processing is most suitable when the formulation is serial. If there are a large number of states and/or quantized values in each stage, then a monadic formulation is more appropriate, and the problem is efficiently solved as a serial string of matrix multiplications. Three efficient systolic arrays for solving monadic-serial dynamic programming problems have been studied in Chapter 6. On the other hand, if the number of stages is large, then the problem should be put into a polyadic formulation. The inputs are grouped into an AND tree and solved by a divide-and-conquer algorithm. When the formulation is nonserial, it may be necessary to transform the problem into a serial formulation before an efficient implementation can be found. A monadic-nonserial

monadic-serial one by grouping state variables. A problem in a polyadic-nonserial formulation can be represented as the search of an optimal solution in an AND/OR graph, which can be transformed into an AND/OR graph for a serial problem by adding dummy nodes. The transformed AND/OR graph can be mapped directly into a planar systolic array by using appropriate control signals. The additional hardware and delay introduced is problem dependent.

One of the important objectives of this thesis is to understand the functional requirements of various search paradigms. It is found that the functional requirements for efficient search of OR-trees with lower-bound tests only are a loosely coupled interconnection of processors with load balancing capability, and a method of concurrent update and broadcast of the incumbent. If dominance tests are applied, the corresponding functional requirements depend on the size of the current dominating nodes, N_d, and the structure of the dominance relation. When $|N_d|$ is small, unstructured dominance tests can be implemented by broadcast busses or global registers. For structured dominance tests, it may be possible to partition the search tree and localize the dominance tests. On the other hand, when $|N_d|$ is large, it is necessary to partition N_d into subsets and to perform the dominance tests in parallel. This results in tight coupling of the processors. Dynamic programming is an approach that applies dominance tests systematically. Dynamic programming problems can be solved by systolic arrays if the dominance relations have some structural features. The functional requirements for both parallel AND-tree and OR-tree searches hold for parallel AND/OR-tree searches, e.g. the granularity of parallelism, the parallel selection of nodes for evaluation, and the intelligent pruning of unnecessary nodes. Processors should know the global state of search to select the nodes for expansion, and be able to inform other processors to prematurely terminate their tasks when necessary. The architecture designed should support the dissemination of this information. Finally,
AND/OR

become an AND/OR tree and evaluated level by level. To search general unstructured AND/OR graphs, a data flow computer may be more suitable. Many researchers have addressed design of graph-reduction and data-flow computers [DaR81, Den80, KLT84, TBH82]. However, Most of these efforts were devoted to AND-graph reduction and numerical computations. Applying data-flow mechanisms to nondeterministic computations, such as searching general AND/OR graphs, is still an open problem [AmH84,WaL85,WaL86].

Parallel processing of combinatorial search is an active research area in which a lot of problems are still unanswered. The following are some key issues that can be studied in the future. First, the performance bounds with respect to time and space for AND/OR-tree (graph) searches, rather than the time only, should be investigated. That is, we need to determine the condition under which an increase in space, i.e., the number of processors and/or memory space, will be worthwhile. An optimal strategy for searching AND/OR-trees (graphs) that minimizes kT^2 is expected as a guideline. To implement AND/OR-tree (graph) searches efficiently, heuristic strategies should be studied. will be designed. Second, some conditions under which local semantic information is useful to guide general AND/OR-tree (graph) searches should be investigated. We have found that for OR-tree searches global information does not have to be collected to determinate the selection order of subproblems in order to minimize the expected completion time. This result should be extended to AND/OR-tree (graph) searches. Third, in many applications, the problems to be solved might not be formulated in regular AND/OR-graphs directly. To take the advantage of VLSI technology, these irregular AND/OR-graphs should be transformed into regular ones before implementation. Lastly, dataflow architectures are suitable to implement nonserial AND/OR-graph searches. An efficient dynamic scheduling method is necessary to achieve a high performance.

LIST OF REFERENCES

LIST OF REFERENCES

[AHU74] A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, 1974.

[ABD82] S. Akl, D. Barnard and R. Doran, "Design, Analysis, and Implementation of a Parallel Tree Search Algorithm," *IEEE Trans. on Pattern Analysis and Machine Intelligence*, Vol. PAMI-4, pp. 192-203, March 1982.

[AmH84] M. Amamiya and R. Hasegawa, "Dataflow Computing and Eager and Lazy Evaluations," *New Generation Computing*, Vol. 2, Ohmsha, Ltd. and Springer-Verlag, pp. 105-129, 1984.

[BaB82] D. H. Ballard and C. Brown, *Computer Vision*, Prentice-Hall, 1982

[BaF81] A. Barr and E. A. Feigenbaum, *The Handbook of Artificial Intelligence*, Vol's 1, 2, and 3, Kaufmann, 1981.

[Bar83] J. Barnett, "Optimal Searching from AND Nodes," *Proc. IJCAI*, pp. 786-788, 1983.

[BaS78] G. Baudet and D. Stevenson, "Optimal Sorting Algorithms for Parallel Computers," *IEEE Trans. on Computers*, Vol. C-27, No. 1, pp 84-87, Jan. 1978.

[Bau78a] G. Baudet, *The Design and Analysis of Algorithms for Asynchronous Multiprocessors*, Tech. Rep., Dept. of Computer Science, Carnegie-Mellon Univ. 1978.

[Bau78b] G. Baudet, "On the Branching Factor of the Alpha-Beta Pruning Algorithm," *Artificial Intelligence* Vol. 10, pp. 173-199, 1978.

[BeB69] U. Bertele and F. Brioschi, "A New Algorithm for the Solution of the Secondary Optimization in Non-serial Dynamic Programming," *J. of Math. Analysis and Applications*, Vol. 27, pp. 565-574, 1969.

[BeB72] U. Bertele and F. Brioschi, *Nonserial Dynamic Programming*, Academic Press, New York, 1972.

[BeD62] R. Bellman and S. Dreyfus, *Applied Dynamic Programming*, Princeton University Press, 1962.

[BeK79] J. Bently and H. T. Kung, "A Tree Machine for Search Problems," *Proc. Int'l Conf. on Parallel Processing*, IEEE, pp. 257-266, 1979.

[Bra84] M. Brady, "Artificial Intelligence and Robotics," in *Robotics and Artificial Intelligence*, M. Brady, L. A. Gerhardt, and H. F. Davidson (eds.), Springer-Verlag, 1984

[BrE70] F. Brioschi and S. Even, "Minimizing the Number of Operations in Certain Discrete-Variable Optimization Problems," *Operations Research*, Vol. 18, pp 67-81, 1970.

[Bro80] S. Browning, "A Tree Machine," *Lambda*, Vol. 1, pp. 31-36, 1980.

[BuH84] F. Burton and Huntbach, "Virtual Tree Machines," *IEEE Trans. on Computers*, Vol. C-33, pp. 278-280, 1984.

[Car84] D. A. Carlson, "Parallel Processing of Tree-Like Computations," *Proc. of 4th Int'l Conf. on Distributed Computing Systems*, pp. 192-198, May 1984.

[CFR85] M. D. Cripps, A. J. Fields, and M. J. Reeve, "The Design and Implementation of ALICE: A Parallel Graph Reduction Machine," *BYTE*, Vol. 10, No. 8, Aug. 1985.

[ClD83] M. Clarke, C. Dyer, "Systolic Array for a Dynamic Programming Application," *Proc. 12'th Workshop on Applied Imagery Pattern Recognition*, 1983.

[CiH84a] A. Ciepielewski and S. Haridi, "Control of Activities in OR-Parallel Token Machine," *IEEE Int'l Symp. on Logic Programming*, 1984.

[CiH84b] A. Ciepielewski and S. Haridi, "Execution of Bagof on the OR-parallel Token Machine," *Proc. Int'l Conf. on Fifth Generation Computer Systems*, pp. 551-560, 1984.

[ClG83] K. Clark and S. Gregory, "PARLOG: A Parallel Logic Programming Language," Research Report DOC 83/5, Imperial College, March 1983.

[CoK83] J. Conery and D. Kibler, "AND Parallelism in Logic Programming," *IJCAI*, pp. 539-543, 1983.

[CMM67] R. W. Conway, W. L. Maxwell, and L. W. Miller, *Theory of Scheduling*, Addison-Wesley, 1967.

[CRL73] J. Casti, M. Richardson, and R. Larson, "Dynamic Programming and Parallel Computers," *J. of Optimization Theory and Appl.*, Vol. 12, No. 4, pp. 423-438, 1973.

[DaR81] J. Darlington and M. Reeve, "ALICE—A Multi-Processor Reduction Machine for The Parallel Evaluation of Applicative Languages," *Proc. of Conf. on Functional Programming Languages and Computer Architecture*, ACM. pp 65-74, 1981.

[DeG84] D. DeGroot, "Restricted AND-Parallelism," *Proc. Int'l Conf. on Fifth Generation Computer Systems*, pp. 471-478, 1984.

[Den80] J. B. Dennis, "Data Flow Supercomputers," *IEEE Computer*, pp. 48-56, Nov. 1980.

[Des78] B. C. Desai, "The BPU: A Staged Parallel Processing System to solve the Zero-One Problem," *Proc. ICS'78, pp. 802-817, Dec. 1978.*

[EHL80] L. D. Erman, F. Hayes-Roth, V. R. Lesser, and D. R. Reddy, "The Hearsay-II Speech-Understanding System: Integrating Knowledge to Resolve Uncertainty," *ACM Computing Surveys*, Vol. 12, No. 2, pp. 213-253, June 1980.

[ElH80] O. I. El-Dessouki and W. H. Huen, "Distributed Enumeration on Network Computers," *IEEE Trans. on Computers*, Vol. C-29, pp. 818-825, Sept. 1980.

[FiF82] R. Finkel and J. Fishburn, "Parallelism in Alpha-Beta Search," *Artificial Intelligence*, pp. 89-106, 1982.

[Fin82] R. Finkel, "Parallelism in Alpha-Beta Search," *Artificial Intelligence*, pp. 89-106, 1982.

[Ful78] S.H. Fuller, et al, "Multi-Microprocessors: A Overview and Working Example," *Proc. of IEEE*, Vol. 66, No. 2, pp. 216-228, Feb. 1978.

[FuN82] K. Furukawa, K. Nitta, and Y. Matsumoto, "Prolog Interpreter Based on Concurrent Programming," *Proc. of First Int'l Conf. Logic Programming*, 1982.

[GaJ76] M. R. Garey, and D. S. Johnson, *Computers and Intractability, A Guide to the Theory of NP-completeness*, W. H. Freeman and Company, 1979.

[Gar73] M. Garey, "Optimal Task Sequencing with Precedence Constraints," *Discrete Mathematics*, Vol. 4, pp. 37-56, 1973.

[GMM81] S.Gensi, U. Montanari, and A. Martelli, "Dynamic Programming as Graph Searching: An Algebraic Approach," *JACM*, Vol. 28, No. 4, pp. 737-751, 1981.

[Gra66] R. L. Graham, "Bounds for Certain Multiprocessing Anomalies", *Bell Syst. Tech. J.* Vol. 45 pp. 1653-1581. 1966

[GrJ76] M. R. Garey and D. S. Johnson. "Performance Guarantees for Heuristic Algorithms: An Annotated Bibliography", *Proc. Symp. on New Direction and Recent Results in Algorithms and Complexity*, Academic Press, 1976.

[GKT79] L. Guibas, H. Kung, and C. Thompson, "Direct VLSI Implementation of Combinatorial Algorithms," *Proc. Conf. VLSI: Architecture, Design, Fabrication*, Caltech, pp. 509-525, 1979.

[GTM84] A. Goto, H. Tanaka, and T. Moto-oka, "Highly Parallel Inference Engine PIE—Goal Rewriting Model and Machine Architecture," *New Generation Computing*, Vol. 2, OHMSHA Ltd. and Springer-Verlag, pp. 37-58, 1984.

[HaA84] R. Hasegawa, M. Amamiya, "Parallel Execution of Logic Programs Based on Dataflow Concept," *Proc. of Int'l Conf. on Fifth Generation Computer Systems*, pp. 507-516, 1984.

[HaB84] K. Hwang and F. A. Briggs, *Computer Architecture and Parallel Processing*, McGraw-Hill, 1984.

[HaS79] J. Harris and D. Smith, "Simulation Experiments of a Tree Organized Multicomputer," *Proc. 6th Annual Symp. Comput. Arch., IEEE/ACM*, pp 83-89, 1979.

[HoS78] E. Horowitz and S. Sahni, *Fundamentals of Computer Algorithms*, Computer Science Press, 1978.

[HoZ83] E. Horowitz and A. Zorat, "Divide-and-Conquer for Parallel Processing," *IEEE Trans. on Computers*, Vol. C-32, No. 6, pp. 582-585, June 1983.

[HWL83] F. Hayes-Roth, D. A. Waterman, and D. B. Lenat, *Building Expert Systems*, Addison-Wesley, 1983.

[Iba73] T. Ibaraki, "Solvable Classes of Discrete Dynamic Programming," *J. of Mathematical Analysis and Applications*, Vol. 43, pp. 642-693, 1973.

[Iba76a] T. Ibaraki, "Computational Efficiency of Approximate Branch-and-Bound Algorithms," *Math. of Operations Research*, Vol. 1, No. 3, pp. 287-298, 1976.

[Iba76b] T. Ibaraki, "Theoretical Comparisons of Search Strategies in Branch-and-Bound Algorithms," *Int'l. J. of Comp. and Info. Sci.*,

[Iba77a] T. Ibaraki, "The Power of Dominance Relations in Branch-and-Bound Algorithms," *JACM*, Vol. 24, No. 2, pp. 264-279, 1977.

[Iba77b] Ibaraki, T., "On the Computational Efficiency of Branch-and-Bound Algorithms," *Journal of the Operations Research Society of Japan*, Vol. 20, No. 1, pp. 16-35, March 1977.

[IFY79] M. Imai, T. Fukumara, and Y. Yoshida, "A Parallelized Branch-and-Bound Algorithm: Implementation and Efficiency," *Syst. Comput. Controls*, Vol. 10, No. 3, pp. 62-70, 1979.

[KaH67] R. Karp and M. Held, "Finite State Processes and Dynamic Programming," *SIAM J. on Appl. Math.*, Vol. 15, pp. 693-718, 1967.

[Kar76] R. M. Karp, *Algorithms and Complexity: New Direction and Resent Result*, J. F. Traub (ed.), Academic Press, pp 1-19, 1976.

[KLT84] R. M. Keller, F. C. H. Lin, and J. Tanaka, "*Rediflow Multiprocessing*," *Proc. COMPCON Spring*, IEEE, pp. 410-417, 1984.

[KKM83] S. Kasif, M. Kohli, and J. Minker, "PRISM: A parallel Inference System for Problem Solving," *Proc. IJCAI*, pp. 544-546, 1983.

[KnM75] D. E. Knuth and R. W. Moore, "An Analysis of Alpha-Beta Pruning," *Artificial Intelligence*, Vol. 6, pp. 293-326, 1975.

[Knu73] D. E. Knuth, *The Art of Computer Programming*, Vol. 3. Addison-Wesley, 1973

[Kun79] H. T. Kung, "Let's Design Algorithms for VLSI Systems," *Proc. Caltech Conf. on VLSI*, pp. 65-90, Jan. 1979.

[KoS74] W. Kohler and K. Steiglitz, "Characterization and Theoretical Comparison of Branch-and-Bound Algorithms for Permutation Problems," *JACM*, Vol. 21, No. 1, pp. 140-156, 1974.

[Kow79] R. Kowalski, *Logic for Problem Solving*, North Holland, 1979.

[Kuc77] D. J. Kuck, "A Survey of Parallel Machine Organization and Programming," *ACM Computing Surveys*, Vol. 9, No. 1, pp. 29-59, March 1977.

[KuK83] 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.

[KuK84] V. Kumar and L. Kanal, "Parallel Branch-and-Bound Formulations for AND/OR Tree Search," *IEEE Trans. on Pattern Analysis and Machine Intelligence*, Vol. PAMI-6, pp. 768-778, Nov. 1984.

[Kum84] V. Kumar, "A General Bottom-up Procedure for Searching AND/OR Graphs," *Proc. of National Conf. Artificial Intelligence*, pp. 182-187, 1984.

[LaS84] T. H. Lai and S. Sahni, "Anomalies of Parallel Branch-and-Bound Algorithms," *Comm. ACM*, Vol. 27, No. 6, June 1984.

[LaS85a] T. H. Lai and A. Sprague, "Performance of Parallel Branch-and-Bound Algorithms," *Proc. Int'l Conf. on Parallel Processing*, IEEE, pp. 194-201, 1985.

[LaS85b] T. H. Lai and A. Sprague, "Performance of Parallel Branch-and-Bound Algorithms," *Vol C-34, No. 10, pp 962-964, Oct. 1985.*

[LaW66] E. L. Lawler and D. W. Wood, "Branch-and-Bound Methods: A Survey," *Operations Research*, Vol. 14, pp. 699-719, 1966.

[LiW84a] G.-J. Li and B. W. Wah, "Computational Efficiency of Parallel Approximate Branch-and-Bound Algorithms," *Proc. Int'l Conf. on Parallel Processing*, IEEE, pp. 473-480, 1984.

[LiW84b] G.-J. Li and B. W. Wah, "How to Cope with Anomalies in Parallel Approximate Branch-and-Bound Algorithms," *Proc. of National Conf. on Artificial Intelligence (AAAI-84)*, pp. 212-215, 1984.

[LiW85a] G.-J. Li and B. W. Wah, "The Design of Optimal Systolic Algorithms," *IEEE Trans. on Computers*, Vol. C-34, No. 1, pp. 66-77, Jan. 1985.

[LiW85b] G.-J. Li and B. W. Wah, "Systolic Processing for Dynamic Programming problems," *Proc. Int'l Conf. on Parallel Processing*, IEEE, pp. 434-441, 1985.

[LiW85c] G.-J. Li and B. W. Wah, "MANIP-2: A Multicomputer Architecture for Evaluating Logic Programs," *Proc. of Int'l Conf. on Parallel Processing*, IEEE, pp. 123-130, 1985.

[LiW85d] G.-J Li and B. W. Wah, "Parallel Processing of Serial Dynamic Programming Problems," *Proc. of Int'l conf. on Computer Software and Applications*, IEEE, 1985.

[LiW86] G.-J Li and B. W. Wah, "Coping With Anomalies in Parallel Branch-and-Bound Algorithms" to appear in IEEE Trans. on Computers, 1986.

[LiP84] G. Lindstrom and P. Panangaden, "Stream-Based Execution of Logic Programs," *Proc. 1984 Int'l Symp. on Logic Programming*, pp. 168-176, Feb. 1984.

[MaC82] T. A. Marsland and M. Campbell, "Parallel Search of Strongly Ordered Game Trees," *ACM Computing Surveys*, Vol. 14, No. 4, pp. 533-551, Dec. 1982.

[Mag85] G. Mago, "Making Parallel Computation Simple: The FFP Machine," *Proc. IEEE COMPCON Spring*, pp. 424-428, 1985.

[MaM73] A. Martelli and U. Montanari, "Additive AND/OR Graphs," *IJCAI*, 1973.

[MaM82] A. Martelli and U. Montanari, "An Efficient Unification Algo *ACM Trans. on Prog. Lang. and Systems*, Vol. 4, pp. 258-282, 198

[Mar79] A. J. Martin, "A Distributed Architecture for Parallel Recursive C tations," Internal report, AJM 18, Eindhoven, Philips Researc 1979.

[Mit70] L. Mitten, "Branch-and-Bound Methods: General Formulation anc perties," *Operations Research*, Vol. 18, pp. 24-34, 1970.

[Moh83] J. Mohan, "Experience with Two Parallel Programs Solving Traveling-Salesman Problem," *Proc. of the 1983 Int'l Conf. on Pa Processing*, IEEE, pp. 191-193, 1983.

[MoM76] T. Morin and R. Marsten, "Branch-and-Bound Strategies for Dyna Programming," *Operations Research*, Vol. 24, No. 4, pp. 611-627, 197

[Mot84] T. Moto-Oka, et al., "The Architecture of a Parallel Inference Engin PIE," *Proc. Int'l Conf. on Fifth Generation Computer Systems*, ICOT, 479-488, 1984.

[NKK84] D. Nau, V. Kumar and L. Kanal, "General Branch and Bound, and Relation to A* and AO*, *Artificial Intelligence* vol. 15, pp 29-58, 1984.

[New79] M. M. Newborn, "Recent Progress in Computer Chess," *Advances i Computers*, Vol. 18, M. C. Yovits (ed.), Academic Press, pp. 59-117 1979.

[Ney82] H. Ney, "Dynamic Programming as a Technique for Pattern Recognition," *Proc. 6th Int'l Conf. Pattern Recognition*, pp. 1119-1125, Oct. 1982.

[Nil80] N. J. Nilsson, *Principles of Artificial Intelligence*, Tioga, 1980.

[PaW78] M.Paterson and M. Wegman, "Linear Unification", *J. Computer and System Science*" Academic Press, Vol. 16, pp. 158-167, 1978.

[PaW79] E. Page and L. Wilson, *Introduction to Computational Combinatorics*, Cambridge University Press, 1979.

[Pea82] J. Pearl, "The solution for the Branching Factor of the Alpha-Beta Pruning Algorithm and Its Optimality," *CACM*, pp. 559-564, 1982.

[Pea84] J. Pearl, *Heuristics*, Addison-Wesley, 1984.

[Pet81] F. Peters, "Tree Machine and Divide-and-Conquer Algorithms," *CONPAR 81*, Lecture Notes CS111, pp 25-35, 1981.

[Ros82] A. Rosenthal, "Dynamic Programming is Optimal for Nonserial optimization problems," *SIAM J. Compu*. Vol. 11, No. 1, 1982.

[Sah76] S. Sahni, "Algorithms for scheduling Independent Tasks," *J. ACM*, Vol. 23, No. 1, pp. 116-127, Jan. 1976.

[Sah77] S. Sahni, "General Techniques for Combinational Approximation," *Operations Research*, Vol. 25, No. 6, pp. 920-936, 1977.

[Sav78] C. Savage, *Parallel Algorithms for Graph Theoretic Problems*, Ph.D. Dissertation, University Illinois, Urbana, IL, August, 1978.

[Sav81] J. E. Savage, "Area-Time Tradeoffs for Matrix Multiplication and Related Problems in VLSI Models," *J. Computer and System Sciences*, Vol. 20, No. 3, pp. 230-242, 1981.

[Sha83] E. Shapiro, *A Subset of Concurrent Prolog and its Interpreter*, ICOT Tech. Report TR-003, .1983, Tokyo, Japan, Feb. 1983.

[Sha74] A. C. Shaw, *The Logical Design of Operating System*, Prentice-Hall, 1974.

[Sek81] Y. Sekiguchi, "A Unifying Framework of Combinatorial Optimization Algorithms; Tree Programming and its Validity," *J. Oper. Res. of Japan*, Vol. 24, No. 1, March 1981.

[SiK75] H. A. Simon and J. Kadane, "Optimal Problem-solving Search: All-or-None Solutions," *Artificial Intelligence*, Vol. 6, pp. 235-246, 1975.

[Smi84] D. R. Smith, "Random Trees and the Analysis of Branch-and-Bound Procedures," *Journal of ACM*, Vol. 31, No. 1, pp. 163-188, Jan. 1984.

[Smi85] D. R. Smith, "Top-Down Synthesis of Divide-and-Conquer Algorithms" *Artificial Intelligence*, Vol. 27, No. 1, pp. 43-96, 1985.

[StM84] S. J Stolfo and D. P. Miranker, "DADO: A Parallel Processor for Expert Systems," *Proc. Int'l Conf. on Parallel Processing*, IEEE, pp. 74-82, 1984.

[Sto79] G. Stockman, "A Minimax Algorithm Better than Alpha-Beta?" *Artificial Intelligence*, Vol. 12, pp 179-196, 1979.

[TaL83] C. Tang and R. C. T. Lee, "Optimal Speedup of Parallel Algorithm Based on the Divide-and-Conquer Strategy," personal communication.

[Tho81] C. D. Thompson, *A Complexity Theory for VLSI*, Ph.D. Thesis, Carnegie-Mellon University, Pittsburgh, Pa. 1981.

[VaR84] P. Varman and V. Ramakrishnan, "Dynamic Programming and Transitive Closure on Linear Pipelines," *Proc. 1984 Conf. on Parallel Processing*, IEEE, pp. 359-364, 1984.

[WaC84] B. W. Wah and K. L. Chen, "A Partitioning Approach to the Design of Selection Networks," *IEEE Trans. on Computers*, Vol. C-33, No. 3, pp. 261-268, March 1984.

[WaL85] B. W. Wah and G.-J. Li, *Computers for Artificial Intelligence Applications*, Tutorial Notes No. 10, IEEE, Nov. 1985.

[WaL86] B. W. Wah and G.-J. Li, *Tutorial on Computers for Artificial Intelligence Applications*, IEEE, 1986.

[WaM84] B. W. Wah and E. Y. W. Ma, "MANIP--A Multicomputer Architecture for Solving Combinatorial Extremum Search Problems," *IEEE Trans. on Comput.*, Vol. C-33, No. 5, pp. 377-390, May 1984.

[WaY85] B. W. Wah and C. F. Yu, "Probabilistic Modeling of Branch-and-Bound Algorithms under a Best-First Search," *IEEE Trans. on Software

Engineering, Vol. SE-11, No 10, Oct. 1985.

[Wei82] B. W. Weide, "Modeling Unusual Behavior of Parallel Algorithms," *IEEE Trans. on Comp.* Vol. C-31, No. 11, pp. 1126-1130, Nov. 1982.

[Whi69] D. White, *Dynamic Programming*, Oliver & Boyd, Edinburgh, 1969.

[WLY84] B. W. Wah, G.-J. Li, and C. F. Yu, "The Status of MANIP--A Multicomputer Architecture for solving Combinatorial Extremum-Search Problems," *Proc. 11'th Annual Int'l Symp. Comput. Architecture*, pp. 56-63, 1984.

[WLY85] B. W. Wah, G.-J. Li, and C. F. Yu, "Multiprocessing of Combinatorial Search Problems," *IEEE Computer*, pp. 93-108, June 1985.

[YaN84] H. Yasuhara and K. Nitadori, "ORBIT: A Parallel Computing Model of Prolog," *New Generation Computing*, ICOT Vol. 2, pp. 277-288, 1984.

[YuW83] C. F. Yu and B. W. Wah, "Virtual-Memory Support for Branch-and-Bound Algorithms," *Proc. COMPSAC*, IEEE, pp. 618-626, Nov. 1983.

[YuW84] C. F. Yu and B. W. Wah, "Efficient Branch-and-Bound Algorithms on a Two-Level Memory Hierarchy," *Proc. COMPSAC*, IEEE, pp. 504-514, Nov. 1984.

APPENDICES

Appendix I: Proof of Theorem 4.1

Proof of Theorem 4.1: The proof can be discussed with respect to Figure 3.3. Suppose that there is a node $P_{i_1} \in \Phi^k$ and $P_{i_1} \notin \Phi^l$, and that there is a node $P_{i_2} \in \Phi^l$ and $P_{i_2} \notin \Phi^k$. Suppose further that P_{i_1} has been terminated by a lower-bound test (due to P_{i_2}) or a dominance test (due to P_{i_2}) in the serial case, then $h(P_{i_2}) < h(P_{i_1})$ since P_{i_2} is selected before P_{i_1}. $P_{i_1} \in \Phi^k$ implies that nodes P_{i_4} and P_{i_5} exist such that $P_{i_4}\,L\,P_{i_5}$ or $P_{i_5}\,D\,P_{i_4}$, and P_{i_5} has not been generated when P_{i_1} is expanded in the serial case (otherwise P_{i_1} would have been terminated). Without loss of generality, let P_{i_3} be the proper ancestor of P_{i_4}. There are two possibilities.

First, $h(P_{i_3}) > h(P_{i_2})$. If $P_{i_5}\,D\,P_{i_1}$, then $h(P_{i_5}) < h(P_{i_1})$ according to the assumption that D is consistent with h, a contradiction! If $P_{i_4}\,L\,P_{i_5}$ and $P_{i_5}\,D\,P_{i_1}$, then $f(P_{i_4}) \leqslant g(P_{i_5})$. From condition (a), D is consistent with g, we have $g(P_{i_5}) \leqslant g(P_{i_1})$ which implies that $f(P_{i_4}) \leqslant g(P_{i_1})$. This means that P_{i_1} must be eliminated in the parallel case, and that it contradicts that $P_{i_1} \in \Phi^k$.

Second, $h(P_{i_3})<h(P_{i_2})$, and P_{i_3}, as well as its descendant P_{i_4}, have been terminated in the serial case and not in the parallel case. We can then apply the above argument again to P_{i_3}, and eventually obtain a sequence of nodes $P_{i_1}, P_{i_2}, \ldots, P_{i_m}$ as depicted in Figure 3.3. There are three possibilities:

(a) The first node P_{i_m} occurs in the serial case (Figure 3.3a). Since $h(P_{i_{k+2}})<h(P_{i_k})$ (otherwise P_{i_k} cannot be terminated by $P_{i_{k+1}}$ in the serial case) and $h(P_{i_{k+4}})<h(P_{i_{k+2}})$ (by the same argument as $h(P_{i_3})<h(P_{i_1})$), we have $h(P_{i_{k+4}})<h(P_{i_k})$. Repeating this, we get $h(P_{i_m})<h(P_{i_1})$. By Lemma 3.1, P_{i_m} must be expanded in the parallel case and terminates P_{i_1} which contradicts that $P_{i_1} \in \Phi^k$.

(b) P_{i_m} occurs in the parallel case (Figure 3.3b). As argued in the case of $h(P_{i_4}) > h(P_{i_3})$, this contradicts with the assumption that D is consistent with h and g.

(c) There is a cycle of cutoffs such that $P_{i_{m-1}} L P_{i_{m-2}}$ (or $P_{i_m} D P_{i_{m-2}}$) ... $P_{i_{k+1}} L P_{i_k}$ (or $P_{i_{k+2}} D P_{i_k}$), and $P_{i_{k-1}} L P_{i_m}$ (or $P_{i_k} D P_{i_m}$) (Figure 3.3c). This implies that $h(P_{i_k}) > h(P_{i_m})$ and that $P_{i_k} D P_{i_m}$, which contradict the assumptions that D is consistent with h or $f(P_{i_m})=f(P_{i_{m-1}})=f(P_{i_{k-1}})$. In the latter case, the heuristic values of all the nodes of the cycle are less than $h(P_{i_1})$ and a feasible solution has been obtained before P_{i_1} is selected, thus P_{i_1} must be terminated in the parallel case.

So far we have proved that at least one node of Φ^1 is expanded in each iteration of a parallel heuristic search. Furthermore, it is easy to show that once all the nodes of Φ^1 are expanded or terminated in the parallel heuristic search, no active node remains, that is, the OR-tree search procedure is completed. Since approximate lower-bound tests are not applied, the optimal solution cannot be terminated by lower-bound or dominance tests (according to Eq. (3.9) and Assumption (b) of this theorem). Hence, during a parallel heuristic search, once all the nodes in Φ^1 have been expanded or terminated, the optimal solution node must be generated. All remaining unexpanded nodes must belong to one of the following three classes: (a) infeasible solution nodes; (b) nodes whose lower bounds are greater than the optimal solution; (c) nodes that are dominated by nodes in Φ^1. Nodes in Class (b) will be eliminated by lower-bound tests. Those in Class (c) will be eliminated by dominance or lower-bound tests according to the transitivity of D and Assumption (a) of this theorem. All these nodes will be eventually drained from the subproblem lists. □

Appendix 2: Proof of Theorem 4.4

Proof: If $\Phi^1 \subset \Phi^k$ because at least $\left\lceil |\Phi^1|/k \right\rceil$ iterations are needed to expand the nodes in Φ^1. Hence the proof is based on the assumption that a node $P_{i_1} \in \Phi^1$ exists and $P_{i_1} \notin \Phi^k$. This means that P_{i_1} is terminated by either a lower-bound or a dominance test in the parallel case. That is, there is a feasible-solution node $P_{i_2} \in \Phi^k$ such that $f(P_{i_2}) \leqslant g(P_{i_1})$, or there is a dominating node $P_{i_3} \in \Phi^k$ such that $P_{i_3} D P_{i_1}$. P_{i_2} or P_{i_3} do not exist in the serial case when P_{i_1} is expanded. Referring to Figure A.1, two cases are possible:

First, $h(P_{i_3}) > h(P_{i_1})$. This means that P_{i_3} has not been generated when P_{i_1} is selected in the serial case. If $P_{i_3} D P_{i_1}$, then D is inconsistent with h. On the other hand, if $P_{i_2} L P_{i_1}$, then $g(P_{i_1}) \leqslant g(P_{i_2}) \leqslant f(P_{i_2}) \leqslant g(P_{i_1})$ which implies that h is not completely consistent with g.

Second, $h(P_{i_3}) < h(P_{i_1})$. In order for P_{i_1} to exist in the serial case, $P_{i_4}, P_{i_5} \in \Phi^1$ must exist so that $P_{i_4} L P_{i_2}$ or $P_{i_5} D P_{i_3}$, and $h(P_{i_4}) < h(P_{i_2})$ or $h(P_{i_5}) < h(P_{i_3})$. There are four combinations of cases: (a) $P_{i_2} L P_{i_1}$ and $P_{i_4} L P_{i_2}$; (b) $P_{i_3} D P_{i_1}$ and $P_{i_5} D P_{i_3}$; (c) $P_{i_3} D P_{i_1}$ and $P_{i_4} L P_{i_2}$; (d) $P_{i_2} L P_{i_1}$ and $P_{i_5} D P_{i_3}$. The first two cases contradict with the assumption that $P_{i_1} \in \Phi^1$ owing to the transitivity of lower-bound and dominance tests. In Case (c), $f(P_{i_4}) \leqslant g(P_{i_2})$. However, P_{i_1} is not terminated by the lower-bound test of P_{i_4}, i.e., $f(P_{i_4}) > g(P_{i_1})$. As a result, $g(P_{i_1}) < g(P_{i_2})$. Since $h(P_{i_2}) < h(P_{i_1})$, this implies that h is not completely consistent with g. In Case (d), $P_{i_2} L P_{i_1}$ implies that $f(P_{i_3}) \leqslant f(P_{i_2}) \leqslant g(P_{i_1})$. On the other hand, $P_{i_5} D P_{i_3}$ implies that $g(P_{i_5}) \leqslant f(P_{i_5}) \leqslant f(P_{i_3})$. Without loss of generality, let P_{i_6} be the best feasible solution in the subtree rooted at P_{i_5}, and hence $f(P_{i_6}) = f(P_{i_5})$. From these inequalities, we get $g(P_{i_6}) = f(P_{i_6}) = f(P_{i_5}) \leqslant f(P_{i_3}) \leqslant g(P_{i_1})$. Since $P_{i_6} \in \Phi^1$, and P_{i_1} is not terminated by P_{i_6}, there must exist an ancestor P_{i_7} of P_{i_6} such that either

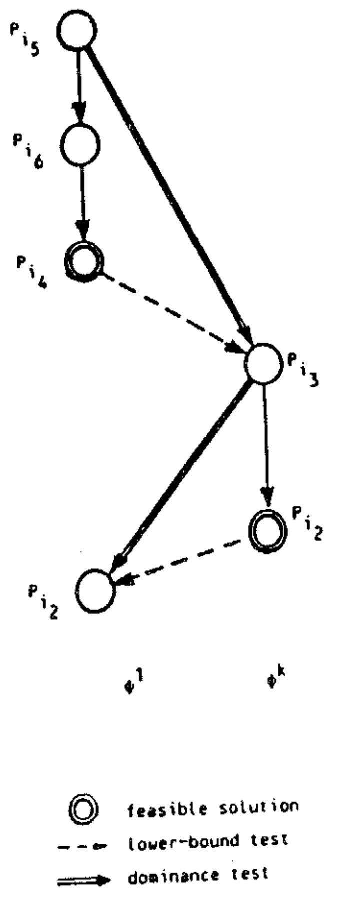


Figure A.1 Proof of Theorem 4.4

$h(P_{i_4}) > h(P_{i_3})$ or $h(P_{i_4}) < h(P_{i_3})$. If $h(P_{i_4}) > h(P_{i_3})$, then h is not completely consistent with g because $g(P_{i_4}) \leqslant g(P_{i_4}) \leqslant g(P_{i_3})$. On the other hand, if $h(P_{i_4}) < h(P_{i_3})$, then the above argument can be repeated such that P_{i_4} is terminated by a lower-bound or dominance test. Since the number of feasible solutions is finite, the same necessary conditions are also true. We conclude that $T_f(k_2,0) \leqslant T_f(k_1,0)$. □

VITA

VITA

Guo-Jie Li was born on May 29, 1943 in Hunan, the People's Republic of China. He graduated from the Department of Physics, Peking University in 1968, and received the M.S. degree in computer science and engineering from the University of Science and Technology of China and the Institute of Computing Technology, Chinese Academy of Science, in 1981. From 1968 to 1978, he joined two manufacturers as an engineer working on designing semiconductor devices and minicomputers.

Guo-Jie Li came to Purdue University as a graduate student in September, 1981. During his stay at Purdue, he was a research assistant.