Computational Efficiency of Parallel Combinatorial OR-Tree Searches

GUO-JIE LI AND BENJAMIN W. WAH, SENIOR MEMBER, IEEE

Abstract—**The performance of parallel combinatorial OR-tree searches (or OR-tree searches in short) is analytically evaluated in this paper. This performance depends on the complexity of the problem to be solved, the error allowance function, the dominance relation, and the search strategies. The exact performance may be difficult to predict due to the nondeterminism and anomalies of parallelism. We derive the performance bounds of parallel OR-tree searches with respect to the best-first, depth-first, and breadth-first strategies, and verify these bound by simulations. We show that a near-linear speedup can be achieved with respect to a large number of processors for parallel OR-tree searches. Using the performance bound developed, we derive sufficient conditions for assuring that parallelism will not degrade the performance, and necessary conditions for allowing parallelism to have a speedup greater than the ratio of number of processors. These bounds and conditions provide the theoretical foundation for determining the number of processors to assure a near-linear speedup.**

Index Terms—**Anomalies, approximation, combinatorial search, dominance test, heuristic search, lower-bound test, OR-tree, parallel processing, performance bounds.**

I. Introduction

In a wide class of combinatorial search problems in artificial intelligence, operations research, decision making, and various scientific and engineering fields, it is necessary to find one or more, optimal, suboptimal, or feasible solutions in a large problem space. A combinatorial search, in short a search, enumerates some or all elements of the problem space until the solutions are found. Combinatorial search problems can be classified into two types. The first type is decision problems that decide whether at least one solution exists and satisfies a given set of constraints [30]. Theorem-proving and expert systems belong to this class. The second type is combinatorial extremum-search or optimization problems which are characterized by an objective function to be minimized or maximized and a set of constraints to be satisfied. Many practical problems, such as finding the shortest path, planning, finding the shortest tour of a traveling salesman, job-shop scheduling, packing a knapsack, vertex cover, and integer programming, belong to this class.

Approximations and parallel processing are two major approaches to enhance the efficiency of combinatorial OR-tree searches. Owing to the exponential nature of many search problems, optimal solutions are usually infeasible to obtain. In practice, approximate solutions are acceptable alternatives. Experimental results on vertex-cover, 0-1 knapsack, and some integer-programming problems reveal that a linear reduction in accuracy may result in an exponential reduction of the average computational time [33]. On the other hand, 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 [31], or when the problem is heuristically solvable in polynomial time. Parallel processing is generally useful for improving the computational efficiency of solving a given problem, and not in substantially extending the solvable problem size of the problem [34].

A search problem can be represented as either a tree or a graph. The nonterminal nodes in a search tree (or graph) can be classified as AND-nodes and OR-nodes. An *AND-node* represents a problem (or subproblem) that is solved only if all its descendent nodes have been solved. An *OR-node* represents a problem (or subproblem) that is solved only if any of its immediate descendents is solved. Based on these two kinds of nodes, a combinatorial search can be classified into an OR-tree search, an AND-tree search, and an AND/OR-tree search [34].

An OR-tree is a state-space tree in which all nonterminal nodes are OR-nodes. It is important because it represents nondeterminism, which is a natural property of many decision problems. Moreover, methods for solving OR-tree-search problems provides a basis to solving many general AND/OR-tree (or AND/OR-graph) search problems. A lot of the heuristic search procedures, such as A*, B*, AO*, SSS*, and dynamic programming, can be formulated as a general branch-and-bound (B&B) procedure [9], [16], [26]. Likewise, evaluating a logic program can be represented as an OR-tree search [15].

An OR-tree can be represented as an OR-graph in which all common nodes in the tree are grouped into a single node. Although more compact in representation, searching OR-graphs is inefficient in a parallel processing system in which the communication overhead is dominant. Frequent broadcasts of information from a parent node to multiple descendent nodes that it connects have to be made

Manuscript received March 4, 1986; revised August 28, 1989. Recommended by C. P. Kruskal. This work was supported in part by the National Science Foundation under Grants MIP 85-19649 and MIP 88-10584 and by the National Aeronautics and Space Administration under Grant NCC 2-481.

G. J. Li is with the Institute of Computing Technology, Academia Sinica, Beijing, People's Republic of China.

B. W. Wah is with the Department of Electrical and Computer Engineering and the Coordinated Science Laboratory, University of Illinois at Urbana-Champaign, 1101 W. Springfield Avenue, Urbana, IL 61801.

IEEE Log Number 8931756.

when the parent and its descendants are searched in different processors. In the corresponding OR-tree search, these broadcasts are eliminated because state information is only passed between the parent and one of the descendents that it activates. This savings in communication overhead is achieved at the expense of performing redundant search in the tree. A second reason for preferring OR-trees over OR-graphs is that, for some problems, the redundant nodes in the search tree are not apparent at program design time, and have to be found, usually at great overhead, at run time. For these reasons, we focus only on the efficiency of OR-tree searches in this paper.

OR-parallelism and its efficiency has been studied extensively in the literature [2], [17], [23]. One of the important issues addressed is linear scaling, that is, when a large number of processors are used, how can an OR-tree search be scheduled without any reprogramming such that the speed of computation increases in direct proportion to the number of computing elements. Rao and Kumar analyzed linear scaling in terms of an iso-efficiency function [17]. It has been reported that linear scaling does not usually hold for parallel game-tree searches [1], [4] and parallel processing of forward-chaining rule-based expert systems [5]. Simulation results have also revealed that using more processors in parallel OR-tree searches might degrade the performance or might result in superlinear speedup when the communication overhead is ignored [18], [20], [28]. A number of conditions and methods to cope with anomalous behavior of branch-and-bound algorithms have been developed [18], [19], [21]-[23].

This paper presents new results on the efficiency of evaluating parallel OR-tree searches in the presence of anomalies. Results are derived with respect to finding one of the optimal (or suboptimal) solutions in the problem space, although they can be generalized to finding all optimal solutions. Our results are useful for designers to understand the existence of anomalies and modify existing algorithms to cope with these anomalies. They can also be used to determinate a good search strategy and predict the maximum number of processors under which a near-linear speedup can be achieved. Our results are derived with respect to the conventional best-first, depth-first, and breadth-first search strategies. Performance is measured in terms of the number of nodes expanded for a given number of processors. Modified search strategies, such as the IDA* [13], RTA* [14], and modified A* search using a virtual memory system [35], consider tradeoffs between computational resources and evaluation efficiency. The derivation of performance bounds for these modified strategies is much harder as performance has to be evaluated with respect to the actual execution time, rather than the number of nodes expanded.

II. Parallel Combinatorial OR-Tree Searches

A combinatorial OR-tree search procedure 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 eliminate subproblems generated that are not better than the ones already known. Appropriately ordering the search and restricting the region searched are key ideas behind any OR-tree search algorithm.

In contrast to conventional B&B algorithms, combinatorial OR-tree search algorithms allow approximate lower-bound and dominance tests as general elimination rules, and a generalized heuristic function as the selection rule. These rules are briefly explained in this section [6].

Let $\boldsymbol{P}$, $\boldsymbol{E}$, and $\boldsymbol{T}$ be, respectively, sets of subproblems, edges, and feasible solutions in the OR-tree. Let P_i be the ith subproblem, P_{i_j} be the jth subproblem decomposable from P_i, and $f(P_i)$ be the value of the best solution obtained by evaluating all subproblems decomposable from P_i. For minimization problem, the lower-bound function satisfies the following properties.

$$\text{a) } g(P_i) \le f(P_i) \qquad \text{for } P_i \in \boldsymbol{P}. \tag{2.1}$$

$$\text{b) } g(P_i) = f(P_i) \qquad \text{for } P_i \in \boldsymbol{T}. \tag{2.2}$$

$$\text{c) } g(P_i) \le g(P_{i_j}) \qquad \text{for } (P_i, P_{i_j}) \in \boldsymbol{E}. \tag{2.3}$$

If a subproblem is a feasible solution with the best objective-function value so far, then its solution value becomes the *incumbent* z. The incumbent represents the best solution obtained so far in the process. During the search of an OR-tree, an active node P_i is terminated if

$$g(P_i) \ge \frac{z}{1+\epsilon} \qquad \epsilon \ge 0, z \ge 0 \tag{2.4}$$

where ϵ is an *allowance function* specifying the allowable deviation of a suboptimal value from the exact optimal value, and an *active node* is a subproblem that has been generated but not expanded. (In searching for an optimal solution, the corresponding elimination condition is $g(P_i) \ge z$.) The final incumbent value z_F obtained by the approximate lower-bound test deviates from z_O, the optimal solution value, by

$$\frac{z_F}{1+\epsilon} \le z_O \le z_F \qquad \epsilon \ge 0, z_O \ge 0. \tag{2.5}$$

Approximations significantly reduce the amount of intermediate storage and the time needed to arrive at a suboptimal solution. In the following sections, $\boldsymbol{L}$ denotes the lower-bound cutoff test, that is, $P_j \boldsymbol{L} P_i$ means that P_j is a feasible solution and that $f(P_j)/(1+\epsilon) \le g(P_i)$, $\epsilon \ge 0$.

Dominance tests are powerful elimination rules that are systematically applied in dynamic programming algorithms to reduce their complexity of enumeration [25]. In contrast to lower-bound tests which compare the incumbent with the lower bounds of all active nodes, dominance tests use a set of parameters to compare pairs of nodes in the search tree. These nodes can be either nonterminal or terminal nodes, and either active or expanded nodes. In this sense, lower-bound tests can be regarded as a special case of dominance tests. A lot of the well-known elimination rules, such as α-β pruning in game-tree searches, are actually dominance tests. Dominance tests, in the best case, can reduce the problem complexity from exponential to polynomial. 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 [8].

A dominance relation is a binary relation such that $P_i \boldsymbol{D} P_j$ implies that P_i dominates P_j [7], [12]. Hence, for minimization problems, the subtree rooted at P_i contains a solution node whose value is no more than the minimum solution value of the subtree rooted at P_j. Thus if P_i and P_j are generated and $P_i \boldsymbol{D} P_j$, then P_j can be terminated. As an example, in the knapsack problem, it is necessary to pack objects into a knapsack of fixed capacity such that the total profit is maximized. Subproblem P_i dominates Subproblem P_j if the two subproblems involve the same set of objects, and the total profit of P_i is larger than or equal to that of P_j, while the total weight of P_i is smaller than or equal to that of P_j. Some results on the parallel implementation of dominance tests have been reported [20], [34].

Another method to reduce the search space is to order the subproblems expanded. If the list of active subproblems is maintained in a first-in/first-out order, then the algorithm is called a *breadth-first search*. If the list is maintained in a last-in/first-out order, then the algorithm is called a *depth-first search*. Lastly, if the list is maintained in increasing order of lower bounds, then the search algorithm is called a *best-first search*. Ibaraki mapped these searches into a general form called *heuristic searches*[1] [6]. A heuristic function defines 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 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.

A generalized heuristic function can be used to unify depth-first, breadth-first, and best-first searches. A heuristic function is said to be *monotone* when

$$h(P_i) < h(P_j) \qquad \text{if } P_j \text{ is a descendant of } P_i. \tag{2.6}$$

Ibaraki also proved that, for any heuristic function, there exists an equivalent monotone heuristic function [6]. A heuristic function is said to be *unambiguous* when

$$h(P_i) \neq h(P_j) \qquad \text{if } P_i \neq P_j \; P_i,\, P_j \in \boldsymbol{P}. \tag{2.7}$$

Notice that the heuristic functions defined conventionally for best-first, depth-first, and breadth-first searches are ambiguous because more than one node in the active list can have the same heuristic value. In subsequent sections, the properties on monotonicity and unambiguity are discussed with respect to predicating the performance and coping with anomalies of parallel OR-tree searches.

To resolve the ambiguity on the selection of subproblems, distinct heuristic values must be defined for the nodes to allow ties to be broken. A path number can be used to define an unambiguous heuristic function [20], [23]. The *path number* of a node in a tree is a sequence of $(h + 1)$ integers representing the path from the root to this node, where h is the maximum number of levels of the tree. The path number $E = e_0 e_1 \cdots e_h$ is defined recursively as follows. The root P_0 exists at Level 0 with a path number $000 \cdots 0$. A node P_{ij} on Level L, which is the jth child (counting from the left and starting from zero) of P_i with path number $E_i = e_0 e_1 \cdots e_{L-1} 000 \cdots$, has path number $E_{ij} = e_0 e_1 \cdots e_{L-1} j 00 \cdots$. According to this definition, nodes can have equal path numbers only if they have the ancestor–descendant relationship. Since these nodes never coexist simultaneously in the active list, the subproblems in the active list always have distinct path numbers. For example, the path numbers of all nodes in the tree in Fig. 1 are shown next to the nodes. Relations "$>$" and "$<$" on path numbers are defined in the usual sense as those of integers.

In best-first and breadth-first searches, the path number can be included in the general heuristic function as a secondary key to resolve ties in the primary key, which is still the lower-bound value or the level number. In depth-first searches, the path number serves as a primary key. That is,

$$h(P_i) = \begin{cases} (\text{level number, path number}) \\ \qquad \text{breadth-first search} \\ (\text{path number, level number}) \\ \qquad \text{depth-first search} \\ (\text{lower bound, level number, path number}) \\ \text{or } (\text{lower bound, path number, level number}) \\ \qquad \text{best-first search} \end{cases} \tag{2.8}$$

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. In general, unambiguous heuristic functions are not restricted to the use of path numbers. Any tie-breaking rule can be adopted as long as (2.7) is satisfied.

The definition of the path number dictates that a partial order exists among successors of any given node in the search tree, and such an order is used consistently by both

[1]The definition of general heuristic searches used in this paper is taken from Ibaraki's definition [6]. Depth-first, breadth-first, and best-first searches are considered as special cases of heuristic searches. This is different from the conventional definition of heuristic searches in artificial intelligence.

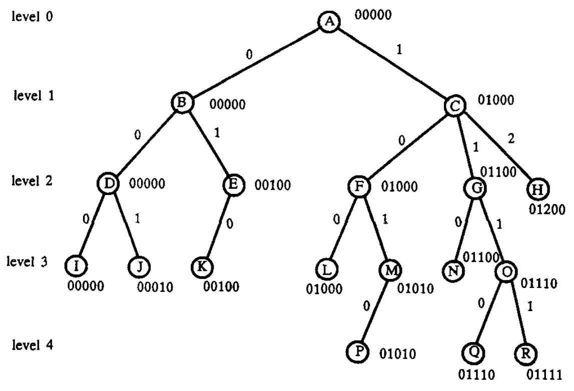

Fig. 1. The path numbers of a tree.

the sequential and parallel algorithms. Such a partial order implies that the same algorithm for decomposing a given node into successors is consistently used in both the sequential and parallel search. An attribute for defining this partial order may be needed if successors of a given node are generated in a random order. The lower-bound values can serve to define this partial order.

OR-tree searches have inherent parallelism. Each of the four rules of serial OR-tree search algorithms can be implemented by parallel processing. When k processors are used, if the number of active subproblems is greater than or equal to k in an iteration, then k subproblems with the k smallest heuristic values can be selected for decomposition; otherwise, all active subproblems can be selected. Parallel computer architecture for OR-tree searches have been studied elsewhere [20], [32].

III. Bounds on Computational Efficiency of Parallel OR-Tree Searches

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

A. Model of Efficiency Analysis

In analyzing the performance bounds, a synchronous model is assumed, that is, all processors must finish the current iteration before proceeding to the next iteration. The performance is difficult to evaluate if the parallel search algorithm were evaluated asynchronously. The performance results for synchronous models form a lower bound to that of asynchronous models.

The parallel computational model used here consists of a set of processors connected to a shared memory. In each iteration, 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 [7] in order to reduce the memory space required.

The shared memory in the above 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. We have found that the efficiency of parallel OR-tree search algorithms in the transformed model is very close to that of the original model, and the performance is not affected by whether the active subproblems are kept in a single list or multiple lists [23], [32]. 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 since this overhead is negligible as compared to that of branching operations.

The results proved in subsequent sections show the performance bounds of parallel best-first, depth-first, and breadth-first searches, respectively. The proofs of these theorems require the following definitions on essential nodes and basic nodes. A node expanded in a *serial* OR-tree search under a given heuristic function is called an *essential node* for that heuristic function; otherwise, it is called a nonessential node. The speedup of a parallel OR-tree search under a given heuristic function depends on the number of essential nodes selected in each iteration. An iteration is said to be *perfect* if the number of essential nodes selected is equal to the number of processors; otherwise, it is said to be *imperfect*. We denote $T_b(k, \epsilon)$, $T_d(k, \epsilon)$, and $T_r(k, \epsilon)$ as the number of iterations required to find a single optimal (or suboptimal) solution using k, $k \geq 1$, processors and an allowance function ϵ in a best-first, depth-first, and breadth-first search, respectively. The subscripts b, d, or r are omitted when the context refers to more than one search strategy.

A *basic node* is the node with the smallest heuristic value in each iteration. It is easy to show that if P_i is a basic node and P_j is any node with a smaller heuristic value, then P_j must be either expanded or terminated when P_i is expanded [20].

Let Φ^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. The following theorem states that any heuristic search with an unambiguous heuristic function can be guaranteed to expand at least one node in Φ^1 in each iteration of the parallel search, when an exact optimal solution is sought and dominance tests are inactive.

Theorem 3.1: Let $\epsilon = 0$ and $\boldsymbol{D} = \boldsymbol{I}$, i.e., an exact optimal solution is sought and dominance tests are inactive. For any parallel heuristic search with a heuristic satisfying (2.6) and (2.7), all basic nodes are essential nodes, i.e., all basic nodes belong to Φ^1.

Proof: The proof is omitted due to space limitation [22], [23]. □

B. Parallel Update of Incumbent

When the search is done in parallel, the way in which the incumbent is updated plays an important role in determining the lower bound of $T(k, \epsilon)$ and in coping with anomalies in approximate search. When more than one feasible solution is obtained in an iteration, updating the incumbent with the minimum feasible solution does not always result in a smaller search overhead and better solution. This phenomenon is demonstrated in an OR-tree search in Fig. 2(a) with $\epsilon = 0.1$. The suboptimal solution found in a serial depth-first search is F_7 (with value 70), and subtree T is eliminated. When the same tree is searched by four processors using a parallel depth-first search, five feasible solutions, $F_1, \cdots, F_5$, are generated in the fourth iteration. If the minimum among the five feasible solutions is chosen as the incumbent, F_5 (with value 75) will become the incumbent, which eliminates P_4 and P_5 by approximate lower-bound tests (2.4). As a result, Subtree T, which may be very large, has to be expanded. To avoid this problem, the following algorithm updates the incumbent in a consistent fashion when multiple feasible solutions are obtained in an iteration.

Serial/Parallel Incumbent-Update Algorithm:

a) When multiple feasible solutions are obtained in an iteration, they are compared to the incumbent. A feasible solution is eliminated if either its value is larger than or equal to the current incumbent, or the lower bound of its parent is not less than $[\text{incumbent}/(1 + \epsilon)]$. If all feasible solutions are eliminated, then exit.

b) If there is one or more feasible solution whose value is less than the current incumbent and the lower bound of its parent is less than $[\text{incumbent}/(1 + \epsilon)]$, then the feasible solution with the smallest heuristic value is chosen as the new incumbent and eliminated from further consideration. If there are no remaining feasible solutions, then the algorithm exists; otherwise, the entire Incumbent-Update algorithm is repeated.

Since the number of incumbents in an OR-tree search is usually small, it rarely happens that two or more incumbent-updates are needed in the same iteration.

To uniquely define the sequence of feasible solutions generated under a given heuristic function and a fixed number of processors, a feasible solution F_i is said to *appear before* another feasible solution F_j in the sequence of feasible solutions if either F_i and F_j are obtained in the t_ith and t_jth, $t_i < t_j$, iterations, or they are obtained in the same iteration and $h(F_i) < h(F_j)$. The following theorem proves the correctness of the Incumbent-Update algorithm.

Theorem 3.2: In an OR-tree search with $D = I$, the sequence of distinct incumbents obtained in the serial and parallel searches are identical regardless of E and the number of processors used if the following conditions are satisfied: a) a feasible solution F_i appearing before another feasible solution F_j implies that $h(F_i) < h(F_j)$ in both the serial and parallel cases, and b) the Serial/Parallel Incumbent-Update algorithm is used.

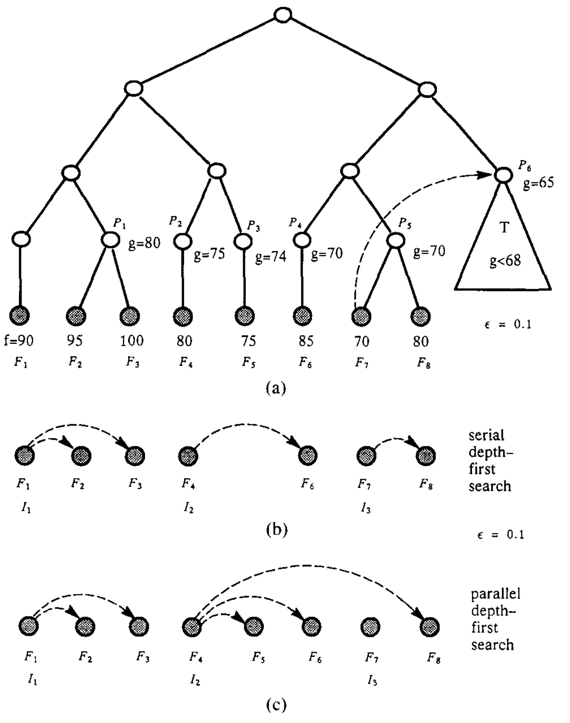

Fig. 2. The Serial/Parallel Incumbent-Update algorithm. (Nodes pointed to by dashed arrows are eliminated by lower-bound tests. I_1, I_2, and I_3 are three distinct incumbents obtained during the serial and parallel searches.) (a) Complete search tree with feasible solutions indicated as shaded nodes. (b) Sequence of incumbents generated and pruned in a sequential depth-first search. (c) Sequence of incumbents generated and pruned using the Serial/Parallel Incumbent-Update algorithm.

Proof: The theorem is proved by induction. Let the sequence of all possible feasible solutions as ordered by the corresponding heuristic values be $F_1, \cdots, F_n$. The initial feasible solutions (or incumbents) in the serial and parallel cases before the search begins are identical. (An infinite value is taken as the first incumbent if there is no heuristic method to find the initial feasible solution.) Suppose that the sequence of the first k distinct incumbents in the serial and parallel searches are identical and that F_i is the kth incumbent. Let $F_j, j > i$, be the $(k + 1)$th distinct incumbent in the serial case. We now show in the parallel search that any feasible solution F_x, $i < x < j$, cannot become an incumbent and that F_j must be the $(k + 1)$th distinct incumbent. Note that for all possible feasible solutions F_x, $i < x < j$, either a) $f(F_x) > f(F_i)$, or b) $g(P_x) \geq f(F_m)/(1 + \epsilon)$, $m \leq i$, where P_x is the parent of F_x. In Case a), F_x will be eliminated by F_i in the parallel case. In Case b), F_x must also be eliminated by F_i in the parallel search according to (2.3) and Step a) of the Serial/Parallel Incumbent-Update algorithm. Since $f(F_j) < f(F_i)$ and $g(P_j) < f(F_i)/(1 + \epsilon)$, F_j cannot be eliminated in the parallel search, where P_j is the parent of F_j. Further, $h(F_j)$ has the smallest heuristic values among those of all possible feasible solutions F_y, $j \leq y \leq n$, hence F_j has to be selected as $(k + 1)$th incumbent in the parallel search. □

The example in Fig. 2 illustrates the Serial/Parallel Incumbent-Update algorithm and demonstrates the result in Theorem 3.2. In this OR-tree, using a depth-first search satisfies the condition that F_i appearing before F_j implies that $h(F_i) < h(F_j)$. The sequence of feasible solutions obtained in the parallel depth-first search with four processors is $F_1, \cdots, F_8$. During the parallel search, five feasible solutions, $F_1, \cdots, F_5$, are obtained in the fourth iteration. According to the Serial/Parallel Incumbent-Update algorithm, F_2, F_3, and F_5 are eliminated in Step a), and F_4 becomes the second incumbent. In the next iteration, F_6 and F_8 are eliminated, and F_7 is the final incumbent. Hence the sequence of incumbents is F_1, F_4, and F_7, which is identical to that of the serial case.

To derive the performance bounds of parallel depth-first and breadth-first searches, the following corollary is needed.

Corollary 3.1: In a parallel OR-tree search with $\boldsymbol{D} = \boldsymbol{I}$, all essential nodes under a given heuristic function must be expanded if the following conditions are satisfied: a) the sequences of distinct incumbents are the same in the serial and parallel cases, and b) $P_i \boldsymbol{L} P_j$ and P_k is the parent of P_i imply 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, then $h(P_3)$ must be less than $h(P_1)$ from Condition a). Since in a serial search, the nodes are expanded according to the heuristic values, P_3 should be selected before P_1. From Condition b), the feasible solution P_2 must be obtained in the serial case as well, hence P_1 must also be eliminated and cannot be an essential node. A contradiction! □

Note that Condition a) of Theorem 3.2 and Condition b) of Corollary 3.1 are not satisfied for all OR-trees and search strategies even if the Serial/Parallel Incumbent-Update Algorithm is used. In particular, we have found that for $\epsilon \geq 0$, the conditions are satisfied for all OR-trees searched by a breadth-first search, and an OR-tree with feasible solutions at the bottom-most level and searched by a depth-first search. The proof for the case using a depth-first search is shown in Corollary 3.2. The proof for the case with a breadth-first search is straightforward and is not shown here. In both cases, the same sequence of distinct incumbents to be generated in the serial and parallel searches can be maintained by the Incumbent-Update Algorithm. Counterexamples can be designed to show that the conditions are invalid when an OR-tree with an arbitrary structure is searched by a best-first or a depth-first search.

C. Parallel Best-First Searches

The performance bounds of a parallel best-first search are shown in the following theorem.

Theorem 3.3: Suppose that $\epsilon = 0$, $\boldsymbol{D} = \boldsymbol{I}$. Let the root be in Level 0 and h be the maximum number of levels in an OR-tree. If the heuristic function satisfies (2.6) and (2.7), then the following bounds hold for the parallel best-first search with k processors.

$$\left(\frac{T_b'(1,0) - 1}{k} + 1\right) \leq T_b(k,0) \leq \left(\frac{T_b(1,0)}{k} + \frac{k-1}{k}h\right) \quad (3.1)$$

where $T_b'(1, 0)$ is the number of essential nodes in a serial best-first search with lower bounds *less than* the optimal-solution value.

Proof: All the required iterations to find an optimal solution can be classified into either perfect or imperfect, and imperfect iterations cause degradation in performance in parallel processing. The proof of the upper bound centers on finding the maximum number of imperfect iterations. Let $h_{\min}(x)$ be the level with the minimum level number in which some active essential nodes reside in the xth iteration, which is an imperfect iteration. For levels less than $h_{\min}(x)$, all active nodes are nonessential. We show that Iteration x may be imperfect only if all essential nodes in $h_{\min}(x)$ are selected for expansion. Suppose that Iteration x is imperfect and that an essential node, say P_i, in $h_{\min}(x)$ is not selected for expansion, then this contradicts the selection rule of a heuristic search, since in this case at least one node with a heuristic value greater than $h(P_i)$ is selected. (Note that all nonessential nodes have heuristic values greater than those of essential nodes.) Thus after 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 h, the maximum number of levels, and the root is defined at Level 0, there can be at most h imperfect iterations. When $\epsilon = 0$ and $\boldsymbol{D} = \boldsymbol{I}$, it is true that once all essential nodes are expanded, the optimal solution must be found. Recall that in each iteration at least one essential node must be selected according to Theorem 3.1. The upper bound of $T_b(k, 0)$ satisfies

$$T_b(k,0) \leq \left(\left\lfloor\frac{T_b(1,0) - h}{k}\right\rfloor + h\right) \leq \left(\frac{T_b(1,0)}{k} + \frac{k-1}{k}h\right). \quad (3.2)$$

To find the lower bound of $T_b(k, 0)$, we know that when $\epsilon = 0$ and $\boldsymbol{D} = \boldsymbol{I}$, all essential nodes with lower bounds less than the optimal-solution value must 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

$$T_b(k,0) \geq \left(\frac{T_b'(1,0) - 1}{k} + 1\right). \quad □$$

The bounds in (3.1) are tight in the sense that examples can be generated to achieve one of these bounds. In the 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 paper, we assume that $[(T_b'(1, 0) - 1)/k + 1]$

$\geq h'$. Notice that if we know $L_{\max}$, the maximum level in which at least one essential node exists, i.e., there is no essential node in levels larger than $L_{\max}$, then the upper bound can be tightened by substituting $L_{\max}$ for h in (3.1). In practice, the information on $L_{\max}$ is not available in advance [19], [20].

In (3.1), h is a function of the problem size. However, $T_b(1, 0)$ and $T_b'(1, 0)$ reflect the complexity of the problem to be solved and are unknown before the solution is found. $T_b(1, 0)$ can be estimated, as in the analysis of Alpha-Beta algorithms [1], [11], [27], by defining a branching factor α.

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

The branching factor measures the average number of branches of an essential node and can be estimated statistically. For example, α is close to 1.1 for knapsack problems when all profits and weights are independent and are generated from uniform distributions. $T_b'(1, 0)$ can be estimated by sampling methods from the problem characteristics and the distribution on the number of subproblems generated.

Let w be $T_b(1, 0)/h$. w can be viewed as the "average width" of an OR-tree, which only consists of essential nodes. Equation (3.2) can be rewritten as

$$\frac{T_b(1, 0)}{T_b(k, 0)} \geq \frac{k \cdot w}{w + (k - 1)}. \tag{3.3}$$

Equation (3.3) shows that if $w >> k$, then the speedup is close to k; whereas if $w << k$, then the lower-bound speedup is close to w.

Equation (3.1) shows that a near-linear speedup can be achieved when a large number of processors are used. 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. In Table I, the theoretical bounds derived above are compared with the simulation results of parallel best-first and depth-first searches for solving two 35-object knapsack problems. In generating the knapsack problems, the weights, $w(i)$, were chosen randomly between 0 and 100 with a uniform distribution, and the profits were set to be $p(i) = (w(i) + 10)$. This assignment is intended to increase the complexity of the randomly generated problems. The results demonstrate that the bounds on parallel best-first searches are very tight, hence its performance can be predicted accurately. Table I also shows that the speedup depends strongly on w. In Case 1, $w \approx 2023$ and a near-linear speedup of $0.93k$ is achieved with 256 processors. In Case 2, $w \approx 188$ and a speedup of $0.48k$ is obtained with 256 processors. Other simulation results also demonstrate a similar behavior.

From Theorem 3.3, it is easy to determinate the maximum number of processors to guarantee a near-linear speedup. Assume that the speedup required is $T_b(1, 0)/T_b(k, 0) \geq (\eta k)$, $0 < \eta < 1$. From (3.1),

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

or

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

For instance, if $\eta = 0.9$, $h = 50$, 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.

Note that in parallel best-first searches with $\epsilon = 0$, essential nodes can be eliminated only if their lower-bounds are equal to the optimal-solution value. When $\epsilon > 0$, other essential nodes can also be eliminated because it is possible for a feasible solution whose value is slightly larger than the optimal-solution value to be found early in the parallel case, while this solution is found quite late or even not found in the serial case. A looser lower bound with respect to approximate best-first searches is derived in Sections IV-C-3.

TABLE I
COMPARISON BETWEEN THEORETICAL BOUNDS AND SIMULATION RESULTS ON PARALLEL BEST-FIRST AND DEPTH-FIRST SEARCHES FOR KNAPSACK PROBLEMS WITH 35 OBJECTS. ($T_b'(1, 0) = T_b(1, 0)$. DURING DEPTH-FIRST SEARCHES, $c = 22$ IN CASE 1 AND $c = 12$ IN CASE 2.)

Num. of proc.	Case 1				Case 2			
	Lower bound	No. of iterat.	Upper bound	Speedup	Lower bound	No. of iterat.	Upper bound	Speedup
Parallel Best-First Search								
1	70790	70790	70790	1.000	6566	6566	6566	1.000
2	35395	35405	35413	1.999	3283	3292	3301	1.995
4	17698	17705	17724	3.998	1642	1659	1668	3.958
8	8849	8857	8880	7.993	821	836	852	7.854
16	4425	4436	4458	15.959	411	431	444	15.234
32	2213	2230	2247	31.745	206	226	240	29.053
64	1107	1124	1141	62.982	103	129	138	50.899
128	554	575	588	123.117	52	78	87	84.179
256	277	298	312	237.557	26	53	61	123.887
Parallel Depth-First Search								
1	70790	70790	70790	1.000	6566	6582	6582	1.000
2	35395	35630	35787	1.987	3283	3488	3513	1.887
4	17698	18044	18285	3.923	1642	1940	1978	3.393
8	8849	8884	9534	7.968	821	1161	1211	5.669
16	4425	4460	5159	15.872	411	777	827	8.471
32	2213	2247	2971	31.504	206	584	635	11.271
64	1107	1143	1877	61.934	103	485	539	13.571
128	554	592	1330	119.578	52	219	491	30.055
256	277	316	1057	224.019	26	90	467	73.133

D. 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 shows the performance bounds of parallel depth-first OR-tree searches. The range of bounds on parallel depth-first searches are larger than that of parallel best-first searches.

Theorem 3.4: For a parallel depth-first search with k processors, $\epsilon = 0$, $\boldsymbol{D} = \boldsymbol{I}$, and a generalized heuristic function $h(P_i)$ = (path number, level number), then

$$\left(\frac{T_b'(1, 0) - 1}{k} + 1\right) \le T_d(k, 0)$$
$$\le \left(\frac{T_d(1, 0)}{k} + \frac{k-1}{k}\left[(c+1)h - c\right]\right) \quad (3.4)$$

where c is the number of the distinct incumbents obtained during the serial depth-first search, and $T_b'(1, 0)$ is the number of essential nodes in a serial best-first search with lower bounds *less than* the optimal-solution value.

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 corresponding parents be denoted by $F_1, \cdots, F_c$, and $P_1, \cdots, P_c$. Further, assume that $F_1, \cdots, F_c$ are obtained in the i_1th, $\cdots$, i_cth iterations, respectively. Hence iterations from 1 to i_1 belong to the first subsequence, and iterations from $i_j + 1$ to i_{j+1} belong to the jth subsequence.

Consider the first subsequence of iterations. Suppose that the jth, $1 \le j \le i_1$, iteration is imperfect. This imperfect iteration occurs because either less than k nodes are selected due to insufficient active subproblems, or some nodes are expanded in the parallel depth-first search but are eliminated in the serial case. In the latter case the heuristic values of these nodes must be greater than $h(P_1)$ because the incumbents used initially in the serial and parallel cases were identical, and if the heuristic values of all expanded nodes in an iteration were 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 with heuristic values less than $h(P_1)$ reside in the xth imperfect iteration. Similar to the proof of Theorem 3.3, we have

$$h_{\min}(x + 1) \ge (h_{\min}(x) + 1).$$

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

Analogous to the above argument, we can prove that in the ith, $1 < i \le c$, subsequence of iterations, at most h imperfect iterations will be encountered before a better feasible solution is obtained. During the last subsequence of iterations, since the optimal solution has been generated, all iterations are imperfect only if less than k nodes are selected in each iteration. In other words, an imperfect iteration implies that all currently active nodes are selected and expanded, and only descendents of these nodes 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 iteration in the parallel case belongs to Φ^1, the upper bound of $T_d(k, 0)$ can be derived as

$$T_d(k, 0) \le \left(\frac{T_d(1, 0) - (c+1)h}{k} + (c+1)h\right).$$

In the above discussion, the expansion of the root is counted in each of the $(c + 1)$ subsequences. Since the root is only expanded once, the above upper bound should be compensated by the additional number of times that the root is expanded (3.4).

The lower bound on $T_d(k, 0)$ can be proved easily because all essential nodes in a serial best-first search with lower bounds less than the optimal solution must be expanded in the parallel depth-first search. □

For problems such as integer programming and 0-1 knapsack, all feasible solutions are located in the bottommost level of the OR-tree. In this case, the following corollary shows that all essential nodes of a serial depth-first search must be expanded in a parallel depth-first search, and a tighter lower bound is obtained.

Corollary 3.2: In searching an OR-tree using a parallel depth-first search and a heuristic function of (path number, level number), if all feasible solutions are in Level h, and the Serial/Parallel Incumbent-Update strategy is used, then

$$\left(\frac{T_d(1, \epsilon) - 1}{k} + 1\right) \le T_d(k, \epsilon) \quad (3.5)$$

where h is the maximum number of levels of the OR-tree.

Proof: First we show that under the given assumptions, $P_i L P_j$ implies that $h(P_k) < h(P_j)$, where P_i is in Level h, P_j is in Level j, $j < h$, and P_k is the parent node of P_i. Suppose that $P_i L P_j$ and $h(P_j) < h(P_k)$, then when P_k is expanded, an ancestor of P_j, say P_a, has to be expanded simultaneously from the defined selection rule and (2.6). Note that when P_a is active, $h(P_a) < h(P_k)$, that P_k is in Level $(h - 1)$, and that P_a is in a level less than $(h - 1)$. Let P_m be the first common ancestor of P_i and P_j. By carrying out the above argument repeatedly, P_m has a child P_a' which is expanded simultaneously with an indirect descendant P_k', where P_a' is an ancestor of P_a, and P_k' is an ancestor of P_k. It is also seen that P_k' is at a level number larger than that of P_a'. Thus when P_m is expanded, all descendant nodes between P_m and P_k' must be generated simultaneously, which is an impossible situation.

The above argument also shows that a feasible solution P_x obtained before P_i implies that $h(P_x) < h(P_i)$. By using the Serial/Parallel Incumbent-Update strategy discussed in Section III-B, the sequences of distinct incumbents are the same in the serial and parallel cases according to Theorem 3.2. By Corollary 3.1, all essential nodes must be expanded in a parallel depth-first search. The lower bound of $T_d(k, \epsilon)$ is derived immediately. □

The bounds in Theorem 3.4 are tight in the sense that we can construct examples to achieve the lower-bound and upper-bound computational times. These degenerate cases occur rarely. Simulations have revealed that for a number of OR-tree search problems, $T_d(k, 0)$ may be very close to $T_b(k, 0)$. The second part of Table I shows the simulation results of solving two cases of the 0-1 knapsack problems using a depth-first search. Note that when the number of processors is large, the number of essential nodes in each imperfect iteration of the parallel depth-first search is usually larger than one. In contrast to the upper bound in (3.4), which was derived with the assumption of one essential node in each imperfect iteration, $T_d(k, 0)$ may be much smaller than the upper bound.

Although c, the number of distinct incumbents, is unknown until the solution is found, c is usually small and can be estimated when integral solutions are sought. It has been observed that c is less than 10 for vertex-cover problems with less than 100 vertices. For most integer programming problems, $c \approx 1$. In these cases, the bounds given in Theorem 3.4 are tight.

E. Parallel Breadth-First Searches

The following theorem presents the performance bounds on parallel breadth-first searches.

Theorem 3.5: For a parallel breadth-first search with k processors, $\boldsymbol{D} = \boldsymbol{I}$, and a generalized heuristic function satisfying (2.6) and (2.7), and using the Serial/Parallel Incumbent-Update strategy, then

$$\left(\frac{T_r(1, \epsilon) - 1}{k} + 1\right) \le T_r(k, \epsilon) \le \left(\frac{T_r(1, \epsilon)}{k} + \frac{k-1}{k}(c + h)\right) \quad (3.6)$$

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

Proof: For a parallel breadth-first search, 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 and may happen at most h times from our discussion in Theorems 3.3 and 3.4. Second, imperfect iterations may occur if there exist some nodes that are eliminated by a feasible solution in the serial case but are expanded simultaneously with the parent of this feasible solution in the parallel case. From Theorem 3.1, it is easy to show that at least one essential node must be expanded in a parallel iteration. $T_r(k, \epsilon)$ is, thereby, bounded by

$$T_r(k, \epsilon) \le \left(\frac{T_r(1, \epsilon)}{k} + \frac{k-1}{k}(c + h)\right).$$

On the other hand, in breadth-first searches, all essential nodes have to be expanded in the parallel case (Theorem 3.2 and Corollary 3.1). Thus $T_r(k, \epsilon)$ has the following lower bound.

$$\left(\frac{T_r(1, \epsilon) - 1}{k} + 1\right) \le T_r(k, \epsilon) \qquad \square$$

Since the performance bounds of $T_r(k, 0)$ are tighter than those of $T_d(k, 0)$, the performance of parallel breadth-first searches can be predicted more accurately.

IV. Coping with Anomalies in Parallel OR-Tree Searches

Up to now, the efficiency of parallel OR-tree searches have been little studied. When comparing the efficiency between using k_1 processors and k_2, $1 \le k_1 < k_2$, processors, a k_2/k_1-fold speedup (ratio of the number of iterations in the two cases in our model) is expected. However, simulations have shown that the speedup can be a) less than one (called a *detrimental anomaly*) [10], [18], [24]; or b) greater than k_2/k_1 (called an *acceleration anomaly*) [10], [18]; or c) between one and k_2/k_1 (called a *deceleration anomaly*) [10], [18], [24]. So far, all known results on parallel OR-tree searches showed a near-linear speedup for only a small number of processors.

The objective of this section is to develop, using the results in Section III, *conditions to cope with the anomalous behavior of parallel OR-tree search algorithms in the possible presence of approximations and dominance tests.* 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. Anomalies are also studied with respect to the assumption that all idle processors are used to expand active subproblems. In fact, detrimental anomalies cannot happen if some processors can be kept idle in the presence of active subproblems. The number of processors to be kept idle is problem dependent and is very difficult to find without first solving the problem.

The conditions to resolve anomalies are described with respect to *serial-to-parallel* processing, which is processing between using one and k processors, and *parallel-to-parallel* processing, which is processing between using k_1 and k_2, $1 < k_1 < k_2$ processors. The conditions to cope with serial-to-parallel anomalies and parallel-to-parallel anomalies are different. These conditions we developed are useful for designers to understand the existence of anomalies and to modify existing algorithms to prevent detrimental anomalies and enhance acceleration anomalies.

A. Anomalies of Parallel OR-Tree Searches

In this section, some anomalies on parallel OR-tree searches are illustrated. A single list of subproblems is assumed, and generalizations to multiple lists are discussed later.

For the OR-tree in Fig. 3(a), the order of nodes expanded in a serial depth-first search is P_0, P_1, P_i, all nodes in T_i resulting in a feasible solution P_i' with value 6, P_k, and all nodes in T_k resulting in the optimal solution of $f(P_k') = 5$. P_j is terminated by the lower-bound test with P_i'. In contrast, when two processors are used, P_1 and P_2 are expanded concurrently. P_i is terminated as a result of dominance by P_k. Since T_i is terminated, T_j is expanded next. If T_j is large, then the combined time of expanding

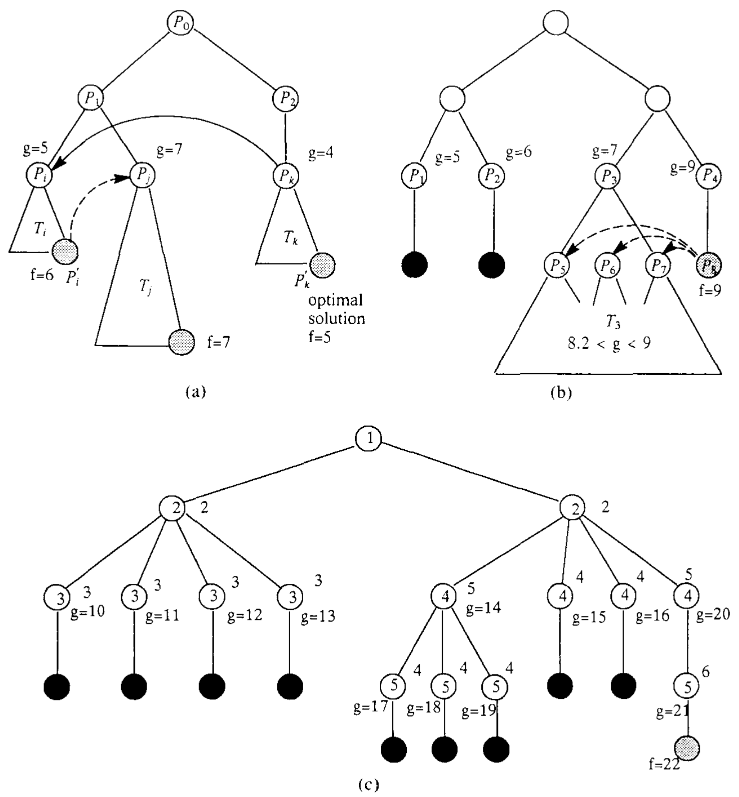

Fig. 3. Examples of detrimental anomalies. (Nodes pointed to by solid arrows are eliminated by dominance tests; nodes pointed to by dashed arrows are eliminated by lower-bound tests. Shaded nodes indicate feasible solutions. Dark nodes indicate infeasible solutions.) (a) With lower-bound and dominance tests, $T_d(2, 0) > T_d(1, 0)$. (b) With approximate lower-bound tests, $T_b(3, 0.1) > T_b(2, 0.1)$; $T_d(3, 0.1) > T_d(2, 0.1)$. (c) Without lower-bound and dominance tests in a depth-first or best-first search $T(4, 0) = 5$, $T(5, 0) = 6$. (The number inside the node is the evaluation order using four processors; the number outside the node is the evaluation order using five processors.)

T_j and T_k using one processor can be smaller than the combined time of expanding T_j and T_k using two processors.

Fig. 3(b) illustrates a detrimental anomaly under an approximate depth-first or best-first search with $\epsilon = 0.1$. When two processors are used, $f(P_8)$, the optimal solution, is found in the fourth iteration. Assuming that the lower bounds of nodes in T_3 are between 8.2 and 9, all nodes in T_3 will be eliminated by lower-bound tests with P_8 since $[9/(1 + \epsilon)] < 8.2$. When three processors are used, P_3 is expanded in the third iteration. P_5, P_6, and P_7 are generated and will be selected in the next iteration. If T_3 is large, $T(2, \epsilon) < T(3, \epsilon)$ will occur.

An example of an acceleration anomaly with an approximate depth-first or best-first search is shown in Fig. 4(a). When three processors are used, the optimal solution is found in the second iteration, and P_4 and P_5 are eliminated. If two processors are used, subtrees T_4 and T_5 have to be expanded. $T(2, 0.1)/T(3, 0.1)$ will be much larger than $3/2$ if T_4 and T_5 are very large.

Fig. 4(b) illustrates another example of acceleration anomalies with dominance tests under a best-first, depth-first, or breadth-first search strategy. When three processors are used, P_4 will be dominated by P_6, and four iterations are required to complete the search. In contrast, when two processors are used, P_4 is expanded before P_6 is generated. If no dominance relation exists between P_6 and the descendants of P_4 (which is possible), then T_4 has 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 inactive. Fig. 3(c) shows a detrimental anomaly in a depth-first or best-first search. The example in Fig. 4(c) illustrates that acceleration anomalies may occur regardless of the search strategy.

Fig. 4(d) shows an acceleration anomaly when the subproblem list is decomposed into local lists, and dominance relations are restricted to subproblems in the local lists only. This phenomenon is described in Section IV-C-4.

Table II illustrates the anomalous behavior of parallel OR-tree search of a vertex-cover problem. In this problem, the minimum number of vertices to cover all edges in an undirected graph is to be found. That is, all edges

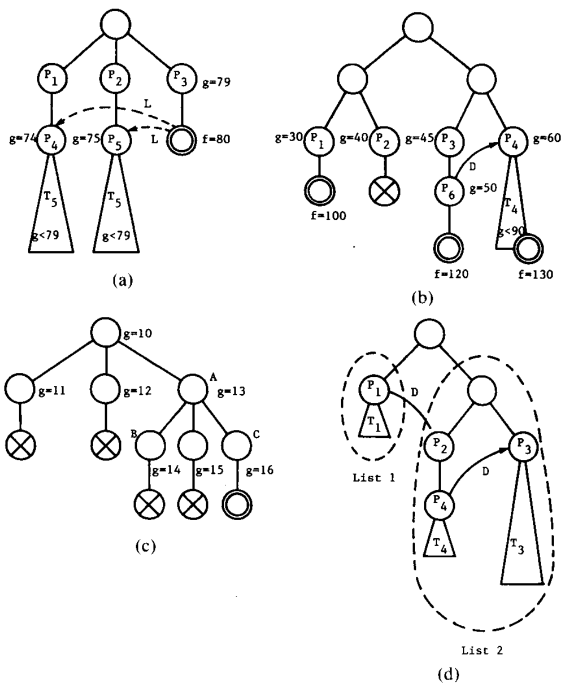

Fig. 4. Examples of acceleration anomalies. (Double nodes indicate feasible solutions. Crossed out nodes indicate infeasible solutions.) (a) With approximate lower-bound tests; $T_b(2, 0.1)/T_b(3, 0.1) > \frac{3}{2}$; $T_d(2, 0.1)/T_d(3, 0.1) > \frac{3}{2}$. (b) With dominance tests only; dominance tests are consistent with g and h for all search strategies; $T(2, 0)/T(3, 0) > \frac{3}{2}$. (c) Without lower-bound and dominance tests for all search strategies; $T(2, 0)/T(3, 0) = \frac{5}{3} > \frac{3}{2}$. (d) With two subproblem lists and local dominance tests only; $T(1, 0)/T(2, 0) = 2$.

TABLE II
EXAMPLES OF ANOMALOUS BEHAVIOR OF PARALLEL OR-TREE SEARCHES FOR A VERTEX-COVER PROBLEM WITH 80 VERTICES, $\epsilon = 0$, AND USING A DEPTH-FIRST SEARCH ($c = 3$). (S-P INDICATES SERIAL-TO-PARALLEL ANOMALIES; P-P INDICATES PARALLEL-TO-PARALLEL ANOMALIES.)

k	$T_d(k,0)$	Speedup $T_d(1,0)/T_d(k,0)$	k/(k−1)	Speedup $T_d(k-1,0)/T_d(k,0)$	Anomaly
1	4134	1.000			
2	2065	2.002	2.000	2.002	S-P, P-P acceleration
3	1474	2.805	1.500	1.401	
4	1042	3.967	1.333	1.415	P-P acceleration
5	854	4.841	1.250	1.220	
6	717	5.766	1.200	1.191	
7	626	6.604	1.167	1.145	
8	516	8.012	1.143	1.213	S-P, P-P acceleration
9	481	8.595	1.125	1.073	
10	438	9.438	1.111	1.098	
11	398	10.387	1.100	1.101	P-P acceleration
12	365	11.326	1.091	1.090	
13	380	10.879	1.083	0.961	P-P detrimental
14	298	13.872	1.077	1.275	P-P acceleration
15	271	15.255	1.071	1.100	S-P, P-P acceleration
16	257	16.086	1.067	1.054	S-P acceleration

in the graph emanate from at least one of the included vertices. In the simulations, the graphs were generated randomly by assigning an edge between a pair of vertices if a random number generated was larger than 0.2.

Many anomalous examples can be created for various combinations of search strategies, allowance functions, and elimination rules. However, the important point here is not in knowing that anomalies exist, but in *understanding why these anomalies occur* and in *developing strategies to cope with these anomalies.* 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 to avoid 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 enumerate them for a given problem. Further, the necessary conditions developed for one problem cannot be generalized to other problems. For a similar reason, sufficient conditions to preserve acceleration anomalies are not evaluated.

B. Coping with Serial-to-Parallel Anomalies

In this section, we discuss results when dominance tests are active. Some results on coping with serial-to-parallel anomalies without dominance tests have been derived elsewhere and are only summarized here [20], [23].

1) Sufficient Condition to Eliminate Serial-to-Parallel Detrimental Anomalies:

a) Finding an Exact Optimal Solution: In this section, we show that $T(k, 0) \leq T(1, 0)$ holds if the heuristic function is monotone and unambiguous, and the dominance relation satisfies some consistency requirements on the lower-bound and heuristic functions.

The following concept on transitivity of lower-bound and dominance tests is needed. Recall that $P_i \boldsymbol{D} P_k$ implies that $f(P_i) \leq f(P_k)$, but the converse is false because some nodes are incomparable (otherwise, the number of active nodes can always be reduced to one). When lower-bound and dominance tests are used together, it is important to note that dominance tests are not transitive with lower-bound tests. That is, $P_i \boldsymbol{L} P_j$ and $P_j \boldsymbol{D} P_k$ do not imply $P_i \boldsymbol{L} P_k$. Similarly, $P_i \boldsymbol{D} P_j$ and $P_j \boldsymbol{L} P_k$ do not imply $P_i \boldsymbol{D} P_k$. In both cases, only $f(P_i) \leq f(P_k)$ can be deduced.

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

Theorem 4.1: $T(k, 0) \leq T(1, 0)$ holds for heuristic searches that satisfy (2.6) and (2.7) and that has dominance relation $\boldsymbol{D}$ which is consistent with the lower-bound function g and the heuristic function h.

Proof: The proof is not shown due to space limitation [21], [22]. □

The requirement on the consistency of $\boldsymbol{D}$ with g is satisfied in many practical problems, such as the shortest-path problem [9], the traveling-salesman problem [25], the n-job two-machine mean-completion-time flowshop problem [3], and the n-job one-machine scheduling problem with deadlines [29]. However, the requirement on the consistency of $\boldsymbol{D}$ with h may not be satisfied in general. The detrimental anomaly illustrated in Fig. 3(a) for a depth-first search is caused by the inconsistency of $\boldsymbol{D}$ with h. The dominance relation for the 0-1 knapsack problem is consistent with the upper-bound function (instead of the lower-bound function for maximization problems). 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 are the same as the corresponding profit and weight of P_j, it is possible that either $P_i\boldsymbol{D}P_j$ or $P_j\boldsymbol{D}P_i$. If $h(P_i) < h(P_j)$ happens but $P_j\boldsymbol{D}P_i$ is true, then $\boldsymbol{D}$ is inconsistent with h. For the knapsack problem, detrimental anomalies can be avoided if the dominance relation is redefined as follows. For P_i and P_j defined on a given subset of objects, $P_i\boldsymbol{D}P_j$ if i) the profit of P_i is larger than or equal to that of P_j and the total weight of P_i is less than that of P_j; or ii) the profit of P_i is larger than that of P_j and the total weight of P_i is less than or equal to that of P_j; or iii) the profit and weight of P_i are equal to the corresponding profit and weight of P_j and the heuristic value of P_i is less than that of P_j.

In general, Theorem 4.1 pinpoints the problem when detrimental anomalies happen and serves as a useful guideline to eliminate them.

b) Finding an Approximate Solution: When approximations are allowed, Theorem 4.1 is not always true. The reason for the detrimental anomaly is that $\boldsymbol{L}$, the lower-bound elimination rules under approximation, are not transitive. That is, $P_i\boldsymbol{L}P_j$ and $P_j\boldsymbol{L}P_k$ do not imply $P_i\boldsymbol{L}P_k$, since $f(P_i)/(1+\epsilon) \le g(P_j)$ and $f(P_j)/(1+\epsilon) \le g(P_k)$ imply $f(P_i)/(1+\epsilon)^2 \le g(P_k)$ rather than $f(P_i)/(1+\epsilon) \le g(P_k)$. Although the lower-bound tests are not transitive, the conditions in Theorem 4.1 are sufficient to avoid detrimental anomalies for best-first searches.

Theorem 4.2: $T_b(k, \epsilon) \le T_b(1, \epsilon)$, $\epsilon > 0$, holds for best-first searches when h, the heuristic function, satisfies (2.6) and (2.7), and $\boldsymbol{D}$, the dominance relation, is consistent with h and g.

Proof: The proof is omitted due to space limitation [22], [23]. □

To avoid detrimental anomalies in approximate breadth-first and depth-first searches, the sufficient conditions are more restricted than those of Theorem 4.2. A possible set of sufficient conditions are shown below.

Theorem 4.3: Suppose i) that the heuristic function satisfies (2.6) and (2.7), ii) that the dominance relation is consistent with h and g, and iii) that the sequences of feasible solutions obtained in the serial and parallel cases are identical, then $T(k, \epsilon) \le T(1, \epsilon)$, $\epsilon \ge 0$.

Proof: Suppose that P_1, a basic node in the parallel search, is eliminated in the serial case by a feasible solution P_2 and that P_3 is a parent of P_2, then $h(P_3) < h(P_1)$. From the definition of basic nodes and the assumption that the sequences of feasible solutions are the same in the serial and parallel cases, P_3 must be expanded in the parallel case, and P_2 is obtained before P_1 is expanded. Hence P_1 has to be eliminated by P_2 in the parallel case. A contradiction! □

In general, the sequences of distinct incumbents in the serial and parallel cases are usually not identical. However, for breadth-first searches and some special cases of depth-first searches, such as the case when all feasible solutions are in the bottommost level of the OR-tree, the conditions of Theorem 4.3 are satisfied if the Serial/Parallel Incumbent-Update algorithm is used in both the serial and parallel cases (Theorem 3.2).

2) Necessary Conditions to Allow Serial-to-Parallel 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 IV-B-1. A dominance relation, $\boldsymbol{D}$, is said to be *inconsistent* with h if there exist two nodes P_i and P_j such that $P_i\boldsymbol{D}P_j$ 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* (resp. *completely consistent*) with the lower-bound function g if $h(P_i) < h(P_j)$ implies that $g(P_i) \le g(P_j)$ (resp. $g(P_i) < g(P_j)$) for all $P_i, P_j \in \boldsymbol{P}$. 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 $g(P_i) \le g(P_j)$. 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 i) that the heuristic function is not completely consistent with g, or ii) that the dominance relation is inconsistent with h.

Proof: The proof is omitted due to space limitation [22], [23]. □

According to Theorem 3.5, it is easy to show that if $\epsilon = 0$ and dominance tests are inactive, then no serial-to-parallel acceleration anomalies exist when a breadth-first search is used. However, if a best-first search is used, then the heuristic function is not completely consistent with the lower-bound function when multiple subproblems can have identical lower bounds. Hence acceleration anomalies can occur, but detrimental anomalies can be prevented by the use of an unambiguous heuristic function. The depth-first search has a similar behavior.

Note that the conditions in Theorem 4.4 are not necessary when approximate solutions are sought; that is, acceleration anomalies may occur in this case even though h is completely consistent with g and the dominance tests are consistent with h. The corresponding necessary conditions are studied in Section IV-C-3.

C. Coping with General Parallel-to-Parallel Anomalies

In Sections III-C through III-E, we have derived three theorems for the performance bounds with respect to different search strategies. From these results, we can investigate the relative efficiency between using k_1 and k_2, $1 < k_1 < k_2$, processors. First, we discuss the simple cases with $\epsilon = 0$ and $\boldsymbol{D} = \boldsymbol{I}$ in Sections IV-C-1 and 2. Then strategies for cases with approximate lower-bound and dominance tests are studied in Sections IV-C-3 and 4.

1) Sufficient Conditions to Eliminate Parallel-to-Parallel Detrimental Anomalies When $\epsilon = 0$ and $\boldsymbol{D} = \boldsymbol{I}$: 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 3.3, then $T_b(k_2, 0) \le T_b(k_1, 0)$ when

$$\frac{T_b(1, 0)}{h} \ge \frac{k_1 k_2}{r_b' k_2 - k_1} \quad \text{and} \quad r_b' > \frac{k_1}{k_2}, \qquad 1 < k_1 < k_2 \tag{4.1}$$

where $r_b' = T_b'(1, 0)/T_b(1, 0)$.

Proof: From Theorem 3.3, at least $((T_b'(1, 0) - 1)/k_1 + 1)$ iterations are needed for k_1 processors, and at most $(T_b(1, 0)/k_2 + (k_2 - 1)/k_2\, h)$ iterations are needed for k_2 processors to find an optimal solution in a given OR-tree using a best-first search. The sufficient condition for $T_b(k_2, 0) \le T_b(k_1, 0)$ is

$$\left(\frac{T_b(1, 0)}{k_2} + \frac{k_2 - 1}{k_2} h\right) \le \left(\frac{T_b'(1, 0) - 1}{k_1} + 1\right).$$

To derive (4.1), we use a stronger condition such that

$$\left(\frac{T_b(1, 0)}{k_2} + h\right) \le \frac{T_b'(1, 0)}{k_1}. \qquad \square$$

For the example in Fig. 3(c), $T_b(1, 0) = 15$, $k_1 = 4$, $k_2 = 5$, $h = 4$, and $r_b' = 1$. Equation (4.1) is not satisfied, hence an increase in the number of processors from four to five may not ensure an improvement in performance. As another example, for $h = 50$, $T_b(1, 0) = 10^6$, and $r_b' = 1$, there will not be any detrimental anomalies for any combinations of k_1 and k_2 such that $1 \le k_1 < k_2 \le 141$.

As mentioned in Section III-C, the term $T_b(1, 0)/h$ can be viewed as the average width of the essential OR-tree consisting of essential nodes only. Intuitively, detrimental anomalies can be prevented if the essential 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, k_2/k_1 is sufficiently large, and $r_b' \approx 1$, then $T_b(k_2, 0)$ will be less than $T_b(k_1, 0)$. If $r_b' << 1$, that is, there are a large number of active nodes with lower bounds equal to the optimal-solution value, then the condition in (4.1) may not be satisfied, and detrimental anomalies may happen.

Similarly, for a depth-first search, the corresponding sufficient conditions can also be determined. The conditions derived are more restricted than (4.1) because the range on $T_d(k, 0)$ are larger (Theorem 3.4). To simplify the sufficient conditions, the following bounds on $T_d(k, 0)$ are used.

$$\frac{T_b'(1, 0)}{k} \le T_d(k, 0) \le \left(\frac{T_d(1, 0)}{k} + (c + 1)h\right).$$

Corollary 4.2: Let $r_d' = T_b'(1, 0)/T_d(1, 0) \le 1$. In a parallel depth-first search that satisfies the assumptions of Theorem 3.4, $T_d(k_2, 0) \le T_d(k_1, 0)$ when

$$\frac{T_d(1,0)}{h} \ge \frac{(c+1)k_1 k_2}{r_d' k_2 - k_1} \quad \text{and} \quad r_d' > \frac{k_1}{k_2}, \qquad 1 < k_1 < k_2 \tag{4.2}$$

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

Proof: The proof is similar to that of Corollary 4.1. $\square$

From Corollary 4.2, we can conclude that the existence of parallel-to-parallel detrimental anomalies in depth-first searches depends on $T_b'(1, 0)$, r_d', and c. If $r_d' \approx 1$, c is small, and $T_b'(1, 0)$ is very large, then (4.2) will 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 $T_b'(1, 0)$, hence $r_d' \approx 1$. Moreover, if the feasible-solution values must be integers, then 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 large and k_2 is relatively small. However, the range of parallel processing within which no detrimental anomalies occur for depth-first searches is smaller than that for best-first searches.

Corollary 4.2 can be verified by using the example in Table II in which $r_d' \approx 1$, $c = 3$, $h = 80$, and $T_d(1, 0) = 4134$. From (4.2), we found that no detrimental anomalies occur when $k_2 \le 4$. The detrimental anomaly observed when k is 13 is beyond the predicted range.

The following theorem proves that detrimental anomalies do not occur when a breadth-first search is used. Note that this theorem is not a corollary of Theorem 3.5.

Theorem 4.5: In a parallel breadth-first search, if $\boldsymbol{D} = \boldsymbol{I}$ and the generalized heuristic function satisfies (2.6) and (2.7), then $T_r(k_2, 0) \le T_r(k_1, 0)$, $1 < k_1 < k_2$.

Proof: Suppose that there are m imperfect iterations when k_2 processors are used. The sequence of iterations can be divided into $(m + 1)$ subsequences, each of which ends in an imperfect iteration except (possibly) the last one. Consider an arbitrary subsequence, say subsequence j, which contains x perfect iterations and ends in an imperfect iteration. The theorem can be proved by showing that, when k_1 processors are used, the total number of iterations required to expand some of the nodes while eliminating other nodes in subsequence j must be greater than or equal to $(x + 1)$.

To account for the iterations expanded in subsequence j when k_1 processors are used, we adopt the convention that any node expanded in conjunction with nodes in subsequence $(j + 1)$ is not counted. When k_2 processors are used, an imperfect iteration occurs in a breadth-first search only when either a) the number of active nodes is less than k_2, or b) at least one new incumbent is found in this iteration, and one or more nodes expanded in this iteration have lower-bound values larger than the new incumbent. Let P_i be the node having the smallest heuristic value in the imperfect iteration of subsequence j considered, and

P_j be the node with the largest heuristic value in subsequence j. If the imperfect iteration in subsequence j belongs to Case a) above, then when k_1 processors are used, all nodes with heuristic values larger than $h(P_j)$ cannot be expanded simultaneously with P_i, because all of them are descendents of nodes belonging to the imperfect iteration. If the imperfect iteration in subsequence j belongs to Case b) above, then when k_1 processors are used, node P_i also cannot be expanded simultaneously with nodes having heuristic values larger than $h(P_j)$; otherwise, the number of nodes expanded in this iteration would be larger than k_1. Therefore, Node P_i has to be included in subsequence j when counting the number of iterations under k_1 processors.

Since there are x perfect iterations in subsequence j, from Corollary 3.1, all $k_2 \cdot x$ essential nodes must be expanded when k_1 processors are used. Among nodes in subsequence j, the minimum number of nodes that must be expanded when k_1 processors are used is $(k_2 \cdot x + 1)$. Hence the minimum number of iterations with respect to k_1 processors is

$$\left\lceil \frac{k_2 \cdot x + 1}{k_1} \right\rceil \geq (x + 1), \qquad 1 < k_1 < k_2.$$

Since the above inequality holds for any subsequence, we obtain $T_r(k_2, 0) \leq T_r(k_1, 0)$. □

2) Necessary Conditions to Allow Parallel-to-Parallel Acceleration Anomalies When $\epsilon = 0$ and $\mathbf{D} = \mathbf{I}$: From Theorem 3.3, we can derive a necessary condition for acceleration anomalies with respect to k_1 and k_2 processors in parallel best-first searches.

Corollary 4.3: In a parallel best-first search that satisfies the assumptions of Theorem 3.3,

$$T_b(k_1, 0)/T_b(k_2, 0) > k_2/k_1$$

only if

$$\big(T_b(1, 0) - T_b'(1, 0)\big) > \big(k_2 - 1 - (k_1 - 1)h\big) \tag{4.3}$$

for $1 < k_1 < k_2$.

Proof: From Theorem 3.3, when k_1 processors are used, at most $(T_b(1, 0)/k_1 + (k_1 - 1)h/k_1$ iterations are needed. When k_2 processors are used, at least $(T_b'(1, 0) - 1)/k_2 + 1)$ iterations are needed. The necessary condition follows directly. □

Note that this necessary condition cannot be obtained from the looser lower bound $T_b(k, 0) \geq T_b(1, 0)/k$ derived by Lai and Sahni [18].

Fig. 4(b) illustrates the acceleration anomaly that $T_b(2, 0)/T_b(3, 0) > 3/2$, which obviously satisfies (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 a 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 bounds on $T_b(k, 0)$.

For depth-first searches, the necessary condition for acceleration anomalies is as follows.

Corollary 4.4: In a parallel depth-first search that satisfies the assumptions of Theorem 3.4,

$$T_d(k_1, 0)/T_d(k_2, 0) > k_2/k_1$$

only if

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

for $1 < k_1 < k_2$.

Proof: The proof is similar to that of Corollary 4.3. □

Obviously, the necessary condition in (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. Table II shows that acceleration anomalies occur frequently. If all solutions are located at the bottommost level of the OR-tree, then the corresponding necessary condition is simplified as (from Corollary 3.2):

$$\big((c + 1)h - c\big) > \frac{k_2 - 1}{k_1 - 1}, \qquad 1 < k_1 < k_2. \tag{4.5}$$

Analogous to Corollaries 4.3 and 4.4, we can derive from Theorem 3.5 the necessary condition for acceleration anomalies to exist in a parallel breadth-first search. That is,

$$(c + h) > \frac{k_2 - 1}{k_1 - 1}, \qquad 1 < k_1 < k_2. \tag{4.6}$$

3) Coping with Parallel-to-Parallel Anomalies in Approximate Searches: It should be noted that Theorem 4.2 is no longer valid when comparing $T_b(k_1, \epsilon)$ and $T_b(k_2, \epsilon)$. Fig. 3(b) gives a counterexample showing that $T_b(3, 0.1) > T_b(2, 0.1)$ even if all lower bounds are distinct. This happens because a node with a larger lower bound (not a basic node) may be expanded before nodes with smaller lower bounds in the parallel case, and nodes with smaller lower bounds may be eliminated by approximate lower-bound tests.

Analogous to the proof of Theorem 3.3, the upper bound on $T_b(k, \epsilon)$ can be derived. To find the lower bound on $T_b(k, \epsilon)$, let f_o be the optimal-solution value and $\text{MINT}_b(\epsilon)$ be the minimum number of nodes that are expanded in the approximate best-first search. $\text{MINT}_b(\epsilon)$ represents the number of nodes whose lower bounds are less than $f_o/(1 + \epsilon)$, since these nodes must be expanded in the best case. $T_b(1, \epsilon)$ may not achieve $\text{MINT}_b(\epsilon)$ because essential nodes may be eliminated by approximate lower-bound tests in the parallel search.

$\text{MINT}_b(\epsilon)$ may be estimated from the distribution of the number of subproblems with respect to lower bounds. From simulations on the 0-1 knapsack and vertex-cover problems, it was observed that the distributions are exponential. In this case, let $a^g \partial g$, $a > 1$, be the number of subproblems whose lower bounds are between g and $(g + \partial g)$. It is easy to show that

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

$$\mathrm{MINT}_b(\epsilon) = \int_{-\infty}^{f_0/(1+\epsilon)} a^g \partial g = (T_b(1,0))^{1/(1+\epsilon)}$$

From the above analysis, we get

$$\left(\frac{\mathrm{MINT}_b(\epsilon) - 1}{k} + 1\right) \le T_b(k, \epsilon) \le \left(\frac{T_b(1, \epsilon)}{k} + \frac{k-1}{k} h\right). \quad (4.7)$$

Similarly, for depth-first searches,

$$\left(\frac{\mathrm{MINT}_b(\epsilon) - 1}{k} + 1\right) \le T_d(k, \epsilon) \le \left(\frac{T_d(1, \epsilon)}{k} + \frac{k-1}{k}[(c+1)h - c]\right). \quad (4.8)$$

Corollary 4.5:[2] In parallel best-first searches that satisfy the assumptions of Theorem 3.3 with the exception that $\epsilon > 0$, $T_b(k_2, \epsilon) \le T_b(k_1, \epsilon)$ when

$$\frac{T_b(1, \epsilon)}{h} \ge \frac{k_1 k_2}{r_b k_2 - k_1} \quad \text{and} \quad r_b > \frac{k_1}{k_2} \qquad 1 < k_1 < k_2 \quad (4.9)$$

where $r_b = \mathrm{MINT}_b(\epsilon)/T_b(1, \epsilon)$. Similarly, $T_b(k_1, \epsilon)/T_b(k_2, \epsilon) > k_2/k_1$ when

$$(T_b(1, \epsilon) - \mathrm{MINT}_b(\epsilon)) > (k_2 - 1 - (k_1 - 1)h), \quad 1 < k_1 < k_2. \quad (4.10)$$

If $(k_1 + 1)/k_2 < r_b$, then we can predict that no detrimental anomalies occur if $k_2 \le \sqrt{T_b(1, \epsilon)/h}$. As (4.10) is quite loose, it is often satisfied. When $k_1 = 1$ and $k_2 = k$, the corresponding necessary condition becomes

$$(T_b(1, \epsilon) - \mathrm{MINT}_b(\epsilon)) > (k - 1). \quad (4.11)$$

For depth-first searches with approximate lower-bound tests, the following corollary shows the required conditions.

Corollary 4.6:[2] In parallel depth-first searches that satisfy the assumptions of Theorem 3.4 with the exception that $\epsilon > 0$, $T_d(k_2, \epsilon) \le T_d(k_1, \epsilon)$ when

$$\frac{T_d(1, \epsilon)}{h} \ge \frac{(c+1)k_1 k_2}{r_d k_2 - k_1} \quad \text{and} \quad r_d > \frac{k_1}{k_2}, \qquad 1 < k_1 < k_2 \quad (4.12)$$

where $r_d = \mathrm{MINT}_b(\epsilon)/T_d(1, \epsilon)$. Similarly, $T_b(k_1, \epsilon)/T_b(k_2, \epsilon) > k_2/k_1$ when

$$(T_d(1, \epsilon) - \mathrm{MINT}_b(\epsilon)) > (k_2 - 1 - (c+1)(k_1 - 1)h), \qquad 1 < k_1 < k_2. \quad (4.13)$$

[2]The proof is similar to that of Corollaries 4.1 and 4.3.

When $k_1 = 1$ and $k_2 = k$, the corresponding necessary condition becomes

$$(T_d(1, \epsilon) - \mathrm{MINT}_b(\epsilon)) > (k - 1). \quad (4.14)$$

If all feasible solutions are located at the bottommost level of the OR-tree and the Serial/Parallel Incumbent-Update strategy is used, then a weaker sufficient condition to eliminate detrimental anomalies can be derived from Corollary 3.2. That is,

$$\frac{T_d(1, \epsilon)}{h} > \frac{(c+1)k_1 k_2}{k_2 - k_1}. \quad (4.15)$$

In this case, the necessary condition to allow acceleration anomalies is the same as that stated in (4.5).

Corollary 4.7: Suppose that a parallel breadth-first search satisfies the assumptions of Theorem 3.5 and that the Serial/Parallel Incumbent-Update Strategy is used, then $T_r(k_2, \epsilon) \le T_r(k_1, \epsilon)$, and the necessary condition to allow acceleration anomalies is the same as (4.6).

Proof: If the Serial/Parallel Incumbent-Update strategy is used, then Theorem 4.5 holds regardless of the value of ϵ.

4) Coping with Parallel-to-Parallel Anomalies Under Dominance Tests: When dominance tests are considered, the above corollaries are no longer valid. Figure 4(b) shows an example in which none of the necessary conditions discussed previously is met, but the acceleration anomaly that $T_b(2, 0)/T_b(3, 0) > 3/2$ still occurs. The reason for the anomaly is that $P_i \boldsymbol{D} P_j$ does not imply the dominance of P_i over the descendants of P_j. For instance, in a 0/1 knapsack problem, $P_i \boldsymbol{D} P_j$ 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 descendants of P_j, their weights may be greater than that of P_i, and 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, then it is necessary 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 this strategy, 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 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. These conditions are loose and are satisfied in most cases.

D. Multiple Subproblem Lists

When there are multiple subproblem lists, one for each processor, a node with the minimum heuristic value is selected from each local list 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 the processors. Hence when dominance tests are inactive ($\boldsymbol{D} = \boldsymbol{I}$), all the theorems and corollaries derived in Sections III and IV 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, then the behavior is similar to that of a centralized list. On the other hand, if dominance tests are performed for subproblems within each local list, then it is possible that $P_i \boldsymbol{D} P_j$ 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 Fig. 4(d), an acceleration anomaly may happen because P_2 is eliminated in the serial case and not terminated in the parallel case, hence T_3 is expanded in the serial case and pruned in the parallel case. Note that this acceleration anomaly will not appear in case of a single subproblem list.

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

V. 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 when all subproblems can fit in the main memory [34]. However, anomalies have been found in parallel searches. This is illustrated in Fig. 5 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. Analogous to the proof of corollaries in Section IV, we can derive for the same problem to be solved the sufficient conditions that assure $T_b(k, 0) < T_r(k, 0)$ and $T_b(k, 0) < T_d(k, 0)$.

$$(T_r(1, 0) - T_b(1, 0)) > (k - 1)(h - 1) \quad (5.1)$$

and

$$(T_d(1, 0) - T_b(1, 0)) > (k - 1)(h - 1). \quad (5.2)$$

Equation (5.2) is valid if all feasible solutions are located in the bottommost level of the OR-tree. In general, if I_d, the number of imperfect iterations in a parallel depth-first search, is greater than $(k - 1)h$, then a parallel best-first search will run faster than a parallel depth-first search. Since the average number of essential nodes in an imperfect iteration is almost the same in both strategies, hence $T_b(k, 0) < T_d(k, 0)$ when $I_d > h$ [22].

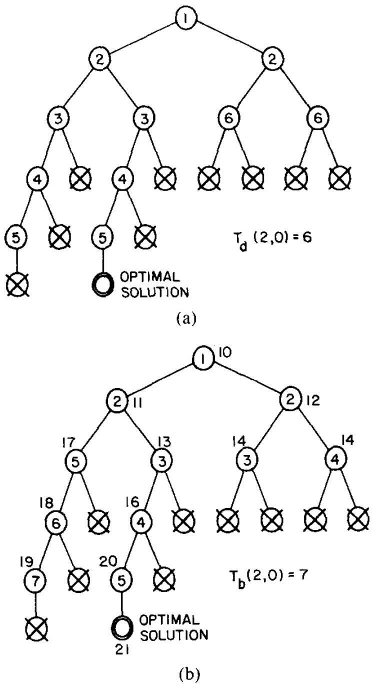

Fig. 5. Anomaly in performance between depth-first and best-first searches. (a) Parallel depth-first search with two processors (number in each node is the selection order). (b) Parallel best-first search with two processors (number in each node is the selection order, number outside each node is the lower bound).

Simulation results have demonstrated that the number of imperfect iterations in a depth-first search is usually larger than that in a best-first search, especially when the number of processors used is relatively large. Therefore, the speedup in the depth-first search drops quickly. In Case 2 of Table I, when more than two processors are used in the depth-first search, the speedup is less than $0.9k$. In contrast, in a best-first search solving the same problem, a $0.9k$ speedup is attained when 32 processors are used.

From the viewpoint of coping with anomalies, a breadth-first search is a conservative 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 a superlinear speedup but suffers from the risk of detrimental anomalies when approximate solutions are sought. For best-first searches, serial-to-parallel detrimental anomalies can be avoided, while acceleration anomalies may occur even if an approximate solution is sought. Generally, 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 the large number of active nodes, the total time, including time spent on 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, a best-first search may not always be desirable. Simulations have shown that the best OR-tree search strategy depends on the accuracy of the problem-dependent lower-bound function [33]. 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. When the accuracy of the lower-bound function is moderate, a best-first search gives a better performance. In this case, either a good memory management system [35] is necessary to support the large memory space required, or a modified best-first strategy, such as the IDA* [13], is applied.

In Fig. 6, 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. As a result, the performance of using the depth-first search is close to that of using the best-first search. (On the average, $T_d(1, 0) = 15\,197$, and $T_b(1, 0) = 15\,180$.) The speedups in best-first searches are a little larger than those of depth-first searches. When the number of processors are very large such that nodes in each level of the OR-tree can be expanded simultaneously, then $T_d(k, 0) = T_b(k, 0) = h$, where h is the height of the tree. Therefore, the two curves on speedups will coincide eventuately.

Finally, we compare the space requirements between depth-first and best-first searches. In a serial search, the space required by a best-first strategy is usually more than that required by a depth-first strategy. Somewhat surprisingly, 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 a near-linear speedup cannot be kept. In contrast, the space required by a parallel depth-first search is almost proportional to the number of processors (Fig. 6). Note that the space efficiency is problem-dependent. For vertex-cover problems, the space required by a parallel best-first search is almost constant for the entire range of the number of processors.

VI. Conclusions

In this paper, 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 in order 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 $T_b(1, 0)/h$, where

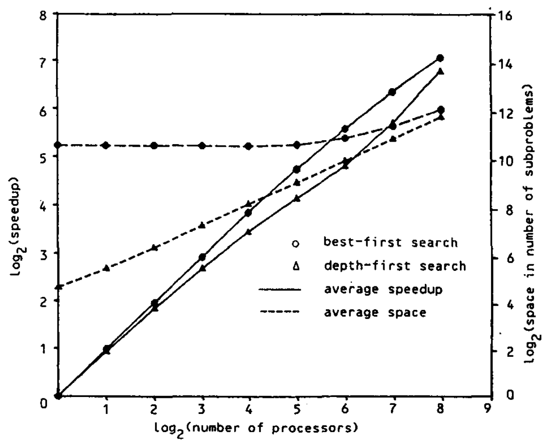

Fig. 6. Average speedup and space requirements of parallel branch-and-bound algorithms for ten knapsack problems with 35 objects.

$T_b(1, 0)$ is the number of iterations in a sequential optimal best-first search, and h is the maximum height of the tree. To guarantee a near-linear speedup, the number of processors must be much less than $T_b(1, 0)/h$. For depth-first and breadth-first searches, the speedups are related to the number of incumbents obtained during the search in addition to the problem complexity. Since the performance bounds on best-first searches are tighter than those on depth-first and breadth-first searches, the range within which a near-linear speedup is maintained is usually larger for best-first searches.

The anomalous behavior of parallel OR-tree searches have been studied thoroughly. Anomalies are caused by a combination of the following reasons: a) there are multiple solution nodes; b) the heuristic function is ambiguous; 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) the feasible solutions are not generated in the same order when different number of processors are used. The existence of a combination of these conditions causes the tree to be searched in a different order when a different number of processors is used. We have analytically investigated the conditions to eliminate detrimental anomalies and to preserve acceleration anomalies with respect to different search strategies. A summary of the results proved in this paper are shown in Table III.

A best-first search is found to be a robust search strategy in the sense of the large range of the number of processors within which a linear speedup is achieved. However, the best OR-tree search strategy depends on the accuracy of the problem-dependent lower-bound function. A best-first search is more suitable for parallel processing when the accuracy of the lower-bound function is moderate and the memory-management scheme is efficient. On the other hand, a depth-first search is more efficient when either the lower-bound function is very accurate in pruning unnecessary searches, or the heuristic function is inaccurate in guiding the search.

TABLE III
CONDITIONS TO COPE WITH ANOMALIES IN PARALLEL OR-TREE SEARCHES

Allow. function	Dom. rel.	Search strategy	Suff. cond. to elim. detrimental anom.		Nec. cond. to allow acceleration anom.	
			(1 vs. k)	(k_1 vs. k_2)	(1 vs. k)	(k_1 vs. k_2)
ε=0	D=I	bfs	Eq's 2.6, 2.7	Cor. 4.1	Th. 4.4	Cor. 4.3
		dfs		Cor. 4.2		Cor. 4.4
		brfs		Th. 4.5		Eq. 4.6
	D≠I	all	Th. 4.1	anom.		exist
ε>0	D=I	bfs	Eq's 2.6, 2.7	Cor. 4.5	Eq. 4.11	Cor. 4.5
		dfs	Th. 4.3	Cor. 4.6	Eq. 4.14	Cor. 4.6
		brfs	Th. 3.5	Cor. 4.7	Cor. 4.7	Cor. 4.7
	D≠I	bfs	Th. 4.2	anom.	exist	exist
		dfs	Th. 4.3			
		brfs	Th. 4.3			

Keys		
	bfs:	best-first search
	dfs:	depth-first search
	brfs:	breadth-first search
	anom.:	when multiple subproblem lists and local dominance tests are used, sufficient conditions are impractical
	exist:	necessary conditions are too loose

REFERENCES

[1] G. Baudet, "On the branching factor of the alpha-beta pruning algorithm," *Artificial Intell.*, vol. 10, no. 2, pp. 173-199, 1978.

[2] A. Ciepielewski and S. Haridi, "Execution of Bagof on the OR-parallel token machine," in *Proc. Int. Conf. Fifth Generation Computer Systems*, ICOT and North-Holland, 1984, pp. 551-560.

[3] R. W. Conway, W. L. Maxwell, and L. W. Miller, *Theory of Scheduling*. Reading, MA: Addison-Wesley, 1967.

[4] R. A. Finkel and J. P. Fishburn, "Parallelism in alpha-beta search," *Artificial Intell.*, vol. 19, no. 1, pp. 89-106, 1982.

[5] C. L. Forgy, A. Gupta, A. Newell, and R. Wedig, "Initial assessment of architectures for production systems," in *Proc. Nat. Conf. Artificial Intelligence*, AAAI, Aug. 1984, pp. 116-120.

[6] T. Ibaraki, "Theoretical comparisons of search strategies in branch-and-bound algorithms," *Int. J. Comput. Inform. Sci.*, vol. 5, no. 4, pp. 315-343, 1976.

[7] —, "Computational efficiency of approximate branch-and-bound algorithms," *Math. Oper. Res.*, vol. 1, no. 3, pp. 287-298, 1976.

[8] —, "On the computational efficiency of branch-and-bound algorithms," *J. Oper. Res. Soc. Japan*, vol. 20, no. 1, pp. 16-35, Mar. 1977.

[9] —, "The power of dominance relations in branch-and-bound algorithms," *J. ACM*, vol. 24, no. 2, pp. 264-279, Apr. 1977.

[10] M. Imai and T. Fukumura, "A parallelized branch-and-bound algorithm implementation and efficiency," *Syst., Comput., Contr.*, vol. 10, no. 3, pp. 62-70, June 1979.

[11] D. E. Knuth and R. W. Moore, "An analysis of alpha-beta pruning," *Artificial Intell.*, vol. 6, no. 4, pp. 293-326, 1975.

[12] W. Kohler and K. Steiglitz, "Characterization and theoretical comparison of branch-and-bound algorithms for permutation problems," *J. ACM*, vol. 21, no. 1, pp. 140-156, 1974.

[13] R. E. Korf, "Depth-first iterative deepening: An optimal admissible tree search," *Artificial Intell.*, vol. 27, pp. 97-109, 1985.

[14] —, "Real-time heuristic search: First results," in *Proc. Sixth Nat. Conf. Artificial Intelligence*, Seattle, WA, June 1987, pp. 133-138.

[15] R. Kowalski, *Logic for Problem Solving*. Amsterdam, The Netherlands: North-Holland, 1979.

[16] V. Kumar and L. N. Kanal, "A general branch and bound formulation for understanding and synthesizing and/or tree search procedures," *Artificial Intell.*, vol. 21, no. 1-2, pp. 179-198, 1983.

[17] V. Kumar and V. N. Rao, "Parallel depth-first search on multiprocessors Part II: Analysis," *Int. J. Parallel Programming*, vol. 16, no. 6, 1987.

[18] T. H. Lai and S. Sahni, "Anomalies in parallel branch-and-bound algorithms," *Commun. ACM*, vol. 27, no. 6, pp. 594-602, June 1984.

[19] T. H. Lai and A. Sprague, "Performance of parallel branch-and-bound algorithms," in *Proc. Int. Conf. Parallel Processing*, IEEE, Aug. 1985, pp. 194-201.

[20] G. J. Li and B. W. Wah, "Computational efficiency of parallel approximate branch-and-bound algorithms," in *Proc. Int. Conf. Parallel Processing*, IEEE, Aug. 1984, pp. 473-480.

[21] —, "Computational efficiency of parallel approximate branch-and-bound algorithms," Purdue Univ., West Lafayette, IN, Tech. Rep. TR-EE-84-6, Mar. 1984.

[22] G. J. Li, "Parallel processing of combinatorial search problems," Ph.D. dissertation, School Elec. Eng., Purdue Univ., West Lafayette, IN, Dec. 1985.

[23] G. J. Li and B. W. Wah, "Coping with anomalies in parallel branch-and-bound algorithms," *IEEE Trans. Comput.*, vol. C-34, no. 6, pp. 568-573, June 1986.

[24] J. Mohan, "Experience with two parallel programs solving the traveling-salesman problem," in *Proc. 1983 Int. Conf. Parallel Processing*, 1983, pp. 191-193.

[25] T. Morin and R. Marsten, "Branch-and-bound strategies for dynamic programming," *Oper. Res.*, vol. 24, no. 4, pp. 611-627, 1976.

[26] D. S. Nau, V. Kumar, and L. Kanal, "General branch and bound, and its relation to A* and AO*," *Artificial Intell.*, vol. 23, pp. 29-58, 1984.

[27] J. Pearl, "The solution for the branching factor of the alpha-beta pruning algorithm and its optimality," *Commun. ACM*, vol. 25, no. 8, pp. 559-564, 1982.

[28] V. N. Rao and V. Kumar, "Superlinear speedup in parallel state-space search," Artificial Intell. Lab., Univ. Texas, Austin, TX, Tech. Rep. AI88-80, 1988.

[29] S. Sahni, "Algorithms for scheduling independent tasks," *J. ACM*, vol. 23, no. 1, pp. 116-127, Jan. 1976.

[30] H. A. Simon and J. B. Kadane, "Optimal problem-solving search: All-or-none solutions," *Artificial Intell.*, vol. 6, no. 3, pp. 235-247, 1975.

[31] D. R. Smith, "Random trees and the analysis of branch-and-bound procedures," *J. ACM*, vol. 31, no. 1, pp. 163-188, 1984.

[32] B. W. Wah and Y. W. Ma, "MANIP—A multicomputer architecture for solving combinatorial extremum problems," *IEEE Trans. Comput.*, vol. C-33, no. 5, pp. 377-390, May 1984.

[33] B. W. Wah and C. F. Yu, "Stochastic modeling of branch-and-bound algorithms with best-first search," *IEEE Trans. Software Eng.*, vol. SE-11, no. 9, pp. 922-934, Sept. 1985.

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

[35] C. F. Yu and B. W. Wah, "Efficient branch-and-bound algorithms on a two-level memory system," *IEEE Trans. Software Eng.*, vol. SE-14, no. 9, pp. 1342-1356, Sept. 1988.

Guo-Jie Li graduated from 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, Academia Sinica, in 1981, and the Ph.D. degree in electrical engineering in 1985 from Purdue University, West Lafayette, IN. Between 1985 and 1986, he was a postdoctoral research associate in the Coordinated Science Laboratory of the University of Illinois at Urbana-Champaign.

He is an Associate Professor at the Institute of Computing Technology, Academia Sinica. His research interests include parallel processing, artificial intelligence, theory of algorithms, and computer architecture. He has published over 40 papers and has coedited the book, *Tutorial: Computers for Artificial Intelligence Applications* (IEEE Computer Society Press, 1986). He has also written chapters in several published books.

Dr. Li is an editor of the *Journal of Computer Science and Technology* and the *Journal of Pattern Recognition and Artificial Intelligence*.

Benjamin W. Wah (S'74-M'79-SM'85) received the Ph.D. degree in computer science from the University of California, Berkeley, in 1979.

He was on the faculty of the School of Electrical Engineering at Purdue University, West Lafayette, IN, between 1979 and 1985. He is now a Professor in the Department of Electrical and Computer Engineering and the Coordinated Science Laboratory of the University of Illinois at Urbana-Champaign, Urbana, IL. Between 1988 and 1989, he served as a Program Director of the Microelectronic Systems Architecture Program, National Science Foundation. He has published extensively in the areas of computer architecture, parallel processing, artificial intelligence, distributed databases, and computer networks. For his contributions to research, teaching, and service, he has been selected as a University Scholar of the University of Illinois in 1989.

Dr. Wah is the Associate Editor-in-Chief of the IEEE TRANSACTIONS ON KNOWLEDGE AND DATA ENGINEERING, an area editor of the *Journal of Parallel and Distributed Computing*, and an editor of *Information Sciences*. He serves as a member of the Governing Board of the IEEE Computer Society and a program evaluator for ABET (computer Engineering) and CSAC (computer science). He is the Co-Program Chairman of the 1990 International Conference on Parallel Processing. Previously, he served as an editor of the IEEE TRANSACTIONS ON SOFTWARE ENGINEERING and a Distinguished Visitor of the IEEE Computer Society.