Parallel Processing of Discrete Optimization Problems *

Grama Y. Ananth and Vipin Kumar
Department of Computer Science,
University of Minnesota
Minneapolis, MN 55455
ananth@cs.umn.edu and *kumar@cs.umn.edu*
and
Panos Pardalos
Department of Industrial and Systems Engineering
University of Florida
Gainesville, FL 32611
pardalos@math.ufl.edu

Abstract

Discrete optimization problems (DOPs) arise in various applications such as planning, scheduling, computer aided design, robotics, game playing and constraint directed reasoning. Often, a DOP is formulated in terms of finding a (minimum cost) solution path in a graph from an initial node to a goal node and solved by graph/tree search methods such as branch-and-bound and dynamic programming. Availability of parallel computers has created substantial interest in exploring the use of parallel processing for solving discrete optimization problems. This article provides an overview of parallel search algorithms for solving discrete optimization problems.

1 Introduction.

Discrete optimization problems (DOPs) arise in various applications such as planning, scheduling, computer aided design, robotics, game playing and constraint directed reasoning. Formally, a DOP can be stated as follows : *Given a finite discrete set X and a function $f(x)$ defined on the elements of X, find an optimal element x_{opt}, such that,* $f(x_{opt}) = min\{f(x)/x \epsilon X\}$. In

*This work was supported by IST/SDIO through the Army Research Office grant # 28408-MA-SDI to the University of Minnesota and by the Army High Performance Computing Research Center at the University of Minnesota.

certain problems, the aim is to find any member of a solution set $S \subset X$. These problems can also be easily stated in the above format by making $f(x) = 0$ for all $x \epsilon S$, and $f(x) = 1$ for all other elements of X.

In most problems of practical interest, the set X is quite large. Consequently, exhaustive enumeration of elements in X to determine x_{opt} is not feasible. Often, elements of X can be viewed as paths in graphs/trees, the cost function can be defined in terms of the cost of the arcs, and the DOP can be formulated in terms of finding a (minimum cost) solution path in the graph from an initial node to a goal node. Branch and bound and dynamic programming methods use the structure of these graphs to solve DOPs without searching the set X exhaustively [36].

Given that DOP is an NP-hard problem [19], one may argue that there is no point in applying parallel processing to these problems, as the worst-case run time can never be reduced to a polynomial unless we have an exponential number of processors. However, the average time complexity of heuristic search algorithms for many problems is polynomial [74, 82]. Also, there are some heuristic search algorithms which find suboptimal solutions in polynomial time (e.g., for certain problems, approximate branch-and-bound algorithms are known to run in polynomial time[81]). In these cases, parallel processing can significantly increase the size of solvable problems. Some applications using search algorithms (e.g. robot motion planning, task scheduling) require real time solutions. For these applications, parallel processing is perhaps the only way to obtain acceptable performance. For some problems, optimal solutions are highly desirable, and can be obtained for moderate sized instances in a reasonable amount of time using parallel search techniques (e.g. VLSI floor-plan optimization [5]).

Parallel computers containing thousands of processing elements are now commercially available. The cost of these machines is similar to that of large mainframes, but they offer significantly more raw computing power. Due to advances in VLSI technology and economy of scale, the cost of these machines is expected to go down drastically over the next decade. It may be possible to construct computers comprising of thousands to millions of processing elements at costs ranging from those of high-end workstations to large mainframes. This technology has created substantial interest in exploring the use of parallel processing for search based applications [1, 2, 14, 32, 33, 35, 63, 66, 68, 78, 79].

This article provides a survey of parallel algorithms for solving DOPs. Section 2 reviews serial algorithms for solving DOPs. Section 3 discusses parallel formulations of depth-first and best-first search algorithms and dynamic programming. Section 4 discusses parallel formulations and applications of 0/1 integer programming. Section 5 discusses parallel formulations and applications of the quadratic assignment problem. Section 6 contains concluding remarks.

2 Sequential Algorithms for Solving Discrete Optimization Problems.

Here we provide a brief overview of sequential search algorithms. For detailed descriptions, see [54, 62, 26].

2.1 Depth-First Search Algorithms.

Depth-first search is a name commonly used for various search techniques for solving DOPs that perform search as follows. The search begins by expanding the initial node, i.e., by generating its successors. At each subsequent step, one of the most recently generated nodes is expanded. (In some problems, heuristic information is used to order the successors of an expanded node. This determines the order in which these successors will be visited by the depth-first search method.) If this most recently generated node does not have any successors (or if it can be determined that the node will not lead to any solutions), then backtracking is done, and a most recently generated node from the remaining (as yet unexpanded) nodes is selected for expansion. A major advantage of depth-first search is that its storage requirement is linear in the depth of the search space being searched. Following are three search methods that use the depth-first search strategy.

Simple Backtracking is a depth-first search method that terminates on finding the first solution. This solution is obviously not guaranteed to be the minimum-cost solution. In the simple version, no heuristic information is used for ordering the successors of an expanded node (which happen to be at the same depth). In its variant, "Ordered Backtracking", heuristics are used for ordering the successors of an expanded node.

Depth-First Branch-and-Bound (DFBB) is a DFS algorithm which searches the whole search space exhaustively; i.e., search continues even after finding the solution path. Whenever a new solution path is found, the current best solution path is updated. Whenever an inferior partial solution path (i.e., a partial solution path whose extensions are guaranteed to be worse than the current best solution path) is generated, it is eliminated.

Iterative Deepening A* (IDA*) performs repeated cost-bounded DFS over the search space. In each iteration, IDA* keeps on expanding nodes in a depth-first fashion until the total cost of the selected node reaches a given threshold which is increased for each successive iteration. The algorithm continues until a goal node is selected for expansion. It might appear that IDA* performs a lot of redundant work in successive iterations. But for many problems of interest, the redundant work is minimal and the algorithm finds an optimal solution [30].

2.2 Best-First Search.

Best-first search techniques use heuristics to direct search to spaces which are more likely to yield solutions. A*/Best-first branch-and-bound search is a commonly used best-first search technique. A* makes use of a heuristic evaluation function, f, defined over the nodes of the search space. For each node n, $f(n)$ gives an estimate of the cost of the optimal solution path passing through node n.

A* maintains a list of nodes called "OPEN" which holds the nodes which have been generated but not expanded. This list is sorted on the basis of the f values of the nodes. The nodes with the lowest f values are expanded first. The main drawback of this scheme is that it runs out of memory very fast since its memory requirement is linear in the size of the search space explored.

2.3 Dynamic Programming

Dynamic programming (DP) is a powerful technique used for solving DOPs. Problems which can be solved efficiently using DP are characterized by the *Principle of Optimality* defined by Bellman [8]. This principle states that an optimal sequence of decisions has the property that irrespective of the initial state and decision, the remaining decisions must constitute an optimal decision sequence with respect to the state resulting from the first decision. Many different formulations of DP have been presented [36, 27, 25].

The essence of many DP algorithms lies in computing solutions to the smallest subproblems and storing the results and using them to compute solution to bigger problems. Thus the solution to the original problem is constructed in a bottom-up fashion [4]. We will illustrate this algorithm using the classical example of finding the multiplication sequence of a sequence of matrices so that the total number of operations (individual multiply / add operations) is minimized [4]. Let the given matrices be represented as $M_1, M_2, ..., M_n$. Also let m_{ij} denote the number of operations required to multiply matrices M_i through M_j in sequence. The DP algorithm proceeds by calculating m_{ij}'s as $\min\{(m_{ik} + m_{k+1,j} + r_{i-1}r_kr_j) \mid\mid i \leq k < j\ \}$, where r_{i-1}, r_k are the dimensions of the product of matrices M_i through M_k, and r_k, r_j are the dimensions of the product of matrices M_{k+1} through M_j. The first term, m_{ik}, represents the number of operations required for multiplying matrices i through k, the second term, $m_{k+1,j}$ represents the number of operations for multiplying matrices $k+1$ through j, and the third term, $r_{i-1}r_kr_j$, represents the number of operations required in computing the product of these two matrices. Clearly, if m_{ik} and $m_{k+1,j}$ have been precomputed, m_{ij} can be computed easily. This process is continued until m_{1n} is computed. The computation is illustrated in Figure 1. Note that the total number of distinct multiplication sequences is exponential in n. However, the DP algorithm has a complexity of $O(n^3)$.

3 Parallel Formulations

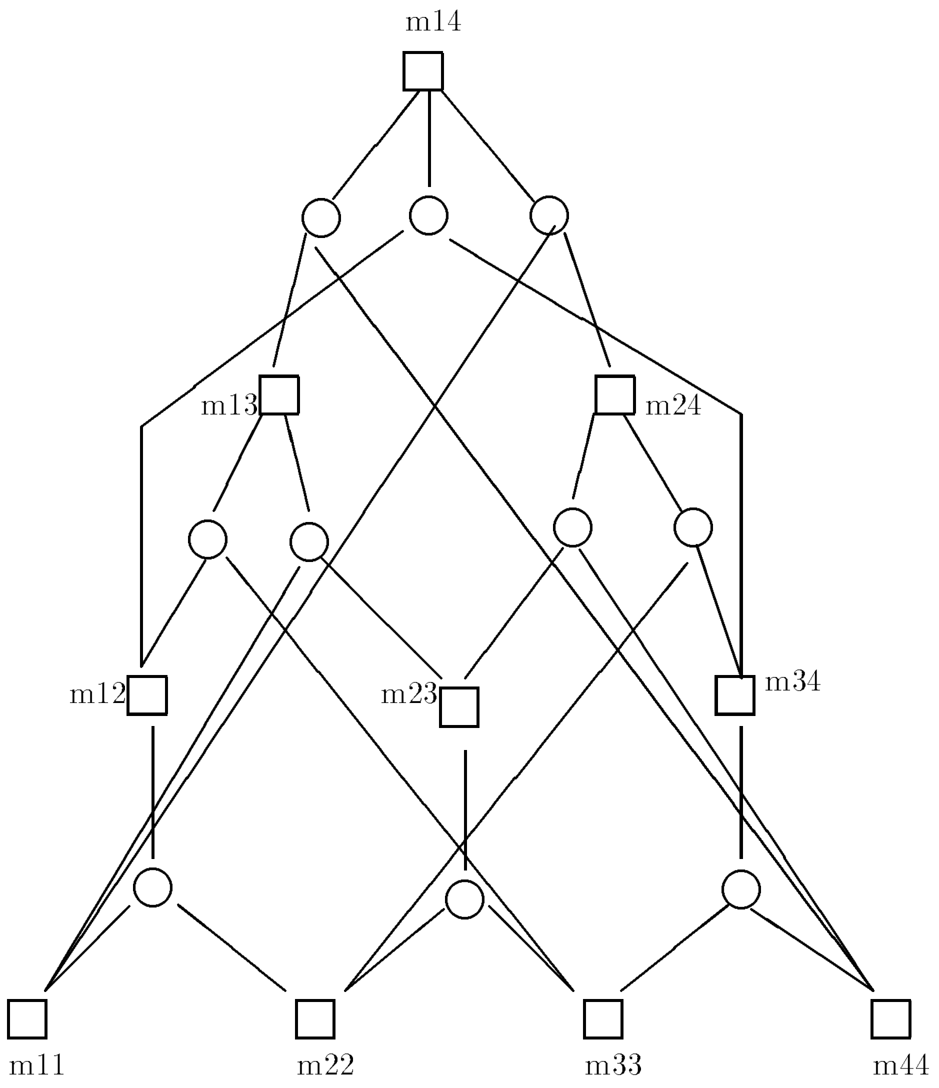
3.1 Parallel Depth-First-Search Algorithms.

The state space tree to be searched by DFS-based algorithms such as backtracking, IDA*, DFBB etc. can easily be partitioned into smaller parts. These parts can be searched by different processors. Searching these parts requires no (e.g., in backtracking and IDA*) or minimal (e.g., in DFBB) communication. However, for most applications, state-space trees generated by DFS tend to be highly irregular, and any static allocation of subtrees to processors is bound to result in significant load imbalance among processors. The core of any parallel formulation of DFS algorithms is thus a dynamic load balancing technique which minimizes communication and processor idling. A number of load distribution techniques have been developed for parallel DFS [72, 73, 34, 39, 48, 67, 28].

A general method for parallelizing DFS is presented in [34, 39]. In this formulation, each processor searches a disjoint part of the search space in a depth-first fashion. When a processor finishes searching its part of the search space, it tries to get an unsearched part of the search space from other processors. When a goal is found, all of them quit. If the search space is finite and has no solutions, then eventually all the processors would run out of work, and the (parallel) search will terminate.

Since each processor searches the space in a depth-first manner, the (part of) state space to be searched can be efficiently represented by a stack. The depth of the stack is the depth of the node being explored currently. Each level of the stack keeps track of the untried alternatives at that level. Each processor maintains its own local stack on which it executes DFS. When the local stack is empty, it takes some of the untried alternatives of another processor's stack. For shared memory architectures [24], this operation can be performed by simply locking the other processor's stack and picking up a part of its search space. For distributed memory machines [24], this has to be accomplished using messages. The detailed schematic of this process is shown in Figure 2. A processor on running out of work selects a target processor for addressing a work request. On receiving a work request, a processor either responds with a part of its own work, or a reject message in case it does not have any work. This process continues until all processors go idle or a solution is found. In such a formulation, all search space is initially assigned to one processor and other processors are given null spaces. Subsequently, the search space is divided and distributed among various processors.

The selection of a target processor for a work request can be done in a number of ways. For



.

Circles indicate calculation of number of operations required in the multiplication of the matrices corresponding to the two incoming arcs.

Figure 1: Dynamic Programming algorithm for finding optimal multiplication sequence for a given ordered set of matrices.

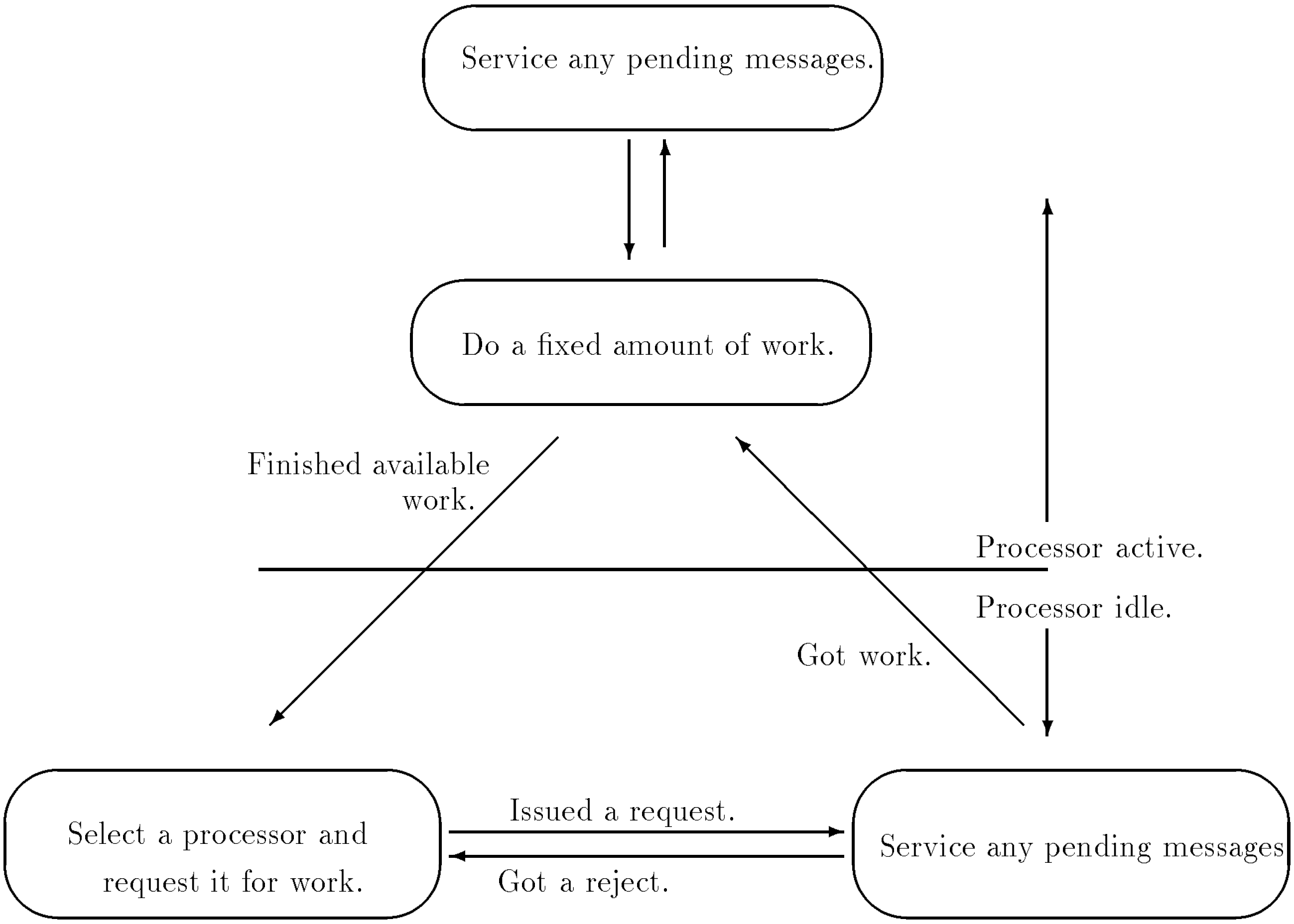


Figure 2: State diagram describing the generic parallel DFS formulation for distributed memory architectures.

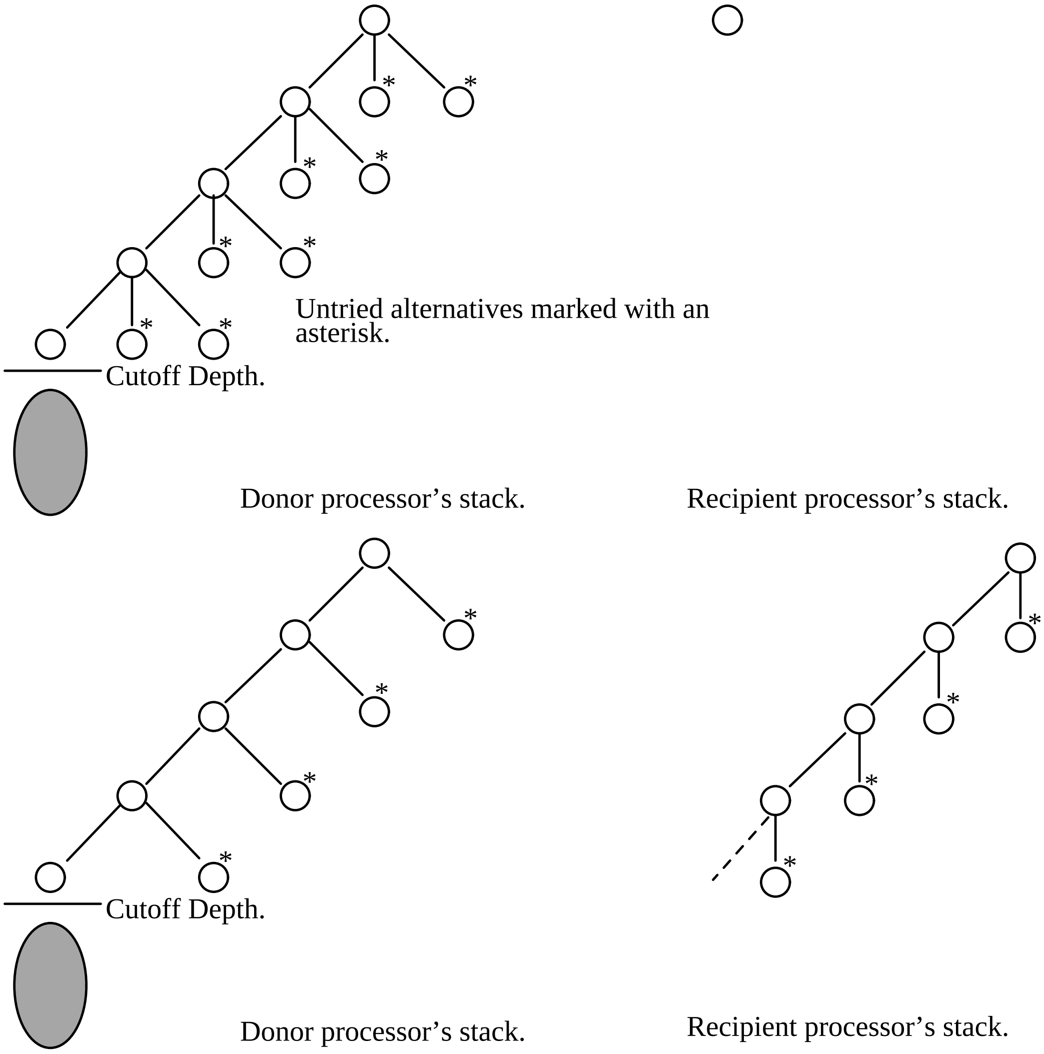


Figure 3: Illustration of the stack splitting mechanism.

example, two techniques discussed in [34] are *random polling* and *global round robin*. In random polling, a processor is selected at random and the work request is targeted to this processor. In global round robin, a global pointer is maintained at a designated processor. This pointer determines the target of a work request. Every time a work request is made, the pointer is read and incremented (modulo the number of processors). Though the global round robin scheme minimizes the total number of work requests for a wide class of problems, accessing the global pointer forms a bottleneck [34]. Consequently, when the number of processors increases, its performance degrades. On the other hand, random polling does not suffer from such a drawback. However, on machines that have hardware support for concurrent access to a global pointer (e.g., the hardware fetch and add [16]), the performance of the global round robin scheme would be better than random polling.

When a work transfer is made, work in the donor's stack is split into two stacks one of which is given to the requester. In other words, some of the nodes (i.e., alternatives) from the donor's stack are removed and added to the requester's stack. Figure 3 illustrates the process of splitting a stack into two parts. Intuitively it is ideal to split the stack into two equal pieces. If the work associated with either of the stacks is too small, the corresponding processor would become idle too soon.

In the above formulation of parallel DFS, a processor requests for work when it becomes idle. An alternate technique is to distribute parts of the search space as they are being generated. For example, every time the successors of a node are generated, they could be sent to selected processors. A number of schemes have been proposed which use such work distribution strategies [67, 18, 17]. A major drawback of these schemes is that an interprocessor communication is required for each node generation. For a detailed discussion on the relative merits of these schemes, see [34].

Parallel Formulations of DFBB. Parallel formulations of DFBB are similar to those of DFS. The parallel formulation of DFS described above can be applied to DFBB with one minor modification. Now we need to keep all the processors informed of the current best solution path. On a shared memory architecture, this can be done by maintaining a global best solution path. On a distributed memory architecture, this can be done by allowing each processor to maintain the current best solution path known to it. Whenever a processor finds a solution path better than the current best known, it broadcasts it to all the other processors which update (if necessary) their current best solution path. Note that if a processor's current best solution path is worse than the global best solution path, then it only affects the efficiency of the search but not its correctness. Parallel formulations of DFBB have been shown to yield linearly increasing speedups for many problems and architectures [5, 6].

Parallel Formulations of IDA*. There are two approaches to parallelizing IDA*. In one approach, different processors work on different iterations of IDA* independently [31, 35]. This approach is not very suitable for finding optimal solutions. This is because a solution found by a node on a particular iteration is not provably optimal until all the other processors have also exhausted search space until that iteration and have not found a better solution. Another approach is to execute each iteration of IDA* via parallel DFS [38, 39, 51]. Since all processors work with the same cost bound, each processor stores this value locally and performs DFS on its own search space.

3.1.1 Speedup Anomalies in Backtracking / DFS.

In parallel DFS, the speedup can differ greatly from one execution to another, as the actual parts of the search space examined by different processors are determined dynamically, and can be different for different executions. Hence, for some execution sequences the parallel version may find a solution by visiting fewer nodes than the sequential version thereby giving superlinear speedup, whereas for others it may find a solution only after visiting more nodes resulting in sublinear speedup. It may appear that on the average the speedup would be either linear or sublinear. This phenomenon of speedup greater than P on P processors in isolated executions of parallel DFS has been reported by many researchers [32, 40, 44, 49, 53] for a variety of problems and is referred to by the term speedup anomaly.

The average speedup in parallel DFS for two different types of models has been analyzed in [52, 69]. In the first model, no heuristic information is available to order the successors of a node. For this model, analysis shows that on the average, the speedup obtained is (i) linear when distribution of solutions is uniform, and (ii) superlinear when distribution of solutions is non-uniform. This model is validated by experiments on synthetic state-space trees modeling the hackers problem[75], the 15-puzzle problem and the N-Queens problem[23]. (In these experiments, serial and parallel DFS do not use any heuristic ordering, and select successors arbitrarily.) The basic reason for this phenomenon is that parallel search can invest resources into multiple regions of the search frontier concurrently. When the solution density in different regions of the search frontier is nonuniform and these nonuniformities are not known a priori, then sequential search has equal chance of searching a low density region or a high density region. On the contrary, parallel search can search all regions at the same time, ensuring faster success rate.

In the second model, the search tree contains a small number of solutions and a strong heuristic is available that directs search to regions that contain solutions. There is, however, some probability that the heuristic makes an error and directs search to regions containing no solutions. The work distribution method used for partitioning the tree does not use any heuristic information. However, each processor searches its own space using the heuristic. For this model, analysis shows that the average speedup is at least linear. This may appear surprising since at any given time most of the processors will be searching spaces that are considered unpromising by the heuristic. An intuitive explanation is that for this model, parallel DFS performs much better than serial DFS when the heuristic makes an error, and thus compensates for the lost performance in the case in which the heuristic is correct. Results from this model have been verified on the parallel formulation of a DFS algorithm, called PODEM, which uses very powerful heuristics to order the search tree [7].

3.2 Parallel Best-First Search.

A number of researchers have investigated parallel formulations of A*/B&B algorithms [35, 37, 43, 48, 64, 70, 77, 80]. An important component of A*/B&B algorithms is the priority queue which is used to maintain the "frontier" (*i.e.*, unexpanded) nodes of the search graph in a heuristic order. In the sequential A*/B&B algorithm, in each cycle a most promising node from the priority queue is removed and expanded, and the newly generated nodes are added to the priority queue.

In most parallel formulations of A*, different processors concurrently expand different frontier nodes. Conceptually, these formulations differ in the data structures used to implement the priority queue of the A* algorithm. In some schemes, a global priority queue is maintained at a designated processor. In each node expansion cycle, a processor picks up the current best node from this queue, expands it and inserts the successors back into the queue. Clearly, accessing such a queue would become a bottleneck for parallel algorithms. Consequently, schemes using global priority queues are suited only for small number of processors. One way to avoid contention due to global priority queues is to let each processor have its own local queue. Initially, the search space is statically divided and given to different processors (by expanding some nodes and distributing them to the local queues of different processors). Now all the processors select and expand nodes simultaneously without causing contention on the shared queue as before. In the absence of any communication between individual processors, it is possible that some processors may work on a good part of the search space, while others may work on bad parts that would have been pruned by sequential search. This would lead to redundant node expansions and poor speedups. Various schemes can be developed to trade off redundant node expansions, communication overheads and contention to optimize performance.

A commonly used technique for implementing distributed queues is to hash every node generated to a unique processor. As shown in [46] this method ensures a good distribution of promising nodes among all processors. This technique also allows checking for duplication of nodes which is required for graph search. For instance, if we are searching for an optimal path through a graph, it might be possible to reach a given vertex in a graph using two different paths. Thus, on reaching any vertex, we first check if it has been reached before from any other path in the graph to avoid extra search. This is accomplished by hashing the node corresponding to the vertex to a unique processor where node duplications can be checked. Note that node duplications need to be checked only for graph search and not for tree search problems. Hashing techniques themselves cause performance degradation as each node generation results in a corresponding communication cycle.

3.3 Parallel Dynamic Programming.

Parallel formulations of DP differ significantly for different serial DP algorithms. First we discuss parallel formulations of those DP algorithms in which subproblems can be statically organized into levels such that solution to a subproblem depends only upon the solutions to the subproblems at preceding levels. For example, consider the optimal matrix multiplication sequence problem discussed in Section 2.3. As seen in Figure 1, computation for a node depends only on the nodes at preceding levels. If there are I nodes at a level, we could assign I/p nodes to each of the p processors. Each processor computes the cost of the nodes assigned to it. This is followed by an all-to-all broadcast during which solution costs of all nodes at that level are made known to all processors. This completes the DP algorithm corresponding to one level. Since each processor has complete information about node costs at preceding levels, no communication is needed other than the all-to-all broadcast. This processor can use no more processors than the maximum number of nodes at any level. In many applications, it is possible to extract a greater degree of parallelism than the above formulation. The first technique uses multiple processors to compute the cost of a node (e.g., [25]). In the second technique, processing of nodes at different levels is pipelined. Thus, processing of a node at level i does not wait for all nodes at levels below i to be completed. Instead, it waits only for the nodes on which its own processing depends. Kung and Leiserson use this technique and present a parallel formulation of the DP algorithm for the matrix multiplication sequence problem which can use $O(n^2)$ processors and run in $O(n)$ time [22].

Not all DP algorithms can, however, be formulated as a multilevel bottom-up tree structure (e.g., see DP algorithms for solving the 0/1 Knapsack problem and the single source shortest path problem). Parallel formulations of these algorithms have to be specially designed. For example, Lee et. al. [42] use the divide-and-conquer strategy for parallelizing the DP algorithm for the 0/1 knapsack problem on a MIMD distributed memory computer.

4 Parallel Integer Linear Programming Algorithms

In the next two sections we discuss parallel algorithms for solving the general linear zero-one and the quadratic assignment problems. Integer linear programming problems appear in many applications and are solved, in general, by some branch and bound type algorithm [58], [65].

In this section, we describe some of the first attempts to parallelize these branch and bound algorithms for integer linear programs.

In [3] the 0-1 integer linear program

$$\min\ f(x) \ = \ c^T x$$

$$\text{s.t. } Ax \ \geq \ b,\ x \in \{0,\ 1\}^n, \tag{1}$$

is considered. We may assume that all cost coefficients c_i are nonnegative (if some $c_i < 0$ then we may replace x_i by $1 - x_i$). The branch and bound algorithm (using best-first search) for solving (6) is described by the following steps:

---

1. Let f_U be the incumbent which contains the best solution found during the search (initially f_U is ∞). The initial subproblem (all variables are free) is created. A list of active subproblems is then created and the initial subproblem is inserted into it.

2. Select a subproblem from the list of active subproblems whose lower bound is smallest.

3. A free variable x_k in the selected subproblem is chosen and is used to generate two new subproblems (corresponding to $x_k = 0$ and $x_k = 1$). The variable x_k is now fixed.

4. The lower bound of each subproblem is computed using

   $$f_L \ = \ \sum_{x_j \text{ fixed}} c_j x_j. \tag{2}$$

   The feasibility of each constraint is checked using the condition

   $$\sum_{x_j \text{ free}} \max(a_{ij},\ 0) \ \geq \ b_j \ - \ \sum_{x_j \text{ fixed}} a_{ij} x_j,\ \ i = 1, ..., m.$$

   Assigning the value of 0 to each free variable in a subproblem is referred to as the "lower bound completion". At this point the feasibility of the lower bound completion is checked using (7) which reduces to checking

   $$\sum_{x_j \text{ fixed}} a_{ij} x_j \ \geq \ b_i,\ \ i = 1, ..., m. \tag{3}$$

5. Delete a subproblem (prune) if any one of the following conditions holds:

   (a) $f_L \ \geq \ f_U$.

   (b) The subproblem is infeasible.

(c) There are no remaining free variables.

(d) The lower bound completion is feasible. In this case, the incumbent is replaced by the lower bound completion if $f_L < f_U$, and all subproblems in the list of active subproblems are deleted if $f_L \geq f_U$.
A subproblem that is not deleted is added to the list of active subproblems.

6. Repeat steps 2-5 until the list of active subproblems is empty. When the list is empty, the algorithm terminates. The optimal solution is the current incumbent.

The simplified "logical model" for the parallel execution of the branch and bound algorithm consists of:

1. A set of N processors.

2. Global data that consists of the list of active subproblems and the incumbent. The global data is accessible by all processors and it is assumed that no overhead is incurred by a processor when it accesses the global data.

3. The processors are synchronized into cycles, and each cycle consists of three steps:

   (a) Each subproblem selects a subproblem from the N subproblems whose lower bounds are the best among all those on the list of active subproblems.

   (b) Each processor (independently of the other processors) expands its subproblem and performs lower bound, feasibility, and termination tests on the newly generated subproblems.

   (c) The processors insert the newly created subproblems back on the list of active subproblems.

The processors continue to iterate until the list of active subproblems becomes empty. The algorithm then terminates and the solution is stored at the incumbent.

Abdelrahman and Mudge [3] propose two parallelization methods on a distributed memory multiprocessor. The first method maintains a centralized list of subproblems (using N slave processes) and a manager (master process). The master process maintains the global data, and the slave processes perform the operations necessary for the expansion of subproblems. The master process selects N subproblems from the list of active subproblems and assigns one subproblem to each slave process. The N subproblems selected have the best bounds among those subproblems in the list of active subproblems. Then each slave process expands its subproblem generating subproblems and computing the corresponding lower bounds. In addition, each slave process performs the tests regarding lower bound, feasibility and termination tests on the subproblems it generated. The results are sent back to the master process which inserts them in the list. The algorithm terminates when the list of active subproblems becomes empty and all the slave processes are idle.

The algorithm has the advantage of expanding subproblems whose bounds are best in the global sense (since subproblems that have smaller lower bounds are most likely to lead to solutions than others that have larger lower bounds). The main disadvantage of this approach is communication cost and the requirement of large memory to maintain the list of active subproblems.

The second method outperforms the first by distributing the list of subproblems and balancing the load among neighboring processors. When all neighboring processors are idle, the algorithm "guesses" to terminate (for details see [3]). The two methods are also capable of incorporating multiple search strategies. Computational results on an NCUBE/six multiprocessor are reported.

In [11] a methodology is proposed which uses a collection of workstations connected by an Ethernet network as a parallel processor for solving 0-1 linear problems. The algorithm is a based on the branch and cut approach and has been used to solve a set of test problems from [13].

5 A Parallel Algorithm for the Quadratic Assignment Problem

In this section we consider one of the most difficult discrete problems, the quadratic assignment problem, and discuss some details on solution techniques using parallel machines [45]. The quadratic assignment problem (QAP) is a mathematical model arising from many location problems in which the cost associated with allocating a facility at a certain location depends, not only on the distances from other facilities, but also on the interaction with other facilities. This model, first proposed by Koopmans and Beckmann in 1957 [29], can be stated as the following minimization problem

$$\min \sum_{i=1}^{n} \sum_{j=1}^{n} f_{ij} d_{p(i)p(j)},$$

$$\text{s.t. } p \in \Pi,$$

where n is a positive integer, $F = (f_{ij}) \in R^{n \times n}$, $D = (d_{ij}) \in R^{n \times n}$, and Π is the set of permutations of the set $N = \{1, ..., n\}$.

In the framework of location problems, the set N represents the set of n locations on which n facilities are to be allocated. For matrix F, the entries $f_{ij},\ i,j = 1,...,n$, represent the flows between facilities i and j. For matrix D, the entries $d_{ij},\ i,j = 1,...,n$, represents the distances between locations i and j. The goal then is to assign the facilities to the locations such that the total cost is minimal. Besides applications in facility allocation, this model can be applied in many other applications, including backboard wiring, machine scheduling, ordering interrelated data on a computer storage device, scheduling of economic lot sizes, and designing typewriter keyboard [45].

Next, we discuss a parallel exact algorithms for the QAP. Roucairol [12] first proposed a parallel branch-and-bound algorithm for solving the QAP and implemented the algorithm on a CRAY XMP/48. However her computational results were not very satisfactory and only problems of sizes ≤ 12 were solved. As a comparison, Pardalos and Crouse [57] proposed a more efficient branch-and-bound algorithm. The discussion in the rest of this section is based on [57].

The exact algorithm presented below is of the branch-and-bound type [57], [61]. The term "solution" and "permutation" are used interchangeably in our discussion.

The algorithm consists of three steps. In the first step, the algorithm computes an initial best known upper bound (BKUB) and sets up an initial branch-and-bound search tree, The initial best known upper bound is computed by using the heuristic algorithm described in [10]. Then the initial search tree consists of $n \times (n-1)$ nodes storing partial permutations p defined by

$$p(1) = i,\ p(2) = j,\ \text{for } i,\ j = 1, 2, \ldots, n \text{ and } i \neq j,$$

i.e., those permutations whose first two assignments are fixed. The tree is organized in the form of a heap keyed by the lower bounds (LWRBND) associated with the partial permutations stored in the tree. The root of the tree has the maximum key value. For each partial permutation (solution) in the heap, the corresponding subproblem is the QAP with part of the facilities being allocated.

In the second step, the four procedures of the branch-and-bound (selection, branching, elimination, and termination [76]) are used as follows:

1. The selection procedure simply selects the partial permutation stored in the root of the heap.
2. The branching procedure splits the selected partial permutation into two new partial permutations. One partial permutation, denoted p^i, includes an additional assignment and the other, denoted p^e, excludes that assignment.
3. The elimination procedure examines the newly formed partial permutations. The new partial permutation p^e is returned to the heap if it does not exclude all possible remaining

assignments. If p^i is a complete permutation, its objective function is evaluated and compared against the current BKUB, which is updated accordingly. Otherwise, a new lower bound is computed for p^i and if this bound is higher than the BKUB, p^i is discarded. This process is repeated until the heap is empty.

4. The termination test checks to see if the heap is empty.

Note that in the selection procedure, since the partial permutation at the root is generally closer to being a complete permutation, it is thus a promising candidate for reducing the BKUB. Furthermore, the partial permutation at the root has the highest lower bound among the lower bounds of the partial permutations stored in the tree. It can be discarded early in the search process, hence keeping the height of the heap small. This is very important since the solution space is extremely large even if the size of the problem is moderate.

In the final step, the best permutation(s) found is taken as the global optimal permutation(s). The algorithm can be described as follows.

Algorithm 1: An Exact Algorithm for the QAP

Input: n, matrices F, D of size $n \times n$.

Output: Optimal permutations for the QAP.

1. Find an initial value of the BKUB and create an initial heap of $n \times (n-1)$ partial permutations.
2. While the heap is not empty, do the following
   - 2.1. Take the root off the heap, choose another assignment from those still available (not already excluded).
   - 2.2. Create two partial permutations: p^e that excludes the assignment, and p^i that includes the assignment.
   - 2.3. Insert p^e into the heap unless it excludes all possible assignments.
   - 2.4. If p^i is a complete assignment, then evaluate the objective function for the QAP corresponding to p^i and update the BKUB accordingly.
   - 2.5. Otherwise, compare the lower bound (LWRBND) for p^i with BKUB; if LWRBND $>$ BKUB then discard p^i else insert p^i into the heap.
3. Print the best permutation(s) found as the optimal permutation(s) p and stop.

For the parallel version of the algorithm, we assume there are a total number of P processors available. One way to parallelize the sequential branch-and-bound algorithm is to use the simple idea of having all P processors accessing the heap in parallel. In practice, this approach will result in a significant amount of overhead in accessing the heap concurrently. Another approach is to simply divide the heap into P disjoint sub-heaps and assign one for each processor. After each processor finishes its heap, the suboptimal solutions of the processors are collected and the global optimal solution(s) can be found. This approach generally introduces a balanced load among the processors, and it was used in our study. The parallel algorithm can be described as follows

---

Algorithm 2: A Parallel Exact Algorithm for the QAP

Input: n, matrices F, D of size $n \times n$.

Output: Optimal permutations for the QAP.

1. Find an initial value of the BKUB and create a heap of the $n \times (n-1)$ partial solutions. Divide the heap into P sub-heaps, one for each processor.
2. For each processor, execute step 2 of Algorithm 1 in parallel.
3. Collect all the suboptimal solution(s) from the P processors, print the best solution(s) found as the optimal solution(s) p, and stop.

The above parallel algorithm was coded in PARALLEL FORTRAN to run on the IBM ES/3090-600S VF computer, that has 6 identical processors capable of processing independent tasks. In our experiment, we used all 6 processors. Given the 6 processors, the parallel algorithm divides the initial tree of $n \times (n-1)$ nodes into 6 sub-heap of $n \times (n-1)/6$ nodes each. Hence, each processor has its own heap to process. Then, 6 parallel tasks are dispatched for finding optimal solutions within their respective sub-heap. This procedure not only balances the load among all processors, but also keeps the processors busy to the fullest extent. The shared variable BKUB is updated in a critical section (using the LOCK and UNLOCK facility). The matrices F, D are shared data. Since they are accessed only by reading, they are not locked in a critical section. The heaps of the processors are not shared among them. The Gilmore-Lawler lower bound [20, 41] is used here as the lower bound of a partial permutation. If the given problem has multiple optimal solutions, the algorithm finds all the optimal solutions.

The algorithm is evaluated with two classes of test problems. The first set of problems include the NUGENT collection of the QAP [55]. The other set of test problems, denoted PALUBETSKES, are generated by the algorithm described in [50, 56].

The NUGENT set of test problems is one of the most widely used in the literature and can be used to test both heuristic and exact algorithms for the QAP. For our study, test problems of sizes $n = 5, 6, 7, 8, 12, 15, 20$, and 30 are used. For problems of sizes 20 and 30, optimal solutions can not be obtained in a reasonable amount of CPU time (due to this difficulty, the exact algorithm was not run on these two cases).

The PALUBETSKES set of test problems are generated according to the test problem generator, which outputs test problems with known optimal solutions, as reported in [50, 56]. The test problem generator contains two positive integer parameters, z and w. A random variable with a uniform distribution in $(0, 1)$ is used also in the generator. For our experiments, seven test problems are created with sizes $n = 10, 11, 12, 13, 14, 15$, and 16, with $z = 9, w = 5$. The optimal objective function value for a test problem generated here is dependent on n and z and is independent of the value of w. Thus, the generated test problems with the same value for the parameter z have the same optimal objective function value for each fixed n.

The computational results are designed to test the efficiency of the parallel algorithm in terms of its speed-up ratios. For our experiments, all 6 processors of the machine are used. For the parallel exact algorithm, each test problem in both sets of test problems is executed 5 times and the computational results reported here are the average over the 5 executions. When there are multiple optimal permutations, all of them are printed by the algorithm. The results are presented in tables 1 and 2. Note that in the following tables, all CPU times are in seconds. The speed-up ratio of a parallel algorithm is computed by dividing the cumulative CPU time by the wall time (elapsed time) of the parallel algorithm.

n	Optimal Value	Alg. Best Value	Cumulative CPU Time	Wall Time	Speed up Ratio
5	50	50	0.11	0.38	0.29
6	86	86	0.12	0.37	0.32
7	148	148	0.15	0.40	0.38
8	214	214	0.31	0.51	0.61
12	578	578	27.58	8.71	3.17
15	1150	1150	1587.30	430.91	3.68

Table 1: Parallel Algorithm - NUGENT Test Problems

n	Optimal Value	Alg. Best Value	Cumulative CPU Time	Wall Time	Speed up Ratio
10	1890	1890	0.61	0.65	0.94
11	3960	3960	47.29	11.60	4.08
12	2772	2772	247.27	51.20	4.83
13	6552	6552	982.33	263.52	3.73
14	4914	4914	324.47	89.31	3.63
15	5040	5040	12872.65	2580.74	4.99
16	5760	5760	12543.98	2340.38	5.36

Table 2: Parallel Algorithm - PALUBETSKES Test Problems

As we can see from the above tables, the parallel exact algorithm has a good speed-up ratio for test problems of sizes larger than 12. For smaller size problems, the parallel algorithm is not as efficient. Details and additional computational results can be found in [57], [45]. Regarding computational results for solving more general problems such as the quadratic zero-one programming problem (using distributed and shared memory machines) can be found in [59] and [60].

6 Concluding Remarks.

Extensive work has been done on development of parallel formulations of search algorithms for solving DOPs. Many of these formulations have been implemented for solving abstract and practical problems on commercially available parallel computers. For example, Kumar et. al. present parallel formulations of depth-first search, depth-first branch-and-bound, IDA*, and A* algorithms. They experimentally evaluate them in the context of problems such as optimizing floorplan of a VLSI chip [5], generating test patterns for combinatorial circuits [7] and tautology verification [34, 21, 6] on various machines including a 1024 processor nCUBE2, 128 processor Intel Hypercube, Symult 2010, a 128 processor BBN Butterfly^{TM}, and a network of 16 SUN^{TM} workstations. Bixby [9] presents a parallel branch and cut algorithm for solving the symmetric traveling salesman problem. He also presents solutions of the LP relaxations of airline crew-scheduling models. Miller et. al. [47] present parallel formulations of the branch and bound technique for solving the asymmetric traveling salesman problem on heterogeneous network computer architectures. Roucairol [71] presents parallel branch and bound formulations for shared memory computers and uses these to solve the Multiknapsack and Quadratic assignment problems. Lee et. al. [42] demonstrate experimentally that it is possible to obtain linear speedup for large instances of the 0/1 knapsack problem, using a divide-and-conquer DP algorithm, provided enough memory is available. Dantas et. al. [15] use vectorized formulations of DP for the Cray to solve optimal control problems. A technique for parallelizing DP algorithms in VLSI is presented in [22].

In summary, multiprocessors offer the potential for effectively speeding up many computationally intensive applications in discrete optimization.

References

[1] *AAAI-88 Workshop on Parallel Algorithms for Machine Intelligence*, St. Paul, Minneapolis, August 1988. Organizers: V. Kumar, L. Kanal, P.S. Gopalakrishnan; Sponsored by American Association for Artificial Intelligence.

[2] *IJCAI-89 Workshop on Parallel Algorithms for Machine Intelligence*, Detroit, Michigan, 20 August 1989. Organizers: V. Kumar, L. Kanal, P.S. Gopalakrishnan; Sponsored by American Association for Artificial Intelligence.

[3] T.S. Abdelrahman and T.N. Mudge. Parallel branch and bound algorithms on hypercube multiprocessors. In *Proceedings of the 3rd Conference on Hypercube Concurrent Computers and Applications - Volume II, ACM Press*, pages 1492–1499, 1988.

[4] A. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, Reading, Massachusetts, 1974.

[5] S. Arvindam, Vipin Kumar, and V. Nageshwara Rao. Floorplan optimization on multiprocessors. In *Proceedings of the 1989 International Conference on Computer Design (ICCD-89)*, 1989. Also published as MCC Tech Report ACT-OODS-241-89.

[6] S. Arvindam, Vipin Kumar, and V. Nageshwara Rao. Efficient parallel algorithms for search problems: Applications in vlsi cad. In *Proceedings of the Frontiers 90 Conference on Massively Parallel Computation*, October 1990.

[7] S. Arvindam, Vipin Kumar, V. Nageshwara Rao, and Vineet Singh. Automatic test pattern generation on multiprocessors. *Parallel Computing*, 17, number 12:1323–1342, December 1991.

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

[9] Robert Bixby. Two applications of linear programming. In *Proceedings of the Workshop On Parallel Computing of Discrete Optimization Problems*, 1991.

[10] R.E. Burkard and U.Derigs. Assignment and matching problems: Solution methods with fortran programs. *Lecture Notes in Economics and Mathematical Systems*, Vol 184, 1980.

[11] T.L. Cannon and K.L. Hoffman. Large scale 0-1 linear programming on distributed workstations. *Working Paper, Dept. of Operations Research, George Mason University*, February, 1989.

[12] C.Roucairol. A parallel branch and bound algorithm for the quadratic assignment problem. *Discrete Applied Mathematics*, 18:211–225, 1987.

[13] H. Crowder, E.L. Johnson, and M. Padberg. Solving large-scale zero-one linear programming problem. *Operations Research*, 2:803–834, 1983.

[14] R. Dehne, A. Ferreira, and A. Rau-Chaplin. A massively parallel knowledge-base server using a hypercube multiprocessor. Technical report, Carleton University, SCS-TR-170, April 1990.

[15] J. D. DeMello, J. L. Calvet, and J. M. Garcia. Vectorization and multitasking of dynamic programming in control: Experiments on a CRAY-2. *Parallel Computing*, 13:261–269, 1990.

[16] A. Gottlieb et al. The NYU ultracomputer - designing a MIMD, shared memory parallel computer. *IEEE Transactions on Computers*, pages 175–189, February 1983.

[17] Chris Ferguson and Richard Korf. Distributed tree search and its application to alpha-beta pruning. In *Proceedings of the 1988 National Conference on Artificial Intelligence*, August 1988.

[18] M. Furuichi, K. Taki, and N. Ichiyoshi. A multi-level load balancing scheme for or-parallel exhaustive search programs on the multi-psi. In *Proceedings of the 2nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 1990. pp.50-59.

[19] M. Garey and D.S. Johnson. *Computers and Intractability.* Freeman, San Francisco, 1979.

[20] P.C. Gilmore. Optimal and suboptimal algorithms for the quadratic assignment problem. *J. SIAM*, 10:305–313, 1962.

[21] Ananth Grama, Vipin Kumar, and V. Nageshwara Rao. Experimental evaluation of load balancing techniques for the hypercube. In *Proceedings of the Parallel Computing 91 Conference*, 1991.

[22] L. Guibas, H. T. Kung, and C. Thompson. Direct VLSI implementation of combinatorial algorithms. In *Proc. Caltech Conf. VLSI: Architecture, Design, Fabrication*, pages 509–525, 1979.

[23] Ellis Horowitz and Sartaj Sahni. *Fundamentals of Computer Algorithms.* Computer Science Press, Rockville, Maryland, 1978.

[24] Kai Hwang. *Advanced Computer Architecture with Parallel Programming.* McGraw Hill Inc., 1992.

[25] Guo jie Li and Benjamin W. Wah. Parallel processing of serial dynamic programming problems. In *Proc. COMPSAC 85*, pages 81–89, 1985.

[26] Laveen Kanal and Vipin Kumar. *Search in Artificial Intelligence.* Springer-Verlag, New York, 1988.

[27] R. M. Karp and M. H. Held. Finite state processes and dynamic programming. *SIAM J. Appl. Math*, 15:693 – 718, 1967.

[28] George Karypis and Vipin Kumar. Unstructured Tree Search on SIMD Parallel Computers. Technical Report 92–21, Computer Science Department, University of Minnesota, 1992. A short version of this paper appears in the Proceedings of Supercomputing 1992 Conference, November 1992.

[29] T.C. Koopmans and M.J. Beckmann. Assignment problems and the location of economic activities. *Econometrica*, 25:53–76, 1957.

[30] R.E. Korf. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence*, 27:97–109, 1985.

[31] Richard Korf. Optimal path finding algorithms. In Laveen Kanal and Vipin Kumar, editors, *Search in Artificial Intelligence.* Springer-Verlag, New York, 1988.

[32] W. Kornfeld. The use of parallelism to implement a heuristic search. In *IJCAI*, pages 575–580, 1981.

[33] V. Kumar, P.S. Gopalkrishnan, and L. Kanal (editors). *Parallel Algorithms for Machine Intelligence and Vision.* Springer Verlag, New York, 1990.

[34] Vipin Kumar, Ananth Grama, and V. Nageshwara Rao. Scalable load balancing techniques for parallel computers. Technical report, Tech Report 91-55, Computer Science Department, University of Minnesota, 1991.

[35] Vipin Kumar and Laveen Kanal. Parallel branch-and-bound formulations for and/or tree search. *IEEE Trans. Pattern. Anal. and Machine Intell.*, PAMI–6:768–778, 1984.

[36] Vipin Kumar and Laveen Kanal. The cdp: A unifying formulation for heuristic search, dynamic programming, and branch-and-bound. In Laveen Kanal and Vipin Kumar, editors, *Search in Artificial Intelligence.* Springer-Verlag, New York, 1988.

[37] Vipin Kumar, K. Ramesh, and V. Nageshwara Rao. Parallel best-first search of state-space graphs: A summary of results. In *Proceedings of the 1988 National Conference on Artificial Intelligence*, pages 122–126, August 1988.

[38] Vipin Kumar and V. N. Rao. Scalable parallel formulations of depth-first search. In Vipin Kumar, P. S. Gopalakrishnan, and Laveen Kanal, editors, *Parallel Algorithms for Machine Intelligence and Vision*. Springer-Verlag, New York, 1990.

[39] Vipin Kumar and V. Nageshwara Rao. Parallel depth-first search, part II: Analysis. *International Journal of Parallel Programming*, 16 (6):501–519, 1987.

[40] T. H. Lai and Sartaj Sahni. Anomalies in parallel branch and bound algorithms. *Communications of the ACM*, pages 594–602, 1984.

[41] E.L. Lawler. The quadratic assignment problem. *Management Science*, 9:586–599, 1963.

[42] J. Lee, E. Shragowitz, and S. Sahni. A hypercube algorithm for the 0/1 knapsack problem. In *Proc. Intl. Conf. on Parallel Processing*, pages 699 – 706, 1987.

[43] D.B. Leifker and L.N. Kanal. A hybrid sss*/alpha-beta algorithm for parallel search of game trees. In *IJCAI*, pages 1044–1046, 1985.

[44] Guo-Jie Li and Benjamin W. Wah. Coping with anomalies in parallel branch-and-bound algorithms. *IEEE Trans on Computers*, C-35, June 1986.

[45] Y. Li and P.M. Pardalos. Parallel algorithms for the quadratic assignment problem. In P.M. Pardalos, editor, *Advances in Optimization and Parallel Computing*, pages 177–189. North-Holland, 1992.

[46] G. Manzini and M. Somalvico. Probabilistic performance analysis of heuristic search using parallel hash tables. In *Proceedings of the International Symposium on Artificial Intelligence and MAthematics*, Ft. Lauderdale, FL, JAnuary, 1990.

[47] Donald Miller. Exact distributed algorithms for travelling salesman problem. In *Proceedings of the Workshop On Parallel Computing of Discre te Optimization Problems*, 1991.

[48] B. Monien and O. Vornberger. Parallel processing of combinatorial search trees. In *Proceedings of International Workshop on Parallel Algorithms and Architectures*, May 1987.

[49] B. Monien, O. Vornberger, and E. Spekenmeyer. Superlinear speedup for parallel backtracking. Technical Report 30, Univ. of Paderborn, FRG, 1986.

[50] K.A. Murthy and P.M. Pardalos. A polynomial-time approximation algorithm for the quadratic assignment problem. Technical report, Technical Report CS-33-90, The Pennsylvania State University, 1990.

[51] V. Nageshwara Rao and Vipin Kumar. Parallel depth-first search, part I: Implementation. *International Journal of Parallel Programming*, 16 (6):479–499, 1987.

[52] V. Nageshwara Rao and Vipin Kumar. Superlinear speedup in state-space search. In *Proceedings of the 1988 Foundation of Software Technology and Theoretcal Computer Science*, December 1988. Lecture Notes in Computer Science number 338, Springer Verlag.

[53] V. Nageshwara Rao, Vipin Kumar, and K. Ramesh. A parallel implementation of iterative-deepening-a*. In *Proceedings of the National Conf. on Artificial Intelligence (AAAI-87)*, pages 878–882, 1987.

[54] Nils J. Nilsson. *Principles of Artificial Intelligence.* Tioga Press, 1980.

[55] C.E. Nugent, T.E. Vollmann, and J. Ruml. An experimental comparison of techniques for the assignment of facilities to locations. *Journal of Operations Research*, 16:150–173, 1969.

[56] G.S. Palubetskes. Generation of quadratic assignment test problems with known optimal solutions (in Russian). *Zh. Vychisl. Mat. Mat. Fiz.*, 28(11):1740–1743, 1988.

[57] P.M. Pardalos and J.Crouse. A parallel algorithm for the quadratic assignment problem. In *Proceedings of the Supercomputing 1989 Conference*, pages 351–360, ACM Press, (1989).

[58] P.M. Pardalos, A.T. Phillips, and J.B. Rosen. *Topics in Parallel Computing in Mathematical Programming.* Science Press, 1992.

[59] P.M. Pardalos and G.P. Rodgers. Parallel branch and bound algorithms for unconstrainted quadratic zero-one programming. In et al. R. Sharda, editor, *Impacts of Recent Computer Advances on Operations Research*, pages 131–143. North Holland, 1989.

[60] P.M. Pardalos and G.P. Rodgers. Parallel branch and bound algorithms for quadratic zero-one programming on a hypercube architecture. *Annals of Operations Research*, 22:271–292, 1990.

[61] P.M. Pardalos and X.Li. Parallel branch and bound algorithms for combinatorial optimization. *Supercomputer*, 39:23–30, 1990.

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

[63] C. Powley and R. Korf. Single agent parallel window search: A summary of results. In *Proceedings of the eleventh International Joint Conference on Artificial Intelligence*, pages 36–41, 1989.

[64] Michael J. Quinn. Analysis and implementation of branch-and-bound algorithms on a Hypercube multicomputer. *IEEE Transactions on Computers*, ?, 1989.

[65] M.J. Quinn. *Designing Efficient Algorithms For Parallel Computers.* McGraw-Hill Book Company, New York, 1987.

[66] Finkel R.A. and J.P. Fishburn. Parallelism in alpha-beta search. *Artificial Intelligence*, 19:89–106, 1982.

[67] Abhiram Ranade. Optimal speedup for backtrack search on a butterfly network. In *Proceedings of the Third ACM Symposium on Parallel Algorithms and Architectures*, 1991.

[68] S. Ranka and S. Sahni. *Hypercube Algorithms for Image Processing and Pattern Recognition*. Springer-Verlag, New York, 1990.

[69] V. Nageshwara Rao and Vipin Kumar. On the efficicency of parallel backtracking. *IEEE Transactions on Parallel and Distributed Systems*, (to appear), 1992. available as a technical report TR 90-55, Computer Science Department, University of Minnesota.

[70] Shie rei Huang and Larry S. Davis. Parallel iterative a* search: An admissible distributed heuristic search algorithm. In *Proceedings of the eleventh International Joint Conference on Artificial Intelligence*, pages 23–29, 1989.

[71] C. Roucairol. Parallel branch-and-bound on shared memory multiprocessors. In *Proceedings of the Workshop On Parallel Computing of Discre te Optimization Problems*, 1991.

[72] Vikram Saletore and L. V. Kale. Consistent linear speedup to a first solution in parallel state-space search. In *Proceedings of the 1990 National Conference on Artificial Intelligence*, pages 227–233, August 1990.

[73] Wei Shu and L. V. Kale. A dynamic scheduling strategy for the chare-kernel system. In *Proceedings of Supercomputing 89*, pages 389–398, 1989.

[74] Douglas R. Smith. Random trees and the analysis of branch and bound proceedures. *Journal of the ACM*, 31 No. 1, 1984.

[75] H. Stone and P. Sipala. The average complexity of depth-first search with backtracking and cutoff. *IBM Journal of Research and Development*, May 1986.

[76] T.Ibaraki. Theoretical comparisons of search strategies in branch-and-bound algorithms. *International Journal of Computer and Information Sciences*, 5(4):315–344, 1976.

[77] Olivier Vornberger. Implementing branch-and-bound in a ring of processors. Technical Report 29, Univ. of Paderborn, FRG, 1986.

[78] Benjamin W. Wah, Guo jie Li, and Chee Fen Yu. Multiprocessing of combinatorial search problems. *IEEE Computer*, pages 93–108, June, 1985.

[79] Benjamin W. Wah, G.J. Li, and C. F. Yu. Multiprocessing of combinatorial search problems. In Vipin Kumar, P. S. Gopalakrishnan, and Laveen Kanal, editors, *Parallel Algorithms for Machine Intelligence and Vision*. Springer-Verlag, New York, 1990.

[80] Benjamin W. Wah and Y. W. Eva Ma. Manip - a multicomputer architecture for solving combinatorial extremum-search problems. *IEEE Transactions on Computers*, c–33, May 1984.

[81] Benjamin W. Wah and C. F. Yu. Stochastic modelling of branch-and-bound algorithms with best-first search. *IEEE Transactions on Software Engineering*, SE-11, September 1985.

[82] Herbert S. Wilf. *Algorithms and Complexity.* Prentice-Hall, 1986.