Parallel Processing of Discrete Optimization Problems: A Survey. *

Ananth Y. Grama and Vipin Kumar
Department of Computer Science,
University of Minnesota
Minneapolis, MN 55455

November 30, 1992

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 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 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

*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 Research Office contract number DAAL03-89-C-0038 with the University of Minnesota Army High Performance Computing Research Center.

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 [70], dynamic programming [14] and heuristic search [96, 47] methods use the structure of these graphs to solve DOPs without searching the set X exhaustively [62].

Given that DOP is an NP-hard problem [34], 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 [119, 131]. 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[129]). In these cases, parallel processing can significantly increase the size of solvable problems. Some applications using search algorithms (e.g. robot motion planning, speech understanding, 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 [10]).

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, 24, 56, 57, 61, 97, 105, 108, 126, 127].

This article provides a survey of parallel algorithms for solving DOPs. Section 2 reviews serial algorithms for solving DOPs. Section 3 discusses general algorithmic issues in parallel formulations. Section 4 discusses parallel formulations of depth-first search. Section 5 discusses parallel formulations of best-first search. Section 6 addresses issues of speedup anomalies in parallel search. Section 7 discusses parallel formulations of dynamic programming. Section 8 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 [90, 96, 47].

2.1 Depth-First Search Algorithms.

Depth-first search is a name commonly used to refer to a class of search techniques used 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 performed, 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. Upon termination, the current best solution is the globally optimal solution.

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 [54].

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. Applications of this technique are in such varied domains as artificial intelligence, task planning, optimal control etc. [14, 15, 89, 130, 71, 6]. Problems which can be solved efficiently using DP are characterized by the *Principle of Optimality* defined by Bellman [14]. 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 [62, 48, 46, 35, 4, 132, 27].

The DP and branch-and-bound methods are related to each other. In [62] Kumar and Kanal present a unified model for representing discrete optimization problems and clarify interrelationships between DP and branch-and-bound. A DP formulation is often expressed as a recursive functional equation whose right-hand side is an expression involving the maximization (or minimization) of values of some cost functions. Li and Wah [46] have developed a classification of DP programming schemes according to the form of the functional equations and the nature of the recursion. A DP formulation is called *monadic* if its cost function involves only one recursive term, otherwise it is called *polyadic*. A DP formulation is *serial* if the subproblems can be grouped in levels, and the solution to any subproblem in a certain level can be found using subproblems that belong only in the immediately preceding levels; otherwise it is *non-serial*.

3 General Algorithmic Issues in Parallel Formulations.

This section discusses some of the general aspects related to parallel formulations of an algorithm. Various terms used in subsequent sections are also defined here.

Sequential Runtime : The sequential runtime T_s is defined as the time taken to solve a particular problem instance on one processing element.

Parallel Runtime, Speedup and Efficiency : The parallel runtime T_p is the time taken to solve a particular problem instance on an ensemble of P processing elements. The Speedup S achieved by a parallel system is defined as the gain in computation speed achieved by using P processing elements with respect to a single processing element. $S = T_s/T_p$. The efficiency E

denotes the effective utilization of computing resources. It is the ratio of the speedup to the number of processing elements used ($E = S/P$).

Parallel Processing Overheads : Processors executing a parallel search algorithm incur a number of overheads. These include the communication overheads and idle time due to load imbalance and contention over shared data structures. Clearly, due to these overheads the speedup should be less than P, assuming that both the sequential and parallel formulations of an algorithm do the same amount of work.

Search Overhead For search based algorithms, the amount of work done by a parallel formulation is often different from that done by the sequential formulation, because the space searched by them is different. Let the amount of work done by a sequential processor be W. Now, if the total amount of work done by P parallel processors is W_P, then the **search overhead** of the parallel system is defined as the ratio of the work done by the parallel formulation to that done by the sequential algorithm, which is given by $\frac{W_P}{W}$. From this, we can also see that the upper bound on the speedup which can be attained from such a parallel system is given by $\frac{P}{\frac{W_P}{W}}$. The actual the speedup, however, may be less than $\frac{P}{\frac{W_P}{W}}$ due to parallel processing overheads.

For search algorithms, the size of the problem solved can be approximated by the number of nodes searched by the algorithm. Thus, search overhead can be viewed as the ratio of number of nodes expanded by the parallel formulation of search to the sequential formulation. In most parallel search algorithms, search overhead is greater than 1. In some cases, it may be less than 1, leading to superlinear speedups with respect to the base sequential algorithm. Superlinearity effects are discussed in Section 6.

Scalability and Scalable Algorithms : If a parallel algorithm is used to solve a problem instance of a fixed size, then the efficiency decreases as number of processors P increases. The reason for this is that the total parallel processing overhead increases with P. For many parallel algorithms, for a fixed P, if the problem size W is increased, then the efficiency becomes higher (and approaches 1), because the total parallel processing overhead grows slower than W. For these parallel algorithms, the efficiency can be maintained at a desired value (between 0 and 1) with increasing number of processors, provided the problem size is also increased. We call such algorithms **scalable** parallel algorithms. The rate at which the problem size must grow with respect to the number of processors to keep the efficiency fixed determines the degree of scalability of a parallel algorithm [59, 66].

Communication versus Computation Tradeoffs Parallel search involves the classical communication versus computation tradeoff. As we had discussed above, search overhead for many algorithms is greater than one, implying that the parallel formulation of the program does more work than the sequential form. We can reduce the search overhead for such formulations at the cost of increased communication, and vice-versa. For instance, if search space is statically divided among different processors which independently search them without communicating, then they together will often perform much more search than one processor. This search overhead can often be reduced if processors are allowed to communicate.

Architecture Related Issues. Suitability of a parallel search formulation depends strongly on the features of the underlying architecture. Parallel computers could be SIMD (Single Instruction, Multiple Data streams) or MIMD (Multiple Instruction, Multiple Data streams). In SIMD machines, all the processors execute the same instructions at any given point of time. In MIMD machines, each processor executes a possibly different program asynchronously. In massively parallel SIMD machines, computations are of a fine granularity, hence communication to computation tradeoffs are very different compared to MIMD machines. Due to architectural constraints in SIMD machines, load balancing needs to be done on a global scale. In contrast, on MIMD machines, load can be balanced among a small subset of processors while the others are busy doing work. Hence, the load balancing schemes developed for MIMD architectures may not perform well on SIMD architectures.

Another important architectural feature is whether the parallel computer has distributed or shared memory. Shared memory parallel computers have global memory that can be directly addressed by all processors. In distributed memory computers, each processor has its own local memory, and processors can communicate only by exchanging messages over the communication network. A number of search techniques use global data structures. These techniques lend themselves naturally to shared memory machines. Direct formulations of these search techniques on distributed memory computers often results in communication and contention over accessing the shared data structures. Typically, this can be circumvented by distributing the data structure over many processors, which may result in additional search overhead. Shared memory architectures are themselves not very scalable; consequently, it is not possible to construct shared memory machines with large number of processors. All these factors influence the selection of a parallel formulation of a search technique.

In subsequent sections, we will discuss these issues in greater detail in the context of state space search.

4 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 [116, 118, 58, 66, 84, 107, 51, 5, 109].

Load balancing can be viewed as a two phase process : task partitioning and subtask distribution. The two most commonly used techniques for task partitioning are *Stack Spiltting* and *Node Splitting*. Stack splitting [58, 66] is illustrated in Figure 2. Here, 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. In *Node Splitting*, each of the n nodes spawned by a node in a tree are themselves given away as individual subtasks. For highly irregular search spaces, stack splitting is more likely to yield subtasks which are of comparable size as it gives out nodes at various levels of the stack. In contrast, node splitting can yield subtasks which are of widely differing sizes since subtrees rooted at different nodes in a tree can be of vastly different sizes. Typically, load balancing schemes utilizing node splitting [33, 107, 30, 99] require more work transfers compared to those using stack splitting.

Subtask distribution schemes can be classified on the basis of the generation and transfer strategies. Subtask generation can be sender initiated or receiver initiated. In receiver initiated subtask generation schemes, work splitting is performed only when an idle processor (receiver) requests for work. In these schemes, when a processor runs out of work, it generates a request for work. In sender initiated subtask generation schemes, the generation of subtasks is independent of the work requests from idle processors. These subtasks are delivered to processors needing them, either on demand (*i.e.*, when they are idle) [33] or without demand [107, 118, 116]. The former is called sender initiated work transfer and the latter is called receiver initiated work transfer. Using these three criteria, we can classify various parallel formulations of depth-first search. Table 1 presents instances of important parallel formulations in each of these classes. Clearly, receiver initiated subtask generation cannot be used in conjunction with sender initiated work transfer.

4.1 Parallel formulations of Backtracking.

A general method for parallelizing DFS is presented in [31, 82, 58, 66]. This technique employs stack splitting in conjunction with receiver initiated subtask distribution schemes. In this formulation, each processor searches a disjoint part of the search space in a depth-first fashion. 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. The detailed schematic of this process is shown in Figure 1. When a processor runs out of work, it selects a target processor for addressing a work request. On receiving a work request, a processor either responds with a

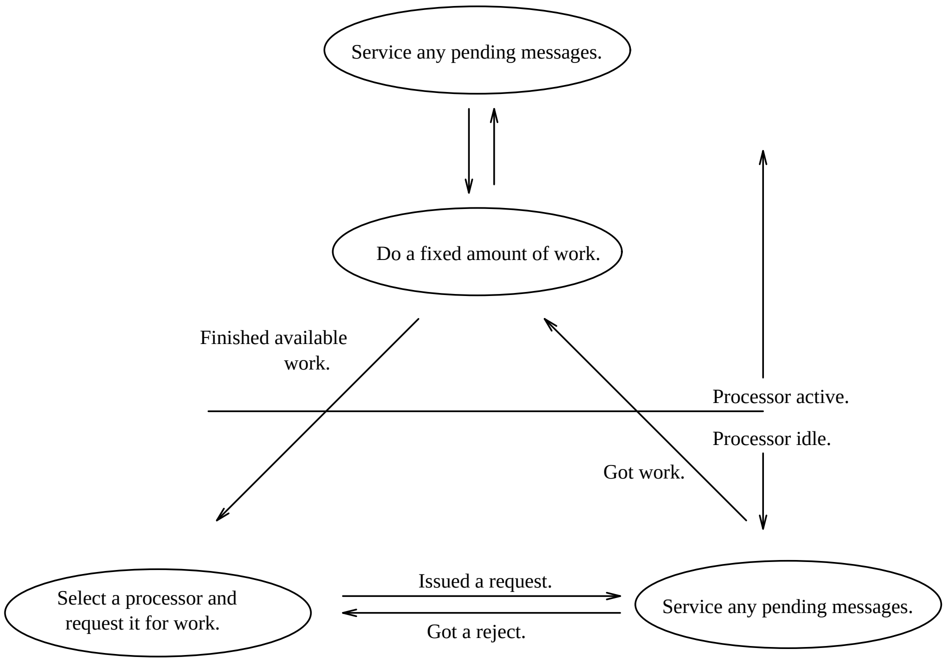

Figure 1: State diagram describing the generic parallel DFS formulation for distributed memory architectures.

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 dynamically among various processors.

The selection of a target processor for a work request can be done in a number of ways. This selection should be done so as to minimize the total number of work requests and balance load evenly between the processors. For example, consider two techniques, *random polling* and *global round robin* discussed in [58, 31]. 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). As shown in [58], the global round robin scheme minimizes the total number of work requests for a wide class of problems; however, accessing the global pointer becomes a bottleneck [58]. Consequently, when the number of processors increases, its performance degrades. In contrast, random polling results in more work requests but does not suffer from contention over shared data structures. However, on machines that have hardware support for concurrent access to a global pointer (e.g., the hardware fetch and add [28]), the performance of the global round robin scheme would be better than random polling.

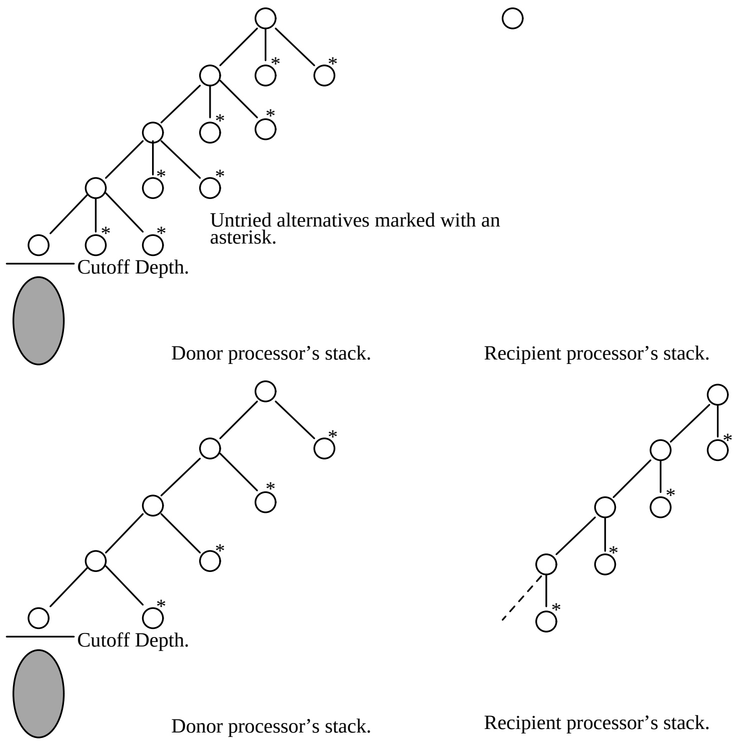

Figure 2: Illustration of the stack splitting mechanism.

A number of load balancing schemes have been explored in [33, 30, 99, 118, 107, 116, 5, 95, 11, 36, 58]. Finkel and Manber [31] present performance results for a number of problems such as the Travelling salesman problem and Knights tour for the Crystal multicomputer developed at the University of Wisconsin. Monien et. al. [84] show linear speedups on a network of transputers for a variety of combinatorial problems. Kumar et. al. [58, 10] show linear speedups for problems such as 15 puzzle and tautology verification for various architectures such as a 128 processor BBN Butterfly, 128 processor Intel Hypercube, a 1024 processor nCUBE2, and a 128 processor Symult 2010. Kumar, Ananth and Rao [36, 58, 66, 86] have investigated the scalability and performance of many of these schemes for a variety of architectures such as hypercubes, meshes and networks of workstations.

Parallel DFS using sender initiated subtask distribution has been proposed by a number of researchers [30, 99, 33, 118, 107]. These use different techniques for task generation and transfer as shown in Table 1. Ichiyoshi et. al. propose the single level and multi level sender based scheme [33]. These schemes are illustrated in 3. The single level scheme balances the load among processors by dividing the task into a large number of subtasks such that each processor is responsible for processing more than one subtask. In doing so it statistically assures that the total amount of work at each processor is roughly the same. The task of subtask generation is handled by a designated processor called MANAGER. The MANAGER generates a specific number of subtasks and gives them one by one to the requesting processors on demand. Since the MANAGER has to generate subtasks fast enough to keep all the other processors busy, subtask generation forms a bottleneck and consequently this scheme does not have a good scalability. The multi level scheme tries to circumvent the subtask generation bottleneck [33] of the single level scheme through multiple level subtask generation. In this scheme, all processors are arranged in the form of an m-ary tree of depth l. The task of super-subtask generation is given to the root processor. It divides the task into super-subtasks and distributes them to its successor processors on demand. These processors subdivide the super-subtasks into subtasks and distribute them to successor processors on request. The leaf processors repeatedly request work from their parents as soon as they finish previously received work. A leaf processor is allocated to another subtask generator when its designated subtask generator runs out of work. For $l = 1$, the multi and single level schemes are identical. Kimura et. al. [52] present detailed scalability analysis of these schemes.

Ferguson and Korf [30, 99] present a work distribution scheme, called Distributed Tree Search (DTS), in which processors are allocated to different parts of the search tree dynamically. Initially all the processors are assigned to the root. When the root node is expanded (by one of the processors) each of the successor nodes is assigned to disjoint subsets of processors. This is done in accordance with a selected processor allocation strategy. This process continues till each processor is individually responsible for searching a part of the search space. If a processor finishes searching its part of the search space sooner than others, it is reassigned to an unexplored part of another processors' search space. DTS has been shown to be quite effective in the context of parallel $\alpha-\beta$

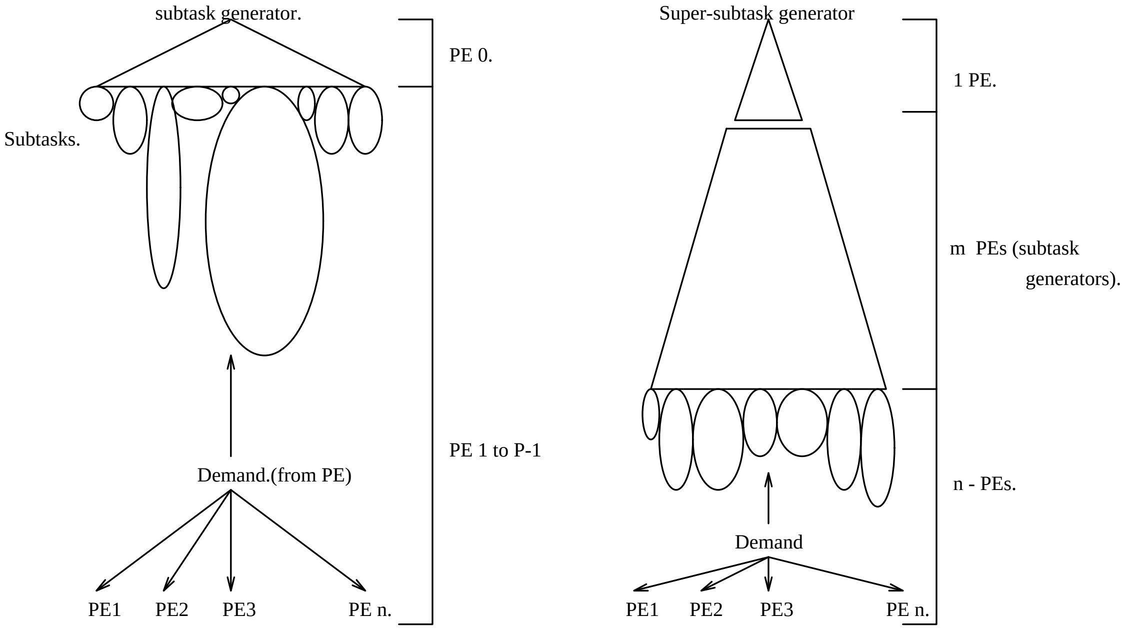

Figure 3: Single and Multi-level subtask distribution schemes.

search on a 32 processor hypercube[30].

A number of techniques using randomized allocation have been presented in the context of parallel depth first search of state space trees [107, 118, 116]. In depth first search of trees, the expansion of a node corresponds to performing a certain amount of useful computation and generation of successor nodes, which can be treated as subtasks.

In the Randomized Allocation Strategy proposed by Shu and Kale [118], every time a node is expanded, all of the newly generated successor nodes are assigned to randomly chosen processors. The random allocation of subtasks ensures a degree of load balance over the processors. There are however some practical difficulties with the implementation of this scheme. Since for each node expansion, there is a communication step, the efficiency is limited by the ratio of time for a single node expansion to the time for a single node expansion and communication to a randomly chosen processor. Hence applicability of this scheme is limited to problems for which the total computation associated with each node is much larger than the communication cost associated with transferring it. This scheme also requires a linear increase in the cross section communication bandwidth with respect to P; hence it is not practical for large number of processors on any practical architecture (*eg.* cube, mesh, networks of workstations). For practical problems, in depth first search, it is much cheaper to incrementally build the state associated with each node rather than copy and/or create the new node from scratch [122, 19]. This also introduces additional inefficiency. Further, the memory requirement at a processor is potentially unbounded, as a processor may be required to store an arbitrarily large number of work pieces during execution. In contrast, for all other load balancing schemes discussed up to this point, the memory requirement of parallel depth first search remains similar to that of serial depth first search.

Ranade [107] presents a variant of the above scheme for execution on butterfly networks or hypercubes. This scheme uses a dynamic algorithm to embed nodes of a binary search tree into a butterfly network. The algorithm works by partitioning the work at each level into two parts and sending them over to the two sons (processors) in the network. Any patterns in work splitting and distributions are randomized by introducing dummy successor nodes. This serves to ensure a degree of load balance between processors. The author shows that the time taken for parallel tree search of a tree with M nodes on P processors is given by $O(M/P + \log P)$ with a high degree of probability. This corresponds to an optimal isoefficiency of $O(P \log P)$ for a hypercube. This scheme has a number of advantages over Shu's scheme for hypercube architectures. By localizing all communications, the communication overhead is reduced by a log factor here. (Note that communicating a fixed length message between a randomly chosen pair of processors in a hypercube takes $O(\log P)$ time.) Hence, Ranade's scheme is physically realizable on arbitrarily large hypercube architectures. To maintain the depth-first nature of search, nodes are assigned priorities and are maintained in local heaps at each processor. This adds an additional overhead of managing heaps, but may help in reducing the overall memory requirement. Apart from these, all the other restrictions on applicability of this scheme are the same as those for Shu and Kale's [118] scheme.

The problem of performing a communication for each node expansion can be alleviated by enforcing a granularity control over the partitioning and transferring process [86, 118]. It is , however, not clear whether mechanisms for effective granularity control can be derived for highly irregular state space trees. One possible method [86] of granularity control works by not giving away nodes below a certain "cutoff" depth. Search below this depth is done sequentially. This clearly reduces the number of communications. The major problem with this mechanism of granularity control is that subtrees below the cutoff depth can be of widely differing sizes. If the cutoff depth is too deep, then it may not result in larger average grain size and if it is too shallow, subtrees to be searched sequentially may be too large and of widely differing sizes.

Saletore and Kale [116] present a parallel formulation in which nodes are assigned priorities and are expanded accordingly. By doing this, they are able to ensure that the nodes are expanded in the same order as the corresponding sequential formulation. In terms of node expansion sequence, this is the only true parallel formulation of DFS. Saletore and Kale show that this prioritized DFS formulation yields consistently increasing speedups with increasing number of processors for sufficiently large problems. This is important since often search overheads in parallel DFS may lead to significant performance degradation. However these schemes appear to be far less scalable than those that do not require prioritization.

Subtask distribution scheme→ Work transfer strategy→ Splitting strategy↓	Sender Initiated Sender Initiated	 Receiver Initiated	Receiver Initiated Receiver Initiated
Node Splitting	Ranade [107] Shu and Kale [118]	Furuichi et. al. [33]	Ferguson and Korf [30] Monien et. al. [82] Finkel and Manber [31]
Stack Splitting			Kumar et. al. [66, 86, 58] Finkel and Manber [31]

Table 1: Taxonomy of parallel formulations of DFS.

4.2 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. In DFBB, 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. For example, Arvindam et. al. [10] show linear speedups on a 1024 processor Ncube^{TM}, a 128 processor Symult^{TM} and a network of 16 SUN workstations in the context of the floorplan optimization problem for VLSI circuits. Patil and Banerjee [95] present a parallel formulation of the branch-and-bound algorithm for test pattern generation and CAD applications.

The classical $\alpha - \beta$ game tree search algorithm can also be viewed as a depth-first branch-and-bound algorithm[60, 63]. Unfortunately, the information about the current best solution (which in this case is the current best winning strategy) cannot be captured in a single number. Hence the simple solution discussed above cannot be used to develop an effective parallel formulation of $\alpha - \beta$. A number of researchers have developed parallel formulations of $\alpha - \beta$ [9, 13, 30, 41, 76, 79, 82, 99]. In particular, methods developed in [30, 82, 99] have shown reasonable speedups on dozens of processors (*e.g.*, in Monien [82], a speedup of 12 is reported on 16 processors for chess). The speedups obtained here are much less than those obtained for DFS formulations (*e.g.*, see [86]). It is not even clear whether parallel $\alpha - \beta$ formulations can make effective use of thousands of processors.

There are a number of conflicting factors impacting the performance of parallel DFS in the case of $\alpha - \beta$. Typical game trees are strongly ordered in nature. If this ordering information is not taken into consideration, parallel $\alpha - \beta$ ends up searching a far bigger space than serial $\alpha - \beta$ (which increases the search overhead). The techniques that use ordering information reduce the overall effectiveness of load balancing [9, 106]. Hence, an appropriate balance needs to be struck in using ordering information in the load balancing algorithm. Another such tradeoff is in terms of the communication overhead for communicating the $\alpha - \beta$ bounds and the pruning achieved due to them. At one extreme, any change in the α and β value of a node is announced to any other processor that may benefit from it. This will maximize pruning but also the communication overhead. At the other extreme, concurrent processes never exchange information about bounds. The exact nature of this tradeoff is not clearly understood and is a subject of continuing research.

4.3 Parallel Formulations of IDA*.

There are two approaches to parallelizing IDA*. In one approach, different processors work on different iterations of IDA* independently [61]. 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 more effective approach is to execute each iteration of IDA* via parallel DFS [65, 66, 86]. Since all processors work with the same cost bound, each processor stores this value locally and performs DFS on its own search space. After termination of each iteration of parallel IDA*, one specifically assigned processor determines the cost bound for the next iteration and restarts parallel depth-first search with the new cost bound. Search stops in the final iteration when one of the processors finds a goal node and informs all others about it. The termination of an iteration can be detected in many ways. On shared-memory architectures (e.g., Sequent, BBN Butterfly), a globally shared variable is used to keep track of the number of idle processors. On distributed memory architectures (e.g., the Intel Hypercube), algorithms such as Dijkstra's token termination detection algorithm [26] can be used. This approach has been used by a number of researchers and tested in the context of different problems and architectures [88, 65, 98, 77]. Clearly, performance and scalability issues for this parallel formulation of IDA* are similar to those of the parallel formulations of DFS algorithm used in parallelizing it.

4.4 Parallel DFS on SIMD computers.

As mentioned in Section 3, the communication and computation tradeoffs in SIMD machines are very different compared to MIMD machines. On SIMD machines, all processors are either in a phase of node expansion or in a phase of load balancing. Hence, if a processor becomes idle, it has to remain idle until all processors enter a load balancing phase. In contrast, in MIMD machines, a processor can request for work the moment it becomes idle (independent of the state of the other processors). Any efficient formulation of DFS on a SIMD machine comprises of two major components (i) a triggering mechanism, which determines when search space redistribution must occur to balance search space over processors; and (ii) the redistribution mechanism itself. The triggering mechanism may be dynamic (changes as the search proceeds) or static (remains the same throughout). In addition, several redistribution techniques have also been proposed. Recent work has shown that SIMD architectures such as $\mathrm{CM2}^{TM}$ can also be used to implement parallel DFS algorithms effectively. Frye [32] presents an implementation of DFS on the CM2 for a block puzzle. Powley et. al.[98] and Mahanti et. al.[77] present parallel IDA* for solving the 15 puzzle problem on $\mathrm{CM2}^{TM}$. Powley's and Mahanti's formulations use different triggering and redistribution mechanisms however, both schemes however report similar results for the 15 puzzle. Karypis and Kumar [51] present a new load balancing technique which is shown to be highly scalable on SIMD architectures. In particular, the scalability of this scheme is shown to be no worse than that of the best load balancing schemes on MIMD architectures.

5 Parallel Best-First Search.

A number of researchers have investigated parallel formulations of A*/B&B algorithms [61, 64, 72, 84, 101, 112, 123, 128, 109]. 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 [81, 83, 84, 102, 101, 103, 111, 128]. Given P processors, the simplest parallel strategy is to let each parallel processor work on one of the current best nodes in the OPEN list. This strategy is called the centralized strategy [81, 103, 111] because each processor gets work from the global OPEN list. As discussed in [45], this strategy should not result in much redundant search. Figure 4 illustrates this strategy. There are two problems with this approach.

(1) The termination criterion of sequential A* does not work any more; i.e., if a current processor picks up a goal-node m for expansion, the node m may not be the best goal node. But the termination criterion can be easily modified to ensure that termination occurs only after a best solution has been found.[72, 100]

(2) Since OPEN will be accessed by all the processors very frequently, it will have to be maintained in shared memory that is easily accessible to all the processors. Hence distributed-memory architectures such as the Hypercube[117] are effectively ruled out. Even on shared memory architectures, contention for OPEN would limit the speedup to $(T_{exp} + T_{access})/T_{access}$, where T_{exp} is the average time for one node expansion, and T_{access} is the average time for accessing OPEN per node expansion [111]. Note that the access to CLOSED does not cause contention, as different

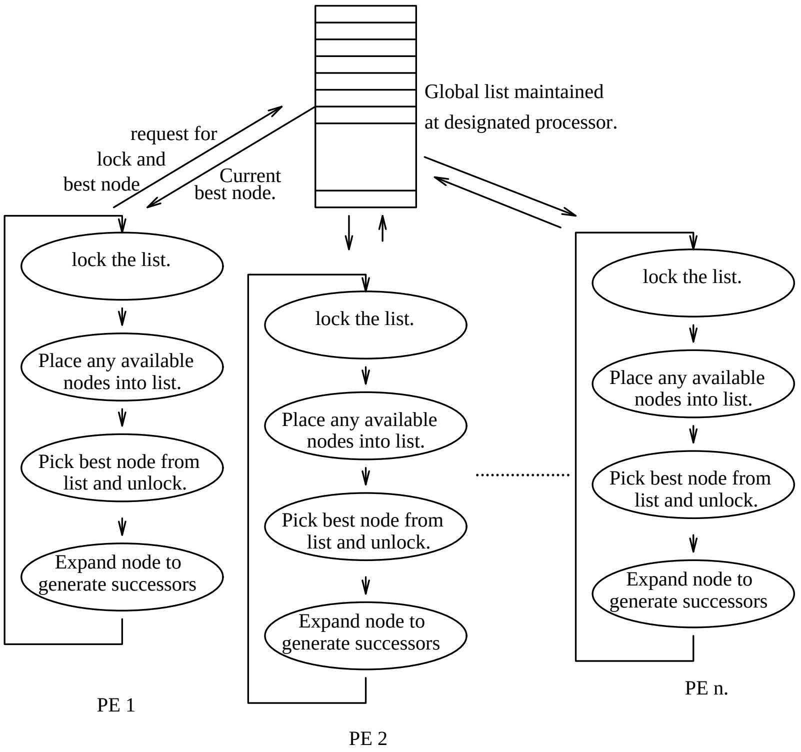

Figure 4: General schematic for parallel best-first search using a centralized strategy.

processors would manipulate different nodes.

One way to avoid the contention due to centralized OPEN list is to let each processor have its own local OPEN list. Initially, the search space is statically divided and given to different processors (by expanding some nodes and distributing them to the local OPEN lists of different processors). All processors now select and expand nodes simultaneously without causing contention on the shared OPEN list 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 the sequential search. This would lead to a high search overhead and poor speedup. Many researchers have proposed communication strategies between local lists to minimize redundancy. Use of distributed lists represents the classical communication versus computation tradeoff. Decreasing the communication coupling between distributed lists increases search overhead and conversely, reducing search overhead using increased communication has the effect of increasing communication overheads.

Communication strategies for parallel best-first search formulations depend on whether the search space is in the form of a tree or a graph. For graphs there is an additional overhead of detecting duplication of nodes. For trees, each new successor is guaranteed to be a new node leading to an unexplored part of the search space. In contrast, in graphs a node can be reached from multiple paths. This means that in the absence of duplication checks, we may potentially replicate search of the space rooted at this node for every single path leading up to this node. Any graph can be unfolded into a tree. For problems such as the 15 puzzle, this unfolding does not cause a significant search overhead. But for some other problems (*e.g.*, finding the shortest path in a reasonably well connected graph), unfolded graphs can be exponentially bigger than the original graphs [55]. Clearly, this would not be acceptable for such applications. We will discuss communication strategies for tree and graph search separately.

5.1 Communication Strategies for Parallel Tree Search Formulations.

A number of communication strategies have been proposed by researchers for use in parallel best-first tree search formulations.

In the random communication strategy, each processor periodically puts some of its best nodes into the OPEN list of a randomly selected processor [124, 23, 64]. This ensures that if some processor has a good part of the search space, then others get a part of it. This strategy can be easily implemented on distributed-memory systems with low diameter(such as Hypercube[117], Torus[23]) as well as shared memory multiprocessors such as the Butterfly. If the frequency of transfer is high, then the redundancy factor can be very small; otherwise it can be very large. The choice of frequency of transfer is effectively determined by the cost of communication. If communication cost is low (e.g., on shared-memory multiprocessors) then it would be best to perform communication after every node expansion.

In the ring communication strategy, different processors are assumed to be connected in a virtual ring [123, 128]. Each processor periodically puts some of its best nodes into the OPEN list of its neighbor in the ring. This allows transfer of good work from one processor to another. This strategy is well suited even for distributed-memory machines with high diameter (e.g., ring). Of course, it can be equally easily implemented on low diameter networks and shared memory architectures. As before, the cost of communication determines the choice of frequency of transfer. Figure 5 illustrates the ring communication strategy.

Rao and Kumar [64] propose the blackboard communication strategy. In this strategy, there is a shared BLACKBOARD through which nodes are switched among processors as follows. After selecting a (least f-value) node from its local OPEN list, the processor proceeds with its expansion only if it is within a "tolerable" limit of the best node in the BLACKBOARD. If the selected node is much better than the best node in the BLACKBOARD, then the processor transfers some of its good nodes to the BLACKBOARD before expanding the current node. If the selected node is much worse than the best node in the BLACKBOARD, then the processor transfers some good nodes from the BLACKBOARD to itself, and then reselects a node for expansion. Figure 6 illustrates the blackboard communication strategy.

The BLACKBOARD formulation is suited only for shared-memory architectures, whereas the

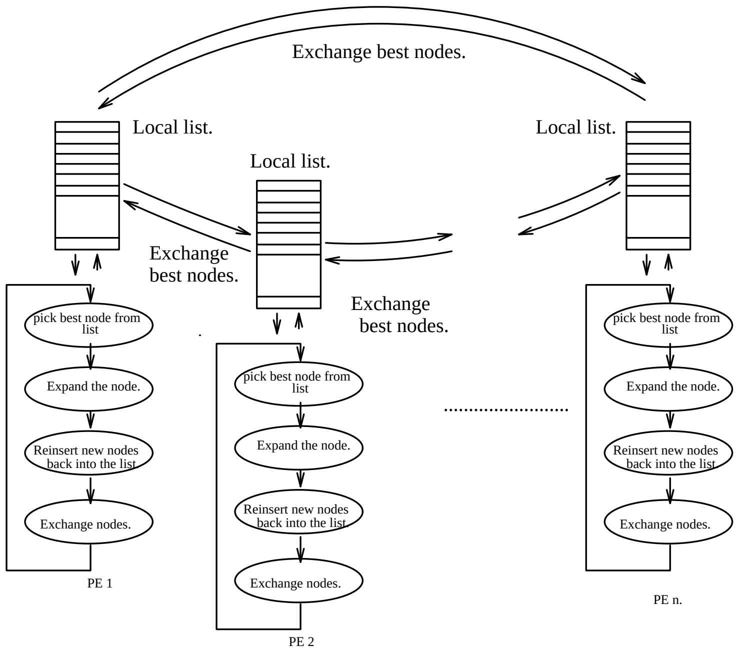

Figure 5: Distributed memory implementation of parallel best-first search using the ring communication strategy.

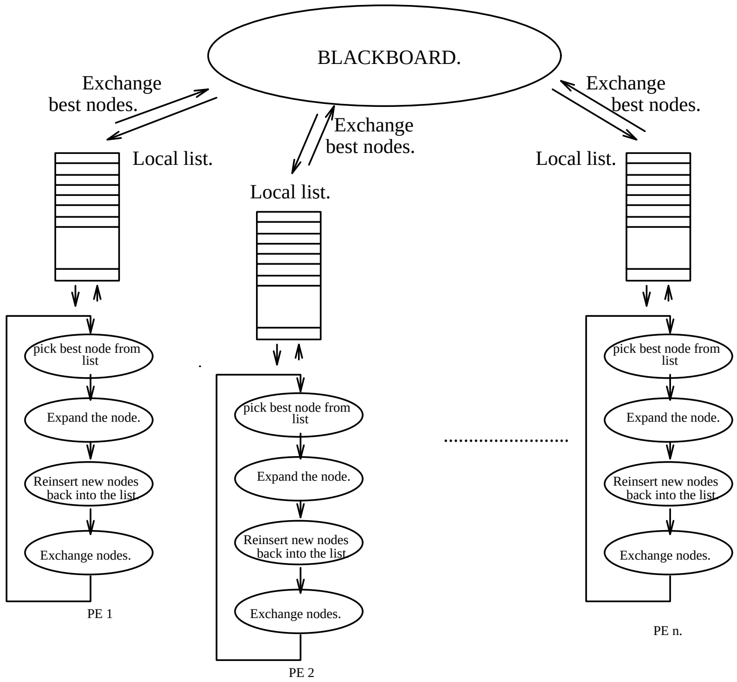

Figure 6: Distributed memory implementation of parallel best-first search using the BLACKBOARD communication strategy.

other two are suited for distributed-memory architectures as well. The effectiveness of different parallel formulations is also strongly dependent upon the characteristics of the problem being solved. Kumar et. al. [64] present an analysis of the relationship between the characteristics of the search spaces and the suitability of various parallel formulations. Many of these schemes have been shown to give good (in some cases even near linear) speedups for moderate (< 100) number of processors for specific problems for large enough search graphs. Karp and Zhang[49] analyze the search overhead of parallel best-first branch-and-bound (*i.e.*, A*) using random distribution of nodes for a specific model of search trees. Renolet et al. [113] use Monte Carlo simulations to model performance of parallel best-first search. Wah and Yu [129] also present stochastic models to analyze performance of parallel formulations of depth first branch and bound and best first branch and bound search.

A number of researchers have applied these strategies to various problems. Bixby [17] 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. [80] present parallel formulations of the best-first branch and bound technique for solving the asymmetric traveling salesman problem on heterogeneous network computer architectures. Roucairol [114] presents parallel best-first branch and bound formulations for shared memory computers and uses these to solve the Multiknapsack and Quadratic assignment problems.

5.2 Communication Strategies for Parallel Graph Search.

The only reasonable way to ensure replication checking in parallel graph search is to implicitly and statically assign each node to a distinct processor. This can be done using a hash function which takes the node descriptor as input and returns a value in the range 0 through $P-1$. When a node is generated, the hash function is applied to it and the node is sent to the processor designated by the return value of the hash function. This processor checks the node for replication and if appropriate installs the node into its OPEN list. If the same node is generated again, it is mapped to the same processor and duplication checks can be performed [78, 29]. This distribution strategy is similar to the random distribution technique discussed in the previous subsection. Manzini [78] shows that this method ensures a good distribution of promising nodes among all processors. Hashing techniques however cause performance degradation as each node generation results in a corresponding communication cycle.

Evett et. al. [29] present a similar scheme called PRA* (parallel retracting A*), which is designed to operate under limited memory conditions. In this formulation, each node is hashed to a unique processor. If a processor receives more nodes than it can locally store, then it retracts nodes with poorer heuristic values are retracted. These retracted nodes can be re-expanded when the more promising nodes failed to yield a solution. Evett et. al. present results from implementation of PRA* on the SIMD CM2 in the context of the 15 puzzle and show good speedups.

5.3 Some Practical Applications of Best-First Branch-and-Bound Search.

In this section, we discuss two important applications that use the best-first branch-and-bound technique, namely, the 0/1 integer linear programming algorithm and the quadratic assignment problem. We also present important parallel formulations of these two problems.

The 0/1 integer linear programming problem can be stated as follows [91] :

$$\min c'x$$

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

The branch and bound algorithm (using best-first search) for solving this problem is described in [37]. The search for the optimal solution proceeds by instantiating elements (x_i's) of the vector x. In each iteration, the subproblem (partially instantiated vector x) in the OPEN list whose heuristic lower bound is the smallest is selected for expansion. Partial instantiations obtained from expansion which are guaranteed to yield worse solutions than the incumbent solution are pruned and the remaining partial instantiations are inserted into the OPEN list.

A number of parallel formulations of this algorithm have been proposed by researchers. Abdelrahman and Mudge [3] propose two parallel formulations for distributed memory computers. The first formulation is based on a centralized OPEN list which is maintained by a master processor. The slave processors select $P-1$ ($P-1$ slave processors) subproblems from the list which have the best bounds. These subproblems are then expanded and after lower bound, feasibility and termination tests, the results are sent back to the master processor. The advantage of such an approach is that the subproblems selected are the ones whose lower bounds are best in the globally. On the other hand, this approach suffers from serious drawbacks. Contention at master processor, significant communication overhead and memory requirement at the master processor all result in significant performance deterioration. The second formulation uses distributed lists with node redistribution between nearest neighbors. When all neighboring processors are idle, the algorithm "guesses" to terminate (for details see [3]). The second approach outperforms the first. Computational results on an NCUBE/six multiprocessor are reported. Cannon and Hoffman [18] propose a parallel formulation based on the branch and cut approach for a network of workstations connected by an ethernet. This approach has been used to solve a set of test problems from [22].

The quadratic assignment problem can be stated in the form of the following minimization problem [53] :

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

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

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 $N = \{1, ..., n\}$.

Such problems arise frequently in mathematical modeling of various location problems where the cost associated with allocating a facility to a certain location is a function of its distance from other facilities and its interaction with other facilities. Common application areas of this problem include backboard wiring, machine scheduling, ordering interrelated data on computer storage devices and scheduling of economic lot sizes [75].

Roucairol [21] proposed among the first parallel formulations of the best-first branch-and-bound algorithm for solving the QAP and implemented the formulation on a CRAY XMP/48. Performance results obtained were however not very impressive and only small to medium sized problems could be solved using this formulation. Pardalos and Crouse [92] propose a parallel formulation of the QAP algorithm based on best-first branch-and-bound search. A heap is used to maintain the OPEN list. To avoid contention, the heap is divided into P disjoint heaps and each processor is assigned one sub-heap. Every processor searches its heap independently and solutions are collected. The global best solution is determined by comparing all the locally optimal solutions. Details and computational results of this formulation are presented in [92, 75].

Experimental results for parallel formulations of more general problems such as the quadratic zero-one programming problem (using distributed and shared memory machines) are discussed in [93, 94].

6 Speedup Anomalies in Parallel Formulations of Search Algorithms.

In parallel search algorithms, 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.

A number of researchers have observed and analyzed the existence of speedup anomalies. Using P processing elements, an acceleration anomaly manifests itself in the form of a speedup greater than P [44, 67, 56, 74, 85, 88]. A speedup of less than P attributed to excess work done by the parallel formulation compared to the sequential formulation is called termed as a deceleration anomaly [44, 67, 81, 43, 125]. The speedup obtained in some cases may actually be less than 1. This is termed as detrimental anomaly [74, 44, 67, 81].

Lai and Sahni [67] present early work on quantifying the existence of speedup anomalies in best-first search. They show that given an instance of a problem, there exists a number k such that there is little advantage in expanding more than k nodes in parallel. For various instances of the 0/1 knapsack problem and a range of number of processors, they experimentally derive the value of k. They also show that for heuristic functions with specific characteristics, anomalies can be avoided. This was shown for cases when the parallel time on P processors was compared with the sequential execution time. However, under identical conditions, anomalies may occur while comparing performance using P_2 processors against that using P_1 processors where $P_2 > P_1 > 1$. They also show that for a one-to-one lower bound function, anomalies occur when $1 < P_1 < P_2 < 2(P_1 - 1)$. Lai and Sprague [69] further show the existence of anomalies even when $1 < P_1 < P_2 < \frac{1}{2}P_1(P_1 - 1)$. Lai and Sprague [68] also present an analytical model and derive characteristics of the lower bound function for which anomalies are guaranteed not to occur as the number of processors is increased.

One reason for speedup anomalies is that many nodes in the search tree can have the same heuristic value, but some of these nodes may be closer to the goal than others. Hence, if a parallel algorithm happens to explore the better nodes first, then it can result in acceleration anomalies, and vice versa. Li and Wah [73, 74] and Wah et. al. [125] investigate dominance relations and heuristic functions and their effect on detrimental and acceleration anomalies. They show that detrimental anomalies in depth first, breadth first and specific classes of best first search can be avoided by augmenting the heuristic values with path numbers. This guarantees that different processors in parallel search expand the nodes of the tree in the same order as that followed by sequential search (provided it also uses the same augmented heuristic functions). Further, they show that acceleration anomalies may occur in one of the following cases: (a) when using breadth first or depth first search; (b) when some nodes have identical lower bounds in heuristic search; (c) where the dominance relation is inconsistent with the heuristic function; (d) when a suboptimal solution is sought ; or (e) when using multiple subproblem lists (lists which hold the nodes sorted according to their heuristic values).

Quinn and Deo [104] derive an upper bound on the speedup attainable by any parallel formulation of branch and bound algorithm using the best bound search strategy. They show that parallel branch and bound can achieve nearly linear or even superlinear speedups under appropriate conditions. Their analytical model indicates that acceleration anomalies are unlikely unless there are a large number of subproblems with the same lower bound as the solution cost.

Kumar and Rao [87, 110] analyze the average speedup in parallel DFS for two different types of models. 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[120], the 15-puzzle problem and the N-Queens problem[39]. (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 [12].

7 Parallel Dynamic Programming.

Parallel formulations of DP differ significantly for different serial DP algorithms. Li and Wah [46] present parallel formulations of classes of DP formulations. Search for a solution for a monadic serial DP formulation can be viewed as computing an optimal path in a multistage graph. The serial DP algorithm finds optimal paths for subproblems at successive stages using solutions to subproblems at preceding stages.

For example, consider the multistage graph shown in Figure 7. Define $C(i)$ as the cost of the least-cost path from node i to the goal node (node T). $C(i)$ is given by

$$C(i) = \min \ \{(c_{i,j} + C(j))/j \text{ is a node in the next stage } \}$$

This equation determines the least cost path by minimizing the sum of the cost of getting to any node in the subsequent level and from there to the goal node. Consider the cost of reaching goal node T from any node in stage Z. Here, $C(Z)$ is given by $\min_{i \epsilon Z}(c_{i,T})$. Similarly, the shortest path to goal node T from any node in layer Y is given by

$$C(Y) = \min_{i \epsilon Y}(c_{i,7} + c_{7,T}, c_{i,8} + c_{8,T}, c_{i,9} + c_{9,T}) \tag{2}$$

Equation 2 is similar to a matrix vector product defined in terms of a closed semiring (R, MIN, +, $+\infty$, 0). Here, 'MIN' corresponds to addition and '+' corresponds to multiplication in the matrix-vector product. Equation 2 can be rewritten as :

$$C(Y) = \begin{bmatrix} C(4) \\ C(5) \\ C(6) \end{bmatrix} = \begin{bmatrix} c_{4,7} & c_{4,8} & c_{4,9} \\ c_{5,7} & c_{5,8} & c_{5,9} \\ c_{6,7} & c_{6,8} & c_{6,9} \end{bmatrix} \begin{bmatrix} c_{7,T} \\ c_{8,T} \\ c_{9,T} \end{bmatrix} \triangleq c_{Y,Z} \cdot c_{Z,T}$$

The shortest path problem now becomes : $C(Y) = c_{Y,Z} * c_{Z,T}, C(X) = c_{X,Y} * (c_{Y,Z} * c_{Z,T})$ and $C(S) = c_{s,X} * (c_{X,Y} * (c_{Y,Z} * c_{Z,T}))$. If there are n nodes at a level, the cost of computing the 'C' vectors is clearly $O(n^2)$. The simplest parallel formulation would be to assign n/P nodes at each stage to each of the P processors. This is akin to vertical slicing of the matrix and the vector. Each processor computes the cost $C(i)$ of the entries assigned to it. Computation of each entry takes $O(n)$ time. This is followed by an all-to-all broadcast [16] during which solution costs of the subproblems at that stage are made known to all the processors. This operation takes $O(P)$ time on a wide range of architectures from a linear array to a hypercube. Since each processor has complete information about subproblem costs at preceding stages, no communication is needed other than the all-to-all broadcast and the next iteration can begin. If $P = n$, then computation for each stage can be performed in $O(n)$ time. Other parallel formulations can be derived using various parallel matrix-vector product algorithms [8, 16].

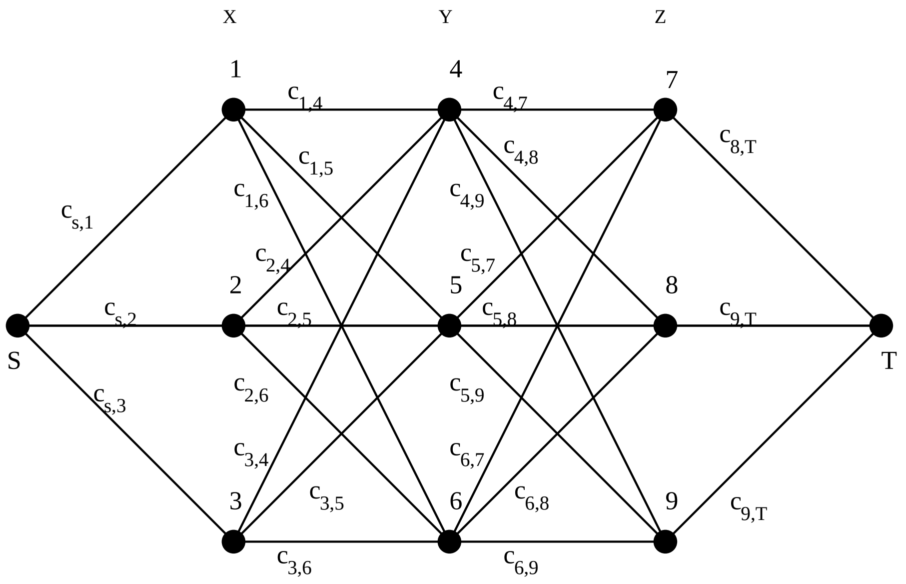

Figure 7: A three-stage multilevel graph with constant arc costs.

Li and Wah [46] show that solving monadic serial DP formulations is equivalent to multiplying a string of matrices. Note that the problem of multiplying two matrices has much more concurrency than the problem of matrix-vector multiplication. Li and Wah present a parallel formulation for monadic serial DP formulations based upon systolic arrays for matrix multiplications.

A large class of optimization problems can be formulated as *polyadic-nonserial* DP formulations.

Examples of these problems are: optimal triangularization of polygons, optimal binary search trees [20], the matrix parenthesization problem [20], and the CYK parser [7]. The serial complexity of all these problems is $O(n^3)$. A number of parallel formulations have been proposed in [42] that use $O(n)$ processors on a hypercube and solves the problem in $O(n^2)$ time. A systolic array algorithm has been proposed in [38] that uses $O(n^2)$ processing cells and solves the problem in $O(n)$ time. Faster algorithms for solving this problem are however not cost optimal. For example, the generalized method of parallelizing such programs described by Valiant et. al. [121] directly leads to formulations which run in $O(\log^2 n)$ time on $O(n^9)$ processors. Rytter [115] uses the parallel pebble game on trees to reduce the number of processors required to $O(n^6/\log n)$ for a CREW-PRAM model. This algorithm also runs in $O(\log^2 n)$ time on $O(n^6)$ processors connected in a perfect shuffle network or a hypercube. For stronger models of computation like a CRCW, the runtime of this algorithm can be reduced to $O(\log n)$ on $O(n^6/\log n)$ processors. Huang et. al. [40] present a similar algorithm for CREW-PRAM models which run in $O(\sqrt{n}\log n)$ time on $O(n^{3.5}\log n)$ processors.

Guibas, Kung and Thomson [38] have developed a systolic algorithm for the parenthesization problem. Their algorithm uses $\frac{n(n+1)}{2}$ processing elements (cells) connected as a two dimension systolic array (TSA). Karypis and Kumar [50] analyze three different mappings of the systolic algorithm presented by Guibas et. al. [38] and experimentally evaluate them in the context of the matrix multiplication parenthesization problem on an nCUBE2. Huang et. al. [40] present an algorithm for solving a generalized parallel formulation of DP on a CREW-PRAM. This formulation can be applied to a number of problems. Dantas et. al. [25] use vectorized formulations of DP for the Cray to solve optimal control problems.

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. [71] use the divide-and-conquer strategy for parallelizing the DP algorithm for the 0/1 knapsack problem on a MIMD distributed memory computer. They demonstrate experimentally that it is possible to obtain linear speedup for large instances of the problem provided enough memory is available.

8 Concluding Remarks.

Multiprocessors offer the potential for effectively speeding up many computationally intensive applications in discrete optimization. A significant amount of work has been done to tap this potential and a lot still remains to be done. A number of DOP formulations have been implemented only for some abstract problems for specific architectures. More experimentation is necessary, particularly on production problems. Given that current parallel computers are much more cost effective than supercomputers, this can can allow us to solve larger instances of practical problems such as scheduling, planning, and various other optimization problems.

More work is needed in deriving parallel formulations of serial DP formulations. A number of new DP algorithms have been proposed which make use of such problem characteristics as sparsity, convexity and concavity [14, 35, 4, 132, 27]. Parallel formulations of many of these need to be investigated.

The scalability of parallel best-first search formulations are not well understood. Current analytical results impose very loose bounds on the scalability. Given that scalability is an important measure for characterizing parallel systems, it is important that a comprehensive analytical study of scalability of parallel best first search be undertaken. Further, the single biggest deterrent to the use of best first search is the memory requirement. A number of parallel formulations have been proposed for best-first search which are designed to operate under limited memory constraints. The nature and performance of these schemes is not clearly understood and more work needs to be done to address limited memory variants of best-first search.

A number of programming environments have been developed for implementing parallel search. These include DIB [31], Chare-Kernel [118], MANIP [128] and PICOS [113]. Work on these programming environments is of prime importance as it has a direct impact on production level software development. Research efforts have also been devoted to developing specialized architectures for implementation of search techniques. These include the PIM [33], MANIP [128] and the Ring Machine [83]. The importance of the problem and the specific goals of these research projects warrants that such efforts be continued.

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. Aggarwal and M.M. Klawe. Applications of generalized matrix searching to geometric algorithms. *Discrete Applied Math.*, 27:3–24, 1990.

[5] D. P. Agrawal, V. K. Janakiram, and Ram Mehrotra. A randomized parallel branch and bound algorithm. In *Proceedings of International conference on Parallel Processing*, 1988.

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

[7] A.V. Aho and J.D. Ullman. *The Theory of Parsing, Translation and Compiling, Vol. 1, Parsing.* Englewood Cliffs, NJ Prentice Hall, 1972.

[8] S. G. Akl. *The Design and Analysis of Parallel Algorithms.* Prentice-Hall, 1989.

[9] S. G. Akl, D. T. Bernard, and R. J. Jordan. Design and implementation of a parallel tree search algorithm. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-4:192–203, 1982.

[10] 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.

[11] 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.

[12] 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.

[13] G. M. Baudet. The design and analysis of algorithms for asynchronous multiprocessors. Technical Report CMU-CS-78-116, Carnegie Mellon University, 1978. PhD. Dissertation.

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

[15] U. Bertele and F. Brioschi. *Nonserial Dynamic Programming.* Academic Press, NY, 1972.

[16] D. P. Bertsekas and J.N. Tsitsiklis. *Parallel and Distributed Computation: Numerical Methods.* Prentice-Hall, 1989.

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

[18] 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.

[19] E. Charniak and D. McDermott. *Introduction to Artificial Intelligence.* Addison Wesley, 1985.

[20] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms.* MIT Press, McGraw-Hill, 1990.

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

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

[23] William Dally. *A VLSI Architecture for Concurrent Data Structures.* Kluwer Academic Publ, Boston, Massachusetts, 1987.

[24] 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.

[25] 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.

[26] E.W. Dijkstra, W.H. Seijen, and A.J.M. Van Gasteren. Derivation of a termination detection algorithm for a distributed computation. *Information Processing Letters*, 16–5:217–219, 1983.

[27] D. Eppstein, Z. Ghalil, and R. Giancarlo. Speeding up dynamic programming. In *Proc. 29th IEEE Symp. Found. Computer Science*, pages 488–496, 1988.

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

[29] M. Evett, James Hendler, Ambujashka Mahanti, and Dana Nau. Pra*: A memory-limited heuristic search procedure for the connection machine. In *Proceedings of the third symposium on the Frontiers of Massively Parallel Computation*, pages 145–149, 1990.

[30] 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.

[31] Raphael A. Finkel and Udi Manber. DIB - a distributed implementation of backtracking. *ACM Trans. of Progr. Lang. and Systems*, 9 No. 2:235–256, April 1987.

[32] Roger Frye and Jacek Myczkowski. Exhaustive search of unstructured trees on the connection machine. In *Thinking Machines Corporation Technical Report*, 1990.

[33] 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.

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

[35] Zvi Ghalil and Kunsoo Park. Dynamic programming with convexity, concavity and sparsity. *Theoretical Computer Science*, 92:49–76, 1992.

[36] 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.

[37] Ananth Y. Grama, V. Kumar, and P. Pardalos. Parallel processing of discrete optimization problems. In *Encyclopaedia of Microcomputers.* Marcel Dekker Inc., New York, 1992.

[38] L. J. Guibas, H. T. Kung, and C. D. Thompson. Direct VLSI Implementation of Combinatorial Algorithms. In *Proceedings of Conference on Very Large Scale Integration, California Institute of Technology*, pages 509–525, 1979.

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

[40] S. S. Huang, H. Liu, and V. Vishwanathan. A sub-linear parallel algorithm for some dynamic programming problems. In *Proc. Internationsl Conf. on Parallel Processing*, pages III–261 – III–264, 1990.

[41] M. M. Huntbach and F. Warren Burton. Alpha-beta search on virtual tree machines. *Information Science*, 44, 3-17, 1988.

[42] Oscar H. Ibara, Ting-Chuen Pong, and Stephen M. Sohn. Parallel Recognition and Parsing on the Hypercube. *IEEE Transactions on Computers*, 40(6):764–770, June 1991.

[43] M. Imai, Tateizumi Y., Y. Yoshida, and T. Fukumura. The architecture and efficiency of DON: A combinatorial problem orianted multicomputer system. In *Proceedings of International Conference on Distributed Computing Systems*, 1984.

[44] M. Imai, Y. Yoshida, and T. Fukumura. A parallel searching scheme for multiprocessor systems and its application to combinatorial problems. In *IJCAI*, pages 416–418, 1979.

[45] Keki B. Irani and Yi fong Shih. Parallel a* and ao* algorithms: An optimality criterion and performance evaluation. In *Proceedings of International conference on Parallel Processing*, pages 274–277, 1986.

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

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

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

[49] Richard Karp and Yanjun Zhang. A randomized parallel branch-and-bound procedure. In *Proceedings of the ACM Annual Symposium on Theory of Computing*, pages 290–300, 1988.

[50] George Karypis and Vipin Kumar. Efficient Parallel Formulations for Some Dynamic Programming Algorithms. Technical Report 92-59, University of Minnesota, Computer Science Departement, October 1992.

[51] 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.

[52] Kouichi Kimura and Ichiyoshi Nobuyuki. Probabilistic analysis of the efficiency of the dynamic load distribution. In *Proceedings of 1991 Distributed Memory and Concurrent Computers*, 1991.

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

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

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

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

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

[58] 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.

[59] Vipin Kumar and Anshul Gupta. Analyzing scalability of parallel algorithms and architectures. Technical report, TR-91-18, Computer Science Department, University of Minnesota,

June 1991. A short version of the paper appears in the Proceedings of the 1991 International Conference on Supercomputing, Germany, and as an invited paper in the Proc. of 29th Annual Allerton Conference on Communuication, Control and Computing, Urbana,IL, October 1991.

[60] Vipin Kumar and Laveen Kanal. A general branch-and-bound formulations for understanding and synthesizing and/or tree search procedures. *Artificial Intelligence*, 21:179–198, 1983.

[61] 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.

[62] 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.

[63] Vipin Kumar, Dana Nau, and Laveen Kanal. General branch-and-bound formulation for and/or graph and game tree search. In Laveen Kanal and Vipin Kumar, editors, *Search in Artificial Intelligence*. Springer-Verlag, New York, 1988.

[64] 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.

[65] 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.

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

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

[68] T. H. Lai and A. Sprague. Performance of parallel branch - and - bound algorithms. *IEEE Transactions on Computers*, C-34, No. 10, October 1985.

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

[70] E. L. Lawler and D. Woods. Branch-and-bound methods: A survey. *Operations Research*, 14, 1966.

[71] 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.

[72] 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.

[73] Guo-Jie Li and Benjamin W. Wah. Computational efficiency of parallel approximate branch-and-bound algorithms. In *International Conf on Paralle Processing*, pages 473–480, 1984.

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

[75] 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.

[76] Gary Lindstrom. The key node method: A highly parallel alpha-beta algorithm. Technical Report 83-101, Computer Science Department, Univ. of Utah, Salt Lake City, 1983.

[77] A. Mahanti and C. Daniels. Simd parallel heuristic search. *To appear in Artificial Intelligence*, 1992.

[78] 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.

[79] T.A. Marsland and M. Campbell. Parallel search of strongly ordered game trees. *Computing Surveys*, 14:533–551, 1982.

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

[81] Joseph Mohan. Experience with two parallel programs solving the traveling salesman problem. In *Proceedings of International conference on Parallel Processing*, pages 191–193, 1983.

[82] B. Monien, R.Feldmann, P.Mysliwietz, and O.Vornberger. Parrallel game tree search by dynamic tree decomposition. In Vipin Kumar, P. S. Gopalakrishnan, and Laveen Kanal, editors, *Parallel Algorithms for Machine Intelligence and Vision*. Springer-Verlag, New York, 1990.

[83] B. Monien and O. Vornberger. The ring machine. Technical report, Univ. of Paderborn, FRG, 1985. Also in Computers and Artificial Intelligence, 3(1987).

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

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

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

[87] 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.

[88] 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.

[89] H. Ney. Dynamic programming as a technique for pattern recognition. In *Proc. 6th Intl. Conf. Pattern Recognition*, pages 1119 – 1125, Oct. 1982.

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

[91] C. H. Papadimitriou and K. Steiglitz. *Combinatorial Optimization : Algorithms and Complexity.* Prentice Hall Inc., Englewood Cliffs, NJ, 1982.

[92] 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).

[93] 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.

[94] 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.

[95] Srinivas Patil and Prithviraj Banerjee. A parallel branch and bound algorithm for test generation. In *IEEE Transactions on Computer Aided Design*, Vol. 9, No. 3, March 1990.

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

[97] 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.

[98] C. Powley, R. Korf, and C. Ferguson. Ida* on the connection machine. *To appear in Artificial Intelligence*, 1992.

[99] Curt Powley, Chris Ferguson, and Richard Korf. Parallel heuristic search: Two approaches. In Vipin Kumar, P. S. Gopalakrishnan, and Laveen Kanal, editors, *Parallel Algorithms for Machine Intelligence and Vision*. Springer-Verlag, New York, 1990.

[100] Michael J. Quinn. *Designing Efficient Algorithms for Parallel Computers*. McGraw Hill, NewYork, 1987.

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

[102] Michael J. Quinn and Narsingh Deo. Parallel graph algorithms. *ACM Computing Surveys*, 16,No 3:319–348, September 1984.

[103] Michael J. Quinn and Narsingh Deo. An upper bound for the speedup of parallel branch-and-bound algorithms. Technical report, Purdue University, Pullman, Washington, 1984.

[104] Michael J. Quinn and Narsingh Deo. An upper bound for the speedup of parallel branch-and-bound algorithms. *BIT*, 26,No 1, March 1986.

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

[106] Finkel R.A. and J.P. Fishburn. Improved speedup bounds for parallel alpha-beta search. *Pattern Analysis and Machine Intelligence*, 5,No.1:89–92, 1983.

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

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

[109] Nageshwara Rao. Parallel processing of heuristic search. Technical report, Univ. of Texas at Austin, PhD dissertation, July 1990.

[110] 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.

[111] Shie rei Huang and Larry Davis. A tight upper bound for the speedup of parallel best-first branch-and-bound algorithms. Technical report, Center for Automation Research, Univ. of Maryland, College Park, Maryland 20742, 1987.

[112] 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.

[113] C. Renolet, M. Diamond, and J. Kimbel. Analytical and heuristic modeling of distributed algorithms. Technical report, Tech. Rep. E3646, FMC Corporation, Advanced Systems Center, Minneapolis, MN, July 1989.

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

[115] W. Rytter. Efficient parallel computations for dynamic programming. *Theoretical Computer Science*, 59:297–307, 1988.

[116] 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.

[117] Charles Seitz. The cosmic cube. *Communications of the ACM*, 28-1:22–33, 1985.

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

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

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

[121] L. Valiant, S. Skyum, S. Berkowitz, and C. Rackoff. Fast parallel computation of polynomials using few processors. *SIAM J. of Comput.*, 12(4):641–644, 1983.

[122] Nageshwara Rao Vempaty, Vipin Kumar, and Richard Korf. Analysis of heuristic search algorithms. In *Proceedings of the National Conf. on Artificial Intelligenc e (AAAI-91)*, 1991.

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

[124] Olivier Vornberger. Load balancing in a network of transputers. In *2nd International Workshop on Distributed Parallel Algorithms*, 1987.

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

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

[127] 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.

[128] 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.

[129] 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.

[130] D. White. *Dynamic Programming*. Oliver and Boyd, Edinburgh, England, 1969.

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

[132] F. F. Yao. Efficient dynamic programming using quadrangle inequalities. In *Proc. 12th ACM Symp. on Theory of Computing*, pages 429–435, 1980.