ORSA Journal on Computing

Publication details, including instructions for authors and subscription information:
http://pubsonline.informs.org

Parallel Search Algorithms for Discrete Optimization Problems

Ananth Grama, Vipin Kumar,

ORSA Journal on Computing
Vol. 7, No 4, Fall 1995

0899-1499/95/0704-0365 $01 25
© 1995 INFORMS

Parallel Search Algorithms for Discrete Optimization Problems

ANANTH GRAMA / *Department of Computer Science, University of Minnesota, 4-192, 200 Union Street S.E., Minneapolis, MN 55455; Email: ananth@cs.umn.edu*

VIPIN KUMAR / *Department of Computer Science, University of Minnesota, 4-192, 200 Union Street S.E., Minneapolis, MN 55455; Email: kumar@cs.umn.edu*

(Received: January 1993; revised June 1994, April 1995; accepted: June 1995)

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. Availability of parallel computers has created substantial interest in exploring parallel formulations of these graph and tree search methods. This article provides a survey of various parallel search algorithms such as Backtracking, IDA*, A*, Branch-and-Bound techniques and Dynamic Programming. It addresses issues related to load balancing, communication costs, scalability and the phenomenon of speedup anomalies in parallel search.

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,* $f(x_{opt}) = \min\{f(x)/x \in X\}$. 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 $x \in S$, 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,[85] dynamic programming,[12] and heuristic search[115,56] methods use the structure of these graphs to solve DOPs without searching the set X exhaustively.[77]

Since DOPs fall into the class of NP-hard problems,[34] one may argue that there is no point in applying parallel processing to them. This is because the NP-hard nature of these problems would strongly suggest that 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.[140,154]. Furthermore, when near-optimal or sub-optimal solutions are acceptable, algorithms exist for finding desired solutions in polynomial time (*e.g.*, for certain problems, approximate branch-and-bound algorithms are known to run in polynomial time[152]). In such 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, it is likely that parallel processing is the only way to obtain acceptable performance. Finally, for problems where optimal solutions are highly desirable, parallel search techniques have been effective for solving moderately difficult real world problems (*e.g.*, VLSI floorplan optimization[81]).

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 the current cost of large mainframes. This technology has created substantial interest in exploring the use of parallel processing for search based applications.[73,72,22,68,69,71,117,29,126,150,149,17]

This article provides a survey of parallel algorithms for solving DOPs. Section I reviews serial algorithms for solving DOPs. Section 2 discusses general algorithmic issues in parallel formulations. Section 3 discusses parallel formulations of depth-first search. Section 4 discusses parallel formulations of best-first search. Section 5 addresses issues of speedup anomalies in parallel search. Section 6 discusses parallel formulations of dynamic programming. Section 7 contains concluding remarks.

1. Sequential Algorithms for Solving Discrete Optimization Problems

Here we provide a brief overview of sequential search algorithms. For detailed descriptions, see [110, 115, 56].

Subject classifications: Optimization, search algorithms, parallel computing.

1.1. Depth-First Search

Depth-first search is a name commonly used to refer to a class of search techniques used for solving DOPs that can be characterized as graph search problems. The graph has an initial node and one or more goal nodes. Paths between the initial and goal nodes correspond to members x of set X. The cost of the path is $f(x)$. Depth-first search of a graph 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. Search terminates when the desired solution is found, or it is determined that no solution exists. The depth-first search technique is illustrated in Figure 1. If a desirable solution can be any path between the initial and a goal node, then search terminates when the first solution path is found. This is the case illustrated in Figure 1. If the objective is to find a solution path that optimizes the value of function f, then search continues until such an element has been found. In this case, the condition in line 12 of Figure 1 should terminate the repeat-until loop only when the stack is empty. Further, a variable needs to be used to store the current best solution. This variable should be updated every time a better solution is found. A major advantage of depth-first search is that its storage requirement is linear in the depth of the search space being explored. 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. Simple backtracking is illustrated in Figure 1. In this program, no order is specified for installing the nodes generated by expanding a parent node. If we had an estimate of the quality of a node, then it is possible to install the newly generated nodes so that the most promising nodes are placed on top of the stack.

Depth-First Branch-and-Bound (DFBB) is a DFS algorithm that keeps track of the current best solution and uses it to prune search of spaces which are guaranteed to yield worse solutions. The value of the current best solution is initialized to an arbitrarily large value. DFBB 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. Figure 2 illustrates the depth-first branch-and-bound algorithm.

Iterative Deepening A* (IDA*) performs repeated cost-bounded DFS over the search space. For many problems, the search space may be very deep. Consequently, backtracking algorithms may get stuck searching deep along a branch of the tree which does not contain a solution when the solution may exist close to the root along another branch. One way of overcoming this problem is to limit the depth to which the tree is searched completely. If no solution is found, the depth bound is increased and the search is repeated. This algorithm would ensure that the solution found is closest to the root (in terms of the number

```
1.   program depth_first_search
2        select initial node and place on stack
3        repeat
4.       begin
5            select node from the top of the stack
6            if (selected node is not the solution)
7.           begin
8                generate successors (if any) of selected node;
9                install generated successors into the stack.
10           end
11       end
12       until (selected node is the solution) or
         (stack is empty)
13.  end_program
```

Figure 1. The sequential depth-first search algorithm.

```
1    program depth_first_branch_and_bound
2        current_best_solution = infinity.
3        select initial node and place on stack.
4        repeat
5        begin
6            select node from the top of the stack.
7            if (selected node is not the solution)
8            begin
9                evaluate best possible solution this node can lead to.
10               assign this value to node_bound.
11               if (node_bound < current_best_solution)
12               begin
13                   generate successors (if any) of selected node;
14                   install generated successors into the stack.
15               end_if
16           end_if
17           else
18               if (cost_of_solution < current_best_solution)
19                   current_best_solution = cost_of_solution.
20       end_repeat
21       until (stack is empty)
22   end_program
```

Figure 2. The depth-first branch-and-bound algorithm.

of edges). Figure 1 can be modified slightly to perform iterative deepening search. The condition in the repeat-until loop is changed so that search is performed until the cutoff depth has been reached, or the stack is empty, or a solution has been found. In the latter two cases, search is terminated. In the first case, the cutoff depth is increased and the depth-first-search procedure is called again.

It is also possible to use cost (instead of depth) to bound the depth of iterative DFS. IDA* performs repeated cost-bounded DFS. 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. IDA* uses an estimate of the lowest cost of the solution $h(x)$ that node x can yield. The cost bound is initialized to $h(x_0)$, where x_0 is the root node. After each iteration, the cost bound is set to the smallest h value of all nodes that exceeded the cost bound in the previous iteration.

It might appear that IDA* performs a lot of redundant work in successive iterations. But for many interesting problems, the redundant work performed is minimal and the algorithm finds an optimal solution.[66] IDA* search procedure can be derived from the iterative deepening depth-first search procedure by replacing the depth bound by the cost bound.

1.2. 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 best-first search algorithm is illustrated in Figure 3. 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.

1.3. 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.[12,13,109,153,86,4] DP views a problem as a set of interdependent subproblems. It solves subproblems and uses the results to solve larger subproblems until the entire problem is solved. The solution to a subproblem is expressed as a function of solutions to one or more subproblems at the preceding levels. A special class of problems solved using DP are characterized by the *Principle of Optimality defined* by Bellman and Dreyfus.[12] 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. A general form of DP and a general version of the principle of optimality is given by Kumar and Kanal in [77].

```
program best_first_search
    select initial node and insert into list
    repeat
    begin
        select node from the head of the list
        if (selected node is not the solution)
        begin
            generate successors (if any) of selected node.
            if (successor doesnt exist in list)
                insert generated successors into the list.
            else
                insert generated successors only if they have
                a better $f$ value and delete other instances.
            sort list according to the $f$ values of the nodes.
        end
    end
    until (selected node is the solution) or
    (list is empty)
end_program
```

Figure 3. The sequential best-first search algorithm.

It is possible to classify various DP formulations on the basis of the dependencies between subproblems and the composition function. If the solution to a subproblem is composed of a single subproblem from one of the previous levels, then the DP formulation is termed monadic, else it is termed polyadic. The dependencies between subproblems in a DP formulation can be represented by a directed graph. Each node in the graph represents a subproblem. A directed edge from node i to node j indicates that the solution to the subproblem represented by node i is used to compute the solution to the subproblem represented by node j. If the graph is acyclic, then the nodes of the graph can be organized into levels such that subproblems at a particular level depend only on subproblems at previous levels. In this case, the DP formulation can be categorized as follows. If subproblems at all levels depend only on the results at the immediately preceding levels, the formulation is called a serial DP formulation; otherwise, it is called a nonserial DP formulation. Based on these two classification criteria, it is possible to classify most DP formulations as serial-monadic, serial-polyadic, nonserial-monadic, and nonserial-polyadic. This classification, first proposed by Li and Wah[91] is important because it helps in designing parallel formulations for these algorithms. Different formulations of DP have been presented.[77,59,91,35,2,156,26]

2. 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. Since different algorithms for solving the same problem may take different times, T_s is defined as the time taken by the fastest sequential algorithm 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. Parallel runtime T_p is a function of the hardware related parameters and the algorithm used. 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 ($E = S/P$).

Search Overhead: Parallel search algorithms incur overhead from several sources. These include communication overhead, idle time due to load imbalance, and contention for shared data structures. Thus, if both the sequential and parallel formulations of an algorithm do the same amount of work, the speedup of parallel search on P processors is less than P. However, the amount of work done by a parallel formulation is often different from that done by the corresponding sequential formulation because they may explore different parts of the search space.

Let W be the amount of work done by a single processor and W_P be the total amount of work done by P processors. The **search overhead factor** of the parallel system is defined as the ratio of the work done by the parallel formulation to that done by the sequential formulation, or W_P/W. Thus, the upper bound on speedup for the parallel system is given by $P \times (W/W_P)$. The actual speedup, however, may be less due to other parallel processing overhead. In most parallel search algorithms, the search overhead factor is greater than one. However, in some cases, it may be less than one, leading to superlinear speedup. If the search overhead factor is less than one on the average, then it indicates that the serial search algorithm is not the fastest algorithm for solving the problem.

Communication versus Computation Tradeoffs: Parallel search involves the classical communication versus computation tradeoff. As we have 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.

In DFBB, for example, a sequential search algorithm may be able to use information gleaned from one branch of the search to prune search in subsequent branches. If these branches are searched in parallel with no exchange of information, this type of information may not be automatically available to each processor. By exchanging pruning information, parallel searches may regain some of the efficiency of the sequential algorithm in terms of search overhead, but with higher communication overhead.

Architecture Related Issues: Suitability of a parallel search formulation depends strongly on the features of the underlying architecture. Parallel computers can be thought of as belonging to one of two classes: SIMD (Single Instruction, Multiple Data streams), or MIMD (Multiple Instruction, Multiple Data streams) machines.[74] 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 trade-offs 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 is a message passing computer or a shared address space computer.[74] Shared address space parallel computers have global memory that can be directly addressed by all processors. In message passing 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 address space machines. Direct formulations of these search techniques on message passing computers often results in a higher communication cost for accessing the shared data structures. Often, this can be circumvented by distributing the data structure over many processors, which may result in additional search overhead. Shared address space computers are not very scalable, particularly, if the entire global memory is equally accessible to all processors. Consequently, it is not possible to construct such machines with a large number of processors. Therefore, most shared address space computers have an underlying distributed memory architecture with hardware support for global addressing. In such systems, access to global memory is much slower than local memory. Hence, the distributed data structures necessary for message passing computers are useful even here. All these factors influence the selection of a parallel formulation of a search technique.

Scalability and Scalable Algorithms: If a parallel algorithm is used to solve a problem instance of a fixed size, then the speedup does not increase linearly as the number of processors P is increased. The speedup curve tends to saturate. In other words, the efficiency drops with increasing number of processors. This phenomenon is true for all parallel systems, and is often referred to as Amdahl's law. The reason for this is that every algorithm has a certain sequential component. This sequential component cannot be reduced, and consequently as the number of processors increases, it causes the efficiency to drop.

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[37,76,81,74] and is called the isoefficiency function.

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

3. Parallel Depth-First-Search Algorithms

The 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. It would therefore seem that by statically assigning nodes in a tree to processors, it is possible to derive parallel formulations. However, for most applications, 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. This is illustrated in Figure 4. The unstructured tree shown in the figure when partitioned among two and four processors yields subtrees with widely differing number of nodes at each processor. This results in significant load imbalances. 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.[137,139,75,81,107,125,62,3,127,51,52,145,57,58]

Load balancing can be viewed as a two phase process: task partitioning and subtask distribution. We discuss these two processes in some detail here.

Task Partitioning: The two most commonly used techniques for task partitioning are *Stack Splitting* and *Node Splitting*. Stack splitting and node splitting[75,81] are illustrated in Figures 5(b) and (c), respectively. 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[33,125,28,119,58,57] require more work transfers compared to those using stack splitting.

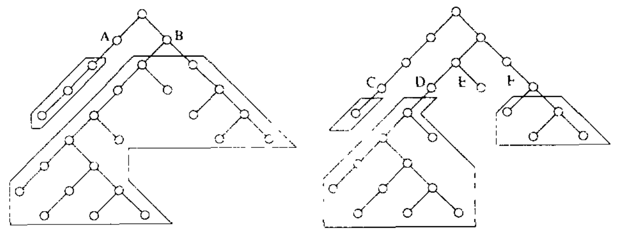

Figure 4. The unstructured nature of tree search and the imbalance resulting from static partitioning.

Subtask Distribution: Issues relating to when work available at a processor is split and how it is distributed are critical parameters of a parallel formulation. The former is referred to as the generation strategy and the latter is referred to as the transfer strategy.

Work available at a processor can either be split independent of a work request from another processor or on receiving a request from another processor. This can be used as a criteria for classifying subtask generation schemes. If work splitting is performed only when an idle processor (receiver) requests for work, it is called receiver initiated subtask generation. In such schemes, when a processor runs out of work, it generates a request for work. If the generation of subtasks is independent of the work requests from idle processors the scheme is referred to as a sender initiated subtask generation scheme.

Subtasks generated using the selected generation strategy must be distributed among other processors. Subtasks can be delivered to processors needing them, either on demand (*i.e.*, when they are idle)[33] or without demand.[125,139,137] The former is called sender initiated work transfer and the latter is called receiver initiated work transfer.

Based on the two generation strategies and two transfer strategies, it is possible to visualize four classes of subtask distribution schemes. It can be seen that receiver initiated subtask generation cannot be used in conjunction with sender initiated work transfer. We can therefore classify various parallel formulations of depth-first search into one of three classes. Table I presents instances of important parallel formulations in each of these classes.

3.1. Parallel Formulations of Backtracking

Let us consider a parallel formulation of DFS using stack splitting in conjunction with receiver initiated subtask generation and transfer. Parallel formulations of DFS in this class are discussed in [31, 105, 75, 81]. In this formulation, each processor searches a disjoint part of the tree in a depth-first fashion. When a goal is found, all of them quit. If the tree 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 6. When a processor runs out of

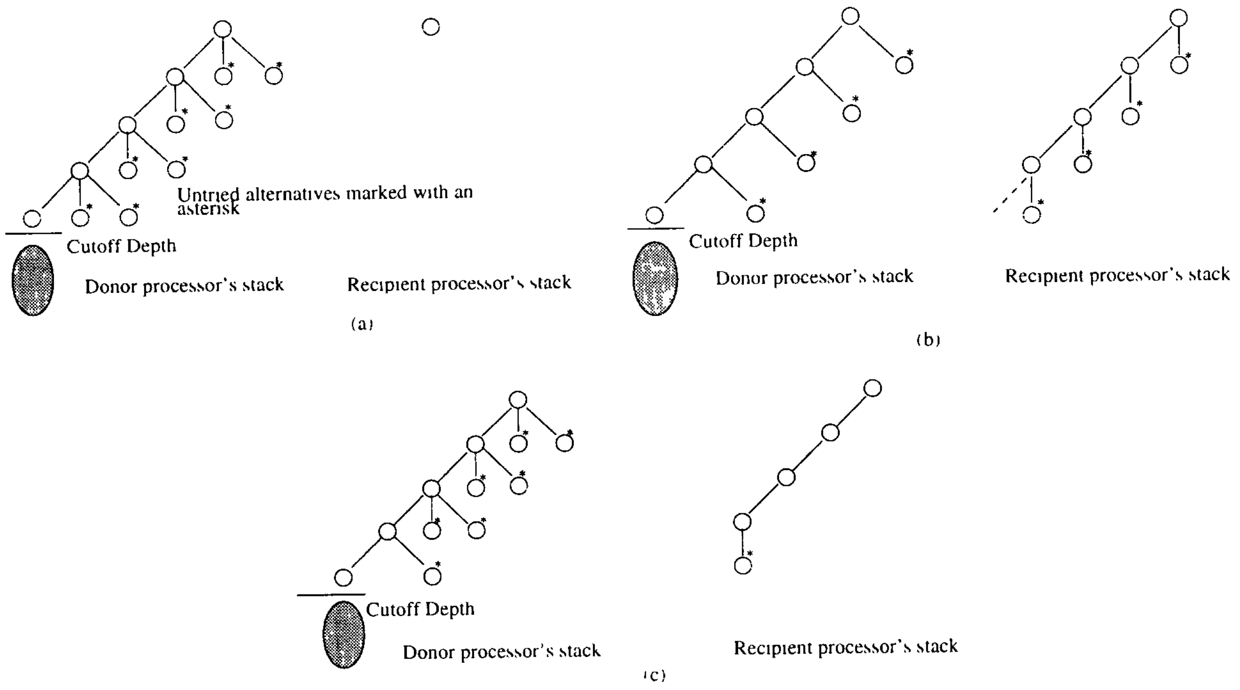
Untried alternatives marked with an asterisk
Cutoff Depth
Donor processor's stack
Recipient processor's stack
(a)
Cutoff Depth
Donor processor's stack
Recipient processor's stack
(b)
Cutoff Depth
Donor processor's stack
Recipient processor's stack
(c)

Figure 5. Illustration of the splitting mechanisms: (a) original tree; (b) stack splitting; (c) node splitting.

work, it 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 indicating that it does not have any work. This process continues until all processors go idle or a solution is found. In such a formulation, the whole tree is initially assigned to one processor and other processors are given null spaces. Subsequently, the tree 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 made so as to minimize overheads due to load imbalance, communication requests, and contention over any shared data structures. For example, consider two techniques, *random polling* and *global round robin* discussed in [75, 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 [75], the global round robin scheme minimizes the total number of work requests for a wide class of search trees. However, contention among processors for the global pointer causes a bottleneck.[75] 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[36]), the global round robin scheme would perform better than random polling.

Table I. Taxonomy of Parallel Formulations of DFS.

Subtask generation scheme →	Sender Initiated		Receiver Initiated
Work transfer strategy → Splitting strategy ↓	Sender Initiated	Receiver Initiated	Receiver Initiated
Node Splitting	Ranade[125] Shu and Kale[139]	Furuichi et al.[33]	Ferguson and Korf[28] Monien et al.[105] Finkel and Manber[31] Karp and Zhang[57,58]
Stack Splitting			Kumar et al.[81,128,75] Finkel and Manber[31]

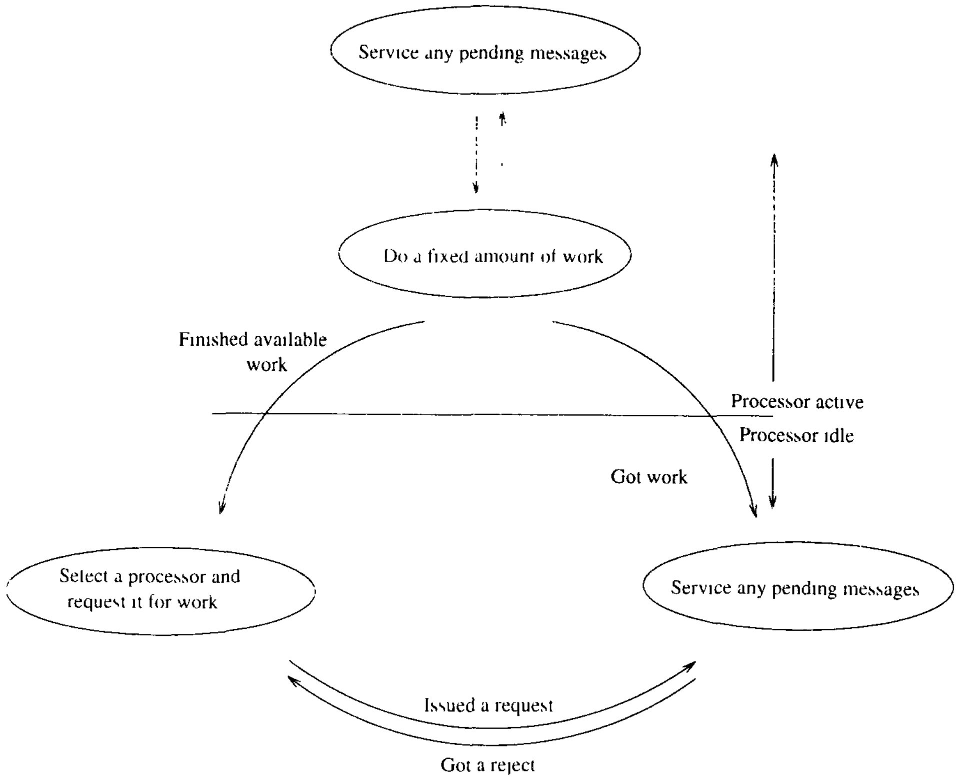
Service any pending messages
Do a fixed amount of work
Finished available work
Processor active
Processor idle
Got work
Select a processor and request it for work
Service any pending messages
Issued a request
Got a reject

Figure 8. State diagram describing the generic parallel DFS formulation for message massing architectures.

Parallel DFS using sender initiated subtask distribution has been proposed by a number of researchers.[28,119,33,139,125] These use different techniques for task generation and transfer as shown in Table I Furuichi et al. propose the single level and multi-level sender based scheme.[33] These schemes are illustrated in Figure 7. In the single level scheme, a designated processor called MANAGER generates a large number of subtasks and gives them one by one to the requesting processors on demand. This scheme therefore uses a receiver initiated transfer in conjunction with a sender initiated generation. Work splitting is based on node splitting. Subtasks are generated by traversing the search tree in depth-first fashion to a pre-determined cutoff depth, and giving nodes at that depth as subtasks. By increasing the cutoff depth, the number of subtasks can be increased. Processors request another subtask from the manager only after finishing the previous one. Hence, if a processor gets subtasks corresponding to large subtrees, it will request the MANAGER fewer number of times. Clearly, if the cutoff depth is large enough, this scheme will result in good load balance among processors. However, if the cutoff depth is too large, the subtasks given out to processors become small and the processors request the MANAGER more frequently. In this case, the MANAGER becomes the bottleneck. Hence, 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[64] present detailed scalability analysis of these schemes.

Ferguson and Korf[28] 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 tree. If a processor finishes searching its part of the tree sooner than others, it is reassigned to an unexplored part of another processors' tree. DTS has been shown to be quite effective in the context of parallel $\alpha - \beta$ search on a 32 processor hypercube.[28]

A number of techniques using randomized allocation have been presented in the context of parallel depth-first search.[125,139,137,53] In depth-first search of trees, the expan-

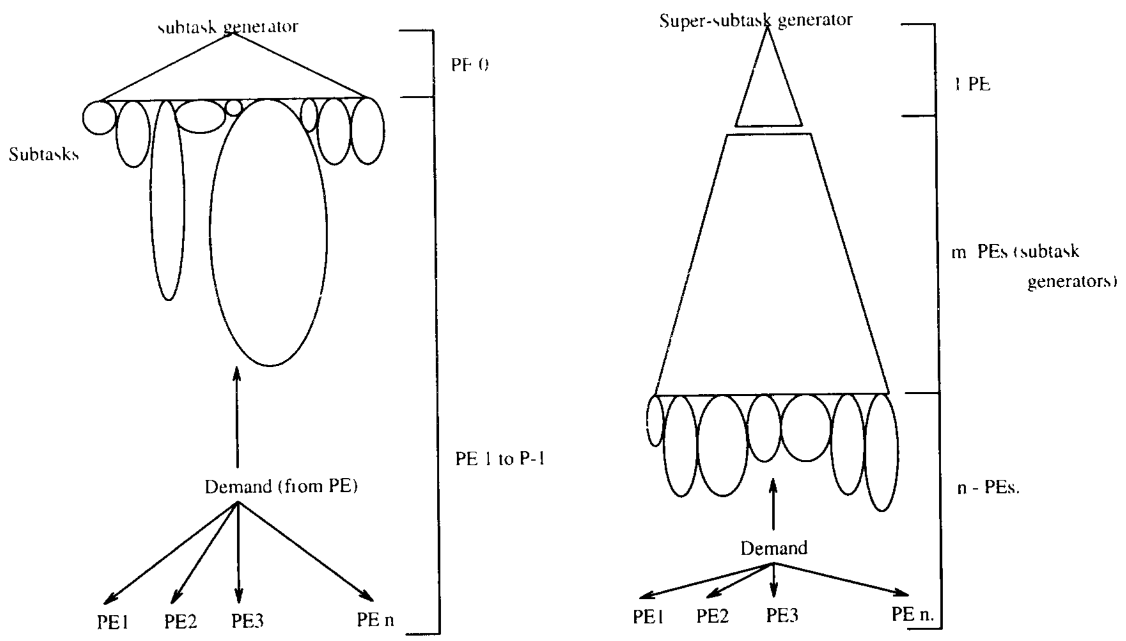

Figure 7. Single and multi-level subtask distribution schemes.

sion 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,[139] 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 many practical architectures (*e.g.*, mesh, ring). 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.[132,18] This also introduces additional inefficiency. Furthermore, 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[125] 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 neighboring 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 scalability of $O(P \log P)$ for a hypercube using the isoefficiency metric described in [81, 75]. This scheme localizes all communication, and thus has less communication overhead compared to Shu's scheme. 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 [139] 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.[128,139] It is, however, not clear whether mechanisms for effective granularity control can be derived for highly irregular state space trees. One possible method[128] 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[137] and Li and Wah[89] present parallel formulations 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. 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 in some cases 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.

A number of parallel formulations (based on receiver and sender initiated generation/transfer schemes) have been experimentally and analytically studied.[33,28,119,139,125,137,3,114,9,39,75] Finkel and Manber[31] present performance results for a number of problems such as the Traveling salesman problem and Knights tour for the Crystal multicomputer developed at the University of Wisconsin. Monien and Vornberger[107] show linear speedups on a network of transputers for a variety of combinatorial problems. Kumar et al. [75, 8, 9, 10] show linear speedups for problems such as 15 puzzle, tautology verification, and automatic test pattern generation 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[39,74,75,81,128] have investigated the scalability and performance of many of these schemes for a variety of architectures such as hypercubes, meshes, and networks of workstations.

3.2. 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 address space architecture, this can be done by maintaining a global best solution path. On a message passing computer, 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 near linear speedups for many problems and architectures. For example, Arvindam et al.[8] show near 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.

The classical $\alpha - \beta$ game tree search algorithm can also be viewed as a depth-first branch-and-bound algorithm.[70,78] 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 $\alpha - \beta$.[7,11,28,46,94,99,29,105,119,100] In particular, methods developed in [28, 105, 119] have shown reasonable speedups on dozens of processors (*e.g.*, in Monien et al.,[105] 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 [128]). 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.[7,30] 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.

3.3. Parallel Formulations of IDA*

There are two approaches to parallelizing IDA*. In one approach, different processors work on different iterations of IDA* independently.[71] This approach is not very suitable for finding optimal solutions. A solution found by a processor is provably optimal only if all the other processors have also exhausted search space until that iteration and no better solution has been found. This is a consequence of the fact that a processor working with a smaller cost bound may find a solution with a lower cost after another processor working with a higher cost bound finds one. Another more effective approach is to execute each iteration of IDA* via parallel DFS.[80,81,128] Since all processors work with the same cost bound, each processor stores this value locally and performs DFS on its own part of the tree. 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 address space architectures (*e.g.*, Sequent, BBN Butterfly), a globally shared variable is used to keep track of the number of idle processors. On message passing computers (*e.g.*, the Intel Hypercube, nCUBE), algorithms such as Dijkstra's token termination detection algorithm[24] can be used. This approach has been used by a number of researchers and tested in the context of different problems and architectures.[129,80,55,118,95] Clearly, performance and scalability issues for this parallel formulation of IDA* are similar to those of the parallel formulations of DFS algorithm.

3.4. Parallel DFS on SIMD computers

As mentioned in Section 2, 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 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 CM2^{TM} can also be used to implement parallel DFS algorithms effectively. Frye and Myczkowski[32] present an implementation of DFS on the CM2 for a block puzzle. Powley et al.[118] and Mahanti and Daniels[95] present parallel IDA* for solving the 15 puzzle problem on CM2^{TM}. Powley's and Mahanti's formulations use different triggering and redistribution mechanisms. Both schemes, however, report similar results for the 15 puzzle. Karypis and Kumar[62] present a new load balancing technique that 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.

4. Parallel Best-First Search

A number of researchers have investigated parallel formulations of A*/best-first branch-and-bound (B & B) algorithms.[71,79,88,107,121,45,146,151,127,38,1,16,20,65,93,134,111,112,113,116,142,102,144,101,97] An important component of A*/B & B algorithms is the priority queue that 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.[104,106,107,122,121,123,44,151] Given P processors, the simplest parallel strategy is to let each parallel processor work on one of the P current best node in the OPEN list. This strategy is called the centralized strategy[104,123,44] because each processor gets work from the global OPEN list. As discussed in [50], this strategy should not result in much redundant search. Figure 8 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.[88,120]

(2) Since OPEN will be accessed by all the processors frequently, it will have to be maintained in globally accessible memory that is easily accessible to all the processors. Hence message passing architectures are effectively ruled out. Even on shared address space computers, 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.[44] Note that the access to CLOSED does not cause contention, as different 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 strategies for communication 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, and it is better to treat it like a tree search problem. But for some other problems (*e.g.*, finding the shortest path in a reasonably well connected graph), unfolded graphs can be

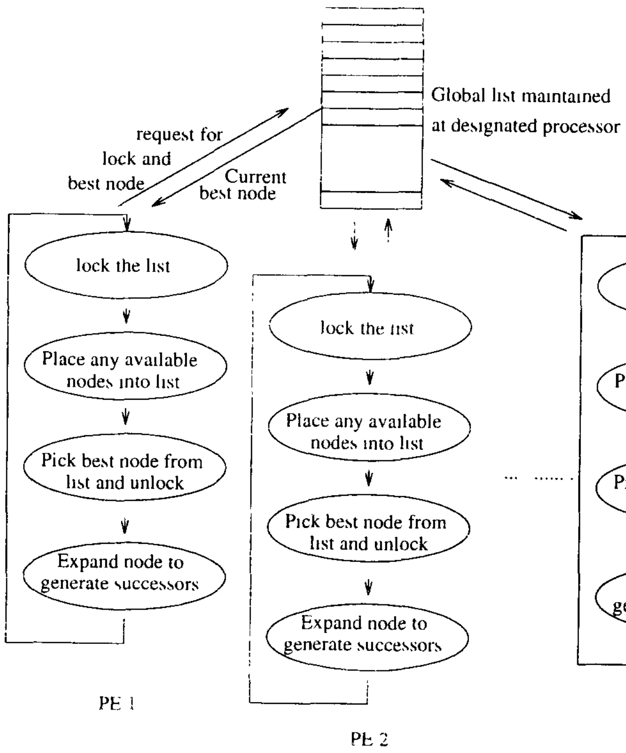
Global list maintained
at designated processor
request for
lock and
best node
Current
best node
lock the list
Place any available nodes into list
Pick best node from list and unlock
Expand node to generate successors
lock the list
Place any available nodes into list
Pick best node from list and unlock
Expand node to generate successors
lock the list
Place any available nodes into list.
Pick best node from list and unlock.
Expand node to generate successors
PE 1
PE 2
PE n

Figure 8. General schematic for parallel best-first search using a centralized strategy.

exponentially bigger than the original graphs.[67] We will discuss communication strategies for tree and graph search separately.

4.1. Communication Strategies for Parallel Tree Search Formulations

Parallel formulations of best-first search must address two issues:

- Quantitative load balancing: This ensures that all processors expand approximately equal number of nodes in the tree. This objective minimizes load imbalance and idling overhead.
- Qualitative load balancing: The objective of this is to minimize the amount of non-essential work (work that is not done by the sequential counterpart). To realize this objective, the promising nodes have to be evenly distributed among various processors.

A number of communication strategies have been proposed by researchers that address these two objectives.

4.1.1. Quantitative Load Balancing Strategies

Random Communication Strategy: In the random communication strategy, each processor periodically puts some of its best nodes into the OPEN list of a randomly selected processor.[147,21,79] This ensures that if some processor has a good part of the search space, then others get a part of it. Although the scheme is designed to explicitly address quantitative load balancing, it also implicitly ensures a measure of qualitative load balance. This strategy can be easily implemented on message passing computers with a high bisection width (such as those using hypercube and fat tree interconnection networks) as well as shared address space computers such as the Butterfly. If the frequency of transfer is high, then the redundancy factor can be very small; otherwise it can be large. The choice of frequency of transfer is effectively determined by the cost of communication. If communication cost is low (*e.g.*, on shared address space multiprocessors) then it would be best to perform communication after every node expansion.

Ring Communication Strategy: In the ring communication strategy, different processors are assumed to be connected in a virtual ring.[146,151] 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 to message passing machines with high diameter and low bisection width (*e.g.*, ring). Of course, it can be equally easily implemented on low diameter message passing computers and shared address space computers. As before, the cost of communication determines the choice of frequency of transfer. Figure 9 illustrates the ring communication strategy. Unless the search space is highly uniform, the search overhead factor of this scheme is very high. The reason is that this scheme takes a long time to distribute good nodes from one processor to all other processors.

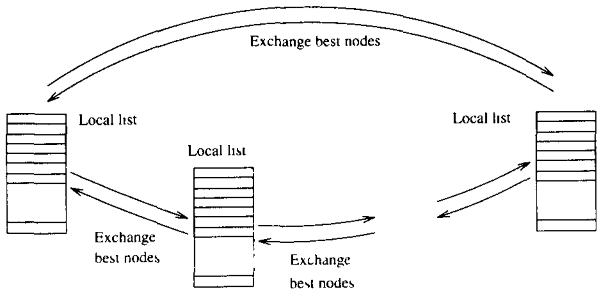

Figure 9. Message passing implementation of parallel best-first search using the ring communication strategy.

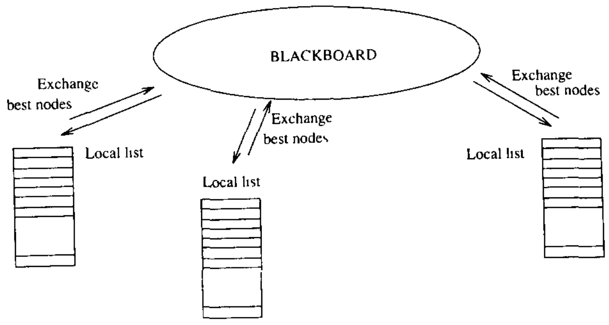

Figure 10. Implementation of parallel best-first search using the BLACKBOARD communication strategy.

Round-Robin Communication Strategies: In the round-robin communication strategy, each processor communicates by periodically putting some of its best nodes in the OPEN list of a processor chosen in a round-robin fashion.[45] Since this potentially requires communication between all pairs of processors, the scheme is particularly suited to high bandwidth, low diameter communication networks. This scheme can be modified for other networks. One such modification is the nearest-neighbor round-robin scheme in which each processor communicates only with its nearest neighbors in a round-robin fashion.[25,96] A generalization of this scheme defines a neighborhood for each processor within which round-robin communication is performed.

4.1.2. Qualitative Load Balancing Strategies

Kumar et al.[79] propose a communication scheme referred to as 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. This scheme is designed to explicitly balance the quality of the nodes at various processors. It implicitly provides a measure of quantitative load balance. Figure 10 illustrates the blackboard communication strategy. The BLACKBOARD formulation is suited only for shared address space architectures.

The quantitative load balancing schemes described in Section 4.1.1 can be modified to provide qualitative load balance too. This is done by allowing transfer of nodes only if the heuristic value of the node is better than a certain threshold. This ensures that good nodes in the search tree are uniformly distributed among various processors. In [25, 96], the nearest-neighbor round-robin strategy is combined with a qualitative load balancing scheme. This scheme, referred to as quality equalizing (QE) strategy in [25], is analytically studied and compared with round-robin and random communication strategies. The performance of the QE strategy is shown to be better than the round-robin and random communication strategies without qualitative load balance.[25]

4.1.3. Experimental and Analytical Results

The effectiveness of different parallel formulations is strongly dependent upon the characteristics of the problem being solved. Kumar et al.[79] experimentally evaluate the relationship between the characteristics of the search spaces and the suitability of various parallel formulations. They studied the performance of the random, ring, and blackboard communication strategies in the context of the 15 puzzle, the vertex cover and the traveling salesman problem on a BBN butterfly. Many parallel formulations of best-first search have been applied to various problems. These have been shown to yield good (in some cases even near linear) speedups for moderate (<100) number of processors on large problem instances. Bixby[15] presents a parallel branch-and-bound algorithm for solving the symmetric traveling salesman problem. He also presents solutions of the LP relaxations of airline crew-scheduling models. Miller[103] present parallel formulations of the best-first branch and bound technique for solving the asymmetric traveling salesman problem on heterogeneous network computer architectures. Roucairol[134] presents parallel best-first branch and bound formulations for shared address space computers and uses these to solve the Multiknapsack and Quadratic assignment problems.

Some parallel best-first search algorithms have been analytically studied. Karp and Zhang[60] 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.[133] use Monte Carlo simulations to model performance of parallel best-first search. Wah and Yu[152] also present stochastic models to analyze performance of parallel formulations of depth-first branch-and-bound and best-first branch-and-bound search.

4.2. Communication Strategies for Parallel Graph Search

A natural 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. Upon receiving the node, this processor checks whether it already has the node. If the node does not exist locally, it is installed in the OPEN list. If the node already exists at the processor, and if the new node has a better cost associated with it, then the previous version of the node is eliminated and the new node is installed in the OPEN list. For a random hash function, the load balancing property of this distribution strategy is similar to the random distribution technique discussed in the previous subsection. Manzini[98] shows that this method ensures a good distribution of promising nodes among all processors. Distributing nodes in this fashion will cause performance degradation, as each node generation results in a corresponding communication cycle.

Evett et al.[27] 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. 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.

Most of the search techniques discussed so far can be used to search OR trees. All internal nodes in such trees are OR nodes. This implies that it is not necessary to search all the subtrees rooted at a node before a solution can be determined. In contrast, there are AND trees in which all internal nodes are AND nodes. In such trees, a solution corresponding to a node can be determined only by searching all its off-springs. Examples of such trees occur in divide-and-conquer algorithms. Many problems can be modeled in the form of tree search of an AND/OR tree. In such trees, some internal nodes are AND nodes while others are OR nodes. An example of AND/OR tree search is the solution of optimization problems using dynamic programming techniques. Parallel formulations of AND/OR tree search are more complicated than AND tree or OR tree search. AND/OR tree search will be discussed here in the context of dynamic programming problems. A more rigorous discussion of parallel AND/OR tree search can be found in [149].

5. Speedup Anomalies in Parallel Formulations of Search Algorithms

In parallel search algorithms, the speedup can differ greatly from one execution to another. This is because the search space examined by different processors 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.[49,82,68,92,108,129] An instance of speedup anomaly is shown in Figure 11(a). 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.[49,82,104,48,148] An instance of deceleration anomaly is illustrated in Figure 11(b). The speedup obtained in some cases may actually be less than 1, *i.e.*, the parallel formulation is in fact slower than the sequential formulation. This is termed as detrimental anomaly.[92,49,82,104]

Lai and Sahni[82] 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 proces-

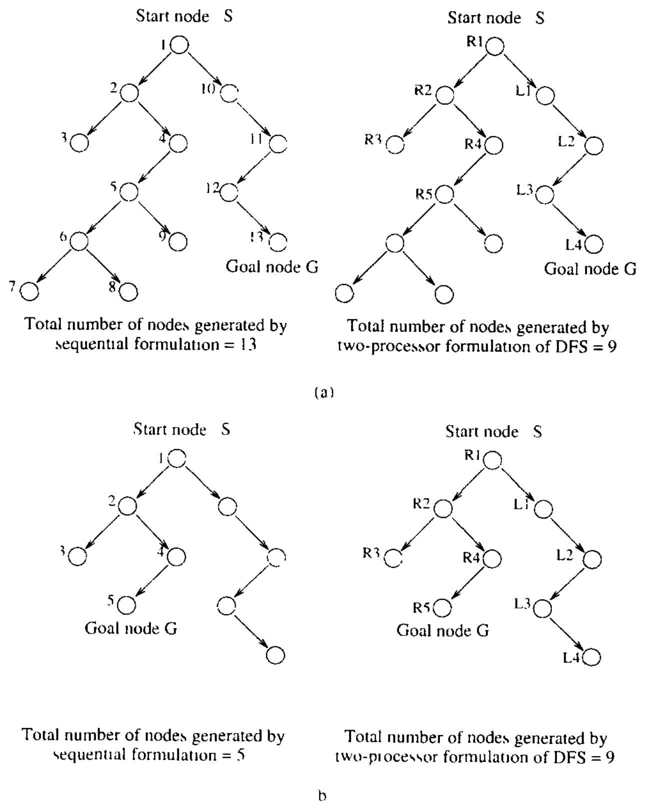
Start node S
Goal node G
Total number of nodes generated by sequential formulation = 13
Start node S
R1
R2
R3
R4
R5
L1
L2
L3
L4
Goal node G
Total number of nodes generated by two-processor formulation of DFS = 9
(a)
Start node S
Goal node G
Total number of nodes generated by sequential formulation = 5
Start node S
Goal node G
Total number of nodes generated by two-processor formulation of DFS = 9
b

Figure 11. Acceleration and deceleration anomalies in parallel depth-first search.

sors 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[84,83] 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[90,92] and Wah et al.[148] 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). Furthermore, 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[124] 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.

Rao and Kumar[130,131] 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,[141] the 15-puzzle problem and the n-Queens problem.[42] (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. Thus parallel DFS 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.[10] PODEM is a popular algorithm used in automatic test pattern generation for combinatorial circuits. In the parallel formulation of PODEM,[10] consistently superlinear speedups were demonstrated on a 128 processor Symult s2010 computer for problems with small number of solutions (very few faults in the circuit).

6. Parallel Dynamic Programming

Parallel formulations of DP differ significantly for different serial DP algorithms. We had earlier classified DP algorithms into four classes, namely, serial-monadic, serial-polyadic, nonserial-monadic, and nonserial-polyadic. Parallel formulations of the problems in each of the four DP categories have certain similarities. Each of these are investigated separately in [74]. In this section we, describe problems in two classes, serial-monadic and nonserial-polyadic.

The graph corresponding to a serial-monadic DP formulation is a multistage (or levelized graph). This is a direct consequence of the fact that solution to a subproblem at a particular level depends only on solutions to subproblems at immediately preceding levels. A number of problems are solved using serial-monadic DP formulations. These include special cases of the shortest path problem and the 0/1 knapsack problem. Parallel formulations of these DP formulations depend on the sparsity of dependencies between subproblems at consecutive levels.

As an example of a serial-monadic DP formulation, consider the multistage graph shown in Figure 12. Let $c(i, j)$ be the cost of the edge connecting node i to j, and the cost of the path connecting two nodes be defined as the sum of costs of the edges of the path. Further, let $C(i)$ be the cost of the least-cost path from node i to the goal node (node G). The goal in this example is to find $C(0)$. Clearly, $C(G) = 0$ and for all other nodes, $C(i)$ 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.

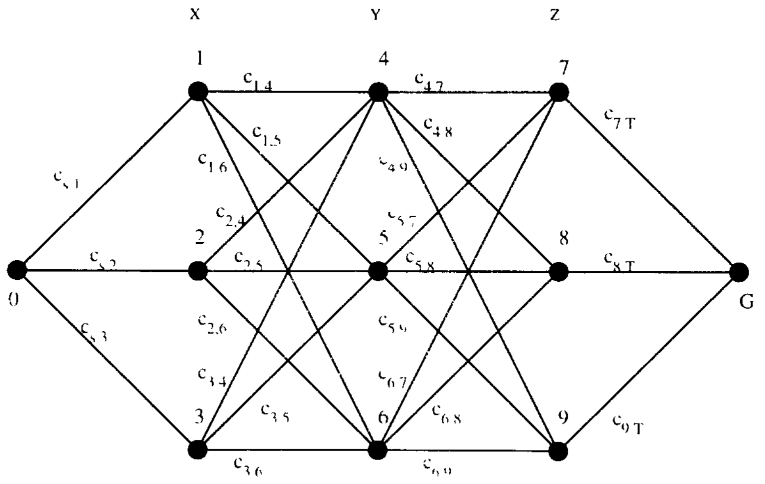

Figure 12. A three-stage graph with constant arc costs.

Let $R(S)$ be a vector of the "C" values of the nodes in stage S. Now, $R(S_i)$ for any stage can be computed as a function of $R(S_{i+1})$ (where S_{i+1} is the next stage after S_i) and the matrix of arc costs between stages S_i and S_{i+1}. It can be shown easily that the computation of $R(S_i)$ is similar to the product of the arc cost matrix between stages S_i and S_{i+1} and R_{i+1} provided the addition operation is substituted by "MIN" and multiplication is substituted by addition.[91] Further, for the example given in Figure 12, $R(S)$ for the first stage is identical to $C(0)$, as it has only one node in the first stage.

If there are n nodes at a level, the cost of computing the "R" vectors is clearly $O(n^2)$. A simple parallel formulation would be to assign one node at each stage to each of the n processors. This is akin to horizontal 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[14] during which solution costs of the subproblems at that stage are made known to all the processors. This operation takes $O(n)$ 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. Faster parallel formulations for hypercubes can be derived using various parallel matrix-vector product algorithms.[74,6,14]

Li and Wah[91] 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 also present a parallel formulation for monadic serial DP formulations based upon systolic arrays for matrix multiplications.

When the dependencies between levels become sparse and irregular, the traditional matrix-vector product methods cannot be used because of their higher computational complexity. For such problems, alternate parallel formulations have to be derived. An example of such a DP formulation is the 0/1 Knapsack problem. If the knapsack has a capacity c, then there are c subproblems to be solved at each of the n levels (n being the total number of objects). Each subproblem uses solutions from only two subproblems at the previous level. Kumar et al.[74] investigate parallel algorithms for this DP formulation.

A large class of optimization problems can be formulated as *nonserial-polyadic* DP formulations. Examples of these problems are: optimal triangularization of polygons, optimal binary search trees,[19] the matrix parenthesization problem,[19] and the CYK parser.[5] The serial complexity of the standard DP formulation for all these problems is $O(n^3)$. Parallel formulations have been proposed in [47] that use $O(n)$ processors on a hypercube and solve these problems in $O(n^2)$ time. A systolic array algorithm has been proposed by Guibas et al.[40] that uses $O(n^2)$ processing cells and solves the problem in $O(n)$ time. These and other parallel formulations are discussed by Kumar et al. in [74]. Faster algorithms for solving this problem use a large number of processors to gain small improvements in time. Therefore, the efficiency obtained from these formulations is poor. For example, the generalized method of parallelizing such programs described by Valiant et al.[143] directly leads to formulations which run in $O(\log^2 n)$ time on $O(n^9)$ processors. Rytter [136] 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 and $O(n^6)$ for a hypercube while still solving it in $O(\log^2 n)$ time. Huang et al.[43] 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.

Karypis and Kumar[61] analyze three different mappings of the systolic algorithm presented by Guibas et al.[40] and experimentally evaluate them in the context of the matrix multiplication parenthesization problem on an nCUBE2. They show that a straightforward mapping of this algorithm on the mesh architecture has an upper bound of 1/6 on the efficiency; *i.e.*, the speedup obtained will always be less than 1/6th of the number of processors. They also present a smarter mapping that does not have this drawback, and show near-linear speedup on a mesh embedded in a 256 processor nCUBE2 parallel computer for the matrix parenthesization problem. DeMello et al.[23] use vectorized formulations of DP for the Cray to solve optimal control problems.

An important characteristic of DP techniques with respect to their parallel formulations is the density of dependencies between various levels in the multistage graph. When the dependencies become sparse, efficient parallel algorithms have to be specially designed using the structure inherent in the DP formulation. Examples of such problems are DP algorithms for solving the 0/1 Knapsack problem and the single source shortest path problem. Lee et al.[87] use a divide-and-conquer strategy for parallelizing the DP algorithm for the 0/1 knapsack problem on a MIMD message passing computer. They demonstrate experimentally that it is possible to obtain linear speedup for large instances of the problem provided enough memory is available.

7. Concluding Remarks

Multiprocessors provide a viable means of effectively speeding up many computationally intensive applications in discrete optimization. A significant amount of work has been done to tap this potential and much work still remains to be done. Here, we list some avenues for future research.

Many DOP formulations have been implemented only for abstract problems. More experimentation is necessary, particularly on production problems. Since current parallel computers are much more cost effective than supercomputers, they will allow us to solve large instances of practical optimization problems such as scheduling and planning.

Near optimal parallel formulations of depth-first search techniques have been derived for various architectures. The performance of many of these formulations has been theoretically shown to be within a small factor off from the optimal. Continuing work in this area has focused on determining tighter bounds for various formulations while making fewer assumptions.[58,57] There are a number of load balancing techniques[41,54,63,137,138,155] that are applicable only if an estimate of the amount of work is available. If we could devise some technique to estimate the amount of unfinished work in a (partially searched) state space tree, then all these parallel formulations could be applied to DFS. Even though the use of extra information cannot improve the scalability of a formulation significantly in asymptotic terms, it can lead to reduction in the constants associated with it.

As we have seen, parallel best-first search techniques require both quantitative and qualitative load balancing techniques. This makes the analysis of parallel best-first search formulations more difficult. The scalability of different parallel best-first search formulations is not well understood and more research is needed on this topic. Further, the single biggest deterrent to the use of best-first search is its large memory requirement. Some parallel best-first search formulations have been proposed that are designed to operate under limited memory constraints.[27] The nature and performance of these schemes is not clearly understood.

The occurrence of consistent superlinear speedup on certain problems implies that the sequential DFS algorithm is suboptimal for these problems and that parallel DFS time sliced on one processor dominates sequential DFS. This becomes particularly significant for problems for which sequential DFS is not dominated by any other known search technique. A number of questions need to be addressed here. What is the best possible sequential search algorithm? Is it the one derived by running parallel DFS on one processor in a time slicing mode? Or is there an entirely different hitherto unknown algorithm that is even better than DFS? If parallel DFS timesliced on one processor is found to dominate other known search algorithms for some search problems, then what is the optimum number of processors to emulate in this mode?

Due to the diverse nature of problems solved using DP, it is not possible to devise generic parallel algorithms for classes of DP formulations. This implies that parallel algorithms have to be tailored to individual applications. It would be useful to identify characteristics of DP formulations and categorize these DP formulations into classes to which known generic parallel algorithms could be applied. Classification of DP formulations as serial/non-serial, monadic/polyadic is a first step in this direction. However, there are DP formulations that do not fall in any of these classes. Furthermore, this categorization is not strong enough to allow development of generic parallel algorithms for each category. A number of new DP algorithms have been proposed which make use of such problem characteristics as sparsity, convexity and concavity.[12,35,2,156,26] Parallel formulations of many of these need to be investigated.

A number of programming environments have been developed for implementing parallel search. These include DIB,[31] Chare-Kernel,[139] MANIP,[151] and PICOS.[133] Continued work on these programming environments is of prime importance, as they can potentially simplify the development of production level software for solving optimization problems.

Acknowledgments

This work was supported by IST/SDIO through the Army Research Office grant DA/DAAH04-93-G-0080 to the University of Minnesota, NSF grant NSG/IRI-9216941, and by the Army Research Office contract number DAAL03-89-C-0038 with the University of Minnesota Army High Performance Computing Research Center. The authors acknowledge the constructive comments provided by Dan Challou. We also thank the anonymous reviewers for comments that improved the quality of the paper.

References

1. T.S. Abdelrahman and T.N. Mudge, 1988. Parallel Branch-and-Bound Algorithms on Hypercube Multiprocessors, in

Proceedings of the Third Conference on Hypercubes, Concurrent Computers, and Applications, ACM Press, New York, pp. 1492–1499.

2. A. AGARWAL and M.M. KLAWE, 1990. Applications of Generalized Matrix Searching to Geometric Algorithms. *Discrete Applied Mathematics 27*, 3–24.
3. D.P. AGRAWAL, V.K. JANAKIRAM and R. MEHROTRA, 1988. A Randomized Parallel Branch and Bound Algorithm, in *Proceedings of International Conference on Parallel Processing*.
4. A.V. AHO, J.E. HOPCROFT and J.D. ULLMAN, 1974. *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, MA.
5. A.V AHO and J.D. ULLMAN, 1972. *The Theory of Parsing, Translation and Compiling: Volume 1, Parsing*, Prentice-Hall, Englewood Cliffs, NJ.
6. S.G. AKL, 1989. *The Design and Analysis of Parallel Algorithms*, Prentice-Hall, Englewood Cliffs, NJ.
7. S.G. AKL, D.T. BERNARD and R.J. JORDAN, 1982. Design and Implementation of a Parallel Tree Search Algorithm. *IEEE Transactions on Pattern Analysis and Machine Intelligence, PAMI-4*, 192–203.
8. S. ARVINDAM, V. KUMAR and V. NAGESHWARA RAO, 1989. Floorplan Optimization on Multiprocessors, in *Proceedings of the 1989 International Conference on Computer Design*. Also published as Technical Report ACT-OODS-241-89, Microelectronics and Computer Corporation, Austin, TX.
9. S. ARVINDAM, V. KUMAR and V. NAGESHWARA RAO, 1990. Efficient Parallel Algorithms for Search Problems: Applications in VLSI CAD, in *Proceedings of the Frontiers 90 Conference on Massively Parallel Computation*.
10. S. ARVINDAM, V. KUMAR, V. NAGESHWARA RAO and V. SINGH, 1991. Automatic Test Pattern Generation on Multiprocessors. *Parallel Computing 17:12*, 1323–1342.
11. G.M. BAUDET, 1978. *The Design and Analysis of Algorithms for Asynchronous Multiprocessors*, PhD thesis, Carnegie-Mellon University, Pittsburgh, PA.
12. R. BELLMAN and S. DREYFUS, 1962. *Applied Dynamic Programming*, Princeton University Press, Princeton, NJ.
13. U. BERTELE and F. BRIOSCHI, 1972. *Nonserial Dynamic Programming*, Academic Press, New York, NY.
14. D.P. BERTSEKAS and J.N. TSITSIKLIS, 1989. *Parallel and Distributed Computation: Numerical Methods*, Prentice-Hall, Englewood Cliffs, NJ.
15. R. BIXBY, 1991. Two Applications of Linear Programming, in *Proceedings of the Workshop on Parallel Computing of Discrete Optimization Problems*, Minneapolis, MN.
16. T.L CANNON and K.L. HOFFMAN, 1989. Large Scale 0-1 Linear Programming on Distributed Workstations. Working Paper.
17. D.J. CHALLON, M. GINI and V. KUMAR, 1992. Parallel Search Algorithms for Robot Motion Planning, Technical Report R 92-65, Department of Computer Science, University of Minnesota, Minneapolis, MN. Also appears in the working notes of the *1993 AAAI Spring Symposium on Innovative Applications of Massive Parallelism*.
18. E. CHARNIAK and D. MCDERMOTT, 1985. *Introduction to Artificial Intelligence*, Addison-Wesley, Reading, MA.
19. T.H. CORMEN, C.E. LEISERSON and R.L. RIVEST, 1990. *Introduction to Algorithms*, MIT Press, McGraw-Hill, New York, NY.
20. H. CROWDER, E.L. JOHNSON and M. PADBERG, 1983. Solving Large-Scale Zero-One Linear Programming Problem. *Operations Research 2*, 803–834.
21. W.J DALLY, 1987. *A VLSI Architecture for Concurrent Data Structures*, Kluwer Academic Publishers, Boston, MA.
22. R. DEHNE, A. FERREIRA and A. RAU-CHAPLIN, 1990. A Massively Parallel Knowledge-Base Server Using a Hypercube Multiprocessor. Technical Report SCS-TR-170, Carleton University.
23. J.D. DEMELLO, J.L. CALVERT, and J.M. GARCIA, 1990. Vectorization and Multitasking of Dynamic Programming in Control: Experiments on a CRAY-2, *Parallel Computing 13*, 261–269.
24. E.W. DIJKSTRA, W.H. SEIJEN and A.J.M. VAN GASTEREN, 1983. Derivation of a Termination Detection Algorithm for a Distributed Computation, *Information Processing Letters* 16:5, 217–219.
25. S. DUTT and N.R. MAHAPATRA, 1993. Parallel A* Algorithms and Their Performance on Hypercube Multiprocessors, in *Proceedings of the Seventh International Parallel Processing Symposium*, pp. 797–803.
26. D. EPPSTEIN, Z. GHALIL and R. GIANCARLO, 1988. Speeding Up Dynamic Programming, in *Proceedings of 29th IEEE Symposium on Foundations of Computer Science*, pp. 488–496.
27. M. EVETT, J. HENDLER, A MAHANTI and D NAU, 1990. PRA*: A Memory-Limited Heuristic Search Procedure for the Connection Machine, in *Proceedings of the Third Symposium on the Frontiers of Massively Parallel Computation*, pp. 145–149.
28. C. FERGUSON and R KORF, 1988. Distributed Tree Search and Its Application to Alpha-Beta Pruning, in *Proceedings of the 1988 National Conference on Artificial Intelligence*.
29. R.A. FINKEL and J.P. FISHBURN, 1982. Parallelism in Alpha-Beta Search, *Artificial Intelligence 19*, 89–106.
30. R.A. FINKEL and J.P. FISHBURN, 1983. Improved Speedup Bounds for Parallel Alpha-Beta Search, *Pattern Analysis and Machine Intelligence* 5:1, 89–92.
31. R.A. FINKEL and U. MANBER, 1987. DIB—A Distributed Implementation of Backtracking, *ACM Transactions of Programming Languages and Systems 9:2*, 235–256.
32. R. FRYE and J. MYCZKOWSKI, 1990. Exhaustive Search of Unstructured Trees on the Connection Machine. Technical Report, Thinking Machines Corporation, Cambridge, MA.
33. M. FURUICHI, K. TAKI and N. ICHIYOSHI, 1990. A Multi-Level Load Balancing Scheme for OR-Parallel Exhaustive Search Programs on the Multi-PSI, in *Proceedings of the Second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pp. 50–59.
34. M. GAREY and D.S. JOHNSON, 1979. *Computers and Intractability*, Freeman, San Francisco, CA.
35. Z. GHALIL and K. PARK, 1992. Dynamic Programming with Convexity, Concavity and Sparsity, *Theoretical Computer Science 92*, 49–76.
36. A. GOTTLIEB, R. GRISHMAN, C.P. KRUSKAL, K.P. MCAULIFFE, L. RUDOLPH and M. SNIR, 1983. The NYU Ultracomputer-Designing a MIMD, Shared Memory Parallel Computer, *IEEE Transactions on Computers, C-32:2*, 175–189.
37. A. GRAMA, A. GUPTA and V. KUMAR, 1993. Isoefficiency Function: A Scalability Metric for Parallel Algorithms and Architectures, *IEEE Parallel and Distributed Technology, Special Issue on Parallel and Distributed Systems: From Theory to Practice 1:3*, 12–21. Also available as Technical Report TR-93-24, Department of Computer Science, University of Minnesota and from anonymous ftp site ftp.cs.umn.edu, file users/kumar/isoeff-tutorial.ps.
38. A. GRAMA, V. KUMAR and P.M. PARDALOS, 1992. Parallel Processing of Discrete Optimization Problems, in *Encyclopaedia of Microcomputers*, Marcel Dekker Inc., New York, NY, pp. 129–157.
39. A. GRAMA, V. KUMAR and V. NAGESHWARA RAO, 1991. Experimental Evaluation of Load Balancing Techniques for the Hy-

percube, in *Proceedings of the Parallel Computing '91 Conference*, pp. 497–514.

40. L.J. GUIBAS, H.T. KUNG and C.D. THOMPSON, 1979. Direct VLSI Implementation of Combinatorial Algorithms, in *Proceedings of Conference on Very Large Scale Integration*, California Institute of Technology, pp. 509–525.
41. J. HONG and X. TAN, 1989. Dynamic Cyclic Load Balancing on Hypercube, in *Proceedings of the Fourth Conference on Hypercubes, Concurrent Computers, and Applications*.
42. E. HOROWITZ and S. SAHNI, 1978. *Fundamentals of Computer Algorithms*, Computer Science Press, Rockville, MD.
43. S.H.S. HUANG, H. LIU and V. VISHWANATHAN, 1990. A Sub-Linear Parallel Algorithm for Some Dynamic Programming Problems, in *Proceedings of 1990 International Conference on Parallel Processing*, pp. III-261–III-264.
44. S.-R. HUANG and L.S. DAVIS, 1987. A Tight Upper Bound for the Speedup of Parallel Best-First Branch-and-Bound Algorithms. Technical report, Center for Automation Research, University of Maryland, College Park, MD.
45. S.-R. HUANG and L.S. DAVIS, 1989. Parallel Iterative A* Search: An Admissible Distributed Heuristic Search Algorithm, in *Proceedings of the International Joint Conference on Artificial Intelligence*, pp. 23–29.
46. M.M. HUNTBACH and F.W BURTON, 1988. Alpha-Beta Search on Virtual Tree Machines, *Information Science 44*, 3–17.
47. O.H. IBARRA, T.C. PONG and S.M. SOHN, 1991. Parallel Recognition and Parsing on the Hypercube, *IEEE Transactions on Computers 40:6*, 764–770.
48. M. IMAI, Y. TATEIZUMI, Y. YOSHIDA and T. FUKUMURA, 1984. The Architecture and Efficiency of DON: A Combinatorial Problem Oriented Multicomputer System, in *Proceedings of International Conference on Distributed Computing Systems*.
49. M. IMAI, Y. YOSHIDA and T. FUKUMURA, 1979. A Parallel Searching Scheme for Multiprocessor Systems and its Application to Combinatorial Problems, in *Proceedings of International Joint Conference on Artificial Intelligence*, pp. 416–418.
50. K.B. IRANI and Y.F. SHIH, 1986. Parallel A* and AO* Algorithms: An Optimality Criterion and Performance Evaluation, in *Proceedings of International Conference on Parallel Processing*, pp. 274–277.
51. V.K. JANAKIRAM, D.P. AGRAWAL and R. MEHROTRA, 1987. Randomized Parallel Algorithms for Prolog Programs and Backtracking Applications, in *Proceedings of International Conference on Parallel Processing*, pp. 278–281.
52. V.K. JANAKIRAM, D.P. AGRAWAL and R. MEHROTRA, 1988. A Randomized Parallel Backtracking Algorithm, *IEEE Transactions on Computers C-37:12*.
53. C. KAKLAMANIS and G. PERSIANO, 1992. Branch-and-Bound and Backtrack Search on Mesh-Connected Arrays of Processors, in *Proceedings of Fourth Annual Symposium on Parallel Algorithms and Architectures*, pp. 118–126.
54. L.V. KALE, 1988. Comparing the Performance of Two Dynamic Load Distribution Methods, in *Proceedings of 1988 International Conference on Parallel Processing*, pp. 8–12.
55. L.V. KALE and V. SALETORE, 1991. Efficient Parallel Execution of IDA* on Shared and Distributed-Memory Multiprocessors, in *The Sixth Distributed Memory Computing Conference Proceedings*.
56. L.N. KANAL and V. KUMAR, 1988. *Search in Artificial Intelligence*, Springer-Verlag, New York, NY.
57. R. KARP and Y. ZHANG, 1988. A Randomized Parallel Branch-and-Bound Procedure, in *Proceedings of 20th Annual ACM Symposium on Theory of Computing*, pp. 290–300.
58. R. KARP and Y. ZHANG, 1993. Randomized Parallel Algorithms for Backtrack Search and Branch-and-Bound Computation, *Journal of ACM 40*, 765–789.
59. R.M. KARP and M.H. HELD, 1967. Finite State Processes and Dynamic Programming, *SIAM Journal of Applied Math 15*, 693–718.
60. R.M. KARP and Y. ZHANG, 1988. A Randomized Parallel Branch-and-Bound Procedure, in *Proceedings of the ACM Annual Symposium on Theory of Computing*, pp. 290–300.
61. G. KARYPIS and V. KUMAR, 1992. Efficient Parallel Mappings of a Dynamic Programming Algorithm. Technical Report TR 92-59, Computer Science Department, University of Minnesota, Minneapolis, MN.
62. G. KARYPIS and V. KUMAR, 1992. Unstructured Tree Search on SIMD Parallel Computers. Technical Report 92-21, Computer Science Department, University of Minnesota. Appears in *IEEE Transactions on Parallel and Distributed Systems 5:* 10, pp. 1057–1072, October 1994. A short version appears in *Supercomputing '92 Proceedings*, pages 453–462, 1992. Available via anonymous ftp from ftp.cs.umn.edu at users/kumar/lb-SIMD.ps.
63. R. KELLER and F. LIN, July, 1984. Simulated Performance of a Reduction Based Multiprocessor, *IEEE Computer*.
64. K. KIMURA and I. NOBUYUKI, 1991. Probabilistic Analysis of the Efficiency of the Dynamic Load Distribution, in *The Sixth Distributed Memory Computing Conference Proceedings*.
65. T.C. KOOPMANS and M.J. BECKMANN, 1957. Assignment Problems and the Location of Economic Activities, *Econometrica 25*, 53–76.
66. R.E. KORF, 1985. Depth-First Iterative-Deepening: An Optimal Admissible Tree Search, *Artificial Intelligence 27*, 97–109.
67. R. KORF, 1988. Optimal Path Finding Algorithms, in L. N. Kanal and Vipin Kumar (eds), *Search in Artificial Intelligence*, Springer-Verlag, New York, NY.
68. W. KORNFELD, 1981. The Use of Parallelism to Implement a Heuristic Search, in *Proceedings of International Joint Conference on Artificial Intelligence*, pp. 575–580.
69. V. KUMAR, P.S. GOPALKRISHNAN and L.N. KANAL, 1990. *Parallel Algorithms for Machine Intelligence and Vision*, Springer-Verlag, New York, NY.
70. V. KUMAR and L.N. KANAL, 1983. A General Branch-and-Bound Formulations for Understanding and Synthesizing and/or Tree Search Procedures. *Artificial Intelligence 21*, 179–198.
71. V. KUMAR and L.N. KANAL, 1984. Parallel Branch-and-Bound Formulations for and/or Tree Search, *IEEE Transactions Pattern Analysis and Machine Intelligence*, PAMI-6, 768–778.
72. V. KUMAR, L.N. KANAL and P.S. GOPALAKRISHNAN, (eds), 1989. *IJCAL-89 Workshop on Parallel Algorithms for Machine Intelligence*, sponsored by American Association for Artificial Intelligence.
73. V. KUMAR, L.N. KANAL and P.S. GOPALAKRISHNAN, (eds), 1988. AAAI-88 Workshop on Parallel Algorithms for Machine Intelligence, St. Paul, Minneapolis. Sponsored by American Association for Artificial Intelligence.
74. V. KUMAR, A. GRAMA, A. GUPTA and G. KARYPIS, 1994. *Introduction to Parallel Computing: Algorithm Design and Analysis*, Benjamin Cummings/Addison Wesley, Redwood City, CA.
75. V. KUMAR, A. GRAMA and V. NAGESHWARA RAO, 1994. Scalable Load Balancing Techniques for Parallel Computers, *Journal of Parallel and Distributed Computing 22:1*, 60–79. Also available as Technical Report 91-55 (November 1991), Department of Computer Science, University of Minnesota, Minneapolis, MN. Available via anonymous ftp from ftp.cs.umn.edu at users/kumar/lb-MIMD.ps.

76. V. KUMAR and A. GUPTA, 1994. Analyzing Scalability of Parallel Algorithms and Architectures, *Journal of Parallel and Distributed Computing (special issue on scalability) 22:3*, 379–391. A short version of the paper appears in *Proceedings of the 1991 International Conference on Supercomputing*, available via anonymous ftp from ftp.cs.umn.edu at users/kumar/survey-scalability.ps.
77. V. KUMAR and L.N. KANAL, 1988. The CDP: A Unifying Formulation for Heuristic Search, Dynamic Programming, and Branch-and-Bound, in L. N. Kanal and Vipin Kumar (eds), *Search in Artificial Intelligence*, Springer-Verlag, New York, NY, pp. 1–27.
78. V. KUMAR, D. NAU and L.N. KANAL, 1988. General Branch-and-Bound Formulation for and/or Graph and Game Tree Search, in L. N. Kanal and Vipin Kumar (eds), *Search in Artificial Intelligence*, Springer-Verlag, New York, NY.
79. V. KUMAR, K. RAMESH and V. NAGESHWARA RAO, 1988. Parallel Best-First Search of State-Space Graphs: A Summary of Results, in *Proceedings of the 1988 National Conference on Artificial Intelligence* pp. 122–126.
80. V. KUMAR and V.N. RAO, 1990. Scalable Parallel Formulations of Depth-First Search, in Vipin Kumar, P.S. Gopalakrishnan, and L.N. Kanal (eds), *Parallel Algorithms for Machine Intelligence and Vision*, Springer-Verlag, New York, NY.
81. V. KUMAR and V. NAGESHWARA RAO, 1987. Parallel Depth-First Search, Part II: Analysis, *International Journal of Parallel Programming 16:6*, 501–519.
82. T.H. LAI and S. SAHNI, 1984. Anomalies in Parallel Branch and Bound Algorithms, *Communications of the ACM 327*, 594–602.
83. T.H. LAI and A. SPRAGUE, 1985. Performance of Parallel Branch-and-Bound Algorithms, *IEEE Transactions on Computers*, C-34:10.
84. T.H. LAI and A. SPRAGUE, 1986. A Note on Anomalies in Parallel Branch-and-Bound Algorithms with One-to One Bounding functions, *Information Processing Letters 23*, 119–122.
85. E.L. LAWLER and D. WOODS, 1966. Branch-and-Bound Methods: A Survey, *Operations Research 14*.
86. J. LEE, E. SHRAGOWITZ and S. SAHNI, 1987. A hypercube Algorithm for the 0/1 Knapsack Problem, in *Proceedings of International Conference on Parallel Processing*, pp. 699–706.
87. J. LEE, E. SHRAGOWITZ and S. SAHNI, 1988. A Hypercube Algorithm for the 0/1 Knapsack Problem, *Journal of Parallel and Distributed Computing 5*, 438–456.
88. D.B. LEIFKER and L.N. KANAL, 1985. A Hybrid sss*/Alpha-Beta algorithm for Parallel Search of Game Trees, in *Proceedings of International Joint Conference on Artificial Intelligence*, pp. 1044–1046.
89. G.-J. LI and B.W. WAH, 1986. How Good are Parallel and Ordered Depth-First Searches, in K. Hwang, S. M. Jacobs, and E. E. Swartzlander (eds), *Proceedings of the 1986 International Conference on Parallel Processing*, IEEE Computer Society Press, pp. 992–999.
90. G.-J. LI and B.W. WAH, 1984. Computational Efficiency of Parallel Approximate Branch-and-Bound Algorithms, in *International Conference on Parallel Processing*, pp. 473–480.
91. G -J. LI and B.W. WAH, 1985. Parallel Processing of Serial Dynamic Programming Problems, in *Proceedings of COMPSAC 85*, 81–89.
92. G.-J. LI and B.W. WAH, 1986. Coping with Anomalies in Parallel Branch-and-Bound Algorithms, *IEEE Transactions on Computers C-35*, 568–573.
93 Y. LI and M. PARDALOS, 1992. Parallel Algorithms for the Quadratic Assignment Problem, in P. M. Pardalos (ed.) *Advances in Optimization and Parallel Computing*, North-Holland, Amsterdam, The Netherlands, pp. 177–189.
94. G. LINDSTROM, 1983. The Key Node Method: A Highly Parallel Alpha-Beta Algorithm, Technical Report 83-101, Computer Science Department, University of Utah, Salt Lake City.
95. A. MAHANTI and C. DANIELS, 1993. SIMD Parallel Heuristic Search, *Artificial Intelligence 60:2*, 243–282.
96. N.R. MAHAPATRA and S. DUTT, 1993. Scalable Duplicate Pruning Strategies for Parallel a* Graph Search, in *Proceedings of the Fifth IEEE Symposium on Parallel and Distributed Processing*.
97. B. MANS, T. MAUTOR and C. BOUCAIROL, 1993. Recent Exact and Approximate Algorithms for the Quadratic Assignment Problem, in *Proceedings of APMOD 93, Volume of Extended Abstracts, Budapest, Hungary*, pp. 395–402.
98. G. MANZINI and M. SOMALVICO, 1990. Probabilistic Performance Analysis of Heuristic Search Using Parallel Hash Tables, in *Proceedings of the International Symposium on Artificial Intelligence and Mathematics*.
99. T.A. MARSLAND and M. CAMPBELL, 1982. Parallel Search of Strongly Ordered Game Trees, *Computing Surveys 14*, 533–551.
100. T.A. MARSLAND and F. POPOWICH, 1985. Parallel Game Tree Search, *IEEE Transactions on Pattern Analysis and Machine Intelligence, PAMI-7:4*, 442–452.
101. T. MAUTOR and C. ROUCAIROL, 1993. Difficulties of Exact Methods for Solving the Quadratic Assignment Problem, in *DIMACS Series in Discrete Mathematics and Theoretical Computer Science—DIMACS Workshop on Quadratic Assignment Problems*, Rutgers, NJ.
102. G.P. MCKEOWN, V.J. RAYWARD-SMITH and S.A. RUSH, 1992. Parallel Branch-and-Bound, in *Advanced Topics in Computer Science*, Blackwell Scientific Publications, Oxford, UK, pp. 111–150.
103. D. MILLER, 1991. Exact Distributed Algorithms for Traveling Salesman Problem, in *Proceedings of the Workshop On Parallel Computing of Discrete Optimization Problems*.
104. J. MOHAN, 1983. Experience with Two Parallel Programs Solving the Traveling Salesman Problem, in *Proceedings of International Conference on Parallel Processing*, pp. 191–193.
105. B. MONIEN, R. FELDMAN, P. MYSLIWIETZ and O. VORNBERGER, 1990. Parallel Game Tree Search by Dynamic Tree Decomposition. In V. Kumar, P.S. Gopalakrishnan and L.N. Kanal (eds), *Parallel Algorithms for Machine Intelligence and Vision*, Springer-Verlag, New York, NY.
106. B. MONIEN and O. VORNBERGER, 1985. The Ring Machine, Technical Report, University of Paderborn, FRG. Also in *Computers and Artificial Intelligence 3*, (1987).
107. B. MONIEN and O. VORNBERGER, 1987. Parallel Processing of Combinatorial Search Trees, In *Proceedings of International Workshop on Parallel Algorithms and Architectures*.
108. B MONIEN, O. VORNBERGER and E. SPEKENMEYER, 1986. Superlinear Speedup for Parallel Backtracking, Technical Report 30, University of Paderborn, FRG.
109. H. NEY, 1982. Dynamic Programming as a Technique for Pattern Recognition, in *Proceedings of 6th International Conference on Pattern Recognition*, pp. 1119–1125.
110. N. J. NILSSON, 1980. *Principles of Artificial Intelligence*, Tioga, Palo Alto, CA.
111. P.M. PARDALOS and J. CROUSE, 1989. A Parallel Algorithm for the Quadratic Assignment Problem, in *Supercomputing '89 Proceedings*, ACM Press, New York, NY, pp. 351–360.
112. P.M. PARDALOS and G.P. RODGERS, 1989. Parallel Branch-and-Bound Algorithms for Unconstrained Quadratic Zero-One Programming, in R. Sharda et al., (eds), *Impacts of Recent*

Computer Advances on Operations Research, North-Holland, Amsterdam, The Netherlands, pp. 131–143.

113. P.M. PARDALOS and G.P. RODGERS, 1990. Parallel Branch-and-Bound Algorithms for Quadratic Zero-One Programming on a Hypercube Architecture, *Annals of Operations Research 22*, 271–292.
114. S. PATIL and P. BANERJEE, 1990. A Parallel Branch-and-Bound Algorithm for Test Generation, *IEEE Transactions on CAD 9:3*.
115. J. PEARL, 1984. *Heuristics-Intelligent Search Strategies for Computer Problem Solving*, Addison-Wesley, Reading, MA.
116. G. PLATEAU, C. ROUCAIROL and I. VALABREGUE, 1988. Algorithm PR2 for the Parallel Size Reduction of the 0/1 Multi-knapsack Problem, in *INRIA Rapports de Recherche*, No. 811.
117. C. POWLEY and R. KORF, 1989. Single Agent Parallel Window Search: A Summary of Results, in *Proceedings of the International Joint Conference on Artificial Intelligence*, pp. 36–41.
118 C. POWLEY, R. KORF and C. FERGUSON, 1993. IDA* on the Connection Machine, *Artificial Intelligence 60:2*, 199–242.
119. C. POWLEY, C. FERGUSON and R. KORF, 1990. Parallel Heuristic Search: Two Approaches, in V. Kumar, P.S. Gopalakrishnan and L.N. Kanal, (eds) *Parallel Algorithms for Machines Intelligence and Vision*, Springer-Verlag, New York, NY.
120. M. J. QUINN, 1987. *Designing Efficient Algorithms for Parallel Computers*, McGraw-Hill, New York, NY.
121. M. J. QUINN, 1990. Analysis and Implementation of Branch-and-Bound Algorithms on a Hypercube Multicomputer, *IEEE Transactions on Computers, C39:3*, 319–348.
122. M. J. QUINN and N. DEO, 1984. Parallel Graph Algorithms, *ACM Computing Surveys, 16:3*, 319–348.
123. M. J. QUINN and N. DEO, 1984. An Upper Bound for the Speedup of Parallel Branch-and-Bound Algorithms, Technical Report, Purdue University, West Lafayette, IN.
124. M. J. QUINN and N. DEO, 1986. An Upper Bound for the Speedup of Parallel Branch-and-Bound Algorithms, *BIT 26:1*.
125. A.G. RANADE, 1991. Optimal Speedup for Backtrack Search on a Butterfly Network, in *Proceedings of the Third ACM Symposium on Parallel Algorithms and Architectures*.
126. S. RANKA and S. SAHNI, 1990. *Hypercube Algorithms for Image Processing and Pattern Recognition*, Springer-Verlag, New York, NY.
127. V. NAGESHWARA RAO, 1990. *Parallel Processing of Heuristic Search*, PhD thesis, University of Texas, Austin, TX.
128. V. NAGESHWARA RAO and V. KUMAR, 1987. Parallel Depth-First Search, Part I: Implementation, *International Journal of Parallel Programming 16:6*, 479–499.
129. V. NAGESHWARA RAO, V. KUMAR and K. RAMESH, 1987. A Parallel Implementation of Iterative-Deepening-a*, in *Proceedings of the National Conference on Artificial Intelligence (AAAI-87)*, pp. 878–882.
130. V. NAGESHWARA RAO and V. KUMAR, 1988. Superlinear Speedup in State-Space Search, in *Proceedings of the 1988 Foundation of Software Technology and Theoretical Computer Science*, number 338 in Lecutre Notes in Computer Science, Springer-Verlag, 161–174.
131. V. NAGESHWARA RAO and V. KUMAR, 1993. On the Efficiency of Parallel Backtracking, *IEEE Transactions on Parallel and Distributed Systems, 4:4*, 427–437. Also available as Technical Report TR 90-55, Department of Computer Science, University of Minnesota, Minneapolis, MN. Available via anonymous ftp from ftp.cs.umn.edu at users/kumar/suplin.ps.
132. V. NAGESHWARA RAO, V. KUMAR and R. KORF, 1991. Depth-First vs Best-First Search, In *Proceedings of the National Conference on Artificial Intelligence e (AAAI-91)*.
133. C. RENOLET, M. DIAMOND and J. KIMBEL, 1989. Analytical and Heuristic Modeling of Distributed Algorithms, Technical Report E3646, FMC Corporation, Advanced Systems Center, Minneapolis, MN.
134. C. ROUCAIROL, 1987. A Parallel Branch-and-Bound Algorithm for the Quadratic Assignment Problem, *Discrete Applied Mathematics 18*, 211–225.
135. C. BOUCAIROL, 1991. Parallel Branch-and-Bound on Shared-Memory Multiprocessors, in *Proceedings of the Workshop on Parallel Computing of Discrete Optimization Problems*.
136. W. BYTTER, 1988. Efficient Parallel Computations for Dynamic Programming, *Theoretical Computer Science 59*, 297–307.
137. V. SALETORE and L.V. KALE, 1990. Consistent Linear Speedup to a First Solution in Parallel State-Space Search, in *Proceedings of the 1990 National Conference on Artificial Intelligence*, pp. 227–233.
138. K. SHIN and Y. C. CHANG, 1989. Load Sharing in Hypercube Multicomputers for Real Time Applications, in *Proceedings of the Fourth Conference on Hypercubes, Concurrent Computers, and Applications*.
139. W. SHU and L.V. KALE, 1989. A Dynamic Scheduling Strategy for the Chare-Kernel System, in *Proceedings of Supercomputing Conference*, 389–398.
140. D.R. SMITH, 1984. Random Trees and the Analysis of Branch and Bound Procedures, *Journal of the ACM 31:1*, 163–188.
141. H.S. STONE and P. SIPALA, 1986. The Average Complexity of Depth-First Search with Backtracking and Cutoff, *IBM Journal of Research and Development*.
142. S. TENG, 1990. Adaptive Parallel Algorithms for Integral Knapsack Problems, *Journal of Parallel and Distributed Computing 8*, 400–406.
143. L.G. VALIANT, S. SKYUM, S. BERKOWITZ and C. RACKOFF, 1983. Fast Parallel Computation of Polynomials Using Few Processors, *SIAM Journal of Computing 12:4*, 641–644.
144. O. VORNBERGER, 1987. Load Balancing in a Network of Transputers, in *Proceedings of Second International Workshop on Distributed Algorithms*.
145. O. VORNBERGER, 1987. The Personal Supercomputer: A Network of Transputers, in *Proceedings of the 1987 International Conference on Supercomputing*.
146. O. VORNBERGER, 1986. Implementing Branch-and-Bound in a Ring of Processors, Technical Report 29, University of Paderborn, FRG.
147. O. VORNBERGER, 1987. Load Balancing in a Network of Transputers, in *2nd International Workshop on Distributed Parallel Algorithms*.
148. B.W. WAH, G.-J. LI and C.F. YU, 1984. The Status of MANIP—A Multicomputer Architecture for Solving Combinatorial Extremum-Search Problems, in *Proceedings of 11th Annual International Symposium on Computer Architecture*, pp. 56–63.
149. B.W. WAH, G.-J. LI and C.F. YU, 1990. Multiprocessing of Combinatorial Search Problems, in Vipin Kumar, P.S. Gopalakrishnan, and L.N. Kanal (eds) *Parallel Algorithms for Machine Intelligence and Vision*, Springer-Verlag, New York, NY.
150. B.W. WAH, G.-J. LI and C.F. YU, 1985. Multiprocessing of Combinatorial Search Problems, *IEEE Computer*, pp. 93–108.
151. B.W. WAH and Y.W. EVA MA, 1984. Manip—A Multicomputer Architecture for Solving Combinatorial Extremum-Search Problems, *IEEE Transactions on Computers 33, C33:12*, 377–390.
152. B.W. WAH and C.F. YU, 1985. Stochastic Modelling of Branch-and-Bound Algorithms with Best-First Search, *IEEE Transactions on Software Engineering SE-11*.

153. D. White, 1969. *Dynamic Programming*, Olıver and Boyd, Edinburgh, UK.

154. H.S. Wilf, 1986. *Algorithms and Complexity*, Prentice-Hall, Englewood Cliffs, NJ.

155. M Willebeek-LeMair and A.P. Reeves, 1989. Dıstributed Dynamic Load Balancing, in *Proceedings of the Fourth Conference on Hypercubes, Concurrent Computers, and Applıcatıons.*

156. F.F Yao, 1980. Efficient Dynamic Programming Using Quadrangle Inequalities, in *Proceedings of 12th ACM Symposium on Theory of Computıng*, pp. 429–435.