PARALLEL HEURISTIC SEARCH: TWO APPROACHES[1]

Curt Powley, Chris Ferguson, Richard E. Korf
Computer Science Department
University of California, Los Angeles
Los Angeles, CA 90024

ABSTRACT

We explore two approaches to parallel heuristic search: one based on tree decomposition, in which different processors search different parts of the tree, and the other based on parallel window search, in which each processor searches the whole tree but with different cost bounds. In the first, we present a generic distributed tree search algorithm that effectively searches irregular trees using an arbitrary number of processors without shared memory or centralized control. For brute-force search the algorithm achieves almost linear speedup. For alpha-beta search, the straightforward approach of allocating P processors in a breadth-first manner achieves an overall speedup with random node ordering of $P^{.75}$. Furthermore we present a novel processor allocation strategy, called Bound-and-Branch, for parallel alpha-beta search that achieves linear speedup in the case of perfect node ordering. In practice, we achieve a speedup of 12 with 32 processors on a 32-node Hypercube multiprocessor for the game of Othello.

In the second approach, we show how node ordering can be combined with parallel window search to quickly find a near-optimal solution to single-agent problems. First, we show how node ordering by maximum g among nodes with equal $f = g + h$ values can improve the performance of iterative-deepening-A* (IDA*). We then consider a window search where different processes perform IDA* simultaneously on the same problem but with different cost thresholds. Next, we combine the two ideas to produce a parallel window search algorithm in which node ordering information is shared among the different processes. Finally, we show how to combine distributed tree search with parallel window search in single-agent or two-player game searches.

[1]This chapter is based upon two articles [FERG88] and [POWL89] that appeared originally in AAAI 88 and IJCAI 89 proceedings. This research was supported by an NSF Presidential Young Investigator Award to the third author, NSF grant IRI-8801939; by an Intel Hypercube, JPL contract number 957523; by DARPA contract MDA 903-87-C0663; and by a Hewlett-Packard equipment grant.

1 INTRODUCTION

1.1 Heuristic Search

Heuristic search is a fundamental problem-solving method in artificial intelligence. Search applications include both single-agent problems and two-player games. Common examples of two-player games are chess, checkers, and Othello. Examples of single-agent search problems include the Eight Puzzle and its larger relative the Fifteen Puzzle. The Eight Puzzle consists of a 3x3 square frame containing 8 numbered square tiles and an empty position called the 'blank'. The legal operators slide any tile horizontally or vertically adjacent to the blank into the blank position. The task is to rearrange the tiles from some random initial configuration into a particular desired goal configuration. A real world example of single-agent search is the traveling salesman problem of finding the shortest simply-connected tour among a set of cities to be visited.

In both single-agent problems and two-player games, search is guided by a heuristic evaluation function. In a game such as chess, a simple evaluation function would return the relative material advantage of one player over the other. A common heuristic function for the Eight and Fifteen Puzzles is called Manhattan Distance. It is computed by counting, for each tile not in its goal position, the number of moves along the grid it is away from its goal position, and summing these values over all tiles, excluding the blank. A common heuristic for the traveling salesman problem is the cost of the minimum spanning tree covering the cities not yet visited.

Shortest-path algorithms for single-agent heuristic search include A* [HART68] and iterative-deepening-A* (IDA*) [KORF85]. In both cases, $g(n)$ refers to the cost of a path from the initial state to node n, while $h(n)$ is the heuristic estimate of the cost of a path from node n to a goal. The merit of a node, $f(n)$, is the sum of $g(n)$ and $h(n)$, and gives an estimate of the cost of a path from the initial state to a goal state that passes through node n. A* is a best-first search that always expands next a node of minimum $f(n)$, until a goal node is chosen for expansion. IDA* performs a series of depth-first searches, where a branch is cutoff when the $f(n)$ value of the last node on the path exceeds a cost threshold for that iteration. The threshold for the first iteration is set at the heuristic value of the initial state, and each succeeding threshold is set at the minimum f value that exceeded the previous threshold. Both A* and IDA* are guaranteed to find optimal (lowest cost) solutions if the heuristic function never over-estimates the cost of the cheapest path to the goal. The main virtue of IDA* over A* is that because it is a depth-first search, its memory requirement is only linear in the solution depth, as opposed to exponential for best-first searches such as A*.

The classic algorithm for two-player game searches is minimax search with alpha-beta pruning. The search tree is expanded to a fixed depth depending on the computational resources available per move. The frontier nodes are

evaluated by a heuristic function, where large positive values represent advantageous positions for one player, called MAX, and negative values with large magnitude favor the opponent, MIN. At each interior node in the tree, the maximum or minimum value of the children is backed up, depending on the player to move at that node. Alpha-beta pruning dramatically reduces the number of nodes that must be evaluated by minimax search without affecting the decisions made. Alpha is a lower bound on the minimax value of a node while Beta is an upper bound. The algorithm works by successively refining the values of these bounds over the course of the search, and pruning nodes whose minimax values lie outside the bounds.

IDA* and minimax search with alpha-beta pruning have several features in common. First, both are depth-first searches to minimize memory requirements. Second, the threshold in IDA* is similar to alpha and beta in minimax in that both are cutoff bounds that prune the search space. Third, in both cases the order in which nodes are generated in the tree can have a large effect on efficiency. In IDA* it determines the time to search the final or goal threshold, while in minimax search it determines the efficiency of alpha-beta pruning. Finally, iterative-deepening, or successive searches to greater depths, plays an important role in both algorithms.

1.2 Parallel Heuristic Search

The main limitation of search is its computational complexity. Parallel processing can significantly increase the number of nodes evaluated in a given amount of time. This can either result in the ability to find optimal solutions to larger problems, or in significant improvements in decision quality for very large problems. While there is a significant body of literature on heuristic search algorithms, work on parallel search algorithms is relatively sparse. Due to the commonalities among single-agent and two-player depth-first searches, similar techniques can be applied to parallelizing them.

There are essentially three different approaches to parallelizing search algorithms. One is to parallelize the processing of individual nodes, such as move generation and heuristic evaluation. This is the approach taken by HITECH, a chess machine that uses 64 processors in an eight by eight array to compute moves and evaluations [EBEL87]. The speedup achievable in this scheme is limited, however, by the degree of parallelism available in move generation and evaluation. In addition, this approach is inherently domain-specific and unlikely to lead to general techniques for using parallel processors to speedup search. We will not consider it further in this paper.

1.2.1 Distributed tree search

A second approach is tree decomposition, in which different processors are assigned different parts of the tree to search. In principle, tree decomposition allows the effective use of an arbitrary number of processors. Recent experimental work on this paradigm is that of Rao et. al. [Rao87/88, KUMA88] in parallelizing IDA*. They have been able to achieve approximately linear speedups on a 30 processor Sequent, and a 128 processor BBN Butterfly and Intel Hypercube. In IDA*, however, the value of a threshold is set at the beginning of the corresponding iteration and does not change in the course of that iteration.

This is not true of branch-and-bound algorithms such as alpha-beta, where the bounds are updated during an iteration. Thus, whether or not a node must be evaluated depends upon values found elsewhere in the tree. The main issue in a parallel branch-and-bound search is how to keep processors from wasting effort searching parts of the tree that will eventually be pruned. In the first part of this paper we describe a different tree decomposition algorithm, distributed tree search, and apply it to minimax search with alpha-beta pruning. This section is based on [FERG88]

Finkel and Manber [FINK87] present a generalized tree search algorithm similar to ours. They do not, however, allow explicit control over the allocation of work among processors, and as a result do not achieve high speedup for branch-and-bound algorithms. The most successful work to date on the specific problem of parallel alpha-beta search has been presented by Feldmann et. al. [FELD89] and Felten and Otto [FELT88]. Feldmann achieves a speedup 12 on 16 processors for evaluating chess positions. Felten has applied his techniques to a larger number of processors and obtains a speedup of 101 on 256 processors. This is the highest speedup reported so far for parallel alpha-beta search.

1.2.2 Parallel window search

A third approach, called parallel window search, was pioneered by Gerard Baudet [BAUD78] in the context of two-player games. In this algorithm, different processors each search the entire game tree, but starting with different values for alpha and beta. The total range of values is divided into non-overlapping windows, with one processor assigned to each window. The processor having the true minimax value within its window will find it faster by virtue of starting with narrower bounds. Unfortunately, this approach by itself is severely limited in speedup since even if alpha and beta both equal the minimax value for some processor, verifying that it is indeed the minimax value requires searching $O(B^{D/2})$ nodes on a tree with branching factor B and depth D. This is the minimum number of nodes which must be searched by a single processor to determine a tree's minimax value. In experiments, speedup was limited to about five or six, regardless of the number of processors.

While the use of windows in two-player games has been considered at length [BAUD78, KUMA84], until now it has not been applied to single-agent search. In the second part of this paper we apply this idea to IDA*. Single-agent window search, though related to and inspired by the use of

windows in two-player game search is a fundamentally distinct approach. This section is based on [POWL89].

Finally, we address the issue of how to combine combine parallel window search with distributed tree search in both single-agent and two-player game searches.

2 DISTRIBUTED TREE SEARCH

2.1 Distributed Tree Search

Given a tree with non-uniform branching factor and depth, the problem is to search it in parallel with an arbitrary number of processors as fast as possible. We have developed an algorithm, called Distributed Tree Search (DTS), to solve this problem. At the top level, the algorithm makes no commitment to a particular type of tree search, but can easily be specialized to IDA*, minimax with alpha-beta pruning, etc. It can also be specialized to perform most tasks that can be expressed as tree recursive procedures, such as sorting and parsing. We make no assumptions about the number of processors, and reject algorithms based on centralized control or shared memory, since they do not scale up to very large numbers of processors.

DTS consists of multiple copies of a single process that combines both searching and coordination functions. Each process is associated with a node of the tree being searched, and has a set of processors assigned to it. Its task is to search the subtree rooted at its node with its set of processors. DTS is initialized by creating a process, the root process, and assigning the root node and all available processors to it. The process expands its node, generating its children, and allocates all of its processors among its children according to some processor allocation strategy. For example, in a breadth-first allocation scheme, it would deal out the processors one at a time to the children until the processors are exhausted. It then spawns a process for each child that is assigned at least one processor. The parent process then blocks, awaiting a message. If a process is given a terminal node, it returns the value of that node and the processors it was assigned immediately to its parent, and terminates.

As soon as the first child process completes the search of its subtree, it terminates, sending a message to its parent with its results, plus the set of processors it was assigned. Those results may consist of success or failure, a minimax value, values for alpha or beta, etc., depending on the application. This message wakes up the parent process to reallocate the freed processors to the remaining children, and possibly send them new values for alpha or beta, for example. Thus, when a set of processors completes its work, the processors are reassigned to help evaluate other parts of the tree. This results in efficient load balancing in irregular trees. A process may also be awakened by its parent with new processors or bounds to be sent to its children. Once

the reallocation is completed, the parent process blocks once again, awaiting another message. Once all child processes have completed, the parent process returns its results and processors to its parent and terminates. DTS completes when the original root process terminates.

In practice, the blocked processes, corresponding to high level nodes in the tree, exist on one of the processors assigned to the children. When such a process is awakened, it receives priority over lower level processes for the resources of its processor. Once the processors get down to the level in the tree where there is only one processor per node, the corresponding processor executes a depth-first search.

In fact, uniprocessor depth-first search is simply a special case of DTS when it is given only one processor. Given a node with one processor, the processor is allocated to the first child, and then the parent process blocks, waiting for the child to complete. The child then allocates its processor to its leftmost child and blocks, awaiting its return. When the grandchild returns, the child allocates the processor to the next grandchild, etc. This is identical to depth-first search where the blocked processes correspond to suspended frames in the recursion stack.

Conversely, if DTS is given as many processors as there are leaves in the tree, and the allocation scheme is breadth-first as described above, it simulates breadth-first search. In effect, each of the children of each node are searched in parallel by their own processor. With an intermediate number of processors, DTS executes a hybrid between depth-first and breadth-first search, depending on the number of processors and the allocation scheme.

2.2 Brute-Force Search

DTS has been implemented to search Othello game trees using a static evaluation function we developed. It runs on a 32 node Intel Hypercube multiprocessor. When the algorithm is applied to brute-force minimax search without alpha-beta pruning, perfect speedup is obtained to within less than 2%. This 2% difference is due to communication and idle processor overhead. This demonstrates that even though the branching factor is irregular, the reallocation of processors performs effective load balancing. As a result, we expect near-optimal speedup for most forms of brute-force search.

2.3 Parallel Branch-and-Bound

Achieving linear speedup for branch-and-bound algorithms, such as alpha-beta search, is much more challenging. There are two sources of inefficiency in parallel branch-and-bound algorithms. One is the **communication overhead** associated with message passing and idle processor time. This also occurs in brute-force search but is negligible for DTS, as shown above. The other source of inefficiency derives from the additional nodes that are evaluated by a parallel algorithm but avoided by the serial version. In branch-

and-bound algorithms, the information obtained in searching one branch of the tree may cause other branches to be pruned. Thus, if the children are searched in parallel, one cannot take advantage of all the information that is available to a serial search, resulting in wasted work, which we call the **search overhead**.

2.4 Analysis of Breadth-First Allocation

Consider a parallel branch-and-bound algorithm on a uniform tree with brute-force branching factor B and depth D. The **heuristic branching factor** b is a measure of the efficiency of the pruning and is defined as the D^{th} root of the total number of leaf nodes that are actually generated by a serial branch-and-bound algorithm searching to a depth D. While the brute-force branching factor B is constant, the heuristic branching factor b depends on the order in which the tree is searched. In the worst case, when children are searched in order from worst to best, no pruning takes place and thus $b = B$. In the best case of alpha-beta pruning, in which the best child at each node is searched first, $b = B^{1/2}$. If a tree is searched in random order, then alpha-beta produces an heuristic branching factor of about $b = B^{.75}$ [PEAR85]. Surprisingly, for breadth-first allocation, the more effective the pruning, the smaller the speedup over uniprocessor search.

Theorem 1: If $b = B^X$ on a uniform tree, then DTS using breadth-first allocation will achieve a speedup of $O(P^X)$, where P is the number of processors.

Proof: The speedup of a parallel algorithm is the time taken by the best serial algorithm, divided by the time taken by the parallel algorithm. The serial algorithm will evaluate $b^D = B^{XD}$ leaf nodes, resulting in a running time proportional to B^{XD}. The parallel algorithm uses P processors allocated in a breadth-first manner. Processors will be passed down the tree until there is one processor assigned per node. This occurs at a depth of $\log_B P$, in time $O(\log_B P)$. Each one of these processors will have to evaluate $O(B^{X(D-\log_B P)})$ nodes since each is searching a tree of depth $D - \log_B P$ by itself. Because the tree is assumed to be uniform, all processors will complete simultaneously in $O(B^{X(D-\log_B P)})$ time. The final step of propagating the values back the root will be done in $O(\log_B P)$ time. Therefore, the speedup is on the order of $B^{XD}/(B^{X(D-\log_B P)} + 2\log_B P)$. As D becomes large the second term of the denominator can be ignored, and hence speedup is $O(B^{XD-X(D-\log_B P)}) = O(B^{(\log_B P)X})$, or $O(P^X)$. □

2.5 Empirical Results for Breadth-First Allocation

We have searched 70 mid-game Othello positions to a depth of 6 using the breadth-first allocation scheme on 1, 2, 4, 8, 16 and 32 processors on a 32-node Intel Hypercube. Results were also obtained for 64, 128, and 256 processors by simulating multiple virtual processors on each of the 32 actual processors available. With one processor the algorithm performs serial alpha-beta search. The communication overhead for the parallel versions is always less than 5%, leading to an almost linear relation between the number of processors and number of node evaluations per unit time. This is expected due to the near perfect speedup obtained for brute-force search. This also allows us to estimate speedup by counting the total number of node evaluations in serial, and dividing that by the number of evaluations per processor performed in parallel. On 32 processors, the parallel alpha-beta algorithm evaluates about 3 times as many leaf nodes as the serial version. This results in a speedup of only 10 over uniprocessor alpha-beta search. Our program uses a very primitive, but reasonably effective, form of node ordering. From previous research, this ordering was found to produce a heuristic branching factor of $b \approx B^{.66}$ for serial alpha-beta with our Othello heuristic function. This predicts a parallel speedup of approximately $P^{.66}$. Figure 1 is a graph on a log-log scale of speedup verses number of processors. The analytical and actual speedup results for breadth-first allocation are represented by curves B and C. From these curves it can be seen that the results for breadth-first allocation fit the analytical curve very closely, thus supporting our analytical results.

If the node ordering is improved, however, even though the parallel algorithm will run faster, the relative speedup over uniprocessor alpha-beta search will decrease. In particular, if the best move from a position is always searched first (perfect move ordering), serial alpha-beta will evaluate only $B^{D/2}$ leaf nodes, and our formula predicts a speedup of only $P^{1/2}$. This is also the lower bound speedup predicted by Finkel and Fishburn for their algorithm in [FINK82]. While one may think that perfect or near-perfect node ordering is impossible to achieve in practice, state-of-the-art chess programs such as HITECH [EBEL87] only search about 1.5 times the number of nodes searched under perfect ordering. In this case our algorithm would have a predicted speedup very close to its lower bound of $P^{1/2}$. Thus the performance of the breadth-first allocation scheme is relatively poor under good node ordering, and a better allocation strategy is required.

2.6 Bound-and-Branch Allocation

We have developed another processor allocation strategy for alpha-beta search that we call Bound-and-Branch. To explain this strategy, we introduce the idea of a cutoff bound. A cutoff bound is an alpha (lower) bound at a max node or a beta (upper) bound at a min node. A cutoff bound allows each child of a node to possibly be pruned by searching only one grandchild under each child. If no cutoff bound exists at a node, then the processors are assigned depth first, i.e. all processors are assigned to the leftmost child. This is the fastest way of establishing a cutoff bound at a node. If a cutoff bound is initially passed to a node, or has been established by searching its first child, then the processors are assigned in the usual breadth-first manner. This al-

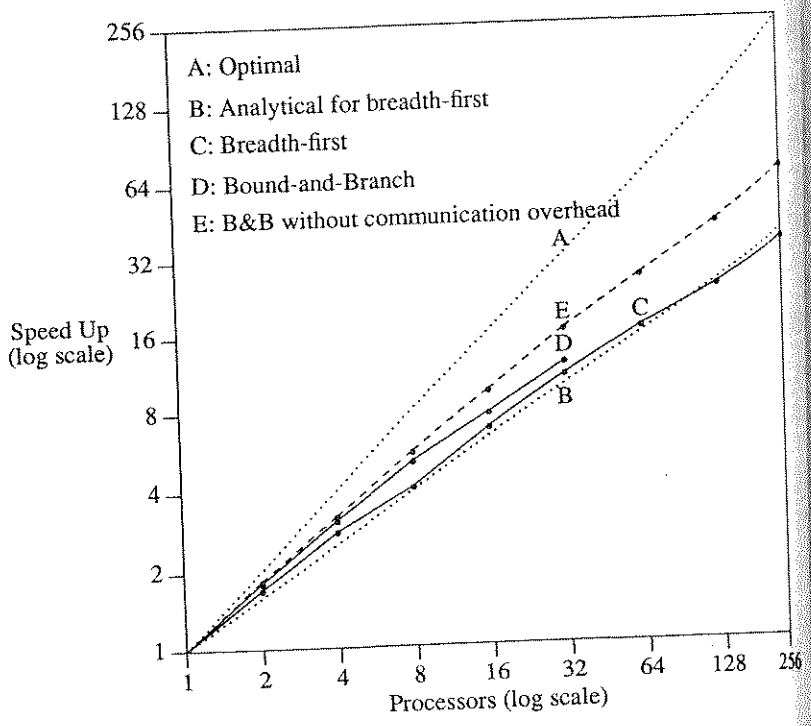

Figure 1: Speedup vs. Number of Processors

gorithm first establishes a bound, and then, once this bound is established, branches its processors off to its children, thus the name Bound-and-Branch. The underlying idea is to establish useful bounds before searching children in parallel, thus hopefully avoiding evaluating extra nodes that would be pruned by the serial version because of better available bounds.

Lines D and E in Figure 1 represent real speedup for the Bound-and-Branch allocation scheme (D), and the speedup not counting communication overhead for the Bound-and-Branch allocation strategy (E). The communication overhead for the Bound-and-Branch allocation strategy is about 25%. This is caused by the added idle processor time and the added communications associated with splitting up processors lower in the tree as opposed to splitting them up as soon as possible. Despite this, the Bound-And-Branch allocation strategy outperforms breadth-first allocation even without good node ordering. Thus this strategy is also useful for the general case of imperfect node ordering. Furthermore, we will show that its speedup over serial alpha-beta actually improves with better node ordering.

Theorem 2: In the case of perfect node ordering, DTS with the bound-and-branch allocation strategy will examine the same nodes as serial alpha-beta.

Proof: The proof is by induction on the height of the tree. For the basis step, consider trees of height one, or a root whose children are all leaves. In this case, serial alpha-beta must evaluate all the leaf children of the root. Parallel alpha-beta will first evaluate the left child, and then in parallel will evaluate the remaining children. Thus, both algorithms evaluate all the terminal nodes.

For the induction step, we assume that both algorithms evaluate the same set of nodes for all trees of height n, and consider a tree of height $n+1$. Assume that the root is a MAX node. Serial alpha-beta first evaluates the left child, returning a value of α. Since there is no initial bound, parallel alpha-beta first assigns all its processors to evaluate the left child, also returning α. Since the left child is a tree of height n, by the induction assumption, both algorithms examine the same set of nodes to evaluate the leftmost subtree.

α now serves as a lower bound on the value of the root. In the case of perfect node ordering, the first child of the root is the best, and hence the value of each of the remaining children will be found to be less than α. In other words, an upper bound no greater than α will be established for each remaining MIN children of the MAX root. To establish this upper bound on a root MIN child, only its leftmost MAX child (the root's grandchild) will be examined in the case of perfect ordering. To establish the upper bound on these MAX nodes (root grandchildren), all of their MIN children (root great-grandchildren) will have to be examined. Thus, among the remaining children of the root, one child of each MIN node generated will be examined, and all children of each MAX node generated will be examined by serial alpha-beta. Since each of these nodes will be passed the lower bound of α, the MAX nodes (root grandchildren) will have a cutoff bound and the

bound-and-branch allocation will search all the children of the MAX nodes. The MIN nodes (root great-grandchildren) will not have a cutoff bound, and hence all processors will be allocated to first search the left child of the MIN nodes. When the processors have examined the left child, they will have established an upper bound less than α and the remaining children of the MIN nodes will be pruned. Thus, the parallel algorithm will examine all children of each MAX node generated, and one child of each MIN node generated. Therefore both serial and parallel algorithms will evaluate the exact same set of nodes among the remaining children of the root.

Since the same nodes are evaluated for both the leftmost child and for the remaining children, the same nodes are evaluated for a tree of height $n+1$ where MAX is at the root. For the case where MIN is at the root, we simply repeat the above argument, interchanging MAX and MIN, lower and upper bounds, and replacing α with β. Thus, for any tree of height $n+1$ parallel alpha-beta with bound-and-branch allocation examines the same nodes as serial alpha-beta. Therefore, by induction, theorem 2 holds for all perfectly ordered minimax trees. □

2.7 Empirical Results for Bound-and-Branch Allocation

How well does this strategy work in the case of no node ordering, good node ordering, and near-perfect node ordering? To obtain better node ordering, the Othello program was modified to perform iterative-deepening alpha-beta search [SLAT77]. In an iterative-deepening search, the game tree is successively searched to depths of 1, 2, 3, 4 ... D. At each iteration, the successors of a position are searched in order of their backed-up values from the previous iteration. While iterative-deepening evaluates many internal nodes, the speedup gained through the node ordering more than compensates for the extra internal node evaluations. Iterative-deepening requires that a representation of the game tree must be kept somewhere in memory. In the parallel version this tree must be distributed among the processors since we do not use shared memory. Our program does this successfully, but in doing so increases the communication overhead to 40%, because of the added information about the game tree that must be passed between processors.

The effect that the communication overhead has on speedup can be drastically reduced by extending the depth of the search. As the depth becomes sufficiently large, the relative effect of the communication overhead is reduced to zero because the communication that must occur is almost constant, independent of the depth of the tree, while the run time is greatly increased by extending the depth. Thus, as search depth increases, the search overhead is the dominant factor in limiting speedup. We refer to the speedup achievable by ignoring the communication overhead as the potential speedup.

In the case of perfect ordering, we have shown that the number of nodes searched by our parallel algorithm is the same as for serial alpha-beta resulting in perfect potential speedup. Likewise, if the worst possible ordering is used, then both methods will have to search all the leaf nodes in the tree, again yielding perfect speedup. These represent two extreme data points on the node ordering spectrum. In the case of the primitive node ordering used in the Othello program, the Bound-and-Branch strategy has a much greater communication overhead, but evaluates far fewer nodes, and as a result achieves an actual speedup of 12 on 32 processors. The potential speedup, however, is 16. Thus the graph of potential speedup vs. quality of node ordering is "U" shaped; high at both extremes, and low in the middle. By performing an iterative-deepening search, we know the node ordering is improved since the search runs faster on a uniprocessor. The 32 processor iterative-deepening search only examines about 50% more nodes than uniprocessor iterative-deepening search. This results in an potential speedup of over 20 on 32 processors. Actual speedup is still only 12 because of the increased communication overhead for the iterative-deepening search. Since better node ordering is obtained from iterative-deepening, and our parallel algorithm increases in efficiency for better ordering, this data point must lie on the upward arc of the "U" shaped curve assuming that the curve is unimodal. Thus greater improvements in node ordering should be reflected in even greater speedups over uniprocessor alpha-beta, and we would expect this approach to have greater speedup when applied to those problems for which a good node ordering scheme can be found. These problems include state-of-the-art chess programs.

3 PARALLEL WINDOW SEARCH

While Distributed Tree Search divides the search tree among the processors, parallel window search gives each processor the entire tree, but divides the range of the cost bounds among the processors. In this section we'll explore its application to single-agent search, in particular IDA*. First, we discuss how global node ordering by minimum h among nodes with equal f values can reduce the time complexity of serial IDA* by reducing the time of the last iteration. This approach is limited by the time to perform the iterations prior to the final iteration. We then discuss the notion of pure parallel window search in which different processors look to different thresholds simultaneously. This approach, however, is limited by the time to perform the final iteration. Finally, we show how the combination of node ordering with pure parallel window search retains the good qualities of both approaches while ameliorating the bad.

Parallel window search continues running and finds better solutions until an optimal solution is found and guaranteed. Even so, it is not meant to compete with IDA* (section 1.1) for finding optimal solutions; its strength comes from relaxing the optimality constraint. After describing our implementation of an ordered parallel window search, we analyze the performance

of parallel window search with perfect ordering, and then present empirical data showing not only high speed, but high solution quality of the first solution found.

Recall from section 1.1 that IDA* works by iteratively searching in a depth-first manner. A branch of the depth-first search is cutoff when the $f(n) = g(n) + h(n)$ value of the last node on the path exceeds the threshold for that iteration. Each successive iteration searches deeper into the tree by using a higher threshold than the previous iteration.

3.1 Node Ordering

An obvious strategy for reducing the time of the last iteration of IDA* is to order the children generated in the depth-first search in the hope of reducing the nodes generated before a goal is reached. Given a perfect ordering scheme, the goal iteration would immediately choose a goal node for expansion. In this case, the time complexity of the last iteration would be reduced from exponential to linear in the depth of solution.

The simplest type of node ordering is to explore the children of a node in increasing order of their static heuristic values, $h(n)$. Experiments with this node ordering show that the improvement that results does not compensate for the additional overhead incurred by the ordering.

A more sophisticated form of node ordering is to order all nodes that are candidates for expansion, not just the children of a particular node. In a depth-first search such as IDA*, because all the nodes of a search frontier are not available at the same time, some modification is necessary to perform ordering; we describe one such modification below. Since a search frontier typically consists of a set of nodes with equal f values, we order the nodes in increasing order of h, or equivalently, in decreasing order of g. This is beneficial for two reasons.

3.1.1 Advantages of node ordering

The main intuition behind this scheme is that, for heuristic functions which never over-estimate the cost of the cheapest path to the goal, we expect smaller h values to be more accurate. This tie-breaking rule among nodes of equal f values is probably well-known, but we know of no work that has studied its effect.

To illustrate this idea, consider two frontier nodes which have equal f values but different g and h components. If our heuristic function were a perfect estimator, both nodes would in fact be on solution paths of the same length. Since h is only an estimate, however, we intuitively expect it to be more accurate as nodes get closer to the goal, and thus we expect the smaller h estimate to be more accurate. Thus, we would choose the node with the the smaller h value, or equivalently, the larger g value. To illustrate, consider two nodes with f values of 30: node A is at a depth (g) of 28 from the root and has an h estimate of 2 moves, whereas Node B is at a depth of 2 and has an h estimate of 28 moves. If we assume the heuristic function underestimates and has a constant relative error of 10%, then the expected f value of node A is $28 + 2 + .2 = 30.2$, while the expected f value of node B is $2 + 28 + 2.8 = 32.8$; thus node A is the better bet.

If we think of nodes in the search tree being explored in a 'left-to-right' manner, our primary motivation for using node ordering is to shift the first goal found to the left. However, in addition to this beneficial **shift effect**, node ordering also reduces the time to find a goal through the **depth effect**.

For each node expanded on the final iteration that does not lead to a goal within the threshold, a subtree of nodes must be explored to verify that fact. By picking a frontier node from the next-to-last iteration of minimum h, we reduce the average size of the subtree beneath it. This is because the maximum depth we can go below a non-goal node without a change in f is limited by the magnitude of h. In order for f to stay constant, as g (depth) increases, h must correspondingly decrease. The amount that h can decrease without reaching a goal, however, is limited by its starting value, since if h decreases to 0 it indicates a goal node. Thus, nodes with smaller h values will tend to have smaller subtrees under them within a given iteration. As a result, even if node ordering did not shift the goal to the left, the nodes explored to find the goal would still be reduced by the depth effect.

The shift and depth effects thus combine to reduce the number of nodes that must be explored unsuccessfully until a goal is found. This is analogous to the problem of searching for buried treasure on an island given the probability of finding treasure in each location as well as the cost required to dig in each location [SIMO75]. The goal is to maximize the treasure found and minimize the cost to find it. Thus it is not only important to find a correct location quickly, but also to minimize the time spent digging empty holes. Similarly, in a state space search, nodes should be ordered so that the goal is located under one of the first few nodes checked (shift effect), while at the same time minimizing the nodes explored under incorrect choices (depth effect). Fortuitously, in this case the same ordering maximizes both effects.

3.1.2 Limitation of serial IDA* node ordering

Even though good node ordering in IDA* may dramatically decrease the time spent in the last iteration, it can have no effect on the previous iterations. The reason is that if a goal is not found, the entire iteration must be completed. If we consider nodes to be explored in a 'left-to-right' manner, the best possible improvement occurs when the first goal found is the rightmost node in the unordered case and the leftmost node in the ordered case. Then the unordered search explores the entire goal iteration frontier, whereas the ordered search explores a single path during the goal iteration. This best case results in a $1/b$ reduction of the nodes generated in the unordered case, where b is the heuristic branching factor. b is formally defined

to be $\lim_{D\to\infty} \sqrt[D]{T(D)}$, where $T(D)$ is the number of leaf nodes at threshold D. This definition is the same as the definition of heuristic branching factor for two-player games in section 2.4, except D now represents number of iterations instead of depth. For single-agent search, the difference between the brute-force branching factor and the heuristic branching factor reflects the pruning effect of using a heuristic function, whereas in two-player games it reflects the pruning effect of using alpha-beta bounds.

Note that even with perfect ordering, in problem instances where the unordered search happens to have a goal as the leftmost node, there is no improvement with perfect ordering. In general, the goal in the unordered case will be some percentage of the way across the frontier, so perfect ordering would produce less than a factor of b improvement on the average.

3.1.3 Implementation of node ordering

The ideal ordering scheme is to fully exploit the information available from the next-to-the-last iteration. This can be done by saving the entire frontier of nodes from the most recent iteration, and then ordering them for expansion by maximum g. We collected data from 47 15-puzzle problem instances selected from [KORF85], and compared the nodes generated in the ordered and unordered cases. On the average, the ordered case generated only 0.2% of the nodes in the final iteration that were generated by the unordered case. Unfortunately, this approach to ordering is impractical since it must store the complete next-to-last iteration, requiring exponential memory and inordinate ordering overhead. It is of interest, however, because it reflects the best that can be achieved using all available information.

To avoid an exponential memory requirement, we save only a constant number of the 'best' nodes. We do this by saving an earlier frontier and dynamically reordering it based on later iterations. At the start, the root node is expanded to a relatively small, fixed frontier-set, on the order of 100-1000 nodes. This frontier-set is searched sequentially by doing a complete search of each frontier-set node to the current threshold. The deepest path achieved in searching under a frontier-set node is recorded, and is used to order the nodes in the following iteration. The frontier-set nodes are searched in decreasing order of the depth of this path (which corresponds to minimum h). A process searches first beneath the saved path, and then searches the rest of the nodes beneath the frontier-set node. This approach is also used in our parallel implementation of node ordering.

To evaluate node ordering empirically, we compared unordered IDA* with ordered IDA* using the ordering scheme just discussed. For the 50 easiest problems from [KORF85], the average ratio of nodes generated in the unordered case to the nodes generated in the ordered case was 1.83. Though almost a factor of two, this improvement is not of practical use because it is mitigated by the overhead associated with ordering. Our ordering program runs about 60% slower than our serial IDA* program (1.4 million versus 2.4 million node expansions per minute on a Hewlett Packard 9000/350 workstation), resulting in no real-time improvement. However, this technique might still be worthwhile with a better ordering scheme or for problem domains with a higher branching factor.

In the next section we discuss a complementary approach to improving IDA* which can overcome the limitations of serial IDA* ordering.

3.2 Pure Parallel Window Search (PWS)

The general idea of window search is to use different processes to search to different thresholds (windows) simultaneously, hoping that one of them will find a solution. In single-agent search, having each process search to a different threshold is equivalent to having each of them perform a separate iteration of IDA*, except that some may go beyond the goal iteration. Because thresholds are not explored sequentially, the first solution found may not be optimal. Optimality can still be guaranteed, however, by completing all shallower thresholds than that of the best goal found. This guarantee, however, requires at least the same amount of total work as serial IDA* does. Typically, it will require more since some processes may explore deeper than the optimal threshold and hence expand nodes that IDA* doesn't. Because the cost of guaranteeing an optimal solution is probably more expensive, PWS is not proposed as a competitor to IDA* for finding optimal solutions. Instead, it is a technique for finding very good, but not necessarily optimal, solutions. Even so, we will compare its speed with IDA*'s speed in the remainder of the paper because it is informative.

Unfortunately, this approach by itself produces limited improvement. The reason is that the search time will still be dominated by the time to perform the last iteration, even if the others are performed in parallel. In the next section, we discuss the expected improvement due to pure parallel window search.

3.2.1 Limitation of pure PWS

For the moment, assume that pure PWS finds only an optimal solution. What is the best improvement in elapsed time relative to IDA* that we can achieve? Assume there are enough processes so that the optimal solution threshold is being searched. Expected improvement can be analyzed as a function of goal location. The analysis shows that improvement relative to IDA* is approximately

$$1 + \frac{1}{a(b-1)}$$

where b is the heuristic branching factor, and a is the fraction of the frontier nodes that must be searched to find the first goal. In the extreme case of a rightmost goal, $a = 1$ and this reduces to $\frac{b}{b-1}$; this represents the lowest

possible improvement of pure parallel window search. For example, if b is 6, the minimum improvement is 1.2. If the goal is midway, $a = 1/2$ and improvement $= 1 + \frac{2}{b-1}$, or 1.4. As a approaches 0 improvement increases and then approaches infinity. This is because the time for window search to find a leftmost goal is insignificant compared to the time for IDA* to find a leftmost goal. The expected value of a will depend on the average number of goals, but in general it will not be low enough to produce large improvement. Hence, the time to search the goal iteration will limit the improvement of pure parallel window search.

3.2.2 The density effect

In the previous section we only considered the improvement for finding an optimal solution. Now we consider whether a non-optimal solution might be found before an optimal solution. Consider two identical processors, P_0 and P_1. P_0 searches to the optimal solution threshold and P_1 searches one threshold past optimal. If the goal of P_1 is located the same fraction of the way across its frontier as the goal of P_0 (that is if $a_0 = a_1$, where a is as defined in the previous section), then P_0 will find a solution first because P_1 must explore a factor of b times more frontier nodes to find the goal than P_0, where b is the heuristic branching factor. However, if P_1's goal is shifted to the left (again exploring in a 'left-to-right' manner) so that a_1 is reduced to less than $\frac{a_0}{b}$, then the non-optimal solution will be found first.

In general, there may be many solutions, both optimal and non-optimal. If the average non-optimal solution density is greater than the average optimal solution density, we would expect to find a non-optimal solution first. Average solution density is a function of the problem domain. In the 8 and 15 puzzle problems, for example, the average non-optimal solution density appears to be higher than the optimal solution density, at least for several thresholds past optimal. In the 8 puzzle, for 1000 random problem instances, we measured the average ratio of non-optimal goal density to optimal goal density for each of the first 5 thresholds past optimal. For thresholds 1, 2, 3, 4, and 5 past optimal, the ratios were 1.20, 1.37, 1.55, 1.68, and 1.87, respectively.

Because of the size of the search space, we were unable to get similar data for the 15-puzzle, but we believe the corresponding numbers for the 15-puzzle are even higher because almost all the initial solutions found by our implementation of PWS are non-optimal.

Thus, solution density also affects the time for parallel window search to find its first solution. In problem domains which have relatively high non-optimal solution density, this effect will increase the average improvement possible from window search.

3.3 Parallel Window Search With Node Ordering

Fortuitously, the limitations and strengths of window search and node ordering are complementary. Node ordering is limited by the non-goal iterations but performs the goal iteration very efficiently. Conversely, pure parallel window search is limited by the time to perform the goal iteration and incurs no additional cost for the non-goal iterations. This suggests that a combination of the two approaches might be effective. We combine node ordering with pure parallel window search and refer to the combination as simply parallel window search. Basically, once a process completes its iteration, it broadcasts the resulting ordering information to the other processes, and then searches the next unsearched threshold. Processes always use the ordering information of the most recently completed threshold to guide their current search. The algorithm is described in more detail below.

3.3.1 Parallel window search algorithm

Given P processes, each process is assigned a different one of the first P thresholds (windows) to search. Processes use the ordering scheme of section 3.1.3; at the start, each process expands the root node to an identical frontier-set, and then searches the frontier-set to the assigned threshold. The deepest path achieved in searching under each frontier-set node is recorded, along with the associated minimum h value. When its entire frontier-set has been searched to a threshold, the process broadcasts ordering information to all other processes. The message consists of: (1) a minimum h value for each of the frontier-set nodes, and the associated path, and (2) the value of the threshold on which the ordering information is based. The process also saves the ordering information for itself. Then the process 'leapfrogs over' the other processes to the next threshold to be searched. When more than one ordering message is received by a process, only the message associated with the deepest threshold is saved. A process picks unsearched frontier-set nodes of minimum associated h value based on the best ordering information received.

At some point, a process will find a solution. After finding a solution, those processes searching smaller thresholds than that of the best solution found continue searching. In this manner, the solution is improved until it is verified to be optimal. Verification of optimality requires completing all thresholds less than optimal; thus the time between finding an optimal solution and verifying its optimality can be large.

Because of the sequence of finding solutions, we can identify three milestones in program execution: when the first solution is found, when the first optimal solution is found, and when optimality is verified. In some cases the first two milestones occur together; that is, sometimes the first solution found is optimal. At program completion, PWS exits with a verified optimal solution, just as IDA* does. The overall difference is that even though PWS may perform more work in guaranteeing an optimal solution, it will

find non-optimal solutions and an optimal solution quickly in the course of guaranteeing an optimal solution.

3.3.2 Analysis of PWS algorithm with perfect ordering

What is the minimum time required by our PWS algorithm on the average to find a solution in the case of perfect ordering? We make the following assumptions:

(1) the complexity of finding any solution is an exponential function of depth (otherwise, we could find a solution in linear time), (2) ordering information from completed thresholds is perfect, (3) ordering information below completed thresholds is random, and (4) when a process receives new ordering information it restarts its search using the new information. Assume that the first solution is found by the process that searched the D^{th} threshold; alternatively, D is the length of the first solution found. D may or may not be optimal.

Now consider the process that searches to threshold D and finds the first solution. When it finds the solution, it will be using ordering information from some threshold d. (see Figure 2). Furthermore, since ordering information is perfect, it will find the solution under the first frontier node of threshold d; call this node x. However, since it has no further ordering information, its search below node x is randomly ordered (assumption 3). Thus, on the average it will have to search $O(b^{D-d})$ nodes below node x to find the goal, where b is the heuristic branching factor.

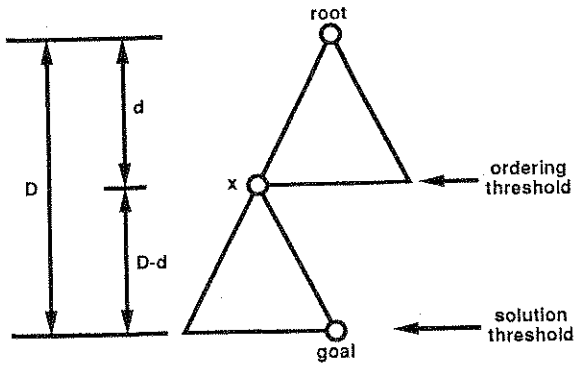

Figure 2: Space Searched in Finding Solution

Because a process is interrupted and restarts when it receives new ordering information, each process searching to a depth d will be interrupted by all processes searching to a shallower depth. This implies that the time to search to depth d is the sum of the times to search to each shallower depth.

Since the tree grows exponentially, this has no effect on the asymptotic time complexity, which is still $O(b^d)$.

How long does it take to find a solution under these assumptions? The total time is the time for the ordering process to complete its search to depth d, plus the time for the solution process to complete the search below node x, which is of depth $D - d$. This is $O(b^d + b^{D-d})$. For small values of d, the running time is dominated by the time to perform the unordered search below depth d. For large values of d, the running time is dominated by the time to determine the ordering information. Note that ordering information doesn't help the ordering process, since the entire ordering threshold must be completed. The running time is minimized when $d = D/2$, which balances the two searches. This results in an overall complexity of $O(b^{D/2})$ in the best case.

In the above analysis, we defined D as the length of the first solution found. If we changed the definition of D to be the length of an optimal solution, then the same result applies to finding optimal solutions.

Consider the minimum number of processes required to achieve this $O(b^{D/2})$ time complexity. All we need is enough processes so that the solution process can start as soon as the ordering process completes, or sooner. This is achieved with a minimum number of processes when the ordering process leapfrogs directly to become the solution process; this requires $D/2$ processes. Thus, $D/2$ processes are sufficient to find a solution in $O(b^{D/2})$ time. Note that having too few processes will delay the availability of a process for the solution threshold and thus increase the time to find a solution.

What is the effect of having more than the minimum number of required processes? Let D be the length of the optimal solution and consider what happens as we add more than $D/2$ processes. As discussed in section 3.2.2, if non-optimal solution density is greater than optimal solution density, a non-optimal solution is likely to be found first and reduce the time to the first solution. On the other hand, if we add processes which search thresholds with relatively low non-optimal solution densities, then these processes will be unproductive since shallower solutions will generally be found first. This implies that, depending on the problem domain, we may want to limit the number of windows for efficiency. Note that adding extra processes cannot possibly increase the elapsed time to find a solution, but it may increase the total work done and thus decrease the efficiency of process work.

3.3.3 Empirical results for first solution found

We ran the initial 93 of 100 problems of [KORF85] (ordered by nodes generated by serial IDA*) using a number of processors ranging from 5 to 9, and measured the time to find the first solution. The 7 most difficult problems weren't used because they would take too long to solve using 5 or 6 processors since this number is less than the minimum required as discussed in the previous section. All 100 problems, however were run using 7-9 processors

with consistent results. In each case, we calculated the average exponential reduction in nodes generated. Since the serial search requires $O(b^D)$ time and the parallel search in the best case requires $O(b^{D/2}) = O((b^D)^{.5})$, where b is the heuristic branching factor, we expect the exponential reduction to be at least .5 (a lower exponent implies a greater reduction).

For 5, 6, 7, 8, and 9 processors, the corresponding average exponents were .8, .78, .75, .75, and .73, respectively. That is, we see a reduction of the nodes generated to about $b^{3/4D}$. We believe the exponent monotonically decreases for two reasons: (1) for the most difficult problems, adding extra processes helps achieve the minimum number of processes required (section 3.3.2); and (2) adding additional processes reduces the time to a solution because of the density effect (section 3.2.2). The consistency of this reduction lends support to the claim that parallel window search reduces the exponential complexity of IDA* for finding non-optimal solutions.

To give an idea of specific performance, when we ran all 100 problems using seven Hewlett Packard 9000/320 workstations, the average nodes generated in finding a solution was 1.6 million compared to an average of 357 million nodes generated in IDA*. More specifically, the most difficult problem IDA* solved required 42 hours and 6.01 billion node generations. Using 7 processors, our program found a solution to this problem that was 4 moves from optimal (6%) in 4 minutes and 6.5 million node generations (by the process finding the solution). If total processor effort is considered and not just elapsed real time, this corresponds to 28 minutes and 45 million nodes generated; this is the time a single processor running PWS with 7 processes would take.

The relative speed of the program is only one measure of its performance, with the quality of solutions being the other primary measure. For seven processors, the solution lengths range from optimal to twelve moves over optimal, with the mode and average being six moves over optimal. Since the average optimal solution length for this puzzle is 53 moves, the first solutions found are on the average within 11 percent of optimal.

To get a better indication of the strength of PWS in finding non-optimal solutions, we also compared it to RTA* [Korf, 1988/89], a real-time variant of A* in which optimality is sacrificed in order to make real-time moves within fixed time limits. In a sense, this is an unfair comparison since RTA* only searches from its current position in the sequence of moves made so far, but it is still instructive.

We compared the quality of solutions of PWS with 7 processors and RTA* as follows. RTA* was run with increasing search horizons until it generated more nodes in finding a solution than all 7 processors of PWS. After removing any cycles from the solution, its length was compared to the PWS solution length. On the average the PWS solution lengths were half (47%) of the RTA* solution lengths. In only three cases was the RTA* solution shorter than the initial PWS solution, and in those cases it was 6 moves less in one case and 2 moves less in the other two cases. These results show that

PWS is not dominated by either IDA* or RTA*, and thus it appears to be a competitive algorithm in terms of search effort versus solution quality.

4 CONCLUSIONS

We have presented two different approaches to parallel heuristic search. Distributed tree search is a generic algorithm that can be easily specialized to different types of search problems and different processor allocation strategies. The algorithm produces near perfect speedup in brute-force searches of irregular trees without relying on centralized control or shared memory. We have shown that under a breadth-first processor allocation strategy, the speedup achievable with parallel branch-and-bound is proportional to P^X, where P is the number of processors, and X is a measure of how effective the pruning is. We also introduced a novel processor allocation strategy for parallel alpha-beta called Bound-and-Branch that does no more work than serial alpha-beta in the case of perfect node ordering and in general increases speedup as the ordering technique improves. These algorithms have been implemented to perform alpha-beta search on a Hypercube and currently produce speedups of 12 on a 32-node Hypercube.

Instead of dividing the search tree among processors, parallel window search gives the entire tree to each processor, along with different cost bounds to use. In the second part of this paper, we considered how to apply this idea to single-agent search. We showed that the effectiveness of node ordering for improving the efficiency of IDA* is limited by the time to complete the previous iterations of the search. Conversely, window search in which different processes simultaneously perform different iterations, is limited by the time to complete the final iteration. Combining effective node ordering with window search by sharing ordering information among processes makes it possible to find near-optimal solutions quickly. If the algorithm continues to run past the first solution found, it finds increasingly better solutions until an optimal solution is guaranteed or the available time is exhausted.

Parallel window search can easily be combined with distributed tree search in either single-agent or two-player game applications. The combination is best viewed at the top level as a parallel window search. Each window uses DTS to share the work among multiple processors. As various windows complete their searches, the freed processors are reallocated to participate in the ongoing searches. For example, in single-agent parallel window search, more than one processor can search to a given threshold; furthermore, once a solution is found, all the processors searching to equal or greater thresholds may be terminated and re-assigned to shallower thresholds.

5 ACKNOWLEDGEMENTS

Draft reviews by and discussions with Bob Felderman, Othar Hansson, and Vipin Kumar substantially improved the paper. Discussions with Andy Mayer also contributed to this research. Also, special thanks to K.P.B. Severance, to Greta Wright, and to Jack and Barbara Powley.

References

[BAUD78] Gerard Baudet. 'The Design and Analysis of Algorithms for Asynchronous Multiprocessors'. Ph.D. dissertation, Computer Science Department, Carnegie-Mellon Univ., Pittsburgh, Pa., April 1978.

[EBEL87] Carl Ebeling. *All The Right Moves*, MIT Press, Cambridge, Mass., 1987.

[FELD89] R. Feldmann, B. Monien, P. Mysliwietz, and O. Vornberger. 'Distributed Game Tree Search'. *Parallel Algorithms for Machine Intelligence*, editors: Kanal, Kumar, and Gopalakrishnan. Springer-Verlag, 1989.

[FELT88] E. Felten and S. Otto. 'A Highly Parallel Chess Program', Proceedings of the International Conference on Fifth Generation Computer Systems, Tokyo, 1988.

[FERG88] Chris Ferguson and Richard E. Korf. 'Distributed Tree Search and its Application to Alpha-Beta Pruning', *Proceedings of the Seventh National Conference on Artificial Intelligence (AAAI 88)*, Saint Paul, Minnesota, pages 128-132, August, 1988.

[FINK82] Raphael A. Finkel and John P. Fishburn. 'Parallelism in Alpha-Beta Search'. *Artificial Intelligence*, Vol. 19, No. 1, pages 89-106, Sept. 1982.

[FINK87] Raphael Finkel and Udi Manber. 'DIB - A Distributed Implementation of Backtracking', *ACM Transactions on Programming Languages and Systems*, Vol. 9, No. 2, pages 235-256, Apr. 1987.

[HART68] P. E. Hart, N.J. Nilsson, and B. Raphael. 'A Formal Basis For The Heuristic Determination of Minimum Cost Paths'. *IEEE Trans. Systems Sci. Cybernet*, 4(2), pages 100-107, 1968.

[KORF85] Richard E. Korf. 'Depth-First Iterative-Deepening: An Optimal Admissible Tree Search'. *Artificial Intelligence*, Vol. 27, pages 97-109, 1985.

[KORF88] Richard E. Korf. 'Real-time Heuristic Search: New Results'. *Proceedings of the Seventh National Conference on Artificial Intelligence (AAAI 88)*, 1988. Vol. 25, pages 97-109, 1985.

[KORF89] Richard E. Korf. 'Real-time Heuristic Search'. *Artificial Intelligence*, to appear, 1989.

[KUMA84] Vipin Kumar and Laveen N. Kanal. 'Parallel Branch-and-Bound Formulations for AND/OR Tree Search'. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Vol. PAMI-6, No. 6, pages 768-778, November, 1984.

[KUMA88] Vipin Kumar and V. Nageshwara Rao. 'Parallel Depth-First Search, Part II. Analysis'. *International Journal of Parallel Programming*, Vol. 16(6), pages 501-519, 1987.

[PEAR85] Judea Pearl. *Heuristics: Intelligent Search Strategies for Computer Problem Solving*. Addison-Wesley, 1985.

[POWL89] Curt Powley and Richard E. Korf. 'Single-Agent Parallel Window Search: A Summary of Results'. *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence (IJCAI 89)*, Detroit, Michigan, 1989.

[RAO87] V. Nageshwara Rao, Vipin Kumar, and K. Ramesh. 'A Parallel Implementation of Iterative-Deepening-A*' *Proceedings of the Sixth National Conference on Artificial Intelligence (AAAI 87)*, Seattle, Washington, pages 178-182, July 1987.

[RAO88] V. Nageshwara Rao and Vipin Kumar. 'Parallel Depth-First Search, Part I. Implementation'. *International Journal of Parallel Programming*, Vol. 16(6) pages 479-499, 1987.

[SIMO75] Herbert A. Simon and Joseph B. Kadane. 'Optimal Problem-Solving Search: All-or-None Solutions'. *Artificial Intelligence*, Vol. 6, pages 235-247, 1975.

[SLAT77] D. J. Slate, L. R. Atkin. 'CHESS 4.5 - The Northwestern University Chess Program', Springer-Verlag, New York, 1977.

[VORN87] O. Vornberger, 'Parallel Alpha-Beta versus Parallel SSS*'. *Proceedings of the IFIP Conference on Distributed Processing*, Amsterdam, October 1987.