Distributed Tree Search and its Application to Alpha-Beta Pruning*

Chris Ferguson and Richard E. Korf
Computer Science Department
University of California, Los Angeles
Los Angeles, Ca. 90024

Abstract

We propose a parallel tree search algorithm based on the idea of tree-decomposition in which different processors search different parts of the tree. This generic algorithm effectively searches irregular trees using an arbitrary number of processors without shared memory or centralized control. The algorithm is independent of the particular type of tree search, such as single-agent or two-player game, and independent of any particular processor allocation strategy. Uniprocessor depth-first and breadth-first search are special cases of this generic algorithm. The algorithm has been implemented for alpha-beta search in the game of Othello on a 32-node Hypercube multiprocessor. The number of node evaluations grows approximately linearly with the number of processors P, resulting in an overall speedup for alpha-beta 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. Using this strategy, an actual speedup of 12 is obtained with 32 processors.

1 Parallel Search Algorithms

Heuristic search is a fundamental problem-solving method in artificial intelligence. 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.

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 [2]. 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.

Another approach, called parallel window search, was pioneered by Gerard Baudet [1] in the context of two-player games. In this algorithm, different processors are assigned non-overlapping ranges for alpha and beta, with one processor having the true minimax value within its window, and finding it faster by virtue of starting with narrow bounds. Unfortunately, this approach 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 is fairly expensive. In experiments, its speedup is limited to about five or six, regardless of the number of processors.

The third, and most promising approach for large numbers of processors, 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. The most recent experimental work on this paradigm is that of Kumar et. al. [6] in parallelizing IDA* [5], a linear-space variant of A*. 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 total amount of work that must be done is independent of the order in which the tree is searched.

This is not true of branch-and-bound algorithms such as alpha-beta pruning, since 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. Finkel and Manber [4] present a generalized tree search algorithm similar to ours, however, they do not allow explicit control over the allocation of work among processors, and hence they do not achieve high speedup for branch-and-bound algorithms. The best work to date on the specific problem of parallel alpha-beta search has been presented by Oliver Vornberger [9]. He achieves a relatively large speedup of 8 on 16 processors for evaluating chess positions.

2 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,

*This research was supported by an NSF Presidential Young Investigator Award to the second author, Jet Propulsion Laboratory contract number 957523, and by the Defense Advanced Research Projects Agency under contract MDA 903-87-C0663, Parallel Systems Laboratory.

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.

3 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.

4 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.*

5 Analysis of a Simple Model

Consider a parallel branch-and-bound algorithm on a uniform tree with brute force branching factor B and depth D. The *effective 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 effective 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 effective branching factor of about $b = B^{.75}$ [7]. Interestingly, 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 $O(P^X)$.*

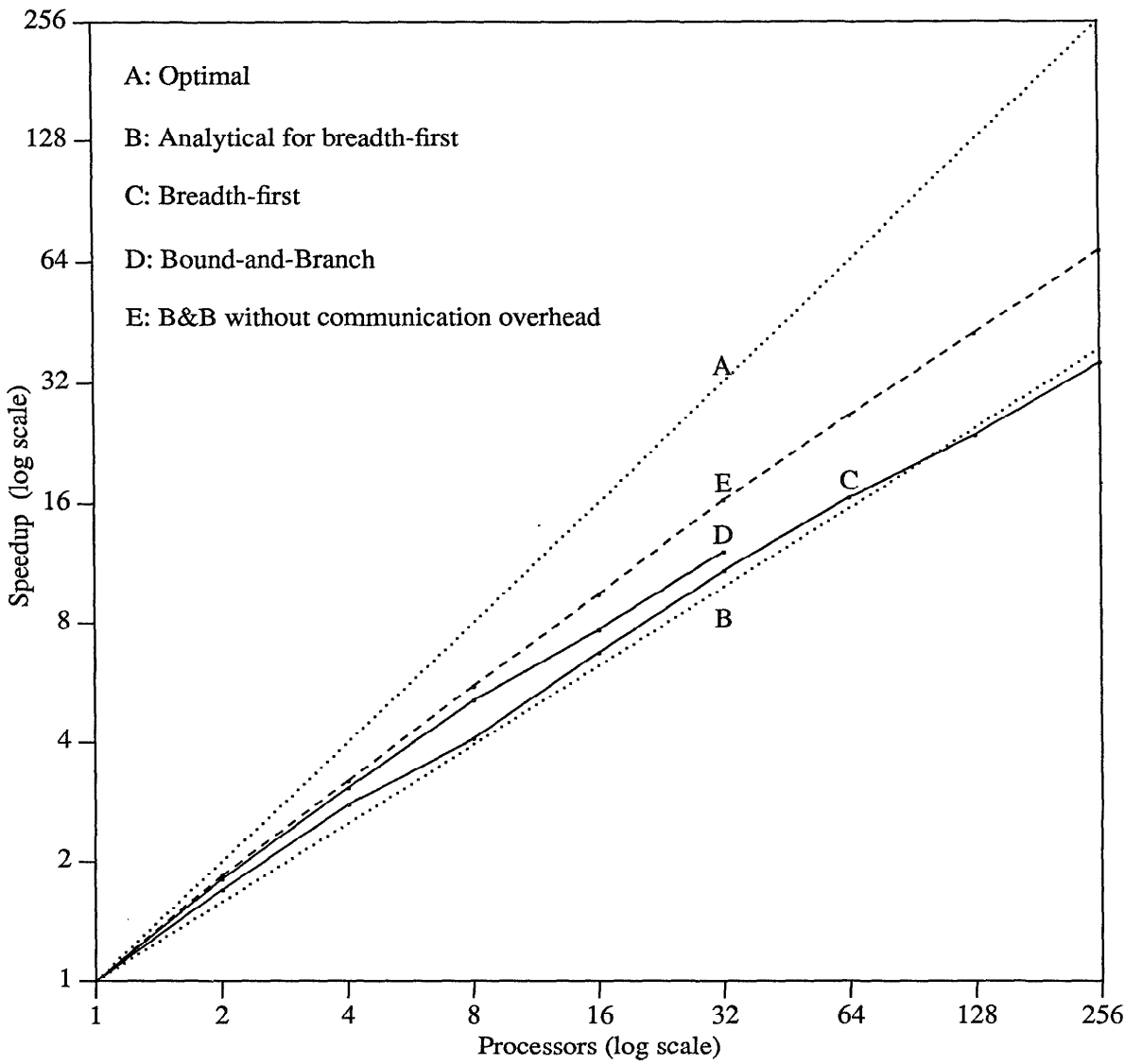

Figure 1

Graph of Speedup Versus Number of Processors

Proof: The speedup of a parallel algorithm is the time taken by a serial algorithm, divided by the time taken by the parallel algorithm. The serial algorithm will evaluate 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 $\log_B P$. Each one of these processors will evaluate $O(B^{X(D-\log_B P)})$ nodes even if no new bounds are passed to it since each is searching a tree of depth $D - \log_B P$ by itself. Thus, this simple parallel algorithm takes time proportional to $B^{X(D-\log_B P)}$. Therefore, the speedup is on the order of $B^{X \log_B P}$, or $O(P^X)$.

6 Breadth-First Allocation

We have searched 40 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 an effective 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 [3]. 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 [2] 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.

7 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 algorithm 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, the Bound-and-Branch allocation strategy will evaluate the same nodes as serial alpha-beta.*

Proof: In perfect node ordering, the first move searched under a node is the best move from that position. Furthermore, once a bound is established for a node, it can never be improved upon without refuting a first child as the best possible move. For a node whose initial bound is a cutoff bound, this bound renders the children of that node independent from one another in the sense that searching any first will never improve the bounds at their parent, and thus cannot cause more cutoffs to occur when searching the others. This implies that these nodes can be searched in parallel without causing any extra evaluations to occur. Thus, since the Bound-and-Branch allocation strategy only branches out in parallel when a cutoff bound is available, no extra node evaluations can occur.

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 [8]. 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 ends up evaluating 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. 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.

8 Conclusions

We have presented a generic distributed tree search 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 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 in 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.

References

[1] G. Baudet, "The design and analysis of algorithms for asynchronous multiprocessors", Ph.D. dissertation, Dept. Computer Science, Carnegie Mellon University, Pittsburgh, PA., Apr. 1978.

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

[3] R. Finkel and J. Fishburn, "Parallelism in Alpha-Beta Search", *Artificial Intelligence*, Vol. 19, No. 1, sept. 1982.

[4] R. Finkel, U. Manber, "DIB - A Distributed Implementation of Backtracking", *ACM Transactions on Programming Languages and Systems*, Vol. 9, No. 2, Apr. 1987.

[5] R. E. Korf, "Depth-first iterative-deepening: An optimal admissible tree search", *Artificial Intelligence*, Vol. 27, No. 1, 1985, pp. 97-109.

[6] V. Nageshwara Rao, V. Kumar, K. Ramesh, "A Parallel Implementation of Iterative-Deepening A*", *Proceedings of the National Conference on Artificial Intelligence (AAAI-87)*, Seattle, Wash., July 1987, pp. 133-138.

[7] Judea Pearl, *Heuristics*, Addison-Wesley, Reading, Mass., 1984.

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

[9] O. Vornberger, "Parallel Alpha-Beta versus Parallel SSS*", *Proceedings of the IFIP Conference on Distributed Processing*, Amsterdam, Oct. 1987.