A STUDY OF PARALLEL TREE SEARCH ALGORITHMS

T.A. Marsland
and
M. Campbell

Technical Report TR82-4

July 1982

A study of parallel tree search algorithms

T.A. Marsland
and
M. Campbell

Computing Science Dept.
University of Alberta
EDMONTON

22July1982
TR82-4

Abstract

A basis for the comparison of algorithms for sequential and parallel search of game trees is presented, one which provides measures of performance on cases of theoretical and practical interest. A number of sequential tree searching algorithms are reviewed and extended. To provide a foundation for the development of parallel search algorithms simulated concurrency of multi-processor systems is used. The comparison involved the generation of a number of independent trees with certain desired properties upon which the algorithms were tested.

KEYWORDS: parallel alpha-beta, aspiration search, strongly ordered trees, minimal window, tree splitting, principal variation splitting, staged SSS*.

A. Introduction

Many game-playing programs build and carry out searches on large trees of possible move sequences. In games like chess it has become clear that increasing the depth of the tree searched can vastly improve the playing ability of a given program [THOM82]. Further reductions in search time may be possible through the use of multiple processors. However, the application of parallelism to game tree search is non-trivial, due to the inherently sequential nature of the most popular search method, the alpha-beta algorithm [KNUT75]. In this paper a number of different algorithms for searching game trees are reviewed. Revisions and extensions are considered which make the algorithms suitable for implementation on multiprocessor systems. These parallel algorithms are compared by measuring their efficiency in searching trees with specified characteristics, particularly random and strongly ordered trees [MARS81].

Games such as chess are classified as two-person zero-sum games of perfect information and produce trees of the type studied here. For any given position p, in such a game, it is possible to represent all the potential continuations from p in the form of a *game tree*. The nodes of the tree correspond to positions, while the branches (edges) represent the moves. The leaves of a game tree are called *terminal* nodes, and are assigned a value by an *evaluation function*. All the others are classified as *interior* nodes. The number of branches leaving any particular interior node is the *branching factor* of that node. A node is at *depth k* if it is k moves, or k *ply*, from the root. Thus, a *uniform game tree* is one in which all interior nodes have the same branching factor, and all terminal nodes are at the same depth in the tree.

B. Sequential Tree Searching Algorithms

The goal of a game tree search is to determihe the *minimax* value of the *root* node. Intuitively, this value is the best score achievable from that node against an opponent who similarly chooses moves which yield his best score. The **minimax algorithm** assumes that there are two players called Max and Min, and assigns a value to every node in a game tree (and in particular to the root) as follows: Terminal nodes are evaluated and a single number is used to represent the desirability of the position from Max's point of view. Working back from the terminal nodes, if Max is to move the value of an interior node p is the maximum over the values of the successors of p. Similarly, if Min is to move the value is the minimum over the values of the successors of p.

The **alpha-beta algorithm** is an implementation of the minimax procedure which employs two bounds, alpha and beta, to control the size of the search. The algorithm is generally implemented in the negamax framework [KNUT75], in which an interior position is calculated as the maximum of the negatives of its successors. The importance of the alpha-beta algorithm stems from its ability to evaluate a game tree at reduced cost by ignoring subtrees that cannot affect the final value of the root node. Such subtrees are said to have been *cut off*. This cut off occurs whenever the score returned from a subtree is greater than beta. The interval enclosed by (alpha,beta) is referred to as the alpha-beta *window*. For the algorithm to be effective, the minimax score of the root node must lie within the window, and this is guaranteed if the initial range is from -infinity to +infinity. Generally speaking the narrower the initial window the faster the search. This provides the motivation for *aspiration searching*, in which the window may be initialized to (V-e,V+e), where V is an estimate of the minimax value and e the expected error [MARS81a]. There are three possible outcomes of an aspiration search on a position p. Either the search fails high or low, or it succeeds. In the latter case the true score of p is found. Searches that fail must be repeated with a window that actually encloses the minimax value for p.

In the development of refinements to alpha-beta, the concept of a *minimal window* [FISH80] was introduced. If scores can only take integer values, then (m,m+1) is an example of a minimal window, and a search using this window will necessarily fail high or low. Though the true score of a position p cannot be found by a minimal window search, it does provide a bound on the score (that is, it determines whether or not negamax(p) > m), while making cut offs that a full window search cannot. For example, if the score of one successor of p is found to exceed a bound m, an immediate cut off can occur without searching the remaining successors of p for the one that exceeds m by the most. In many circumstances a bound of this type on a position is sufficient to cause a cut off elsewhere.

Palphabeta, for "principal-variation alphabeta" [FISH80], is a generalized application of minimal window searching. It can, under certain circumstances, re-examine nodes that have already been evaluated. This occurs whenever the minimal window search does not cause the anticipated cut off. A slightly more symmetric version of the algorithm exists, **SCOUT** [PEAR80], but is not otherwise significantly different. If the first path to a terminal node is in fact the optimal sequence of moves predicted by minimax, the balance of the tree is searched with a minimal window. However each time a minimal window search on a subtree fails high, the search is repeated. Hence there is some risk, if the tree is poorly ordered, that these algorithms will visit more terminal nodes than alphabeta. Since there are techniques, particularly *iterative deepening* [MARS81a], which can provide a good approximation to the actual principal variation with reasonable reliability, algorithms based on palphabeta can be quite productive.

SSS* [STOC79] is an algorithm for determining the minimax value of AND/OR trees, of which game trees are a special case. It is claimed that SSS* *dominates* alpha-beta in terms of terminal nodes evaluated. That is to say, SSS* never evaluates a node that alpha-beta can ignore [STOC79], and this is indeed the case if a simple modification is made to the algorithm [CAMP81]. However SSS* requires a very large data structure so that a number of alternate solution paths throughout the tree may be maintained. One proposal to reduce this storage requirement employs SSS* to some fixed depth D, whereupon the 'terminal nodes' at D ply are evaluated by a further SSS* search of depth D [CAMP81]. The staging reduces storage requirements so that they are linear with search depth. This approach uses SSS* in layers, or stages, and will be called *staged SSS**. The primary disadvantage of this and other refinements to SSS* is the fact that a lower bound is not always available on any given node's score.

In uniform trees of width W and depth D, for which the W**D terminal nodes are independent identically distributed random variables (with a continous distribution function), a formula for the average number of terminal positions evaluated by alpha-beta has been developed [FULL73]. This formula is computationally intractable, however, and can only be calculated for small values of W and D. Thus for trees of practical depths there is no computationally acceptable performance measure. At present, only **empirical methods** are available to study searching performance on trees with varying types of ordering properties. In one study [CAMP81] a number of algorithms were compared: alphabeta, palphabeta (**PAB**), SCOUT, SSS* and staged SSS*, using a variety of tree sizes and various assumptions about placement of the best move. In particular, random ordering, moderate ordering (geometric with parameter 0.5), strong ordering (0.8 probability the first move is best) [MARS81], and perfect ordering. As expected SSS* and its variations are superior on random trees, while PAB is more effective for well ordered trees.

C. Approaches to Parallel Tree Search

There are a number of methods for applying parallelism to game tree search. Though this paper is primarily concerned with tree decomposition methods, some other possibilities should be mentioned.

C.1 Parallelism in Primitive Operations

Two basic operations needed by programs that search game trees are *move generation* and *terminal node evaluation*. Both these functions are promising sites for the use of special purpose multi-processors, particularly in chess. Parallel chess move generation [CORA76], and parallel evaluation [MARS81] have been considered. However, it is important to note that in these cases cooperation between processors is occurring at a very low level, requiring highly specialized interconnection mechanisms.

C.2 Parallel Aspiration Searching

The basis of aspiration searching is the improved performance of the alpha-beta algorithm on a restricted window. Aspiration searching has a parallel counterpart, e.g., searching a number of (disjoint) windows simultaneously. The advantage of this method is that the concurrent searches are relatively independent, reducing the need for a complex communication scheme. The main difficulty with this approach is that the overall search time is bounded below by the search time for alpha-beta under optimal ordering conditions, i.e. there is a minimal tree that must be examined in any successful search. Therefore, regardless of the number of processors available, there is a fixed maximum speedup possible. A typical bound on speedup is a factor of five or six [BAUD78].

In any parallel searching algorithm using the window concept, parallel aspiration search is also applicable. We will omit further mention of parallel aspiration search, on the understanding that it is an additional enhancement which can usually be employed.

C.3 Tree Decomposition

Most discussions of parallel game tree search have concentrated on concurrent examination of independent subtrees. Baudet concludes that parallel aspiration searching must be combined with tree decomposition if large performance improvements are sought [BAUD78]. However there are a number of overheads involved in concurrent search of different subtrees. These overheads can be divided into two broad categories, namely *search overhead* and *communication overhead*.

The efficiency of most search algorithms hinges on the fact that decisions to cut off search on given subtrees are based on all the accumulated information obtained to that point in the search. For various reasons, this information is not always available to parallel search algorithms. Communication delays may make the data arrive too late, or, more importantly, information may not yet be available if it is still being calculated by another concurrent search. The extra effort that a given parallel algorithm must carry out (relative to the sequential algorithm) can be defined as the *search overhead*. A convenient numerical measure of this overhead is defined by:

Let N(A(k),T) be the number of terminal nodes scored by a parallel algorithm A(k) when using K processors to search some game tree T. Then

$$S(A(k),T) = N(A(k),T)/N(A(1),T)$$

is called the *search overhead coefficient* of algorithm A(k) on tree T.

Note that, in general, one would expect S > 1 for k > 1, though this is certainly not always the case. The quantity S provides an indication of how efficiently a searching algorithm distributes information dynamically among the cooperating processors on a particular game tree.

Communication overhead can arise in different ways, depending on the system configuration. Information can be exchanged either via some sort of message passing system, or through a global shared data structure. The former incurs message passing costs, while the latter will require synchronization if a reasonable degree of concurrency is to be maintained. Although the information to be shared is dependent upon the particular search algorithm used, it seems clear that communication overhead is inversely related to search overhead. In other words, if improved sharing of data between independent searches is achieved (at increased communication costs), better cut off decisions can be made by the search algorithm, thus reducing search overhead.

D. Algorithms for Parallel Search

Before discussing parallel search algorithms, it is necessary to state some assumptions about the underlying processor architecture. Tree searching multi-processor systems can be classified into two basic categories, depending upon how they decompose trees for concurrent search. *Static decomposition* systems generate and assign subtree searches in a fixed, pre-determined manner, while *dynamic decomposition* systems assign subtree searches conditional on the current status of the overall search.

An architecture suitable for static decomposition is the *processor tree* [FISH80]. A processor tree consists of processors (the nodes of the tree) and communication lines (the branches of the tree). The successors of a node are its *slaves*, while the predecessor of a node is its *master*. The *root* processor has no master. From this description it is clear that a given processor can communicate directly only with its master and slaves (if any). The processor tree architecture is an excellent one from the implementation point of view. There are limited interconnection requirements for each processor, independent of the total number of processors in the system. Also, the number of processors is extendable in a simple and regular fashion, by increasing the width and/or depth of the tree. The processor tree also provides a fairly flexible means to control the subtrees searched. If, for example, a master processor wants a sub-subtree to be evaluated, it can simply assign one of its slaves (and thereby all the slave's descendants) to the search.

An architecture that employs a dynamic decomposition system has been suggested [AKL82]. Processes (nodes to be searched) are kept in a priority ordered set. Whenever a processor comes available it is allocated to the highest priority process. Dynamic decomposition can reduce the processor idle time substantially, and provides the maximum possible flexibility in directing search towards specific desired subtrees. On the other hand, such a processor pool causes a number of implementation difficulties. In particular, a means of selecting and suspending processes must be found, one which does not involve inordinate synchronization and storage overheads. In addition, an efficient scheme must be devised which enables a periodic information update to currently running processors, a relatively trivial matter in the processor tree architecture. Finally, substantial storage, and its attendant management problems, may be required to hold the intermediate positions temporarily abandoned by processors that are re-assigned.

The algorithms used in this study are based on a processor tree, static decomposition architecture. These choices were made because they appear to be more practical.

D.1 Tree-splitting

One use of a processor tree to implement alpha-beta is called the *tree-splitting algorithm* [FISH80]. In this algorithm, a master processor generates all the successors of a given position, and assigns them to its slave processors. Terminal slaves will carry out a regular alphabeta search on their assigned position, while interior slaves will again generate and assign successors. Master processors maintain a local alpha-beta window, which they pass to their slaves along with a search assignment. The windows are updated when slaves return values from their searches.

This application of a processor tree does have some drawbacks. The width and depth of the tree are bounded by the width and depth of the game tree being searched. However, we will show that processor trees with large fanouts have greater search overheads. Therefore the tendency is to prefer deep, narrow tree structures to wide, shallow ones. For this reason, the maximum depth restriction is likely to be the more serious one.

Although processor trees are relatively powerful at directing search towards relevant game subtrees, there is some difficulty with processor idle time, since a given processors' descendants cannot be reassigned until the initial search is completed. This idle time is directly related to the processor tree width.

Figure 1 illustrates the tree-splitting algorithm in a pseudo code based on the C language. Several constructs have been adapted from the original version [FISH81].

1. j.treesplit indicates the execution of procedure treesplit on processor j.
2. **parfor**, a parallel loop which conceptually creates a separate process for each iteration of the loop. The program continues as a single process when all iterations are complete.
3. **when** waits until its associated condition is true before proceding with the body of the statement.
4. **critical** allows only one process at a time into the critical region.
5. procedure *terminate* kills all currently active processes in

```
treesplit(position p, int α, int β)
{   int w, i, t[MAXWIDTH];
    processor j;
    if (I am a leaf processor)
        return(alphabeta(p, α, β));
                                 /*                         */
    w = generate(p);             /*  determine successors   */
                                 /*      p.1 ... p.w        */
    parfor i = 1 to w do {
        when (a slave j is idle) {
            t[i] = -j.treesplit(p.i, -β, -α);
            critical {
                if (t[i] > α) α = t[i];
            }
            if (α ≥ β) {
                terminate();
                return(α);
            }
        }
    }
    return(α);
}
```

Figure 1: Linear allocation of processors.

```
int alpha[MAXDEPTH], beta[MAXDEPTH];
/*
  each terminal processor keeps its alpha and beta values
  in global arrays instead of passing as parameters
*/
UPDATE(int depth, int score, int bound)
{   if (bound == -1)      /* lower bound */
        alpha[depth] = max(alpha[depth], score);
    else
        beta[depth] = min(beta[depth], score);
    if (depth < MAXDEPTH)
        UPDATE(depth+1, -score, -bound);
}
```

Figure 2: Interrupt driven update of alpha-beta values.

the **parfor** loop.

A mechanism can be provided for dynamically updating the alpha-beta window, used by slaves while they carry out a search, Figure 2. When a master processor receives a new alpha value from one of its slaves, UPDATE is invoked (via an interrupt mechanism) in each of the slaves currently searching.

A naive application of the tree-splitting algorithm might use one master and K slaves, with the master generating all the positions at some fixed common depth C in the tree and assigning them successively to the slaves. Though having the appeal of simplicity, there are a number of drawbacks to such a scheme, based mainly on the tradeoffs involved over the value of the common depth.

For example, if C = 1, i.e. the slaves are assigned the immediate successors of the root node,

a. The degree of concurrency is immediately limited by the branching factor of the game tree.

b. There can be difficulty with system idle time, e.g. 7 slave processors will, on the average, perform only slightly better than 4 when searching a tree with branching factor 8.

c. There may be poor bound sharing between searches, thus increasing search overhead.

This last point deserves further discussion. Consider a uniform game tree T of width 8 and depth 4 which is perfectly ordered. The search overhead coefficient for various processor tree widths, assuming one level of slaves with C = 1, may be computed as follows:

S(A(1),T) = 127/127 = 1.0

S(A(3),T) = 190/127 = 1.496

S(A(5),T) = 308/127 = 2.425

S(A(9),T) = 568/127 = 4.472

In other words, a system with 9 processors, that uses this configuration, each slave examines 568/8 = 71 nodes, producing a speedup factor of only 127/71 = 1.79 on perfectly ordered trees.

Increasing C, the common depth, postpones the limited concurrency problem, and reduces the difficulty with idle time, since the individual searches are shorter. In addition, search overhead is reduced considerably. The problems with larger C values are the greater communication overheads, and the increased complexity and volume of work required by the master processor. In fact, it is clear that the volume of work and the amount of storage needed for a master processor is exponential with C. Thus practical considerations keep the size of C small.

In order to reduce the bottleneck at the master processor it is possible to insert some intermediate level masters between the root processor and the slaves searching at depth C. In this manner each master need only handle a fixed number of slaves, regardless of the total number of processors available. In such a configuration, define P to be the number of game tree plies between a master and its slaves. For a processor tree of depth D, then P * D = C, where C is again the depth at which the terminal slaves begin their search.

There are a number of variations on this technique designed to improve searching performance. If P is small (e.g. 1 or 2), the master processors could be idle much of the time waiting for messages. In this case, the masters may be able to join their slaves in subtree evaluation, although this is probably only practical for the deepest masters [FISH80]. A second optimization could group higher level masters as separate processes on a single processor [FISH80]. The fact that the top level masters are usually the least busy motivates this suggestion, though the value of P again plays a limiting role. A third variation on the tree-splitting algorithm involves the more complex processor assignment strategy of our next proposal.

D.2 Principal Variation Techniques

One parallel search algorithm [AKL82] was based on the observation that alpha-beta must search certain subtrees regardless of the ordering properties of the game tree. Thus these subtrees can advantageously be searched concurrently. However, the described algorithm used a dynamic tree decomposition, the disadvantages of which have already been discussed. *Mandatory work first* [FISH81] is an adaptation of this method to the processor tree architecture.

Our proposal, *Pv-splitting*, relies on the assumption that the current principal variation is correct. Pv-splitting, for "principal variation tree-splitting", also uses a processor tree architecture. The algorithm is motivated by a close examination of the behavior of the sequential alpha-beta algorithm on perfectly ordered trees.

D.2.a Basis for pvsplit

The Dewey decimal system will be used to assign coordinate numbers to nodes. Every position at depth k is represented by a sequence of k positive integers. The root is represented by a null sequence, while the W successors of a node a1.a2 ... ak are a1.a2 ... ak.1 through a1.a2 ... ak.w. It is now possible to precisely define perfect ordering. A tree is perfectly ordered if, for each position p in the tree,

```
    negamax(p) = evaluate(p) if p is a terminal node
               = -negamax(p.1)  otherwise.
```

where p represents the sequence of integers that specify to path to the position. The function evaluate(p) returns a numeric value which measures the relative quality of p. A number of other support procedures exist, but most important is the function generate(p) which produces all the immediate successors of p and returns the number of successors, W.

The nodes visited by alpha-beta in a perfectly ordered tree are called *critical* nodes [KNUT75]. A node a1.a2 ... ak is critical if aj is 1 either for all even values of j, or for all odd values of j. Critical nodes can be divided into three types. In type 1 nodes, all the aj's are 1. A node is of type 2 if ai is its first entry > 1 and k-i is even. When k-i is odd, the nodes are of type 3. Intuitively, type 1 nodes are those on the principal variation, while type 2 nodes are alternatives to the principal variation. Successors of type 2 nodes are of type 3, while type 3 successors are again of type 2.

The following observations can be made about the critical positions in a perfectly ordered game tree:

a. At type 1 and 2 nodes, the best move must be considered first, though this is not necessary for type 3 nodes.

b. At type 1 and 3 nodes, all successors are examined.

c. At type 2 nodes, only the first successor is examined.

Clearly the power of alpha-beta pruning derives from the fact that type 2 nodes can be cut off with less than a full-width search. These cut offs are made possible by the score returned from searching type 1 nodes. If a type 2 node, for example the second successor of the root, is searched without the benefit of the score from the corresponding type 1 node (in this case, the first successor of the root), the node will be explored full-width.

A parallel algorithm must find a means to reduce this search overhead. The structure of palphabeta suggests the algorithm shown in Figure 3, which is run on the root processor of a processor tree. This algorithm, **pvsplit**, concentrates its efforts on fully evaluating type 1 nodes, and then using the resultant score to search type 2 nodes efficiently. There must be some maximum depth that this procedure can be applied, due to the restriction that the processor tree should not be deeper than the game tree. At the maximum depth on the principal variation, a standard version of the tree-splitting algorithm may be used to obtain the initial evaluation, after which pv-splitting can be used.

D.2.b Extensions to pvsplit

An enhancement of pv-splitting arises from the observation that, in optimally ordered trees, type 3 nodes must be explored full-width, while type 2 nodes need only examine their first successor. This suggests that concurrency is more profitably applied at type 3 nodes, since no cut offs can occur there. Type 2 nodes, on the other hand, should be examined with minimal concurrency, since a cut off can occur after scoring only 1 successor. This technique is implemented in a processor tree by assigning slaves type 2 positions only, i.e. instead of assigning the immediate successors of a (type 2) node to the slaves, assign

```
pvsplit(position p, int α, int β, int depth)
{   int w, i, t[MAXWIDTH];
    processor j;
    if (depth == MAXDEPTH)
        return(treesplit(p, α, β));
                                      /*                          */
    w = generate(p);                  /*  determine successors   */
                                      /*        p.1 ... p.w       */
    α = -pvsplit(p.1, -β, -α, depth+1);
    if (α ≥ β)
        return(α);
    parfor i = 2 to w do {
        when (a slave j is idle) {
            t[i] = -j.treesplit(p.i, -β, -α);
            critical {
                if (t[i] > α) α = t[i];
            }
            if (α ≥ β) {
                terminate();
                return(α);
            }
        }
    }
    return(α);
}
```

Figure 3: All processors on candidate principal variation.

the successors' successors. Besides reducing search overhead, this optimization allows processor tree widths that are prohibitively expensive (in terms of search overhead) in standard tree splitting.

So far it has been assumed that the tree being searched is perfectly ordered. Obviously if it is known in advance that the tree is perfectly ordered, there is no point in carrying out a search at all. Therefore as a practical matter, pv-splitting should be examined after relaxing the optimal ordering assumption, though there is still good reason to believe that the tree is strongly ordered, i.e. there is a high probability that the best move from a given position is placed high in the movelist. Application of standard sequential ordering techniques to a parallel environment is available in a recent report [MARS81].

Palphabeta also suggests a modification to the parallel searching algorithm, namely the use of the minimal window bound-testing procedure. If the type 1 nodes are indeed the correct principal variation, the remainder of the search can benefit from the minimal window. Whenever a minimal window search fails high, maximum effort should be made to fully evaluate this subtree, since it contains the new principal variation. In effect, the subtree should be treated as if its root was a type 1 node, since its value is crucial to the efficiency of the remaining searches.

D.3 SSS* Adaptations

Although the effectiveness of sequential SSS* [STOC79] (in terms of nodes evaluated) cannot be disputed, a parallel version is fraught with implementation difficulties. These problems centre around the maintenance of the requisite data structure. If the amount of storage required is not a limiting factor, the synchronization overhead involved in preserving the integrity of the data tends to reduce concurrency gains.

A parallel adaptation of SSS* can be envisioned which works in a dynamic decomposition framework similar to that described by Akl, i.e. processors choosing tasks from a priority ordered set. In this case, the tasks are individual entries in a list. A free processor can remove the top entry of the list and carry out the appropriate action, which will result in further additions or deletions to the list. If efficient means of handling dynamic tree decompositions could be found, this method could be very attractive in terms of overall search speed, particularly for trees that are randomly or poorly ordered.

Staged SSS* [CAMP81] is adaptable to the processor tree architecture, and hence can employ a static game tree decomposition. Each master maintains a list of positions to be searched, and assigns appropriate subtree searches to its slaves. Terminal slaves can employ either SSS*, staged SSS*, or any of a number of other methods to evaluate their assigned nodes. Since the various lists are local to a given processor, no data sharing overhead is required here. In addition, the total storage requirement is not exponential with game tree depth, making deeper searches more practical. Staged SSS*, judging from sequential performance analyses, will be most useful in random or weakly ordered trees.

E. Performance Comparison of Parallel Algorithms

Empirical performance tests of some parallel tree searching algorithms have been done over a range of multi-processor configurations and on trees of various types. All concurrency was simulated. Only algorithms that employ a static tree decomposition were considered, as the practicality of dynamic methods is not clear.

In measuring the performance of different algorithms on a given multi-processor system, the main concern is total elapsed time. If algorithm A is consistently faster than algorithm B, it is fair to say that A is better than B, regardless of the relative number of terminal node evaluations or move generations. For the purposes of this simulation study, elapsed time is broken in three components:

1. EVALTIME - time to evaluate a terminal node.
2. MVGENTIME - time to generate the moves at an interior node
3. MESSAGETIME - time to pass a message between a master and

slave, or vice versa.

For simplicity, our comparison assumes that EVALTIME is set to 1 time unit, while MVGENTIME and MESSAGETIME are negligible. Also, although terminal node evaluation time is consistent over all algorithms, move generation mechanisms and message passing time can be algorithm dependent. It is assumed these other overheads will have roughly the same relative magnitudes as the elapsed EVALTIME's, and thus can safely be ignored.

The algorithms compared here are:

1. tree-splitting (TS),
2. pv-splitting (PV), and
3. staged SSS* (SSS).

Tables 1,2 and 3 contain the simulation results. Each processor-tree/algorithm combination searched 100 independent trees of width 24 and depth 4, and average elapsed times were recorded. The trees themselves were built in the manner described in the Appendix. Note that, on 4-ply trees, PV cannot employ a depth 3 processor tree, and SSS cannot use either a depth 2 or 3 system.

Table 1 indicates the superiority of staged SSS* on random trees. Both TS and PV consistently required about 50% more search time on identical multi-processor systems.

Table 2 contains data from trees with geometric distribution of the best move with parameter 0.8. PV was designed for such

(L,K)	TS	PV	SSS
(1,2)	6443	6101	4305
(1,4)	4384	4028	2854
(1,8)	3273	2958	2032
(1,12)	3021	2644	1696
(2,2)	3689	3967	
(2,4)	1506	1973	
(2,8)	758	1273	
(3,2)	2317		
(3,4)	654		

L = Processor tree length
K = Processor tree branching factor

D = 4
W = 24

Table 1: Search time for randomly ordered trees.

(L,K)	TS	PV	SSS
(1,2)	1264	944	1140
(1,4)	1187	604	690
(1,8)	1232	440	463
(1,12)	1270	412	383
(2,2)	766	686	
(2,4)	391	391	
(2,8)	236	287	
(3,2)	541		
(3,4)	244		

L = Processor tree length
K = Processor tree branching factor

D = 4
W = 24

Table 2: Search time for strongly ordered trees.

strongly ordered trees, and outperforms TS considerably, especially on the wider processor tree configurations. The depth 2 processor trees diminish the advantage somewhat. This is because the simulations only invoked pvsplit at the root processor of the tree. Depth 1 processors ran regular tree-splitting. SSS shows rather well in this data set, but it should be noted that the algorithm is slightly more time-consuming than TS or PV, and thus SSS must do somewhat better than these others in order to be practical.

A very interesting result occurs in the TS data. It appears that a system with 8 terminal slaves does worse than a system with 4 terminal slaves. This strange result is due to the lack of a dynamic updating mechanism in the simulations. When a processor searching a sub-optimal move returns first with a poor score, it will be reassigned but with the returned score as the new alpha value. This poor alpha value allows fewer cut offs, and increases search times. Even if the best possible alpha value is returned one time unit later, it is not available for the other search. These results indicate that a dynamic window updating mechanism is of great importance.

Table 3 contains data from a perfectly ordered tree. Such trees are ideally suited for PV, and the data bears this out. The values are of little practical interest, but they do illustrate dramatically the larger processor tree widths feasible with PV.

(L,K)	TS	PV	SSS
(1,2)	863	599	876
(1,4)	719	311	497
(1,8)	647	167	284
(1,12)	623	119	213
(2,2)	575	443	
(2,4)	287	190	
(2,8)	143	83	
(3,2)	431		
(3,4)	179		

L = Processor tree length D = 4
K = Processor tree branching factor W = 24

Table 3: Search time for optimally ordered trees.

If the data contained in the tables is plotted as a log-log graph of time versus the number of terminal node processors, K, then effective speedups can be computed from a least squares fit. Assuming K terminal processors, SSS achieves speedups of K**0.52 (random ordering), K**0.61 (strong ordering), and K**0.8 (optimal ordering). However, varying the methods used for choosing the next subtree to search can affect this performance. The particular algorithm employed in the simulations was favorable for ordered trees. Other mechanisms could be expected to favor random trees.

TS is best examined on processor trees of small fixed width so as to reduce the effects of the absence of a dynamic updating mechanism. The following values assume a processor tree width of 2. Randomly ordered trees gave a speedup of K**0.74. Strong ordering produced a speedup of K**0.61, while optimal ordering gives a speedup of K**0.5. These results are consistent with theoretical studies [FISH81], which predict a K**0.5 speedup for optimal ordering, with increasingly effective use of parallelism as trees become less ordered. With dynamic updating, the non-optimal ordering speedups can be expected to improve.

Since pv-splitting was not employed in full generality for processor trees of depth 2, the depth 1 values will be used. However the non-optimal orderings have their values seriously affected by the lack of dynamic updating. Random trees allowed a speedup of K**0.52. Strongly ordered trees produced a K**0.55 speedup, and optimal ordering gave a speedup of K**0.92. Only in the last case is the figure actually meaningful.

F. Summary of Results

This paper has described and compared algorithms for parallel search of game trees. Three approaches to parallel tree search have been discussed, and tree decomposition is identified as the focus of this paper. Tree-splitting [FISH80] is one implementation of alpha-beta in parallel. Pv-splitting is presented as a parallel search algorithm which is effective given certain ordering assumptions about the trees searched. A parallel adaptation of staged SSS* is also discussed as a potentially effective search method for random trees. The performance of these parallel algorithms has been compared, given widely ranging multi-processor configurations and tree characteristics. The apparent strengths and weaknesses of the various algorithms were discussed.

Static decompositions have been emphasized, mainly because of their advantages in actual implementation on multi-processor systems. Dynamic decomposition has great promise if a means can be found to reduce the associated synchronization overhead.

Our conclusion is that the parallel algorithms should be tested more thoroughly on a true multi-processor system. In this way a number of questions could be answered concerning the time spent in the different phases of the tree search and interprocessor communication. The effect of search enhancement techniques on a parallel system is also a matter of interest.

REFERENCES

[AKL82] S. Akl, D. Barnard and R. Doran, "Design, analysis and implementation of a parallel tree search algorithm", IEEE Trans. on Pattern Analysis and Machine Intelligence, Vol PAMI-4, No. 2 (1982), 192-203.

[BAUD78] G. Baudet, "The design and analysis of algorithms for asynchronous multiprocessors", Ph.D. thesis, Computer Science Dept., Carnegie-Mellon Univ. Pittsburgh, (1978).

[CAMP81] M. Campbell, "Algorithms for the parallel search of game trees", TR81-8, Computing Sci. Dept., Univ. of Alberta, Edmonton (1981).

[CORA76] L. Coraor and J. Robinson, "Using parallel microprocessors in tree decision problems", Proceedings of the International Symposium on Mini and Micro Computers, IEEE, 51-55, (1976).

[FISH80] J. Fishburn and R. Finkel, "Parallel alpha-beta search on Arachne", TR 394, Computer Science Dept., Univ. Wisconsin, Madison, (1980).

[FISH81] J. Fishburn, "Analysis of speedup in distributed algorithms", Ph.D. thesis, TR 431, Computer Science Dept., Univ. Wisconsin, Madison, (1981).

[FULL73] S. Fuller, J. Gaschnig and J. Gillogly, "Analysis of the alpha-beta pruning algorithm", Computer Science Dept., Carnegie-Mellon University, Pittsburgh, (1973).

[KNUT75] D. Knuth and R. Moore, "An analysis of alpha-beta pruning", *Artificial Intelligence 6*, 293-326, (1975).

[MARS81] T.A. Marsland and M. Campbell "Parallel search of strongly ordered game trees", TR 81-9, Computing Science Dept., Univ. of Alberta, Edmonton, (1981). (submitted to ACM Computing Surveys).

[MARS81a] T.A. Marsland and M. Campbell, "A survey of enhancements to the alpha-beta algorithm", ACM81 Conference Proceedings, Los Angeles, 109-114, (1981).

[PEAR80] J. Pearl, "Asymptotic properties of minimax trees and game-searching procedures", *Artificial Intelligence 14*, 113-138, (1980).

[STOC79] G. Stockman, "A minimax algorithm better than alpha-beta?", *Artificial Intelligence 12*, 179-196, (1979).

[THOM82] K. Thompson, "Computer chess strength", *Advances in Computer Chess 3*, M.R.B. Clarke (ed.), Pergamon, (1982).

Appendix - Tree Generation Methods

For purposes of comparison, it is desirable to have each algorithm search the same trees. For this reason it was decided to generate the trees separately and store them in files so that they can be searched any number of times by any number of different algorithms. A tree file consists of W**D 8-bit bytes, with each byte representing the score of one terminal position. Only uniform trees of width W and depth D were considered. The position of a byte in the file indicates the location of a node in the game tree being represented. The first byte in the file is the leftmost terminal node in the tree, and the last byte the rightmost.

The terminal node values were chosen by a mechanism which depends upon the distribution of the position of the best move at a node. The tree is generated recursively, in the negamax framework. At any given interior node, all the successors are assigned values less than some score, and then the best move is chosen and assigned the value of score. For practical purposes it was found that trees with more than about 400,000 terminal nodes required excessive CPU resources, hence the size restrictions on the trees searched