Parallel Search of Strongly Ordered Game Trees

T. A. Marsland and M. Campbell

Technical Report TR81-9

September 1981

DEPARTMENT OF COMPUTING SCIENCE

The University of Alberta

Edmonton, Alberta, Canada

PARALLEL SEARCH OF STRONGLY ORDERED GAME TREES

T. A. Marsland and M. Campbell

Department of Computing Science
University of Alberta
Edmonton, Alberta, Canada
T6G 2H1

Financial support for this study was provided by the Natural Sciences and Engineering Research Council of Canada.

PARALLEL SEARCH OF STRONGLY ORDERED GAME TREES

T.A. Marsland and M. Campbell
Department of Computing Science
University of Alberta
EDMONTON, Canada
T6G 2H1

DRAFT for submission to ACM Computing Surveys
August 1981

The following quantities should be subscripted:- Ri, KEYj, Ra, Rb, Rw, KEYk, Rf, Rt, Hk, Mx, My,

ABSTRACT

The alpha-beta algorithm forms the basis of many programs that search game trees. Current sequential game-playing programs have therefore developed a number of methods designed to improved the effectiveness of alpha-beta. These enhancements are based on the observation that the alpha-beta algorithm is most beneficial when the best move in each position is to the left of the game tree. Trees that approach this so-called "best-first ordering" are both of practical importance and possess properties that can be exploited in both sequential and parallel environments.

This paper reviews the enhancements to alpha-beta, and examines their applicability to parallel tree searching. Various parallel search algorithms are then compared under the assumption of strongly ordered trees, and an algorithm is proposed which attempts to utilize these ordering properties.

ACKNOWLEDGEMENT

Financial support for this study was provided by the Natural Sciences and Engineering Research Council of Canada.

1. INTRODUCTION

The primary aim of this paper is to present and compare various methods of employing parallelism in the search of two-person, zero-sum game trees. Particular emphasis is placed on using the alpha-beta algorithm to search strongly ordered trees, such as those generated by current chess programs. Typical details about processor and communication considerations are commonly available [WEIT80][ENSL74], and more specific information can be found in a report[MARS80].

With few exceptions [NEWB77], much of the existing theoretical work on both sequential and parallel game tree searching has been restricted to random trees. However, in practice, truly random trees are quite uncommon. In addition, special techniques have been developed to improve the effectiveness of the principal searching method, the alpha-beta algorithm[SLAG69]. Thus, we will assess the applicability of these enhancements to parallel searching methods, contrast various ways of doing parallel alpha-beta searches and propose a parallel algorithm which attempts to take advantage of the characteristics of the trees produced. Strongly ordered trees are both more realistic, and possess properties that can be exploited in a parallel environment.

2. SEQUENTIAL SEARCH ALGORITHMS

A complete description of the alpha-beta algorithm can be found elsewhere [KNUT75]. Rather than duplicate that work we will simply clarify some relevant facts and terminology used in our paper. A typical procedure heading might be alphabeta(p, alpha, beta, depth), where p represents a position, (alpha,beta) the search window or range of values over which the search is to be made, and depth the intended length of the search path. The basic structure of the depth-limited alpha-beta algorithm can be seen in Figure 1.

```
alphabeta(p, alpha, beta, depth)
 position p;
 int alpha, beta, depth;
{
  int w, m, i, t;

  if (depth < 0) return(evaluate(p));
  w = generate(p);
        /* determine successor positions */
        /* p.1 ... p.w and return number */
        /* of moves as function value    */
  if (w == 0) /* no legal moves */
    return(evaluate(p));
  m = alpha;
  for i = 1 to w do
  {  t = -alphabeta(p.i,-beta,-m,depth-1);
     if (t > m) m = t;
     if (m >= beta)  /* cutoff */
       return(m);
  }
  return(m);
}
```

Figure 1: Depth limited alphabeta procedure

For purposes of analysis, it is convenient to study the performance of the minimax and alpha-beta algorithms on uniform trees of depth D and constant width W. It is also usual to measure the relative efficiency of tree-searching algorithms in terms of the number of terminal nodes scored. The minimax

algorithm will always examine M(W,D) = W**D terminal nodes, while under ideal conditions the alpha-beta algorithm scores only

$$B(W,D) = W^{\lceil D/2 \rceil} + W^{\lfloor D/2 \rfloor} - 1 \text{ nodes.}$$

Thus the potential efficiency of the alpha-beta algorithm is very good, examining close to the square root of the maximum number of nodes, while still generating the same solution path (principal variation) from the root node. However, optimal performance is achieved only when the first move considered at each node is the best one. Under more realistic assumptions, we can define the following quantities.

R(W,D) = average number of terminal nodes scored in a random uniform game tree

A(W,D) = average number of terminal nodes scored in a strongly ordered uniform game tree

For the purposes of this paper, we will define a tree to be strongly ordered if the search finds: (1) the first branch from each node best 70% of the time, and (2) the best move in the first 25% of the branches 90% of the time. Static ordering mechanisms, combined with enhancements to alpha-beta (to be discussed later) tend to produce trees with these properties [GILL77] [MARS73].

While the performance of alpha-beta on random trees has a solid theoretical basis [FULL73], at present only empirical evidence is available for strongly ordered trees. Nevertheless, on a statistical basis, it seems clear that we have the relation

$$B(W,D) < A(W,D) << R(W,D) << M(W,D) = W^{**}D$$

Relative values for these terms can be seen from our Monte Carlo

simulation results, presented in Table 1. The simulations were carried out on trees of depths up to 5 and width W, with scores in the range 0 - 127. To estimate R, the values were assigned randomly to the terminal nodes, while the calculation of A relied on branch-dependent scores. The bracketed numbers represent the standard deviation for 100 independent search trials. Table 1 illustrates the relative efficiency of the alpha-beta under different move ordering assumptions.

D=3	B	A	R	M
8	71	105 (21)	181 (36)	512
W 16	271	405 (64)	786 (114)	4096
24	599	857 (115)	1752 (250)	13824

D=4	B	A	R	M
8	127	281 (88)	690 (153)	4096
W 16	511	1286 (430)	4125 (875)	65536
24	1151	2946 (1013)	10425 (1891)	331776

Table 1: Expected search costs for trees of width W and depth D.

A number of modifications to the alpha-beta algorithm have been proposed. They are examined here mainly for compatibility with the other search enhancements discussed.

The concept of a minimal window, an alpha-beta window of (-m-1,-m) where m is the best score so far, was introduced and used to search the last subtree [FISH80]. Slight searching improvement was noted for no cost.

Palphabeta is an interesting modification of alpha-beta

which operates only on nodes along the principal variation[FISH80]. Once a candidate principal variation is obtained, the balance of the tree is searched with a minimal window. However, if the tree is poorly ordered, each subtree that is better than its elder siblings must be searched twice Hence there is some risk that palphabeta will examine more nodes than alpha-beta. Iterative deepening provides a principal variation with reasonable reliability, and makes this technique more feasible. The structure of palphabeta can be seen in Figure 2, which includes an alpha-beta refinement (falphabeta) to enable use of a narrower window whenever the minimal window search fails.

```
palphabeta(position p, int depth)
{
  int w, m, i, t;
  if (depth < 0) return(evaluate(p));
  w = generate(p);
  if (w == 0) return(evaluate(p));
  m = -palphabeta(p.1, depth-1);
  for i = 2 to w do
  {  t = -falphabeta(p.i,-m-1,-m,depth-1);
     if (t > m)
       m = -alphabeta(p.i,-INF,-t,depth-1);
  }
  return(m);
}
```

Figure 2: Principal variation alphabeta.

It could be pointed out that it is not necessary to carry palphabeta all the way to the terminal nodes. In fact, since only the first few moves of a principal continuation are usually reliable, carrying palphabeta to, say, N-2 ply on an N ply iteration may be sufficient.

SCOUT [PEAR] is a further generalization of palphabeta, in which the final call to alphabeta is replaced by

```
m = -palphabeta(p.i, depth-1);
```

In its original form, SCOUT does not use the minimal window idea, but rather an equivalent test procedure. Our initial simulation results indicate that palphabeta out-performs SCOUT on strongly ordered trees.

SSS* [STOC79] and staged SSS* [CAMP81] are effective in the search of random or poorly ordered trees. However these algorithms are not significantly better than the alpha-beta on strongly ordered trees, and require more time and space. This paper will not consider further those methods that are not suitable for search of strongly ordered trees.

3. ENHANCEMENTS TO ALPHA-BETA SEARCHING

Many of the following techniques have been developed in efficiency-conscious full-width chess programs. The basic methods, however, are applicable to most programs that search game trees [MARS81].

3.1. Aspiration search: The interval enclosed by (alpha, beta) is referred to as the window. For the alpha-beta algorithm to be effective, the minimax score of the root position must lie within the initial window. Generally speaking, however, the narrower the initial window, the better the algorithm's performance. In many problem domains such as chess, there are reliable methods to estimate the score that will be returned by the search. Thus, instead of using an initial window of (-INF, +INF) (where INF is a number larger than any that evaluate() will return), one can use (V-e,V+e), where V is the estimated

score, and e the expected error. There are three possible outcomes of this so-called aspiration search, depending on S, the actual (minimax) score of a position p.

1. if S <= V-e, alphabeta(p,V-e,V+e,D) <= V-e
2. if S >= V+e, alphabeta(p,V-e,V+e,D) >= V+e
3. if V-e < S < V+e, alphabeta(p,V-e,V+e,D) = S

Cases 1 and 2 are referred to as failing low and failing high respectively [FISH80]. Only in case 3 is the true score of the position p found, using a smaller search space -- bounded by B(W,D) and A(W,D).

In the failed low case, it is necessary for the search to show that each alternative from the root is less than V-e. Assuming perfect ordering,

$W ** \lceil D/2 \rceil$ nodes must be examined.

In the failed high case, it is sufficient for the search to show one alternative greater than V+e. Again, under perfect ordering conditions, only

$W ** \lfloor D/2 \rfloor$ nodes need be examined.

Either way the search must be repeated, for example alphabeta(p,V+e,+INF,D) for the failed high case. Empirical evidence has shown aspiration searches to be very effective; in TECH^9, search time reductions averaging 23% were noted [GILL78]. This average was also obtained by Baudet by adapting his results for parallel tree search to the sequential case [BAUD78].

Falphabeta, for 'fail-soft alphabeta' [FISH80], is useful when aspiration searching is employed. Though always examining the same nodes as alpha-beta, falphabeta can give a tighter

bound on the true score of the tree when the search fails high or low. Although falphabeta requires a slight constant overhead, any system which uses aspiration searches should find the technique a practical one.

3.2. Transposition Table: In carrying out a search of a chess game tree, it is not uncommon for positions to recur in numerous places throughout the tree. Rather than rebuild the subtrees associated with the transposed positions, it may be possible to simply retrieve the results stored in a table by a previous search. A transposition table is a large hash table, with each entry representing a position. For game modelling, nearly perfect hashing functions can be produced[ZOBR70]. Although there are many table management problems which must be solved, the technique has very low overhead for the large potential gains.

A typical hash index generation method is the one proposed by Zobrist[ZOBR70], who observed that a chess position constitutes placement of up to 12 different piece types {K,Q,R,B,N,P,-K ... -P} onto a 64-square board. Thus a set of 12x64 integers (plus a few for en passant and castling privileges), {Ri}, may be used to represent all the possible piece/square combinations. An index of the position may be given by

KEYj = Ra xor Rb xor ... xor Rw

where the Ra etc. are integers associated with the piece placements for the particular position under consideration. Movement of a piece from a square associated with Rf to the piece/square associated with Rt yields a new index

KEYk = (KEYj xor Rf) xor Rt

More importantly, if the Ri are uniformly distributed in the interval [0,2**N], then so are the KEYk. Typically N is 32 and so 2**N is too large for direct use of KEYk as an index into a transposition table, rather

Hk = KEYk mod T is used, where T << 2**N.

Clearly, all the possible chess positions cannot be represented uniquely by Hk, but even so this is quite sufficient as a basis for a successful entry point. A minimal table entry could have the following format:

lock	move	score	flag	len	prio

lock — to ensure the table position is identical to the tree position,

move — best move in the position, determined from previous search,

score — of subtree computed previously,

flag — indicating whether score is upper bound, lower bound or true score,

len — length of subtree that score is based on,

prio — used in table management, to select entries for deletion.

When a position reached during a search is located in the table (i.e. the lock matches), there are a number of possible actions:

(1) If len is less than remaining length to be searched, score is ignored and the search is carried out as usual. However move is tried first in the position. The main advantage of

this is that it saves a move generation, and also, since move has previously (in a shallower search) proven best, it is likely to be so again. Furthermore, move will direct the search toward positions that have been seen before, hence increasing the effectiveness of the table.

(2) if len >= remaining length to be searched

(a) if score was the true score, this value is returned without further searching

(b) otherwise, score is used to adjust the current alpha-beta bounds. This could either cause an immediate cutoff, or allow the search to continue with a reduced window. If a search must be done, move will be tried first.

There are also further enhancements possible. For example, DUCHESS[6] maintains both upper and lower bounds on the position score, with separate lengths for each [TRUS81].

Transposition tables are most effective in chess endgames, where there are fewer pieces and more reversible moves. Gains of a factor of 5 or more are typical, and in certain types of king and pawn endings, experiments with BLITZ[4] and BELLE[3] have conducted searches of more than 30 ply, representing speedups of well over a hundred-fold. Even in complex middlegames, however, significant performance improvement is observed. Successful use of the transposition table helps make trees look strongly ordered, and makes possible search times less than for optimal alpha-beta, since large subtrees need not be re-evaluated. For greatest effect the transposition table must be integrated into alpha-beta carefully, as illustrated in the Appendix.

Another potential application for transposition tables is to implement the method of analogies[ADEL79]. In simple terms analogies are used to determine if the sphere of influence surrounding an exchange is unaltered, even though the actual position containing the exchange is different. However, the calculations involved to determine the influences are quite complex, perhaps as expensive as the actual exchange tree evaluation. On the other hand it is possible that the number of echanges offered during a move is far less than the number of terminal positions in the move tree. Clearly the method of analogies, and its implementaion through transposition tables, is potentially fruitful area of further research.

3.3. Killer Heuristics: The killer heuristic is based on the premise that if move My 'refutes' move Mx, it is more likely that My (the 'killer') will be effective in other positions[GILL72]. Any move which causes a cutoff at level N is said to have refuted the move at level N-1[CICH73]. There are many ways of using this information. For example, the program CHESS[5] maintains a short list of killers at each level in the tree, and attempts to apply them early in the search in the hope of producing a quick cutoff. A further advantage of the killer heuristic is that it tends to increase the usefulness of the transposition table[TRUS81]. By continually suggesting the same moves, there is a greater possibility of reaching a position already in the table.

In its full generality, the killer heuristic can be used to dynamically reorder moves as the search progresses. For example, if a move My at level N refutes a move at level N-1, and My

remains to be searched at level N-2, it is worth considering next. An additional method, used by AWIT[1], seeks out defensive moves at ply N-1 which counteract killers from level N. The idea behind the generalized killer heuristic mechanism is to allow information gathered deep in the tree to be redistributed to shallower levels. This is not usually done by the full-width programs, however, since it is not yet clear that the potential gains exceed the overhead.

The actual search reductions produced by the killer heuristic are not clear. In TECH[9], no improvement was noted, but CHESS[5], DUCHESS[6], OSTRICH[7] and BLITZ[4] continue to employ the mechanism.

3.4. Iterative Deepening: Iterative deepening (also called staged search) refers to the procedure of using an N-1 ply search to prepare for an N ply search. It has been hypothesized [MARS81] that the cost of such a search is given by a recurrence relation of the form

$$A(W,D) = A(W,D-1) + B(W,D) + (W-1)*F(W-1,D-2)$$

where F(W,D) is the expected cost of an alpha-beta search of strongly ordered trees with W > 20 and D > 4, given the first N-1 moves of the principal variation. Iterative deepening has certain immediately obvious advantages.

(1) It can be used as a method for controlling the time spent in a search. In the simplest case, new iterations can be tried until a preset time threshold is passed.

(2) An N-1 ply search can provide a principal continuation which, with high probability, contains a prefix of the N ply principal continuation. This allows the alpha-beta

search to proceed more quickly.

(3) The score returned from a N-1 ply search can be used as the center of an alpha-beta window for the N ply search. It is probable that this window will contain the N ply score, thus increasing search speed.

These last two points, though significant, are not really complete justifications for the use of iterative deepening from a tree searching point of view. In fact, in experiments with checkers game trees [FISH80], it was found that iterative deepening increased the number of nodes searched by 20% (apparently only using point (2), however). In addition, studies with TECH[9] using a generalized version of (2), but not (3), noted a 5% increase in search times when iterative deepening was applied [GILL78]. It appears that a strong initial move ordering, together with a good alpha-beta window estimate, can approximately match iterative deepening. The real searching advantage of iterative deepening is:

(4) The transposition table and killer lists are filled with useful values and moves.

The importance of this fact is illustrated by the performance of the BELLE[3] chess machine. Typical chess middlegame positions have branching factors of 35-40. It has been found that in such positions, it normally costs BELLE a factor of 5 - 6 to go one further ply, i.e. less than the expected cost of optimal alpha-beta.

A variation of this basic scheme, one which is especially appropriate if transposition tables are not used, is employed by L'EXCENTRIQUE[8]. A 2 or 4-ply minimax search is first performed

to obtain W move-pairs (moves and their best refutation). These are then sorted and a 6, 8, 10 etc -ply iterative deepening cycle initiated. The rationale behind two ply increments is to preserve a consistent theme between iterations, so that the principal variation will not flip-flop between attacking and defensive lines. To our knowledge, no analytical comparison between this and conventional iterative deepening has been done.

4. PARALLEL TREE-SEARCH METHODS

The best way to make K processors perform an alpha-beta search on a tree is not known. Generally, a K-fold increase in computing power is not possible because some inter-communication is necessary, causing losses as processors wait for these messages. More importantly, if independent subtrees are searched concurrently it is likely that redundant nodes will be examined, because the best bounds are not always available. Despite these problems the effective computing power can be substantially higher, depending on the processor configuration employed.

4.1. Parallel Evaluation : Current game-playing programs that carry out full-width searches must come to terms with the tradeoff between depth of search and complexity of terminal node evaluation. Most of the stronger chess programs employ a relatively simplistic scoring function in order to search more deeply. Nevertheless, a considerable portion of the search time is spent in evaluation, on the order of 40% in BLITZ[4] and DUCHESS[6].

An obvious application of concurrency to game tree search

appears to be within the evaluation function itself. A number of processors could be used to simultaneously evaluate different terms in the scoring function, which could be combined to form an overall evaluation of the position. This method is used to a limited extent in the chess machine BEBE[2].

Advantages of this technique are numerous:

(1) Evaluation time could be reduced, allowing deeper searches.

(2) Many small, cheap processors could be used to evaluate individual features in a position.

(3) Since there is no obvious limit to the amount of concurrency possible, the evaluation function could be considerably more complex; large amounts of game-specific knowledge could be utilized, and extended arbitrarily.

Ultimately one could envision an evaluation 'machine', which would consist of a processor hierarchy. For example, bottom level processors would score primitive board features, passing the values to higher level processors, which would combine the features in various (not necessarily linear) ways to form more complex features. The machine could also have the ability to return from a terminal search with an indication that the position is too unstable to score reliably. Admittedly a large proportion of terminal nodes in a full-width search need nothing more than a material evaluation (about 50% in BELLE[3]), but the above scheme could improve positional understanding in the remainder.

4.2. Parallel Aspiration Search: Even though alpha-beta search itself is relatively efficient, the aspiration refinement provides improvement whenever it is successful. One parallel implementation would be to divide the alpha-beta window into non-overlapping sub-intervals and apply a processor to each range [BAUD78]. For example

Processor 1 (-INF, V-e)
Processor 2 (V-e, V+e)
Processor 3 (V+e, +INF)

Hopefully processor 2 will finish first, but in any case one of them will succeed and do so in less time than a uniprocessor searching over (-INF,+INF). Those processors which fail early can cut off or improve the bounds for others. Baudet [BAUD78] has explored optimal ways of window decomposition, including methods which don't initially cover (-INF,+INF).

There are two important results from this aspiration search work:

(a). Maximum expected speed-up is typically 5 or 6, regardless of the number processors available, because the cost of a partial search, i.e., a restricted window search, is bounded below by B(W,D).

(b). When the degree of parallelism is small (K = 2 or 3) the speed-up obtained may be greater than K. These results are based on certain assumptions; in particular, it is assumed that the distribution of the backed-up score is known. The applicability to strongly ordered trees is not clear. In any case, the sequential version of the aspiration search is very powerful for chess game trees, and largely supplants the

parallel methods.

4.3 Tree Decomposition: Most discussions of parallel game tree search have concentrated on concurrent examination of independent subtrees. Even Baudet concludes that parallel aspiration searching must be combined with tree decomposition if large performance improvements are desired [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 arises from the fact that decisions to cutoff 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 as it is 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.

Communication overhead can arise in different ways, depending on the system configuration. Information can be communicated 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 overhead, which increases with the degree of concurrency. Of course the volume of information to be shared is dependent upon the particular search algorithm used, but it seems clear that,

in general, 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 cutoff decisions can be made by the search algorithm, thus reducing search overhead.

4.4 Enhancements in Parallel Search: The searching enhancements of Section 3 are examined for applicability to parallel search. Suggestions are made for those techniques with non-trivial implementation difficulties.

Aspiration searching in parallel has been shown to be relatively ineffective in trees where a good initial window can be chosen. However the sequential version of aspiration searching, when used in conjunction with iterative deepening, is equally applicable to parallel systems, perhaps more so, since a common problem of such systems is inappropriately wide windows.

Transposition tables continue to be effective, provided all the processors access the same table. The method is especially attractive since table usage is a naturally autonomous function, and can be partitioned for parallel execution. Furthermore, something useful can be done while waiting for access to the transposition table, namely proceed with the evaluation of the next subtree. If the position sought is not in the table, then no time is lost, otherwise the first result from either the tree recomputation or the table access is used.

Access delays to the transposition table can be reduced by dividing the table into ranges and providing a different processor for each partition. In any case, the table naturally splits itself into two portions, those positions for white to

move and those for black, Figure 4. This scheme is quite independent of the relationships between the game processors C1, C2 and C3, which share and provide updates for the transposition table memory. A potential bottleneck exists at processor P0, but this should not be severe since P0 has no significant computational functions, beyond those necessary for the routing operations.

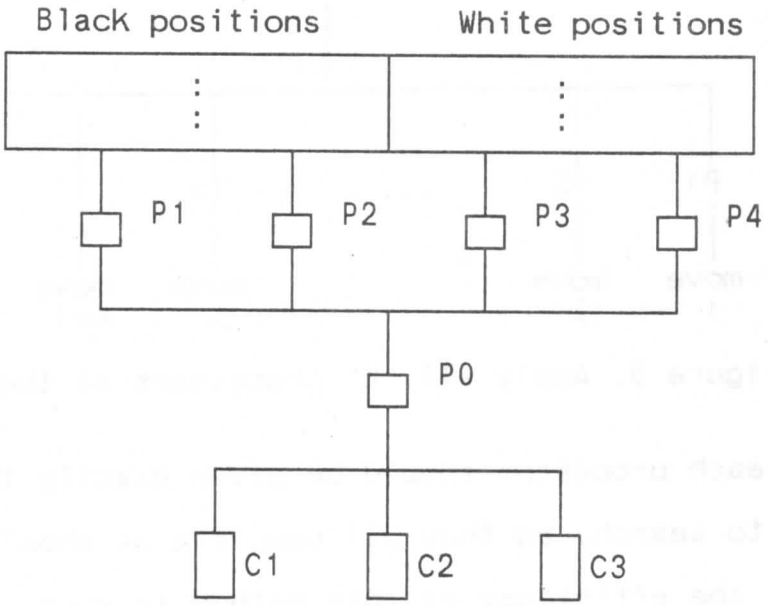

Figure 4. Transposition table access and management.

The killer heuristic presents similar problems to the transposition table. The killer list is so small, however, that the management problems are much reduced.

The alpha-beta modifications discussed are relatively unaffected by parallelism. Falphabeta proceeds identically, with similar advantages to those found in sequential systems. Palphabeta restricts the method of application of parallelism to the tree to ensure the correct minimal windows can be found. These restrictions are not necessarily deleterious however.

5. PARALLEL SEARCH ALGORITHMS

5.1 Naive Method: With a static decomposition, the game tree is split into groups of subtrees, and each subtree is assigned to a different processor, Figure 5. As processors complete they are allocated to the next group of subtrees, until the full tree is evaluated.

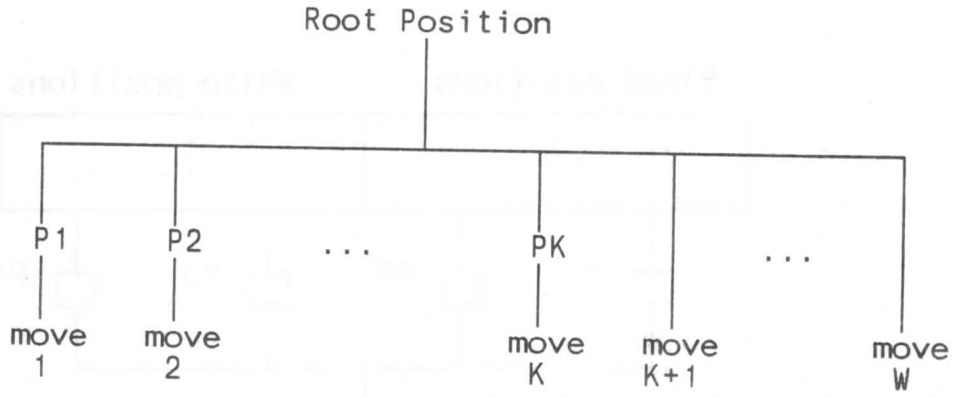

Figure 5. Apply all 'K' processors at the first level.

Ideally each processor should be given exactly the same sized subtree to search, so they all complete at about the same time. However, the efficiency of this method is very sensitive to the ratio W/K.

More importantly, for a typical game-tree with W = 40, alpha-beta pruning itself reduces the search space to one equivalent to a tree with W = 7 [GILL72]. Thus if K = 40 processors are applied at depth 1, the average speed-up over a uniprocessor employing alpha-beta would be only 7. Note that the most serious disadvantage with this scheme is that the processors share alpha-beta values in a very limited way.

5.2. Minimal Tree: The minimal tree that must be searched by the sequential version of the alpha-beta algorithm has a very definite structure. It has been proposed that these subtrees be searched independently and concurrently as the first stage of a parallel algorithm [AKL80]. The resultant alpha-beta window generated by the first phase is applied to the second phase, where an independent search of the remaining subtrees takes place. To simplify the description the following terminology is used:-

> The first son of a node is called the left son, and is contained in the left subtree. All other sons of the node are right sons and are in right subtrees [AKL80].

Phase 1: Search the left subtree of the root node, and the left subtrees only of right sons of the root node. At the end of this phase the left sons will have been fully evaluated, while the right sons will have temporary values (i.e. the values of their left sons). Note this statement of the phase 1 search is actually an oversimplification, since the method is applied recursively to each left subtree. Figure 6 shows the first phase of a search on a 2-ply tree, by marking the branches explored with solid lines and terminal scores.

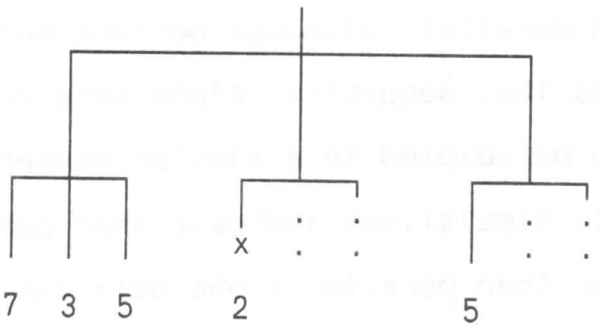

Figure 6. First phase of search, necessary tree.

Phase 2: Those subtrees whose temporary values are insufficient to cause a cutoff are now searched one branch at a time until all right sons have been cut-off or fully explored.

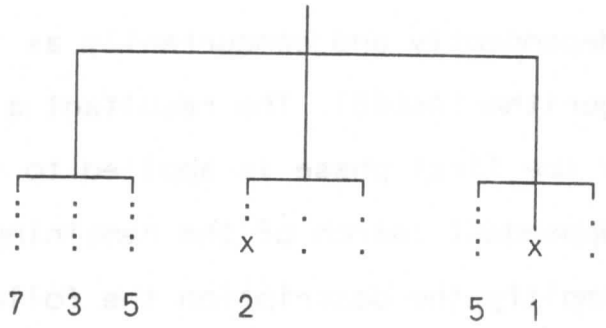

Figure 7. Second phase of search, balance of tree.

The second phase of the search is illustrated in Figure 7, again solid lines show the branches examined during this phase, single dots for lines never considered, and double dots for variations completed during the first phase. Assuming perfect ordering, the search will have cost

$$B(W,D-1) + (W-1)*B(W,D-2).$$

This model has been simulated for the cases $W < 6$ and trees with random terminal nodes [AKL80]. However, although it is not yet clear how effective an actual implementation might be, an important point has been made: certain subtrees must be searched, no matter what the conditions, and so they may as well be searched in parallel, although perhaps not with the narrowest possible bounds that sequential alpha-beta could supply.

SCOUT can be adapted in a similar manner to a parallel system [AKL81]. Simulations indicate that parallel SCOUT is slightly better than parallel alpha-beta for strongly ordered trees, but alpha-beta is better as trees become less ordered [AKL81].

5.3. Processor Tree Hierarchy: In order to limit interprocessor communication it is convenient to attach processors in a very regular way. For example, in the processor tree of Figure 9, each node in the hierarchy has a distinct computational function, and an orderly connection mechanism is used. In the simplest case, all non-terminal nodes of the processor tree execute a Master algorithm. They receive a position and an alpha-beta window from their parent, generate successor positions and assign them to child processors. Whenever a child completes it returns a score for its subtree. If this score causes the alpha bound to change, the master interrupts its children and forces them to update their alpha-beta values. The terminal nodes of the processor tree also receive a position and a window, but simply execute a Slave algorithm to construct the game tree to its maximum permitted depth, evaluate the terminal nodes and return to the master (parent) the best score for the subtree. This is essentially the tree-splitting algorithm [FISH80], see Figure 8.

```
treesplit(position p, int alpha, int beta)
{   int w, i, t[MAXWIDTH];
    processor j;
    if I am a leaf processor
        return(alphabeta(p,alpha,beta));
                                /*                          */
    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,-beta,-alpha);
            critical
            {   if (t[i] > alpha) alpha = t[i];
            }
            if (alpha ≥ beta)
            {   terminate();
                return(alpha);
            }
        }
    }
    return(alpha);
}
```

Figure 8. The Treesplit Algorithm

Several constructs have been adapted from Fishburn [FISH81].

1. j.treesplit indicates the execution of procedure treesplit on processor j.
2. parfor, a parallel for loop, 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 processes in the parfor loop that are still active.

A sample processor tree implementation, employing 4 processors, is shown in Figure 9.

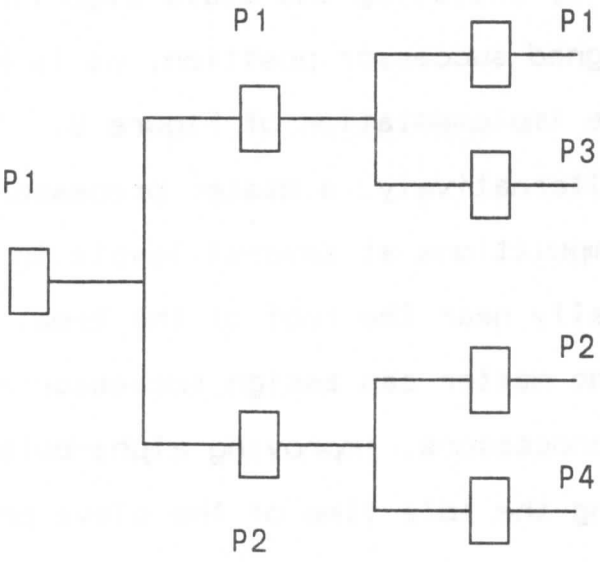

Figure 9. Example of processor tree method.

An important feature from the performance point of view is the dynamic updating of the alpha-beta windows, since this speeds the completion of the child processors. Even though an inexpensive mechanism for dynamically sharing these bounds is available[FISH80], a large amount of time is still spent computing without their benefit. However, the method is relatively simple, as shown by the following pseudo code for an interrupt invoked update mechanism:-

```
int alpha[MAXDEPTH], beta[MAXDEPTH];
/*   bounds are stored in global tables  */
update (depth, side, bound)
{
   if (side > 0)
      alpha[k] = max(alpha[k], bound);
   else
      beta[k] = min(beta[k], bound);
   if (depth > 0)
      update(depth-1, -side, -bound);
}
```

There are a number of refinements to this processor tree scheme.

(a). Since the masters spend most of their time waiting for a child processor to complete, their idle time can be

filled by executing the slave algorithm for the next unassigned successor position, as is essentially the case for the implementation of Figure 9.

(b). Alternatively, a master processor may take charge of the computations at several levels in the game tree, especially near the root of the tree.

(c). The master can assign successor's successors to the child processors, improving alpha-beta value sharing and reducing the idle time of the slave processors.

The disadvantage of these refinements is that either a more involved mechanism is needed to indicate completion of a child process (a and b), or increased interprocessor communication is necessary (c).

5.4. PV-splitting : The algorithm proposed in this section is designed for efficient search of strongly ordered trees. This is carried out by a refinement of the tree-splitting algorithm [FISH80], but which operates on the principal variation, hence the name PV-splitting. This algorithm assumes an underlying hierarchical processor organization. The advantages of this choice are many. Most importantly, the regular configuration limits the complexity of interprocessor communication that is required, and simplifies the control structure for processor initiation and termination. One of the major goals in designing this algorithm was its applicability for an actual physical system.

To understand the basis of the PV-splitting algorithm it is necessary to closely examine the nature of the tree searched by alpha-beta under optimal ordering conditions. Nodes in the tree

can be classified into one of three types. A precise definition of these types can be found in [KNUT75]. Intuitively, type 1 nodes are those on the principal variation, while type 2 nodes are alternatives to the principal variation. Type 3 nodes are successors of type 2, and successors of type 3 are again of type 2. The following observations can be made:

(1) At type 1 and 2 nodes, the best move must be considered first.

(2) At type 1 and 3 nodes, all the successors are examined.

(3) 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 cutoff with less than a full-width search. Maximum benefit from this cutoff is only possible, however, if the best alpha value is available. There is strong reason, therefore, to establish this alpha value before searching type 2 nodes.

Figure 10 illustrates the pvsplit algorithm. Pvsplit makes a call to the treesplit algorithm of Figure 8.

(a). Optimally ordered tree

Alpha-beta takes 1151+671 time units

(L,K)	tree-splitting	PV-splitting
(1,2)	1222	961
(1,4)	922	505
(1,8)	772	277
(2,2)	910	648
(3,2)	778	———

(b). Strongly ordered trees

Alpha-beta takes 4165 time units

(L,K)	tree-splitting	PV-splitting
(1,2)	2700	2264
(1,4)	2030	1425
(1,8)	1859	1084
(2,2)	1724	1587
(3,2)	1172	———

Table 2. Comparison between tree-splitting and PV-splitting for various processor tree configurations.

These preliminary figures indicate that PV-splitting, as expected, outperforms ordinary tree-splitting. The wider the processor tree, the greater the relative discrepancy. The values for processor trees of configuration (2,2) and (3,2) are included for comparison with the (1,4) and (1,8) structures respectively, since the corresponding systems have equal numbers of slave nodes. Apparently PV-splitting still does better, but

this is highly dependent on the ordering of the tree.

6. CONCLUSIONS

This paper has shown that many of the techniques employed by sequential game-playing programs to improve searching efficiency are applicable to parallel systems. Of particular importance is the proposed parallel implementation of transposition tables, since such tables provide significant performance improvement. It is therefore reasonable to assume that the trees to be searched by parallel algorithms will be strongly ordered, and the resultant properties can be used to advantage. Preliminary results on the proposed PV-splitting indicate that this method is able to utilize the ordered-tree characteristics to increase searching speed.

More detailed analysis of PV-splitting is necessary, mainly in conjunction with the alpha-beta search enhancements. Such study is probably only possible in an actual game playing program. The underlying processor tree architecture of the tree-splitting algorithms provides a convenient implementation framework.

REFERENCES

ADEL79 G.M. Adelson-Velsky, V.L. Arlazarov and M.V. Donskoy, "Algorithms of Adaptive Search", Machine Intelligence 9, (J.E. Hayes, D. Michie and L.I. Mikulich, editors), Wiley 1979, pp 373-384.

AKL80 S. Akl, D. Barnard and R. Doran, "Design, analysis and implementation of a parallel alpha-beta algorithm", TR 80-98, Computing and Information Science Dept., Queen's University, Kingston, 1980.

AKL81 S. Akl and R. Doran, "A comparison of parallel implementations of the alpha-beta and scout tree search algorithms using the game of checkers", TR 81-121, Computing and Information Science Dept., Queen's University, Kingston, 1981.

BAUD78 G. Baudet, "The design and analysis of algorithms for asynchronous multiprocessors", Ph.D. Dissertation, Carnegie-Mellon Univ., Pittsburgh, Pa., 1978.

CAMP81 M. Campbell, "Algorithms for parallel search of game trees", M.Sc. thesis, Computing Science Dept., Univ. of Alberta, Edmonton, 1981.

CICH73 R.J. Cichelli, "Research progress report in computer chess", SIGART Newsletter 41, Jun. 1973, pp 32-36.

ENSL74 P. Enslow, Multiprocessors and Parallel Processing, Wiley, 1974.

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

FISH81 J. Fishburn, "Analysis of speedup in distributed algorithms", Ph.D. Dissertation, University of Wisconsin-Madison, 1981.

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

GILL72 J. Gillogly, "The technology chess program", Artificial Intelligence 3 (1972), 145-163.

GILL78 J. Gillogly, "Performance analysis of the Technology chess program", Ph.D. Dissertation, Carnegie-Mellon Univ., Pittsburgh, Pa., 1978.

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

MARS74 T.A. Marsland and P.G. Rushton, "A study of techniques for game-playing programs", Advances in Cybernetics

and Systems, (J. Rose editor), Vol 1, 1974, pp 363-371.

MARS80 T.A. Marsland, M.S. Campbell and A.L. Rivera, "Parallel search of game trees", TR 80-7, Computing Science Dept., Univ. of Alberta, Edmonton, 1980.

MARS81 T.A. Marsland and M.S. Campbell, "A survey of enhancements to the alpha-beta algorithm", ACM81 Conference Proceedings, Los Angeles, pp ?-?.

NEWB77 M. Newborn, "The efficiency of the alpha-beta search in trees with branch dependent terminal node scores", Artificial Intelligence 8 (1977), 137-153.

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

SLAG69 J.R. Slagle and J.K. Dixon, "Experiments with some programs that search game trees", JACM 16, 1969, pp 189-207.

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

TRUS81 T.R. Truscott, "Techniques used in minimax game-playing programs", Masters thesis, Duke Univ., Durham, NC, 1981.

WEIT80 C. Weitzman, Distributed micro/minicomputer systems, Prentice Hall, 1980.

ZOBR70 A.L. Zobrist, "A hashing method with applications for game playing", TR 88, Computer Science Dept., Univ. of Wisconsin-Madison, 1970.

CHESS PROGRAMS REFERENCED

1. AWIT - T.A. Marsland; University of Alberta
2. BEBE - T. Scherzer; SYS-10 Inc.
3. BELLE - K. Thompson, J. Condon; Bell Telephone Laboratories
4. BLITZ - R. Hyatt, A. Gower; Univ. of Southern Mississippi
5. CHESS - D. Slate, L. Atkin; Northwestern University
6. DUCHESS - T. Truscott, B. Wright, E. Jensen; Duke University
7. OSTRICH - M. Newborn; McGill University
8. L'EXCENTRIQUE - C. Jarry; Montreal
9. TECH - J. Gillogly; Carnegie-Mellon University

```
AB(position p, int alpha, int beta, int depth)
{   int i, t, w, type, score, flag;
    position p.opt;
    type = retrieve(p, depth, score, flag, p.opt);
      /* type <  0 - position not in table
         type == 0 - position in table, but length < depth
         type >  0 - position in table, length >= depth
     */
    if (type > 0)
    {   if (flag == VALID) goto done;
        if (flag == LBOUND)
          alpha = max(alpha, score);
        else /* flag == UBOUND */
          beta = min(beta, score);
        if (score >= beta)  goto done;
    }
      /* Note beneficial update of alpha or beta
         bound assumes full width search.
         Score in table insufficient to terminate search
         so continue as usual, but try p.opt (from table)
         before generating other moves, if p is non-terminal.
      */
    score = alpha;
    if  ((type >= 0) and (p.opt != NULL))
    {   t = -AB(p.opt, -beta, -score, depth-1);
        if (t > score) score = t;
        if (score >= beta)  goto done;
    }
      /* no cutoff. Generate moves, put p.opt first.
      */
    w = generate(p);
    if (w == 0)                  /* mate or stalemate */
    {   p.opt = NULL;
        score = evaluate(p);
        goto done;
    }
    for i = 2 to w do
    {
       if (depth == 0)
         t = evaluate(p.i);
       else
          t = -AB(p.i, -beta, -score, depth-1);
        if (t > score)
        {  score = t;
           p.opt = p.i;     /* note best successor */
           if (score >= beta) goto done;
        }
    }
done:
    flag = VALID;
    if (score <= alpha) flag = UBOUND;
    if (score >= beta) flag = LBOUND;
    store(p, depth, score, flag, p.opt);
    return(score);
}
```

Appendix: Alpha-beta implementation using transposition table