Parallel Search of Strongly Ordered Game Trees

T. A. MARSLAND AND M. CAMPBELL

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

The "alpha-beta" algorithm forms the basis of many programs that search game trees. A number of methods have been designed to improve the utility of the sequential version of this algorithm, especially for use in game-playing programs. These enhancements are based on the observation that alpha-beta is most effective when the best move in each position is considered early in the search. Trees that have this so-called "strong ordering" property are not only of practical importance but possess characteristics that can be exploited in both sequential and parallel environments.

This paper draws upon experiences gained during the development of programs which search chess game trees. Over the past decade major enhancements to the alpha-beta algorithm have been developed by people building game-playing programs, and many of these methods will be surveyed and compared here. The balance of the paper contains a study of contemporary methods for searching chess game trees in parallel, using an arbitrary number of independent processors. To make efficient use of these processors, one must have a clear understanding of the basic properties of the trees actually traversed when alpha-beta cutoffs occur. This paper provides such insights and concludes with a brief description of our own refinement to a standard parallel search algorithm for this problem.

Categories and Subject Descriptors: C.0 [**Computer Systems Organization**]: General—*system architectures*; C.1.2 [**Processor Architectures**]: Multiple Data Stream Architectures (Multiprocessors)—*associative processors; parallel processors*; C.4 [**Computer Systems Organization**]: Performance of Systems— *design studies*; F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems—*pattern matching*; I.2.8 [**Artificial Intelligence**]: Problem Solving, Control Methods and Search—*heuristic methods; graph and tree search strategies*

General Terms: Algorithms, Experimentation

Additional Key Words and Phrases: Alpha-beta search, computer chess, game playing, parallel search, tree decomposition

INTRODUCTION

Chess, checkers, kalah, and go are popular examples of two-person "zero-sum" games—that is, games in which one player's losses are the opponent's gains. There are a number of methods for programming a computer to play such games. The simplest (and most successful) programs have as their basis "brute-force" search, in which an exhaustive examination of all possible sequences of moves is carried out until terminal

CONTENTS

positions are reached (no more legal moves). By subsequently backing up through this tree of moves, a player can find the best move for one side by using the *minimax* algorithm. Minimax search assumes that the players will always select the alternative that is best for them in any given position. The advantage of such an approach is that it guarantees perfect play; a “winning” position will always be won, and a drawn position will at least be drawn. This strategy works admirably for games such as tic-tac-toe, but most interesting games are too large to be handled in this fashion. In chess, for example, de Groot [DEGR65] has estimated that the number of positions that could be explored is 38^{84}.

The method used in most current game-playing programs is to approximate the whole game tree by searching a succession of fixed-depth trees. Since the search is truncated before the end of the game is reached, estimates of the value of the discarded portion of the tree are made by an *evaluation function*. Such estimates are inherently unreliable, however, since if one had a perfect evaluation function, there would be no need to conduct a search at all. Empirical evidence suggests that for most common games, the deeper the search, the higher the quality of the play. The *alpha-beta pruning algorithm* is one technique for increasing the speed of minimax search. Alpha-beta is able to avoid searching subtrees that are judged not relevant to the outcome of the search, while always producing the same result as minimax. A complete description of the minimax and alpha-beta algorithms can be found elsewhere [KNUT75, NILS80]; our own summary of them, along with a programming example, appears in the next section.

In this paper we assess the effectiveness of various refinements to the alpha-beta algorithm, especially with regard to their importance in searching trees whose branches are ordered to favor early detection of the ultimate solution. Most theoretical work on both sequential and parallel game-tree searching has been primarily concerned with random trees [FULL73, KNUT75, BAUD78], although there is one major exception [NEWB77]. In practice, truly random trees are quite uncommon, and so, under reasonable assumptions, improvements to the searching algorithm are possible. Also, these game tree problems may be partitioned in a number of different ways to facilitate parallel solutions. We compare various ways of doing parallel alpha-beta searches and present algorithms that attempt to take advantage of the characteristics of strongly ordered trees. The rationale for this work is that well-ordered trees are not only more realistic, but possess properties that can be exploited in a parallel environment. General information about processor selection and communication is not presented here, since it is commonly available elsewhere [WEIT80, ENSL74].

1. SEQUENTIAL SEARCH ALGORITHMS

Given a position p in a two-person zero-sum game, all the potential continuations from p can be represented as a game tree, with nodes corresponding to positions and branches to moves. Leaves of the tree are called *terminal* nodes, and are assigned values by the evaluation function. All remaining nodes are classified as *nonterminal*. The task in searching a game tree is to determine the *minimax value* of the root node p. Intuitively, the minimax value of a node is the best value attainable from that node against an opponent who uses a similar technique to select best moves.

The minimax algorithm assumes that there are two players, called Max and Min, and it assigns a value to every node in a game tree (and in particular to the root) as follows. Terminal nodes are assigned values that represent the desirability of the position from Max's point of view. Nonterminal nodes are assigned a value recursively. If Max is to move at a given nonterminal node, its value is the maximum over the values of its successors. Similarly, if it is Min's move, he will choose the minimum over the values of the successors.

The alpha-beta algorithm produces the same result as minimax, but at reduced cost. Typical usage of the alpha-beta algorithm involves a function call of the form

V := alphabeta (p, alpha, beta, depth);

where p represents a position, (alpha, beta) represents the *search window* or range of values (the bounds) over which the search is to be made, and depth represents the intended length of the search path measured in *ply* (i.e., moves). Typically, p is a pointer to a data structure that describes the state of the game at this node. The exact nature of the structure is very implementation dependent. The value returned by the function, V, is the minimax value for the position p. Figure 1 illustrates a “negamax” [KNUT75] version of the depth-limited alpha-beta algorithm. Use of the negamax framework is particularly attractive since, by maximizing over the negative of the values returned by the search, one avoids the need to select the correct maximum/minimum operation. Our various program excerpts are presented in a PASCAL-like language, extended with the return statement for function termination.

Although the alpha-beta function only returns one value, it is also necessary to keep track of the optimal move in position p. This is a simple matter, but is not illustrated in Figure 1 in order to keep the structure of the program as simple as possible. Note also that our version of alpha-beta includes the functions `evaluate`, to assess a terminal node, and `generate`, to produce $p.1$ through $p.w$, pointers to the immediate successors of position p. Details about the maintenance of these successors have been omitted, although functions `make` and `undo` are included to play and retract the current move. Also, `evaluate` is usually complex, because the whole quality of the play hinges on the assessment made here [SLAT77]. Since the majority of the nodes in the tree are terminal, the function must not be too time consuming. Nevertheless, in chess programs `evaluate` often extends the search using moves that are selected from captures and certain checks. This is done to

```
function alphabeta(p : position; alpha, beta, depth : integer) : integer;
     VAR width, score, i, value : integer;
BEGIN
     IF (depth <= 0) THEN            { a terminal node? }
          return(evaluate(p));
                                     { determine successor positions }
     width := generate(p);           { p.1 .. p.w and return number of }
                                     { successors as function value }
     IF (width = 0) THEN             { no legal moves? }
          return(evaluate(p));
     score := alpha;
     FOR i := 1 TO width DO BEGIN
          make(p.i);
          value := -alphabeta(p.i, -beta, -score, depth-1);
          undo(p.i);

          IF (value > score) THEN    { an improvement? }
               score := value;
          IF (score >= beta) THEN    { a cutoff? }
               return ( score );
     END;
     return( score );
END;
```

Figure 1. Negamax version of the depth-limited alphabeta function.

ensure that only quiescent positions are evaluated.

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 evaluated. The minimax algorithm will always examine $M(W, D) = W^D$ terminal nodes, while at best the alpha-beta algorithm evaluates only [SLAG69]

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

where $\lceil x \rceil$ and $\lfloor x \rfloor$ represent upper and lower integer bounds on x. Thus the efficiency of the alpha-beta algorithm can be very good, potentially visiting as few as two times the square root of the maximum number of nodes, while still generating the same solution path (the *principal variation*) from the root node. However, this optimal performance is achieved only when the first move considered at each node is the best one. That alpha-beta is effective in reducing the number of terminal nodes evaluated is clear from a study of the sample uniform tree (width = 3 and depth = 3) shown in Figure 2. The numbers at the terminal nodes would be produced by an evaluation function. The other numbers are the values of the individual subtrees, as passed back (backed up) to the root node by the alpha-beta algorithm. Thus the minimax value of this tree is 3, and only 16 terminal nodes would be visited, as shown by the solid lines, rather than 27, as would be the

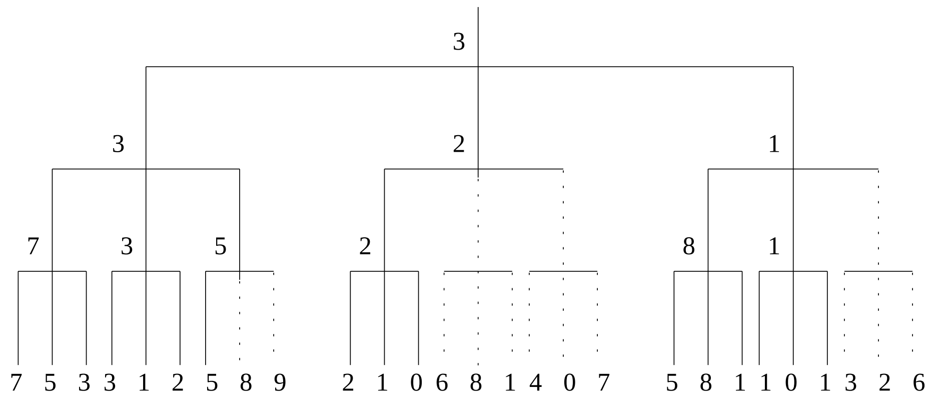

Figure 2. Minimax tree showing alpha-beta cutoffs.

case for an exhaustive search. The dotted branches of the tree are said to have been cut off or pruned by the alpha-beta algorithm.

For the purposes of this paper more realistic assumptions are needed. A *random uniform* game tree will be defined to be one in which the terminal node scores (values of nodes at the maximum depth in the tree) are uniformly distributed across some fixed range of value. Also, trees are defined to be *strongly ordered* if

(1) 70 percent of the time the first branch from each node is best;
(2) 90 percent of the time the best move is in the first quarter of the branches being searched.

Although these numbers may appear to be rather arbitrary, it turns out that static ordering mechanisms, when combined with heuristic methods and memo functions [BIRD80], tend to produce trees with these properties [GILL78, MARS74]. Thus for each variation of the alpha-beta algorithm we can define the following quantities:

$R(W, D)$ = average number of terminal nodes visited in a search of a random uniform game tree;
$S(W, D)$ = average number of terminal nodes visited in a search of a strongly ordered uniform game tree.

At each terminal node visited during the search the evaluation function is invoked to assess the position.

While the performance of alpha-beta on random trees has a solid theoretical basis [FULL73, BAUD78], at present only empirical data are available for strongly ordered trees [GRIF76]. Nevertheless, statistical evidence supports the relationship

$$B(W, D) < S(W, D) < R(W, D) << M(W, D) = W^D.$$

Relative values for these terms can be seen from our Monte Carlo simulation results, presented in Table 1. These results were obtained from trees of depth 4 or less, and terminal node scores were chosen from the range 0-127. To estimate R , the values were assigned randomly to the terminal nodes. Because of the way the scores were chosen, they were not all unique and distinct, and so R is slightly underestimated. The calculation of S, on the other hand, relied on the use of a distribution function at the terminal nodes, to ensure that the best move met our strong ordering criteria. The parenthesized numbers represent the

Table 1. Expected Search Costs for Trees (in Number of Terminal Nodes Visited)

Width	Best	S-Strong		R-Random		Minimax
Depth = 3						
8	71	105	(21)	181	(36)	512
16	271	405	(64)	786	(114)	4096
24	599	857	(115)	1752	(250)	13824
Depth = 4						
8	127	281	(88)	690	(163)	4096
14	511	1286	(430)	4125	(875)	65536
24	1151	2946	(1013)	10425	(1891)	331776

standard deviation for 100 independent search trials. Therefore Table 1 illustrates the relative efficiency of alpha-beta under best, strong, random, and worst move ordering assumptions, and supports the view that $R(W, D)$ is very much less than $M(W, D)$.

2. ENHANCEMENTS TO ALPHA-BETA SEARCHING

Many of the following techniques have been developed in efficiency-conscious chess programs, but ones that discard moves only at the terminal nodes. These programs are often called *full-width* programs because they examine all necessary moves at every node, employing "forward pruning" only at the terminal nodes to eliminate the quiescent moves. Even so, the basic methods are applicable to most programs that search game trees.

2.1 Aspiration Search

The interval enclosed by alpha and beta is referred to as the window. For the alpha-beta algorithm to be effective, the minimax value of the root position must lie within the initial window. Generally speaking, the narrower the initial window, the better is the algorithm's performance. In many problem domains, including chess, there are reliable methods to estimate the value 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 value and e the expected error. There are three possible outcomes of this so-called aspiration search, depending on V^*, the actual minimax value of a position p. Since V^* is in the range –INF $< V* <$ +INF, one of the following conditions is true:

(1) If $V^* \le V - e$, then alphabeta$(p, V - e, V + e, D) = V - e$.
(2) If $V^* \ge V + e$, then alphabeta$(p, V - e, V + e, D) = V + e$.
(3) If $V - e < V^* < V + e$, then alphabeta$(p, V - e, V + e, D) = V^*$.

Cases 1 and 2 are referred to as *failing low* and *failing high,* respectively [FISH80]. Only in Case 3 is the true value of the position p found. In this case the search time will not be greater than that for a full window search, and cannot be less than that for an optimal search. In the failing low case, it is necessary for the search to show that every alternative from the root is less than $V - e$. Thus, assuming a perfectly ordered tree of width W and depth D,

```
{ Assume V = estimated value of position p, and e = expected error limit }
     alpha := V - e;
     beta := V + e;

     V := alphabeta(p, alpha, beta, depth);
     IF (V >= beta) THEN                         { failing high }
          V := alphabeta(p, beta, +INF, depth);
     ELSE
     IF (V <= alpha) THEN                         { failing low }
          V := alphabeta(p, -INF, alpha, depth);
               {       arrive here after successful search     }
```

Figure 3. Aspiration alpha-beta search.

$W^{\lceil D/2 \rceil}$ nodes must be examined.

Conversely, in the failing high case the search stops as soon as an alternative is found which is 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. As illustrated in Figure 3,

V := alphabeta(p, beta, +INF, D);

must be invoked for the failing high case. Empirical evidence has shown aspiration searches to be very effective; in TECH,[1] search time reductions averaging 23 percent were noted [GILL78]. This figure was confirmed 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. Falphabeta is produced through two modifications to the alpha-beta function of Figure 1. The recursive call becomes

value := –falphabeta($p.i$, –beta, –max(alpha, score), depth–1);

and score is initialized to –INF, rather than alpha. The function falphabeta gives a tighter bound on the true value of the tree when the search fails high or low, and does so while searching the same nodes as alpha-beta [FISH80]. Although falphabeta requires a slightly larger overhead, any system that uses aspiration searches should find the technique practical, even though the actual savings may be small, since the next iteration may start with a better window.

[1] All footnotes in this paper refer to chess programs.
TECH was formulated by J. Gillogly, Carnegie-Mellon University. Further details concerning some of these programs can be found in More Chess and Computers, by D. Levy and M. Newborn, Computer Science Press, Rockville, Md., 1982.

Table 2. Components of a Minimal Transposition Table Entry

lock	move	score	flag	length	prio

lock is used to check that the table entry corresponds to the desired tree position;
move is the best move in the position, as determined by a previous search;
score is the value of the subtree as computed previously;
flag indicates whether score is an upper bound, lower bound, or a true value;
length is the height of the subtree upon which score is based;
prio is used in table management, to select entries for deletion.

2.2 Transposition Table

In carrying out a game tree search it is not uncommon for positions to recur in numerous places throughout the tree. Rather than reevaluate these positions, it may be possible simply to retrieve the equivalent search result from a large hash table [SORE78] whose entries represent positions. For game modeling, nearly perfect hashing functions can be produced. Although there are some table management problems that must be solved, the technique has very low overhead and large potential gains.

A typical hash index generation method is the one proposed by Zobrist [ZOBR70], and an illustration of its application to chess can be found in a paper by Marsland and Campbell [MARS81]. A transposition table entry could have the components shown in Table 2.

For best effect the transposition table must be incorporated into alpha-beta carefully, as shown in Figure 4. Note that our implementation employs two functions, store and retrieve, to perform transposition table access, but details have not been included. In addition, functions make and undo, to play and retract moves, are omitted. When a position reached during a search is located in the table (i.e., the *lock* matches), there are two possibilities, depending on whether *length* is smaller than the remaining depth to be searched. If possible, *score* is used to reduce the size of the current alpha-beta window, unless *length* is less than the depth of search. In any case, *move* must be tried immediately, since if it again causes a cutoff, it will save an expensive move generation (Figure 4).

Other possibilities for a transposition table entry also exist. For example, DUCHESS[2] maintains both upper and lower bounds on the position score, with separate lengths for each [TRUS81], thus improving the possibility that one of the bounds may be used to reduce the window size.

Transposition tables are most effective in chess end games, where there are fewer pieces and more reversible moves. CHESS 4.7[3] was the first to demonstrate searches of more than 25 ply in certain types of King and Pawn endings; by taking advantage of this knowledge, gains of a factor of 5 or more are typical [SLAT77]. Even in complex middle games, however, significant (25-50 percent) performance improvement has been observed [THOM81]. Furthermore, if the actual length of the subtree whose result is retrieved from the transposition table is greater than the specified search depth of the current variation, then the effective length of the search for this variation is greater than the maximum specified depth, and so the equivalent of an extended search is done. Also, successful use of the transposition table makes the tree more strongly ordered. Thus search times shorter than those for optimal alpha-beta are possible, since some subtrees need not be reevaluated.

[2]DUCHESS was formulated by T. Truscott, B. Wright, and E. Jensen, Duke University.
[3]CHESS was formulated by D. Slate, and L. Atkin, Northwestern University.

```
function AB(p : position; alpha, beta, depth : integer) : integer;
     VAR i, value, width, length, score, flag : integer;
         p.opt : position;
BEGIN
     retrieve(p, length, score, flag, p.opt);
                { length is the effective subtree height.
                  length < 0 - position not in table.
                  length 2 0 - position in table. }
     IF (length >= depth) THEN BEGIN
          IF (flag = VALID) THEN return(score);
          IF (flag = LBOUND) THEN
               alpha := max(alpha, score);
          IF (flag = UBOUND) THEN
               beta := min(beta, score);
          IF (alpha >= beta) THEN return(score);
     END;
{ 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 (depth <= 0) THEN                { terminal node? }
          return(evaluate(p));
     IF (length >= 0) THEN BEGIN
          score := -AB(p.opt, -beta, -alpha, depth-1);
          IF (score >= beta) THEN goto done;
     END ELSE score := -INF;
                    { No cutoff, generate moves }
     width := generate(p);
     IF (width = 0) THEN             { mate or stalemate? }
          return(evaluate(p));
     FOR i := 1 TO width DO BEGIN
          value := -AB(p.i, -beta, -max(alpha, score), depth-1);
          IF (value > score) THEN BEGIN
               score := value;
               p.opt := p.i; { note best successor }
               IF (score >= beta) THEN goto done;
          END;
     END;
done:
     flag := VALID;
     IF (score <= alpha) THEN flag := UBOUND;
     IF (score >= beta) THEN flag := LBOUND;
     IF (length <= depth) THEN store(p, depth, score, flag, p.opt);
     return(score);
END;
```

Figure 4. Alpha-beta implementation using a transposition table.

```
V := 0;
FOR D := 1 TO depth DO BEGIN
     alpha := V - e;
     beta := V + e;
     V := falphabeta(p, alpha, beta, D);

     IF (V >= beta) THEN
          V := falphabeta(p, V, +INF, D);
     ELSE
     IF (V <= alpha) THEN
          V := falphabeta(p, -INF, V, D);

     sort(p); { best move so far is tried first on next iteration. }
END;
```

Figure 5. Iterative deepening with aspiration search.

2.3 Killer Heuristic

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 [GREE67]. Any move that causes a cutoff at level N in the tree is said to have refuted the move at level $N-1$ [CICH73]. A node is at level N in the tree if it is N ply from the root node. There are many ways of using this information. For example, the program CHESS 4.7 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. By continually trying the same killer moves, there is a greater possibility of reaching a position already in the table [TRUS81], and thus reducing the time spent searching the tree.

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 $N-1$, then it is worth trying My at level $N-2$, if it exists, before generating all the moves for that position and trying them in order [NEWB79]. An additional method, used by AWIT,[4] 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, since it is not clear that the potential gains exceed the overhead. The actual search reductions produced by the killer heuristic cannot be stated with certainty. Even though use of the killer heuristic did not yield improvements for TECH [GILL78], variations of this method were used in CHESS 4.7, DUCHESS, OSTRICH,[5] and BLITZ,[6] and were found to be effective.

[4]AWIT was formulated by T. A. Marsland, University of Alberta.
[5]OSTRICH was formulated by M. Newborn, McGill University.
[6]BLITZ was formulated by R. Hyatt and A. Gower, University of Southern Mississippi.

2.4 Iterative Deepening

Iterative deepening refers to the process of using a $(D-1)$ ply search to prepare for a D ply search. That is, after a $(D-1)$ ply search one can retain the moves of the principal variation and use them as an initial sequence of moves for a D ply search [SLAT77]. The cost of iterative search (again measured in terms of the number of terminal nodes visited) is given by a recurrence relation of the form

$$S(W, D) = S(W, D-1) + F(W, D),$$

where $F(W, D)$ is the expected cost of an alpha-beta search given the first $D-1$ moves of the principal variation, and W is the search width. The exact nature of $F(W, D)$ is not known, but it has been hypothesized for chess programs that employ transposition tables [MARS81]

$$F(W, D) \simeq B(W, D) + (W-1) * B(W-1, D-2)$$

for the cases $W > 20$ with $D > 4$.

Iterative deepening can be used to advantage in the following ways:

(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) A $(D-1)$ ply search can provide a principal variation, which, with high probability, contains a prefix of the D ply principal variation. This allows the alpha-beta search to proceed more quickly.

(3) The value returned from a $(D-1)$ ply search can be used as the center of an (aspiration) alpha-beta window for the D ply search (Figure 5). It is probable that this window will also contain the value for the current search, thus reducing search time.

These last two points, though significant, are not really complete justifications for the use of iterative deepening. In fact, in experiments with checkers game trees [FISH80], it was found that iterative deepening increased the number of nodes searched by 20 percent (apparently only using Point (2), however). In addition, studies with TECH using a generalized version of (2), but not (3), noted a 5 percent 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 the advantages of iterative deepening. The real searching advantage of iterative deepening, however, is that

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

As a consequence, the search may proceed more quickly since the table entries and killer lists tend to direct the search along lines that are sufficiently good to cause immediate cutoffs.

The importance of transposition tables is illustrated by the performance of the BELLE[7] chess machine [COND82]. Typical chess middle-game positions have branching factors of 35–40. It has been found that in such positions, it normally costs BELLE a factor of between 5 and 6 to go one further ply, in fact, slightly less than the expected cost of optimal alpha-beta [THOM81]. Exactly how much each additional ply improves the performance of a program has recently been quantified by Thompson [THOM82]. This was done by playing a series of matches between $(D+1)$ ply and D ply versions of BELLE, for all values of D from 3 to 8.

[7] BELLE was formulated by K. Thompson and J. Condon, Bell Telephone Laboratories.

An alternative form of iterative deepening, one that is especially appropriate if transposition tables are not used, was employed by L'EXCENTRIQUE.[8] A 2 or 4 ply minimax search was first performed to obtain W minimax move pairs (move and best refutation). These were then sorted and a 6, 8, 10,... ply iterative deepening cycle initiated. The rationale behind 2 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 comparison between this and conventional iterative deepening has been done. Likewise, no quantitative study of the advantages of minimax move pairs over conventional alpha-beta move pairs (move and sufficient refutation) seems to have been done. In either case, this *refutation table* usage is a valuable way of guiding the search, since the storage requirement is only $W * D$ (width * depth) entries. For each variation at the next iteration, the corresponding sequence of moves in the refutation table is tried first. Often these sequences will be sufficient to cut off the search, thus reducing the number of necessary move generations. In our experience with chess programs, use of a refutation table to seed the variations often improves iterative deepening searches by 30 percent.

2.5 Other Searching Techniques

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.

An interesting implementation of the alpha-beta algorithm treats the first variation in a special way. The method was originally called Palphabeta [FISH80] and then renamed Calphabeta [FISH81], but will be referred to here as *principal variation search* or PVS for short. Once a candidate principal variation is obtained, the balance of the tree is searched with a *minimal window* , an alpha-beta window of (–score – 1, –score), where score is the best value found so far (Figure 6). On the other hand, if the tree is poorly ordered, each subtree that is better than its elder siblings must be searched again. Hence there is some risk that PVS will examine more nodes than alpha-beta. When iterative deepening is used to provide a principal variation, PVS becomes more effective, because with each iteration it is increasingly likely that the first move tried is best. The structure of PVS can be seen in Figure 6, which includes an alpha-beta refinement, falphabeta, to enable use of a narrower window whenever the minimal window search fails. For simplicity, make and undo are again omitted.

The basic idea behind PVS is that it will assume that the first move made at each node is best. Thus PVS makes a recursive call to the first successor, $p.1$, and determines its value. The remaining successors, $p.i$ for $2 \leq \mathrm{i} \leq$ width, are examined in turn. If one of these successors has a value that is greater than that for the current principal variation, it becomes the new principal variation, and is searched again with the correct window. It is possible that PVS can also benefit from some form of aspiration search, but to our knowledge that has not yet been accomplished. Even so, our experience with PVS suggests that a 13 percent improvement in speed over an aspiration search may be expected. One practical observation about the operation of the minimal window search has been made, namely, after a failing-high minimal window search at the first level in the tree, there is no need to follow immediately with a full window search [THOM81]. Since we now have a new bound on the minimax value, only if an even more successful variation arises will there be any doubt about which one might be the new principal variation. This refinement has been tested in the BELLE program, and search time reductions of up to 50 percent have been noted.

SCOUT [PEAR80] is a further generalization of PVS, in which the final call to falphabeta is replaced by score := –PVS($p.i$, depth – 1);
In its original form, SCOUT did not use the minimal window idea, but rather an equivalent test proce-

[8]L'EXCENTRIQUE was formulated by C. Jarry, CIP Inc., Sun Life Building, Montreal.

```
function PVS(p : position; depth : integer) : integer;
     VAR width, score, i, value : integer;
BEGIN
     IF (depth <= 0) THEN
          return(evaluate(p));
     width := generate(p);
     IF (width = 0) THEN
          return(evaluate(p));
     score := -PVS(p.1, depth-1);
     FOR i := 2 TO width DO BEGIN
          value := -falphabeta(p.i, -score-1, -score, depth-1);
          IF (value > score) THEN
               score := -falphabeta(p.i, -INF, -value, depth-1);
     END;
     return(score);
END;
```

Figure 6. Minimal window search.

dure. Initial simulation results indicate that PVS is slightly better than SCOUT on strongly ordered trees [CAMP83].

Even though $SSS*$ [STOC79] and *staged SSS** [CAMP83] are effective in the search of random or poorly ordered trees, these algorithms are not significantly better than alpha-beta on strongly ordered trees, and require more time and space. We do not consider further those methods that are not especially suited to the search of strongly ordered trees.

3. APPROACHES TO PARALLEL TREE SEARCH

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 intercommunication between processors is necessary, causing losses as they wait for these messages. More important, if independent subtrees are searched concurrently, it is likely that redundant nodes will be examined, because the best bounds are not always available. In spite of these problems, some processor configurations yield substantially higher effective computing power than others.

3.1 Parallel Evaluation

Current game-playing programs that carry out full-width searches must come to terms with the trade-off between depth of search and complexity of terminal node evaluation. Most of the stronger chess programs employ a rather simplistic scoring (evaluation) function, in order to make time for deeper searches. Nevertheless, a considerable portion of the search time is spent in evaluation: on the order of 40 percent in both BLITZ and DUCHESS.

An obvious application of concurrency to game tree search appears to be within the scoring function itself. A number of processors could be used to evaluate simultaneously 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.[9]

Advantages of this technique are numerous:

(1) Evaluation time can be reduced, allowing deeper searches.
(2) Many small, cheap processors can 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 can be considerably more complex: large amounts of game-specific knowledge can be utilized, and extended arbitrarily.

Admittedly a large proportion of terminal nodes in a full-width search, about 50 percent in CHESS 4.7, need nothing more than a count of the pieces held by each side [SLAT77]. Nevertheless, applying concurrency to the evaluation function, as described above, can improve positional understanding in the remainder. It is these positional factors that are so expensive, and, by this technique, can be computed concurrently with material evaluation and each other.

Another method, employed by BELLE and to a lesser extent by BEBE, is to partition the board and to apply a microprocessor to the maintenance of the data structure associated with each square. Ultimately one could envision an evaluation machine that would consist of a processor hierarchy. For example, bottom-level processors would assess 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 assess reliably. While it is too early to say how far one can successfully pursue this approach in practice, it is clear that there are many opportunities for experimental and theoretical work.

3.2 Parallel Aspiration

Even though the alpha-beta search itself is relatively efficient, the aspiration refinement provides improvement whenever it is successful. One parallel implementation of this idea is to divide the alpha-beta window into nonoverlapping subintervals and apply a processor to each range [BAUD78]. For example, take

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

Ideally Processor 2 will finish first, but, in any case, one of them will succeed, and will do so in less time than a uniprocessor searching over (–INF, +INF). Those processors that fail early can cut off or improve the bounds for others. Baudet [BAUD78] has explored optimal ways of decomposing windows, including in his exposition methods that do not initially cover (–INF, +INF).

There are two important results from this parallel aspiration work:

(1) Maximum expected speedup is typically a factor of 5 or 6, regardless of the number of processors available. This is because the cost of an aspiration search is bounded below by $B(W, D)$.
(2) When the number of processors (K) is small ($K = 2$ or 3), the speedup obtained may be greater than K [BAUD78].

[9] BEBE was formulated by T. Scherzer, SYS-10 Inc., Chicago.

These results are based on certain assumptions; in particular, it is assumed that the distribution of the backed-up value is known. The implications for parallel search of strongly ordered trees are not clear, but since the sequential version of the aspiration search is so effective for chess game trees, one cannot expect the parallel aspiration methods to offer much improvement.

3.3 Tree Decomposition

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

The efficiency of most search algorithms arises from 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 important, 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 exchanged via some sort of message-passing system, or through a globally shared data structure. The former incurs message-passing costs, whereas the latter requires synchronization overhead if high degrees of concurrency are to be achieved. 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.

3.4 Enhancements to Parallel Search

Before considering newer search algorithms, the sequential search enhancements will be assessed to determine their effectiveness in a parallel situation.

Aspiration searching in parallel offers no particular advantage, since a single processor employing a good initial window will do just as well. However, the sequential version of aspiration searching, when used in conjunction with iterative deepening, is equally, if not more, applicable to parallel systems, since a common problem of such systems concerns their inappropriately wide windows.

Transposition tables continue to be effective, provided that all the processors access the same table. Since transposition table usage is a naturally autonomous function, it is an especially attractive parallel application. Furthermore, a processor can do something useful while waiting for access to the transposition table, namely, evaluate 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 7). This scheme is quite independent of the relationships between the game processors, C_1, C_2, and C_3, which share and provide updates for the transposition table memory. The game processors place their transposition requests with a manager, P_0. A potential bottleneck exists there, but this should not be severe since P_0 has no significant computational

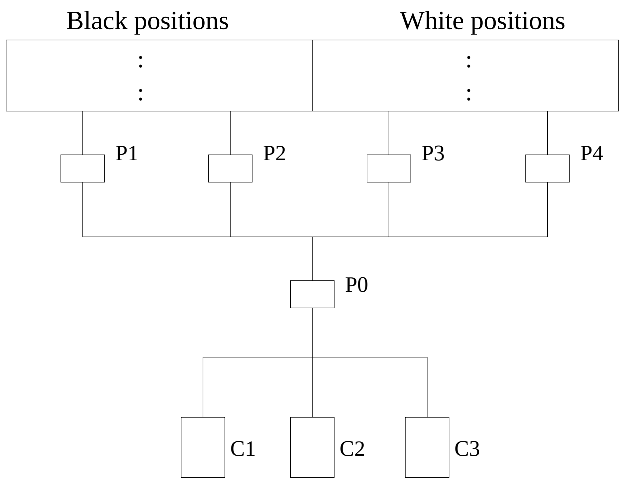

Figure 7. Transposition table access and management.

functions beyond those necessary for the routing operations.

The killer heuristic presents problems similar to the transposition table. The killer list of moves that have been effective in comparable positions is so small, however, that the management problems are much reduced.

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

4. TREE DECOMPOSITION METHODS

4.1 Naive Approach

With a static decomposition, the game tree is split into groups of subtrees, and each subtree is assigned to a different processor (Figure 8). As processors complete, they are allocated to the next group of subtrees, until the full tree is evaluated. Ideally each processor should be given exactly the same size of subtree to search, in order that all may complete at about the same time. Even so, the efficiency of this method is very sensitive to the width/processor ratio W/K.

More important, for a typical game tree with $W = 40$, a direct alpha-beta search is equivalent to an exhaustive search of a tree with $W = 7$ [GILL72]. Thus, if $K = 40$ processors are applied at the root of the tree, the average speedup 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.

4.2 Minimal Tree

The minimal tree that must be searched by the sequential version of the alpha-beta algorithm has a very

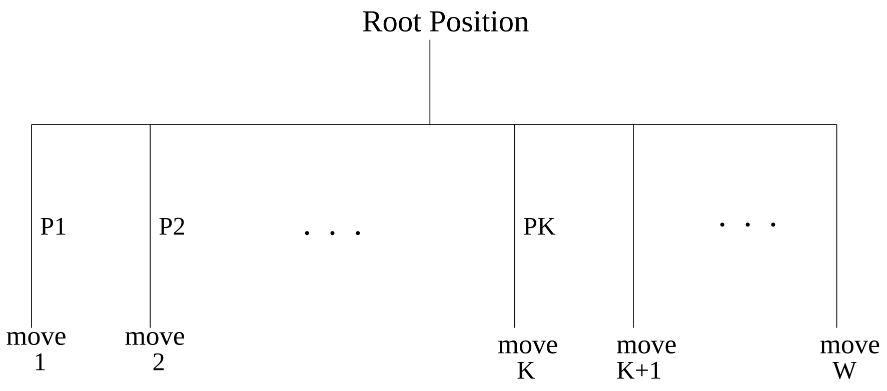

Figure 8. Apply all K processors at the first level.

definite structure. It has been proposed that these subtrees be searched independently and concurrently as the first stage of a parallel algorithm [AKL82]. Akl's method uses the alpha-beta window generated by the first phase to speed a second phase, where an independent parallel search of the remaining subtrees takes place. To simplify the description, the following terminology is used:

The first branch of a node points to the *left son,* and is contained in the *left subtree.* All other branches of the node point to *right sons* and are in *right subtrees.*

Phase 1. Recursively 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. The right sons will have temporary values, which are the values of their left sons. Figure 9 shows the first phase of a search on a 3 ply tree. The branches explored are marked with solid lines and terminal scores.

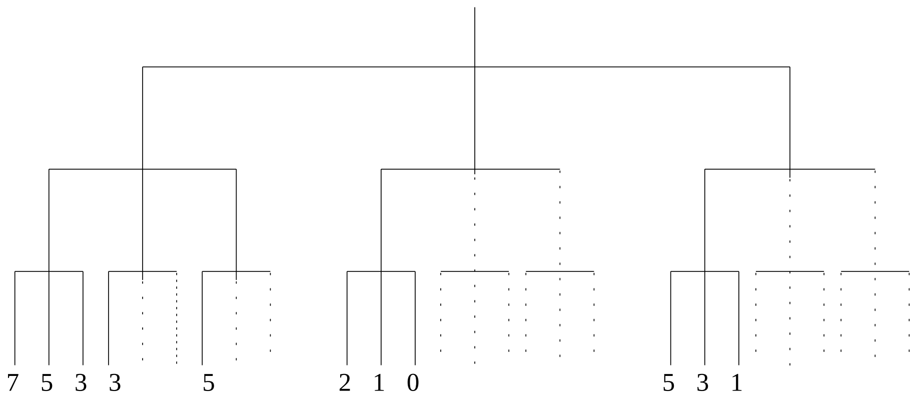

Figure 9. Necessary tree searched during the first phase.

Phase 2. Those subtrees whose temporary values are insufficient to cause a cutoff are now assigned different processors and searched one branch at a time until all right sons have been cut off or fully explored. The second phase of the search is illustrated in Figure 10. Again solid lines show the branches examined during this phase, single dots show lines never considered, and double dots show variations completed during the first phase. Assuming perfect ordering, the search will have cost $B(W, D-1) + (W-1) * B(W, D-2)$, where W is the width of each node and D the maximum depth of search.

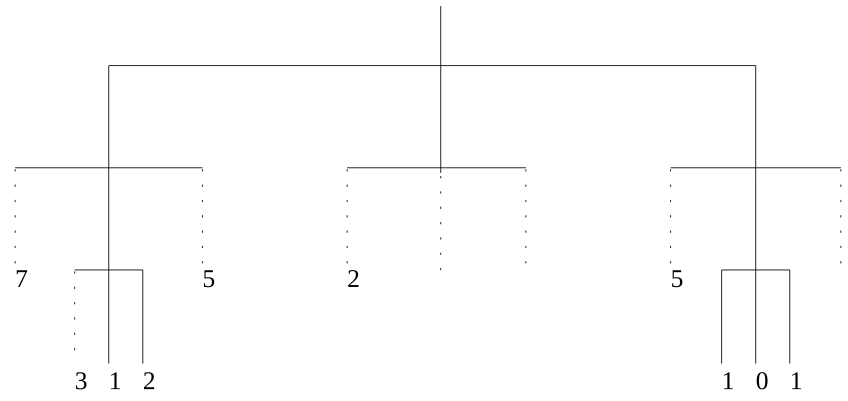

Figure 10. Balance of the tree searched during the second phase.

This model has been simulated for cases with $W \leq 20$ and trees with random terminal nodes [AKL82]. 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 to a parallel system in a similar manner [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.

4.3 Processor Tree Hierarchy

In order to limit interprocessor communication, one should use simple connection mechanisms. For example, in the processor tree of Figure 11 each node in the hierarchy has a *fan-out* of 2 and a distinct computational function. In the simplest case, all nonterminal 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 value for its subtree. If this value causes the alpha bound to change, the master interrupts its children and forces them to update their alpha-beta values, using the mechanism of Figure 12.

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 value for the subtree. This is essentially the tree-splitting algorithm [FISH80], and is informally presented in Figure 13. The processor tree architecture (Figure 11) is well suited to executing the tree-splitting algorithm. Several constructs have been adapted from Fishburn's work [FISH81] for the algorithm presentation in Figure 13, as follows:

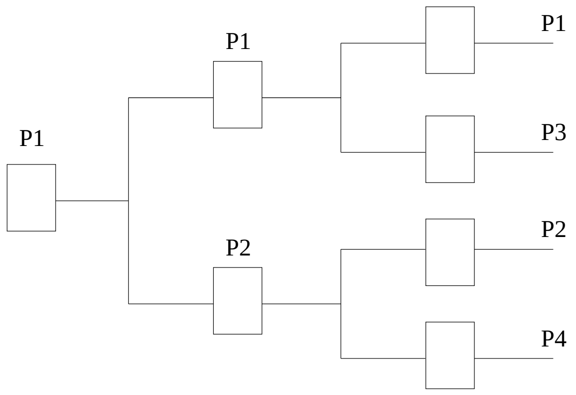

Figure 11. Example of processor tree method.

```
VAR alpha, beta : ARRAY [1..MAXDEPTH] OF integer;
   { alpha-beta bounds are stored in global tables }
procedure update(depth, side, bound : integer);
BEGIN
     IF (side > 0) THEN
          alpha[depth] := max(alpha[depth], bound);
     ELSE
          beta[depth] := min(beta[depth], bound);
     IF (depth > 0) THEN
          update(depth-1, -side, -bound);
END;
```

Figure 12. Dynamic update of alpha-beta values.

(1) j.treesplit is the recursive execution of 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 proceeding with the body of the statement.
(4) critical allows only one process at a time into a critical region.
(5) procedure terminate kills all processes in the parfor loop that are still active.

An important feature of parallel implementations is 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], the processors still spend a large amount of time computing without their benefit. Fortunately, the update method is relatively simple, as shown by the pseudo-code of Figure 12.

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

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

```
function treesplit(p : position; alpha, beta : integer) : integer;
     VAR width, i : integer;
         value : ARRAY [1..MAXWIDTH] OF integer;
         j : processor;
BEGIN
     IF (I am a leaf processor) THEN
          return(alphabeta(p, alpha, beta));

          width := generate(p); { determine successors }
                                {      p.1 .. p.w      }
          parfor i := 1 TO width DO BEGIN
               when (a slave j is idle) BEGIN
                    value[i] := -j.treesplit(p.i, -beta, -alpha);
                    critical BEGIN
                         IF (value[i] > alpha) THEN
                              alpha := value[i];
                    END;
                    IF (alpha >= beta) THEN BEGIN
                         terminate();
                         return(alpha);
                    END;
               END;
          END;
          return(alpha);
END;
```

Figure 13. The tree-splitting algorithm.

time can be filled by executing the slave algorithm for the next unassigned successor position, as is essentially the case for the architecture of Figure 11.

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

(3) The master can assign a 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 (1 and 2), or increased interprocessor communication is necessary (3).

4.4 Principal Variation Search

Algorithms can be designed for even more efficient search of strongly ordered trees. One such method operates on the Principal Variation as a refinement of the tree-splitting algorithm, hence the name *PV-splitting* [CAMP81]. This algorithm assumes an underlying hierarchical processor organization. Its regular configuration limits the complexity of interprocessor communication required, and simplifies the control structure for processor initiation and termination.

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 have been classified into one of three types [KNUT75]. Intuitively, type 1 nodes are those on the principal variation, and 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. For optimal search the following conditions hold:

(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 cut off with less than a full-width search. Maximum benefit from this cutoff is only possible, though, if the best alpha value is available. There is strong reason, therefore, to establish this alpha value before searching type 2 nodes. For this reason we have proposed pvsplit (Figure 14), which follows the principal variation for the number of ply specified by the length parameter, before invoking treesplit to bring all the processors into play on the largest part of the minimal tree that must be searched. A further enhancement is possible by having the master processors assign their slaves successors of successors. This ensures that type 2 nodes are always explored one branch at a time, in case a cutoff occurs. The concurrency is effectively applied at type 3 nodes, which will have to be searched full width in any case.

From a close examination of PV splitting, one can see how it draws on the minimal tree concept [AKL82], but two important differences can be noted. PV splitting assumes an underlying processor hierarchy structure. This contrasts with Akl's algorithm, which employs a pool of processors running a group of priority-ordered processes. Also, the processor tree architecture is conceptually clearer from an implementation point of view. A second point of difference comes about because PV splitting waits for the search value of left subtrees before initiating right subtree searches. This ensures that the best available alpha value is given to the right subtree searches, which is not necessarily the case in other algorithms. The cost for this is increased processor idle time.

The advantages of PV splitting over tree splitting are fairly obvious (assuming, of course, strongly ordered trees). In particular, the width of the processor tree can be much greater, since concurrency is only applied to type 3 positions. Also, much improved sharing of bounds is achieved at the cost of a moderate increase in communication overhead. On the other hand, PV splitting suffers from the restriction that the processor tree must be shallower than the tree being searched, particularly since processors are employed at alternating levels. The possibility for wider processor trees reduces this problem somewhat. Tree splitting and PV splitting have been compared by simulation. Results are given in Table 3a and b. All searches were carried out on trees of depth 4 and width 24. The length parameter to pvsplit was initially 1; thus the principal variation was followed for one ply before the other processors were activated. It was assumed that one time unit of overhead was needed to process a node, terminal or nonterminal, and that communication costs were negligible, relative to this interval.

These preliminary figures indicate that PV splitting, as expected, outperforms ordinary tree splitting. The wider the processor tree, the greater is the relative difference. 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.

```
function pvsplit(p : position; alpha, beta, length : integer) : integer;
     VAR width, i : integer;
         value : ARRAY [1..MAXWIDTH] OF integer;
         j : processor;
BEGIN
     IF (length <= 0) THEN
          return(treesplit(p, alpha, beta));
     width := generate(p); {  determine successors }
                           {      p.1 ..  p.w      }
     alpha := -pvsplit(p.1, -beta, -alpha, length-1);
     IF (alpha >= beta) THEN
          return(alpha);
     parfor i := 2 TO width DO BEGIN
          when (a slave j is idle) BEGIN
               value[i] := -j.treesplit(p.i, -beta, -alpha);
               critical BEGIN
                    IF (value[i] > alpha) THEN
                         alpha := value[i];
               END;
               IF (alpha >= beta) THEN BEGIN
                    terminate();
                    return(alpha);
               END;
          END;
     END;
     return(alpha);
END;
```

Figure 14. Parallel alpha-beta with processor tree architecture: the PV-splitting algorithm.

5. 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 for parallel searches of game trees.

Table 3. Comparison between Tree Splitting and PV Splitting for Various Processor Tree Configurations[a]

(L, F)	Tree splitting	PV splitting
(a) *Optimally ordered trees*		
(1, 2)	1222	961
(1, 4)	922	505
(1, 8)	772	277
(2, 2)	910	648
(3, 2)	778	—
(b) *Strongly ordered trees*		
(1, 2)	2700	2264
(1, 4)	2030	1425
(1, 8)	1859	1084
(2, 2)	1724	1587
(3, 2)	1172	—

[a]L = processor tree length; F = processor tree fan-out. Depth = 4; width = 24.

ACKNOWLEDGMENTS

Financial support for this study was provided by the Natural Sciences and Engineering Research Council of Canada. Our special thanks go to Visiting Professor Anne Brindle, editorial assistant Rachel Rutherford, and the reviewers for their many constructive comments, which helped reduce the potential ambiguities in this paper. Their efforts were much appreciated and are gratefully acknowledged.

REFERENCES

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

AKL82 Akl, S., Barnard, D., and Doran, R. "Design, analysis, and implementation of a parallel tree search algorithm." *IEEE Trans. Pattern Anal. Mach. Intell.* **PAMI-4,** 2 (1982), 192–203.

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

BIRD80 Bird, R. S. "Tabulation techniques for recursive programs." *ACM Comput. Surv.* **12,** 4 (Dec. 1980), 403–417.

CAMP81 Campbell, M. "Algorithms for the parallel search of game trees," M.Sc. thesis, Tech. Rep. 81-8, Computing Science Dep. Univ. of Alberta, Edmonton, Canada, Aug. 1981.

CAMP83 Campbell, M., and Marsland, T. A. "A comparison of minimax tree search algorithms." *Artif. Intell.* (to appear).

CICH73 Cichelli, R. J. "Research progress report in computer chess." *SIGART Newsl.* **41** (Aug. 1973), 32–36.

COND82 Condon, J. H., and Thompson, K. "Belle chess hardware." In M. R. B. Clarke (Ed.), *Advances in Computer Chess,* vol. 3. Pergamon Press, Elmsford, N. Y., 1982, pp. 45–54.

DEGR66 de Groot, A. D. *Thought and Choice in Chess.* Mouton, The Hague, 1965.

ENSL74 Enslow, P. *Multiprocessors and Parallel Processing.* Wiley, New York, 1974.

FISH80 Fishburn, J., and Finkel, R. "Parallel alpha-beta search on Arachne." Tech. Rep. 394, Computer Science Dep., Univ. of Wisconsin, Madison, Wis., July, 1980.

FISH81 Fishburn, J. "Analysis of speedup in distributed algorithms." Ph.D. dissertation, Tech. Rep. 431, Computer Science Dep., Univ. of Wisconsin, Madison, Wis., May, 1981.

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

GILL72 Gillogly, J. "The technology chess program." *Artif. Intell.* **3** (1972), 145–163.

GILL78 Gillogly, J. "Performance analysis of the technology chess program." Ph.D. dissertation, Computer Science Dep., Carnegie-Mellon Univ., Pittsburgh, Pa. March, 1978.

GREE67 Greenblatt, R. D., Eastlake, D. E. and Crocker, S. D. "The Greenblatt chess program." In *Fall J. Computer Conf. Proc.,* vol. 31, Thompson Books, Washington, D.C., 1967, pp. 801–810.

GRIF76 Griffith, A. K. "Empirical exploration of the performance of the alpha-beta tree-searching heuristic." *IEEE Trans. Comput.* **C-25,** 1 (1976), 6–11.

KNUT76 Knuth, D., and Moore, R. "An analysis of alpha-beta pruning." *Artif Intell.* **6** (1975), 293–326.

MARS74 Marsland, T. A., and Rushton, P. G. "A study of techniques for game-playing programs." In J. Rose (Ed.) *Advances in Cybernetics and Systems* vol.1. Gordon and Breach, London, 1971, pp.363–371.

MARS81 Marsland, T. A., and Campbell, M. "A survey of enhancements to the alpha-beta algorithm." In *ACM 81 National Conf. Proc.* (Los Angeles, Calif., Nov. 1981). ACM, New York, pp. 109–114.

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

NEWB79 Newborn, M. M. "Recent progress in computer chess." In M. C. Yovits (Ed.), *Advances in Computers,* vol. 18. Academic Press, New York, 1979, pp. 59–114.

NILS80 Nilsson, N. J. *Principles of Artificial Intelligence.* Tioga Publ., Palo Alto, Calif., 1980.

PEAR80 Pearl, J. "Asymptotic properties of minimax trees and game searching procedures." *Artif. Intell.* **14** (1980), 113–138.

SLAG69 Slagle, J. R., and Dixon, J. K. "Experiments with some programs that search game trees." *J. ACM* **16** , 2 (Apr.1969), 189–207.

SLAT77 Slate, D., and Atkin, L. “CHESS, 4.5—The Northwestern University chess program.” In P. Frey (Ed.), *Chess Skill in Man and Machine,* chap. 4. Springer Verlag, New York, 1977, pp. 82–118.

SORE78 Sorenson, P. G., Tremblay, J. P., and Deutscher, R. F. “Key- to-address transformation techniques.” *INFOR* **16** , 1 (1978), 1–34.

STOC79 Stockman, G. “A minimax algorithm better than alpha-beta?” *Artif. Intell.* **12** (1979), 179–196.

THOM81 Thompson, K. Private communications, Oct.–Nov. 1981.

THOM82 Thompson, K. “Computer chess strength.” In M. R. B. Clarke (Ed.), *Advances in Computer Chess,* vol. 3. Pergamon Press, Elmsford, N.Y., 1982, pp. 55–56.

TRUS81 Truscott, T. R. “Techniques used in minimax game-playing programs.” Master’s thesis, Computer Science Dep., Duke Univ., Durham, N. C., Apr. 1981.

WEIT80 Weitzman C. *Distributed Micro/Minicomputer Systems.* Prentice Hall, New York, 1980.

ZOBR70 Zobrist, A. L. “A hashing method with applications for game playing.” Tech. Rep. 88, Computer Science Dep., Univ. of Wisconsin, Madison, Wis., Apr. 1970.

Received August 1981; final revision accepted September 1982.