Parallel Game-Tree Search

T. A. MARSLAND, SENIOR MEMBER, IEEE, AND FRED POPOWICH

Abstract–**The design issues affecting a parallel implementation of the alpha-beta search algorithm are discussed with emphasis on a tree decomposition scheme that is intended for use on well-ordered trees. In particular, the principal variation splitting method has been implemented, and experimental results are presented which show how such refinements as progressive deepening, narrow window searching, and the use of memory tables affect the performance of multiprocessor-based chess playing programs. When dealing with parallel processing systems, communication delays are perhaps the greatest source of lost time. Therefore, an implementation of our tree decomposition-based algorithm is presented, one that operates with a modest amount of message passing within a network of processors. Since our system has low search overhead, the principal basis for comparison is the communication overhead, which in turn is shown to have two components.**

IndexTerms–**Alpha-beta search, chess programs, concurrent programming, graph and tree search strategies, message passing multiprocessors, tree decomposition.**

I. INTRODUCTION

WHEN sequential versions of the alpha-beta tree searching algorithm are adapted for use on a parallel processing system, the speedup (reduction in search time) is notoriously less than the number of processors in the system. These losses are caused by *search overhead* and *communication overhead.* The search overhead is algorithm-dependent, and measures the extra work that a parallel algorithm does compared to its sequential counterpart. Since an alpha-beta search uses accumulated information to determine when cutoffs are to occur, a parallel implementation may have one processor still calculating the better cutoff value that another could use to end its execution. Thus the total work done, and hence the search overhead, may be higher. In contrast, communication overhead results from the necessary exchange of information between processors. These delays are of two fundamentally different types: waiting for an enquiry response and waiting for other processors to finish their work, and are therefore dependent on the system configuration as well as the algorithms used. Thus, parallel alpha-beta is potentially susceptible to search overhead losses, and compensating for this overhead may entail major communication delays.

One sequential alpha-beta algorithm, the principal variation search (PVS) [1], has been shown to be effective on well-ordered (nonrandom) game trees [2]. The essence of the PVS method is to assume that the leftmost path examined is the best, and to use the value of that path to form tight bounds on the search of the remaining subtrees in a quick attempt to prove their inferiority. If one of these subtrees proves superior, it must be searched again to determine its true value. In such a case, much of the previous preliminary search may be wasted. PVS is efficient on well-ordered trees because the search time saved in determining subtrees to be inferior exceeds the cost of the discarded search when a superior subtree is identified. Since a

Manuscript received February 19, 1984; revised November 5, 1984. This work was supported in part by the Canadian National Science and Engineering Research Council under Grants A5556 and A7902.

The authors are vith the Department of Computer Science, University of Alberta, Edmonton, Alta., Canada.

good ordering of branches is usually possible for typical applications (e.g., computer chess) we have devised a parallel version, dubbed principal variation splitting (PVsplit), which also takes advantage of this partial ordering property in determining how to allocate processors to the search.

Research has not yet determined the best way of using N processors to search a game tree of indefinite size. When many processors ($N > 1000$) are available, it is possible that the most effective organization will be totally different from the one that works best for only a few processors ($N < 10$). Several groups have considered the division of work problem for small values of N. and proposals have been made based on theoretical, experimental, and simulation results.

The approaches to sharing the work differ, and they include partitioning the search window between processors [3], assigning individual processors to separate subtrees [4], and managing a pool of processors [5], [6] . Each has its advantages and disadvantages. For example, by dividing the search window into disjoint ranges and assigning one processor to each partition, one can guarantee that the best continuation can be found, and in no more time than a single processor searching over the full window. In this so-called parallel aspiration search of Baudet's, N processors are used to search N disjoint alpha-beta windows at one time. This approach is especially attractive when applied to random game trees, but Baudet's analysis showed that even with many processors the speedup is limited to a factor of 5–6 [3]. That analysis was done under the optimistic assumption that all the processors complete their work at the same time, and so all communication overhead issues were avoided. Speedup is limited because each processor must search the minimal game tree. Of course on perfectly ordered trees (minimal trees) the parallel aspiration method yields no speedup, and so the method has little potential for success in applications where good branch ordering is possible.

The direct tree decomposition approach (applying one processor per subtree) [4] suffers from a similar minimal tree search disadvantage, although the amount of work done per processor may be smaller. More seriously, it is difficult to share information between processors, yet this is important for best performance. On the other hand, this method has the potential for arbitrary speedup, at the expense of increased search overhead, providing the tree to be searched is big enough. Finally, some methods that hold promise in terms of efficiency and information sharing properties use a pool of processors [5], but here a large memory is needed to store partially evaluated nodes as they await an available processor. Aside from the storage requirement, these methods are difficult to analyze in terms of their communication overhead [6].

Our approach is a tree decomposition scheme in which all the processors are initially applied to the search and decomposition of a candidate principal variation. A loosely coupled network of processors is used to implement our PV splitting algorithm. Our interest is with the application of a few processors (up to a dozen) with a view exceeding the known speedup limit of the parallel window partitioning method [3]. To regulate message flow, a processor tree architecture is used [7].

II. SEQUENTIAL SEARCH ALGORITHM

With PVS, alternatives to the first move are assumed inferior until proven otherwise. A preliminary examination of these alternatives is made with a zero-width (minimal) window of (alpha, alpha+1), based on a bound, alpha, obtained from the search of the best candidate so far. Any move (subtree) searched with such a zero window will necessarily fail. Most of these searches will "fail low" because the move is easily refuted. Whenever a better move is found, the current search "fails high" and it must be repeated with more appropriate bounds to determine the correct value for the new candidate. PVS assumes that game trees are rarely random, and that any knowledge about the application domain can be used to pre-order the move list and, hence, favorably bias the shape of the game tree.

An alpha-beta algorithm suitable for doing this search, since it can return values below the range of the

```
FUNCTION pvs(p : position; aipha,beta,depth : integer) : integer;
                                    { p is pointer to the current node }
                                    { alpha and beta are window bounds }
                                { depth is the remaining search length }
                                { the value of the subtree is returned }
VAR  score, i, value : integer;
     posn : ARRAY[1..MAXWIDTH] OF position;
                                               { assert depth positive }
BEGIN
     IF depth = 0 THEN                        { leaf, maximum depth?   }
          return (evaluate (p));  { produce a pointer to   }
     posn := generate (p);                    { an array of successors }
     IF empty(posn) THEN                      { leaf, no moves? }
          return (evaluate (p));
                                              { principal variation? }
     score := -pvs(posn[l], -beta, -alpha, depth-1);
     FOR i := 2 TO sizeof(posn) DO BEGIN
          IF (score >= beta) THEN             { cutoff? }
               return (score);
          alpha := max(score, alpha);
                                            { zero-width window search }
          value := -pvs (posn[i], -alpha-1, -alpha, depth-1);
          IF (value > alpha) THEN
               IF (value < beta) THEN      { if "fail-high," re-search }
                    score := -pvs(posn[i], -beta, -value, depth-1);
               ELSE score := value;
     END {forloop};
     return (score);
END {pvs};
```

Fig. 1. Depth-limited principal variation search.

given window [8], is shown in Fig. 1. It is called *pvs* and uses application-dependent functions *generate,* to form an array of successors to the given position; *empty,* to determine if no successors exist; *sizeof,* to return the size of the array; and *evaluate,* to estimate the value of the subtrees at a leaf (horizon node).[1] These functions are not presented, but in our case study they form part of a chess playing program. For simplicity some other details have been omitted, particularly use of *make* and *undo,* to update and restore the given move respectively. The mode of presentation of the programs throughout this paper is a Pascal-like pseudocode, extended with a *return* statement for function exit, although our implementations are done in the *C* language [9]. The algorithm ensures that the best available score is returned, even when all the moves

[1] Strictly speaking, a leaf is a terminal node (one without successors) but hereafter we use it synonymously with horizon node, that is, any node at the maximum search depth of the tree. An evaluation function is used to estimate the value of a leaf. These functions are expensive and are designed to do a quiescence search; that is, a limited search over a subset of the available moves (e.g., checks and captures in chess), to limit the error in the estimated value of the discarded subtrees that extend beyond the horizon.

have a value that is less than the alpha bound. In many respects, *pvs* is similar to SCOUT [10], [11], and the Calphabeta algorithm [8], from which the notion of a zero window search is drawn.

There are several enhancements to alpha-beta implementations that improve their performance dramatically. Perhaps the most important is *progressive* or *iterative deepening* [12]. Rather than embarking immediately upon a maximum depth search, a series of successively deeper searches is used. The moves are sorted after each iteration, thus providing a more plausibly ordered move list for the next search phase, which in turn tends to result in a quicker search. This method is only effective when used with a *refutation table* [2] or a *transposition table* [1]. For example, after each D-ply search (that is, to depth D) sequences of moves from the root to leaf nodes are stored in a *refutation table.* For the selected move this table contains the best variation, while for the others it holds a D-ply sequence sufficient to refute the move. Upon a $D+1$ ply search, the table is used to direct each variation along a potential refutation and thus significantly reduce the search time. This knowledge table is of modest size, being linear with search depth, and is cheap to maintain. One can also implement a more comprehensive table, one which holds not only the preferred move in a position, but also bounds on the value of the associated subtree. This *transposition table* (a direct access memory table of results for subtrees already examined) is more powerful since it can also be used to improve search windows, provide more cutoffs, and extend the effective search depth [13]. Progressive deepening is an important idea in game-tree searches because most of the time is spent exploring the principal variation. Any enhancement which increases the probability that the PV will be examined early in the search will improve performance, since the balance of the tree is discarded more quickly.

III. PRINCIPAL VARIATION SPLITTING

The PV splitting algorithm is based on PVS and, unlike other methods, delays tree decomposition until the first path along the principal variation has been searched. Thereafter, individual subtrees are examined by a hierarchy of processors arranged in a tree. A processor tree consists of computers and communication lines corresponding to tree nodes and branches, respectively (see Fig. 2). Any processor can communicate directly with only its parent and its children. The root processor is designated the master, and communicates with the user. There is also a master/slave relationship between the parent and its children but, by supplying appropriate software, one can ensure that an individual processor can behave as both parent and child (cf. processor $P1$ in Fig. 2).

When direct tree decomposition is used for parallel implementations of alpha-beta algorithms, some subtree cutoffs may not occur, since more moves will be examined without the benefit of the best bound [3]. This becomes more evident as processor tree fanout increases. In our PV splitting algorithm, on the other hand, the most plausible move is analyzed by all processors. Thus, not only is the first variation searched faster, but also the subsequent decomposition ensures that all the processors at a given level start with a good window bound. This provides more cutoffs and allows the balance of the search to proceed more quickly. Consequently the search overhead is reduced.

Fig. 3 illustrates a version of the PV splitting algorithm which uses the following constructs adapted from Fishburn [8].

1) *j.treesplit* represents the execution of a direct tree-partitioning algorithm by processor "*j*." *Treesplit* may also use the processor tree architecture, but is not presented here since it is adequately described elsewhere [1].
2) PARFOR initiates a parallel loop which conceptually creates a separate process for each iteration of the loop. The program continues as a single process when all loops are complete.

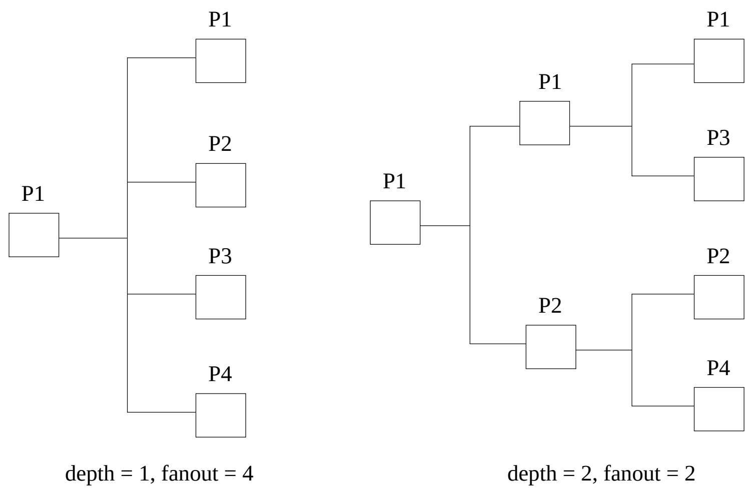

Fig. 2. Processor tree structure.

3) WHEN causes the parent to wait until the associated condition is true before proceeding with the body of the statement.
4) CRITICAL allows only one process at a time into the next block of code.
5) *terminate* kills all currently active processes in the PARFOR loop.

For efficiency, we have assumed in Fig. 3 that the processor tree length is less than the specified depth of the game tree search. *Pvsplit* uses the processor tree in the following way. First, the master processor traverses the principal variation until the length of the processor tree is equal to the remaining depth to be searched in the game tree, then the processors work on the nodes of the game tree that they cover. Once this is complete, the alternatives at the node where the master processor currently resides are divided amongst the children of the master. Note that the master is never idle, since it spawns itself as a child. Once search of this node is complete, the processor tree backs up one level and the next set of alternatives are again divided amongst the first layer of child processors. This backing up and splitting operation continues until the whole tree has been searched.

For the results presented in this paper, our implementation of PV splitting uses a processor tree of length one. Consequently, the *pvsplit* routine may be simplified into one called *pmws,* while *j.treesplit* is replaced by a direct call to *pvs,* (see Fig. 4). It is this simplification which is used in our analytical model and performance studies which follow. *Pmws* stands for parallel minimal window search, since the alternate variations are searched in parallel with a zero-width window. Although some of the details about move updates have been omitted from the *pmws* function, we have managed to illustrate progressive deepening with function *main.* The *transmit.j* and *receive.j* functions are used to exchange information between processor "*j*" and its parent, while *j.child–pvs* represents the execution of the *child–pvs* procedure by processor "*j*." The major disadvantage of this method is that only a single processor is available to search any new principal variation which subsequently emerges, since no tree splitting occurs at that time. To overcome this restriction one would have to apply all the processors to the re-search of the new candidate, just as they were for the primary principal variation, but this could lead to further synchronization delays. Finally, in the interest of simplicity we have indicated by comments the locations where communication is necessary to update the

```
FUNCTION pvsplit(p : position;
                 alpha, beta, depth, length : integer) : integer;
                                { length is the processor tree height }
                                          { assertion length <= depth }
VAR  t, value, score, height : integer;
     j : processor;
     posn : ARRAY[1..MAXWIDTH] OF position;
BEGIN
     IF length = 0 THEN                        { end of processor tree }
          return(pvs(p, alpha, beta, depth));
     posn := generate (p);{ produce a pointer to an array of successors }
     IF empty(posn) THEN                               { no legal move }
          return(evaluate(p));
     height := length;
     IF height >= depth THEN                    { apply processor tree }
          height := length - 1;
     score := -pvsplit(posn[1], -beta, -alpha, depth-1, height);
     PARFOR i := 2 TO sizeof(posn) DO        { loop through successors }
          WHEN (a child processor j is idle) BEGIN
               IF score >= beta THEN BEGIN                  { cutoff? }
                    terminate();
                    return (score);
               END;
               CRITICAL alpha := max(alpha, score);
                                          { keep processor tree rooted }
               value := -j.treesplit(posn[i],-beta,-alpha,depth-1,length-1);
               CRITICAL score := max(score, value);
          END;
     return (score);
END;
```

Fig. 3. General form of principal variation splitting.

best path to a leaf (see Fig. 4). These communications are far less frequent than those illustrated by the presence of transmit/receive invocations.

```
FUNCTION main(p :  position;  maxdepth : integer) : integer;
   VAR score, i, j, value, depth : integer;
       posn : ARRAY[1..MAXWIDTH] OF position;
BEGIN
   posn := generate(p);
   IF empty(posn) THEN return (evaluate (p));
   for depth := 1 to maxdepth DO BEGIN              { progressive deepening }
        score := -pmvs(posn[i], depth-1);
        PARFOR i := 2 TO sizeof(posn) DO
              WHEN (child processor "j" is idle) DO BEGIN
                  transmit.j(posn[i], score, depth); { send best score to "j"}
                  j.child_pvs                     { invoke child_pvs on "j" }
                  receive.j(posn[i].value)           { accept subtree value }
                  CRITICAL score := max(score, posn[i].value)
              END;
        sort(posn);       { communication of refutation table update omitted }
   END;
   return(score);
END;


FUNCTION pmws(p : position; depth : integer) : integer;
                                           { note processor tree length is i }
   VAR score, i, j, value : integer;
       posn ARRAY[1..MAXWIDTH] OF position;
BEGIN
   IF depth = 0 THEN return(evaluate(p));
   posn := generate(p);
   IF empty(posn) THEN return (evaluate (p));
   score := -pmws(posn[i], depth-1);
   PARFOR i := 2 TO sizeof(posn) DO
      WHEN (child processor "j" is idle) DO BEGIN
         transmit.j(posn[i], score, depth);                      { send to j }
         j.child_pvs;                     { processor "j" executes child_pvs }
         receive.j(posn[i].value);
         CRITICAL score := max(score, posn[i] value);
                               { communication of better move update omitted }
   END;
   return (score);
END;


PROCEDURE child_pvs;
   VAR value, score, depth : integer; p : position;
BEGIN
   receive.m(p, score, depth)                          { receive from master }
   value := -pvs(p, -score-1, -score, depth-1);
   IF value > score THEN                     { re-search with correct window }
      value := -pvs(p, -MAXINT, -value, depth-1);
   transmit.m(value);                                     { send to master }
END;
```

Fig. 4. Parallel minimal window search.

IV. SOFTWARE SYSTEM

The underlying strategy in both our sequential and parallel algorithms for game tree search is PVS. This method presumes a good ordering of the moves so that the most likely candidate is searched first, while the remaining variations are examined using a zero-width window. Progressive deepening, an iterative search with dynamic reordering of the moves, along with a refutation table, containing the main continuation for each move at the first level in the tree, are used since they have shown their merit in previous tests [2]. A transposition table, containing the results from nodes seen during the search, is also included so that its effect on the parallel performance may be studied.

To examine overheads and other problems (such as memory table management) associated with a parallel searching algorithm, we have designed *Parabelle,* a chess program which is based on *TinkerBelle.*[2] The results from *Parabelle* are compared to the best available uniprocessor version of that program. *Parabelle* analyzes chess positions on a processor tree network. All the processors recursively analyze the first move, with the remainder being examined by individual processors using a zero-width window. There is a local refutation table for each processor that is updated after each phase of the progressive deepening search. A transposition table, which can be accessed on either a global or local basis, is also included. When a global transposition table is used, then all processors have access to the same subtree results which are stored in the master processor. Any benefits of this arrangement may be cancelled by the increase in the communication overhead of the system. On the other hand, if a local transposition table for each processor is used, then this communication is not necessary, but the table will contain fewer entries since it will not have results from positions seen by other processors.

V. HARDWARE CONFIGURATION

Parabelle is written in *C* for use on a Motorola 68000-based multiple instruction stream multiple data stream (MIMD) system [14]. The program itself requires about 48K bytes of space, while the rest of each unit's 256K bytes of memory may be used for data storage. Each unit contains identical software for searching chess positions, with a master possessing an additional 128K bytes of memory to hold extra code for global table management and communication interface routines. The system used for our experiments with the *pmws* version of *pvsplit* consists of four identical SUN workstation boards [15], in a depth one processor tree configuration (see Fig. 5), but can be extended to handle nine processors. With a system of four processors, preliminary simulations of PV splitting have shown that a processor tree of depth one is slightly superior to one of depth two [1].[3]

Communication between the processors is channeled through an eight-port serial communication interface (SCI) [16] at a maximum rate of 960 bytes/s. This device currently resides in the master workstation where direct memory data transfer can occur between it and the SUN processor. Communication between the SCI and the other units is done over RS-232 data lines. Additional SCI units could be installed to provide processor trees of various depths. Since the SCI takes care of the character by character transmission of data, the host (master) processor need only write to, and read from, the appropriate buffers to communicate with the individual processors. The SCI can also interrupt the host whenever data are available and, consequently allows the host to act as a parent processor when it is handling an interrupt, or as a child when it is in normal operating mode.

[2] A chess program, developed by K. Thompson (Bell Laboratories, Murray Hill, NJ), which participated at the U.S. Computer Chess Championship, ACM National Conference, San Diego, CA, 1975.

[3] That comparison was based on the expected number of leaf nodes visited.

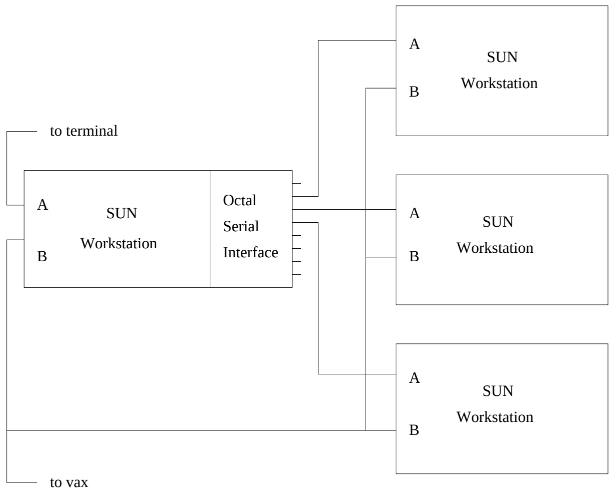

Fig. 5. System configuration.

While the master processor has its input buffered by the SCI, the bare SUN workstation can only buffer two characters. As a result, the master must not be allowed to transmit to the children unless it can be sure that no data will be lost. One way that this problem may be handled is by having the child transmit the request-to-send (RTS) signal when it desires input, so that the SCI will not transmit unless it receives this signal (clear-to-send). Fortunately, this extra protocol is not necessary for the bulk of our messages, since they are of the form "read immediately after write" (with respect to the child), and so blocks of information from the master can always be accepted.

VI. INTERUNIT COMMUNICATION

When a deterministic game tree is searched, it is possible to take advantage of the fact that each processor can generate identical move lists for the current position. As a consequence, they can all search the principal variation recursively, and so explore the same first path (sequence of moves to a leaf). For the remaining moves, the master informs an idle child of the best score, and identifies the next subtree to search. In turn, the child tells the master what value it found for the subtree it was searching. This exchange is fundamental to PV splitting and is referred to as *internode* communication. It is brief, four bytes, and occurs only during searches at nodes along the first variation, marked with a PV in Fig. 6 and named type 1 nodes by Knuth [17]. The remaining nodes in a game tree are either ALL nodes (every successor branch must be searched) or CUT nodes (no more branches are searched than are necessary to produce a cutoff). Fig. 6 represents a minimal alpha-beta tree and the cutoff subtrees are marked with an x.

A few other interunit communications occur. In particular, an *interlevel* message is sent to the master after search of a PV node has been completed. This message is required to pass back the best refutation sequence and its value. The length of the message depends only on the length of the refutation line transferred, that is, the depth of the subtree searched, but the message is only sent by those slaves which actually find a better line.

When using the progressive deepening refinement, *inter-iteration* communication at the root node is also

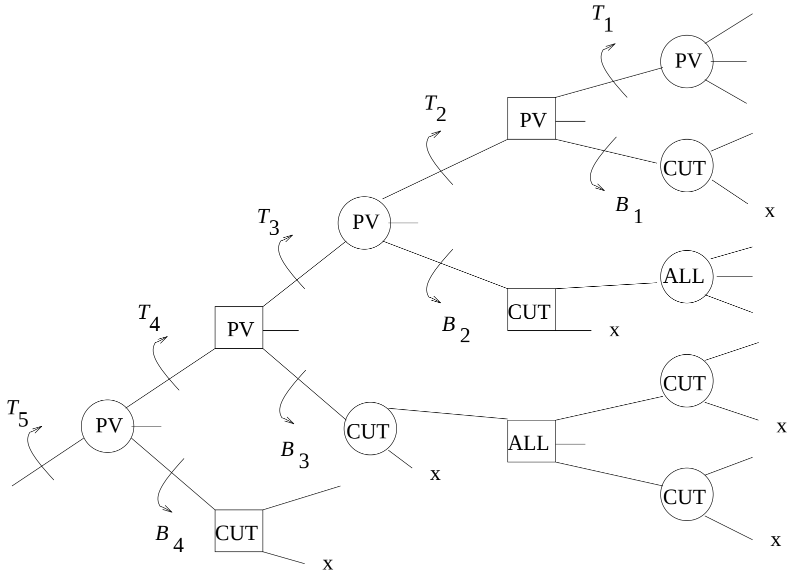

Fig. 6. Properties of an alpha-beta search tree.

required. Between iterations, the move list is first sorted by the master and then passed on to its slaves. The master refutation table is updated and similarly transferred to the children. A large amount of information transfer, proportional to the width of the search tree, must take place at this point, but this communication delay can be tolerated since only a handful of iterations are usually performed. This transfer is necessary to ensure that all the processors again have identical information about the move order and acquired memory (e.g., refutation table). This transfer can be avoided if processors are restricted to searching the same subset of the initial moves considered at the previous iterations, but should be done in order that the new work can be shared equally.

If a transposition table is used, all the processors must be able to search and update this large direct-access hash table. In one of our experiments, the transposition table was managed by the master processor. The children access this global table via *at-node* communication. Whenever a child processor is about to execute *pvs* on a node within its assigned search tree, a 6-byte request message (consisting of a hash access key) is sent to the master to see if a table entry exists for that node. If the access is successful, the master passes back 6 bytes consisting of the move found and its score, along with a measure of the score's reliability [1]. With this information, the child can narrow its search window or even avoid searching the subtree altogether. After completing its search, the child transmits to the master a 12-byte update message consisting of the two information packets just described. Unlike the other forms of communication, *at-node* messages can be frequent.

Finally, there are *interactive* messages, to allow the startup of the individual processors, the setting of system configuration variables, and other forms of external communication. Input to the master processor is echoed to all the children. *Interactive* communication involves tasks such as updating the board configuration, obtaining the opponent's move, and even setting the maximum search depth. Typically, these are messages to the user's console. They do not affect the performance of the system.

TABLE I
TYPES OF INTERPROCESSOR COMMUNICATION

Type	Frequency	Length
at-node	before and after each node in the tree, to probe global transposition table (optional)	6 or 12 bytes
inter-node	before and after each alternative to the first path from a PV node, to indicate next move	4 bytes
inter-level	after each PV node, to retain best refutation line	2 bytes for each ply from leaf
inter-iteration	before and after each progressive deepening phase, to synchronize refutation tables	4w + 2iw bytes. w is tree width at the root, i the iteration number
interactive	occur in response to user requests	varies

The different types of communication, along with their length, are summarized in decreasing order of frequency in Table I. The *at-node* communication is optional in the sense that use of a global transposition table is not mandatory. In fact, since the child may be idle when it is waiting for a response to its request, it may be prudent to suppress this communication and perhaps implement a transposition table local to each individual processor. Certainly our experimental results, presented later, show this to be the case. Another possibility would be for the master to interrupt the child if and only if the position is found in the transposition table. Thus, the child would not have to wait for a negative response and so could proceed immediately with its search, discarding its work only when a successful retrieval occurs.

VII. ANALYTICAL MODEL

It is difficult to develop a mathematical model which can be used to estimate the time to search a multibranch tree, especially if several processors are to be used. Since the size of the minimax search of a game tree is only known in some statistical sense, it is customary to use a uniform (fixed number of branches at each node) game tree of specified depth as a model. Since a uniform tree is regular and well-defined, it is possible to count the nodes in the minimal tree and, hence, to derive a formula for the search time for a designated processor configuration and search strategy. Any speedup factor which may be computed based on the search of a minimal game tree should also be a good estimator of the speedup possible for a progressive deepening alpha-beta search using f processors.

For the purposes of this analysis we consider the search of a uniform game tree of width w and depth d, in which $(1 + g)$ branches are examined at the cutoff nodes. The searching strategy employed in principal variation splitting with a processor tree of length 1 and fanout f. Thus, all f processors traverse the first branch at each PV node (see Fig. 6), and then the balance of the branches are split equally among the processors.

In order to simplify presentation, the following notation is used.

d Depth of search.
e Average time to evaluate a single move at a leaf (horizon) node.
f Number of processors (fanout).
g Average number of branches searched at cutoff node.
k Average time to create a node and generate the move list.
s Setup time in handling a message.
w Number of branches (moves) at any node (position).
t_1 Time to transmit an internode message of length b_1 bytes.
t_2 Time to transmit an interlevel message of length b_2 bytes.
B_d Time to do a zero window search to depth d.
$E_{d,g}$ Time to evaluate the leaf nodes of a uniform tree.
$I_{d,f}$ Idle time for f processors using *pvsplit.*
L_d Time to evaluate leaf nodes during a zero window search.
N_d Number of leaf nodes in a minimal tree.
T_d Time to search a minimal tree to depth d.

Note that both t_1 and t_2 are proportional not only to the communication link speed but also to the message length.

It is well known that for an optimal (minimal) alpha-beta tree the number of leaf nodes in a tree of depth d and width w is given by the equation

$$N_d = w^{\lceil d/2 \rceil} + w^{\lfloor d/2 \rfloor} - 1 \tag{1}$$

where $\lceil x \rceil$ and $\lfloor x \rfloor$ represent the first integer not less than and not greater than x, respectively. Thus, the number of leaf nodes depends on whether d is even or odd. The leaf node expression is too simple for our purpose since it does not recognize that these leaf nodes are of two types: ALL nodes at which every move is considered, and CUT (cutoff) nodes at which only one move is evaluated. Furthermore, although the dominant computations are done at the leaf nodes, the cost of an interior node is not negligible.

From Fig. 6 a recurrence relationship to estimate T_d, the number of processor cycles required to search a minimal alpha-beta tree to depth d, may be derived as follows:

$$T_d = k + T_{d-1} + (w-1)B_{d-1} \quad \text{cycles} \tag{2}$$

where

$$T_0 = e$$

and

$$B_d = 2k + wB_{d-2} \quad \text{for} \quad d > 1$$

where

$$B_1 = k + e \quad \text{and} \quad B_0 = e.$$

We have assumed here that at the leaf nodes a quiescence search (a search limited to disruptive moves like checks and captures in chess) is carried out, and that the cost of evaluating a leaf node depends on whether it is an ALL or a CUT type. After some substitutions, (2) becomes

$$T_d = dk + e + (w-1)\sum_{i=0}^{d-1} B_i \quad \text{cycles.} \tag{3}$$

While (2) and (3) may be used to estimate the time to traverse a minimal tree by a uniprocessor, they account neither for multiprocessors using tree splitting nor for interunit communication. When a processor tree with fan-out f and length 1 executes *pvsplit* the search time for a minimal tree is

$$T_{d,f} = k + T_{d-1,f} + \left\lceil \frac{(w-1)}{f} \right\rceil B_{d-1} \quad \text{cycles for} \quad d > 0 \quad \text{and} \quad f > 0, \tag{4}$$

where $T_{0.f} = e$ cycles. Finally, when *internode* and *interlevel* communications are included,

$$T_{d,f} = k + T_{d-1,f} + \left\lceil \frac{(w-1)}{f} \right\rceil B_{d-1} + (w-1)\frac{(f-1)}{f}(s+t_1) + (f-1)(s+t_2) \quad \text{cycles,} \tag{5}$$

for $d > 0$ and $f > 0$. Equation (5) shows that provided *at-node* communication is excluded (i.e., global transposition tables are not used), the cost of message passing is modest, $O(w)$ compared to $O(w^d)$ for the search itself.

For a variety of reasons the time taken to search a game tree is dominated by the time to evaluate the leaf nodes. This is primarily because e, the estimated value of each subtree that extends beyond the horizon, must be computed. While we use k to denote the cost of an interior node and $k + we$ for the cost of an ALL leaf, the ratio e/k may be highly variable (although $e >> k$). When the alpha-beta algorithm is enhanced with move ordering refinements, the game trees searched are close to minimal [2]. Let us assume that these trees may be modelled by a uniform tree of width w, with an average branching factor of $(1+g)$ at the CUT nodes. Again from Fig. 6 a pair of recurrence relations to measure $E_{d,g}$, the time taken to evaluate the leaf nodes of a uniform tree, is given by

$$E_{2m+1,g} = E_{2m,g} + (w-1)(1+g)^m w^{m-1}. \quad \text{ALL cycles} \tag{6}$$

where ALL represents the cost of a fully evaluated node, so it is equal to we, and

$$E_{2m,g} = E_{(2m-1),g} + (w-1)(1+g)^{m-1} w^{m-1}. \quad \text{CUT cycles,} \tag{7}$$

where the cost of a CUT node is $(1+g)e$, and where

$$E_{1,g} = PV = we \quad \text{cycles.}$$

After some manipulation we obtain

$$E_{2m+1,g} = (w-1)e\left[\sum_{i=0}^{m}(1+g)^i w^i + \sum_{i=0}^{m-1}(1+g)^{i+1} w^i - 1\right] + we \quad \text{cycles.} \tag{8}$$

While it is possible to massage this equation further it is not necessary to do so. To check on the validity of (8) we observe that, when $g = 0$, $E_{2m+1,0}$ should be proportional to the node count value for the minimal tree. For example, for odd depths

$$
\begin{aligned}
E_{2m+1,0} &= (w-1)e\left[\frac{w^{m+1}-1+w^m-1}{w-1}-1\right]+we \\
&= (w^{m+1}+w^m-1)e = eN_{2m+1}
\end{aligned}
$$

as expected. An equivalent result can be obtained for even depths.

A. Overheads

It is possible to develop simple formulas for search, time, and communication overheads. Search overhead has already been defined as the extra work done by f processors over that done by a single processor. Thus

$$
\text{search overhead} \;=\; \frac{\text{number of leaf nodes visited by } f \text{ processors}}{\text{number of leaf nodes visited by uniprocessor}} - 1 \tag{9}
$$

For our results in the next section, this form of the search overhead was used. One might argue that a more appropriate measure is the leaf time overhead, given by

$$
\text{leaf time overhead} \;=\; \frac{\Sigma \text{ of time spent at leaf nodes by } f \text{ processors}}{\text{time spent at leaf nodes by a uniprocessor}} - 1 \tag{10}
$$

Equation (9) is the preferred measure in simulation studies [5], [1], and while it may differ from (10), we believe that it is still an adequate approximation. Similarly, the time or total overhead can be measured accurately as

$$
\text{time overhead} \;=\; \frac{f \times \text{ time taken by } f \text{ processors}}{\text{time taken by a uniprocessor}} - 1 \tag{11}
$$

and in turn may be approximated by the comparable expression for the search of a minimal game tree, i.e.,

$$
\text{time overhead} \;=\; \frac{f \times T_{d,f}}{T_{d,1}} - 1 \tag{12}
$$

where $T_{d,f}$ is given by (4) or (5). Communication overhead, on the other hand, is algorithm- and system-dependent. It encompasses the remaining losses and may be adequately expressed as

$$
\text{communication overhead} \;=\; \text{time overhead} \;-\; \text{search overhead}.
$$

Communication overhead has two distinct components: message passing costs and scheduling costs. The former reflects the time spent updating the master with the results of a subtree search, while the latter occurs whenever a processor is awaiting its work assignment. This scheduling cost has been termed synchronization overhead [18], but was not measured in our system. However, we are able to estimate this component as follows. While $T_{d,f}$ measures the time for f processors to search a minimal game tree, and may include message passing time, it does not account for the reduction in effective speed as processors become idle when no subtrees remain to be searched. Under *pvsplit* these synchronization losses occur only at PV nodes. They arise not only because the time to search a subtree is highly variable, but also because of the imbalance between the node width w and the number of processors f. For example, at a PV node all the processors traverse the first path and then split the remaining $(w-1)$ subtrees. As the search completes, from 1 to $(f-1)$ processors are idle, i.e., at each PV node $(f/2)B_d$ cycles are lost. These final subtrees are searched with a zero window, which for all practical purposes means that $g = 0$ at each CUT node. Again, if the cost of these subtrees is dominated by the cost of evaluating the leaf nodes, the idle time per PV node is approximated by $(f/2)L_d$, where L_d is given by

$$\begin{aligned} L_{2m} &= w^{m-1} \bullet \text{ ALL } = \text{ w}^{\text{m}}\text{e cycles} \\ L_{2m+1} &= w^{m} \bullet \text{ CUT } = \text{ w}^{\text{m}}\text{e cycles,} \end{aligned}$$

that is

$$L_d = w^{\lceil d/2 \rceil} e \quad \text{cycles.} \tag{13}$$

For the purposes of this study we shall define synchronization overhead as

$$\begin{aligned} \text{synchronization overhead} &= \frac{\text{multiprocessor idle time}}{\text{uniprocessor leaf search time}} \\ &= \frac{I_{d,f}}{E_{d,g}}. \end{aligned}$$

Under progressive deepening, d iterations are performed, each to successively deeper depth. Only the last two iterations produce significant losses, so the idle time is most conveniently expressed in the form

$$I_{d,f} = \frac{f}{2}\left[\sum_{i=0}^{d-1} L_i + \sum_{i=0}^{d-2} L_i\right] + I_{d-2,f} \quad \text{cycles.}$$

Also, since even in the uniprocessor case the final subtrees are searched with a zero window, we shall use $E_{d,0}$ as an estimate of $E_{d,g}$. This is reasonable since in our experiments g had an average value of about 0.5. Finally, restricting our interest to odd-ply searches and ignoring all but the powers of w^m in $I_{2m+1,f}$, and $E_{2m+1,0}$ we get

$$\begin{aligned} \text{synchronization overhead} &\approx \frac{f w^m e}{2(w^{m+1} + w^m - 1)e} \\ &\approx \frac{f}{2(w+1)}. \end{aligned} \tag{14}$$

For an even-ply search the synchronization overhead is slightly different, $3f/4(w-1)$, but in both cases they are of order $O(f/w)$. Even so, it should be remembered that this analysis takes no account of the statistical nature of game trees searched under an alpha-beta window. In practice, we observe that for long periods of time $f - 1$ processors are idle, so the cost will be higher. Since most of the synchronization losses occur at the root node, and they increase with increasing processor fanout (f), we might expect that only processor trees with narrow fanout (e.g., 2) will be effective in reducing these losses. Another possibility is to treat the root node of the game tree as a special case (as is normally done anyway), and develop ways of deploying the idle processors to assist the others that are still working.

VIII. PERFORMANCE

The test results, as based on *Parabelle's* performance on a standard of 24 chess positions [19], show that PV splitting has low search overhead. For comparison we present the data from a series of five-ply searches, with and without transposition tables, performed by a system consisting of from one to four processors. This search depth limit was chosen because longer trees may cause overloading of the transposition table, thus obscuring the results.

By contemporary standards, the program for our application is weak at computer chess. Our performance results are not designed to show how well the program plays chess, but rather to assess relative performance of the various components of the system. Also, the chess program we are using is slow. This is partly because it is written wholly in a portable version of *C*, and also because we preferred extensibility of our implementation to speed. Our aim was to explore an efficient way of employing many processors in the search of game trees. We assume that any efficiency improvements in the application program itself will be reflected equally in the various algorithms.

A. Without Transposition Table

By disabling the transposition table, it is easier to observe characteristics of the parallel searching algorithm that may be obscured when the table is in use. Table II summarizes the results obtained on five-ply searches without a transposition table. The *nodes* column corresponds to the number of leaf nodes searched by all the processors combined. The *secs* field of the table is the real time required, truncated to seconds, for the system to search the tree, with *speedup* being the average of the ratio of the time required by the uniprocessor program to that of the multiprocessor system. All the leaf node counts in Table II are the averages of the counts for each of the 24 test positions.

TABLE II
BASIC PERFORMANCE RESULTS WITHOUT TRANSPOSITION TABLES

No Transposition Table					
fanout f	leaf nodes	time secs	average speedup	% search overhead	% time overhead
1	60065	1611	1.0	0	0
2	63268	942	1.84	5.3	16.9
3	64503	664	2.53	7.4	26.6
4	64470	532	3.06	7.3	32.1

The search overhead of the system is reflected in the general increase in the number of leaf nodes examined as more processors are used. With a few exceptions this is true for all the test positions. To understand why one or two tests do not display this characteristic, one must first reconsider the behavior of the parallel algorithm as compared to its sequential counterpart. If the first move is not best, new candidates will arise and these will have to be re-searched with the correct window as they are recognized. When this is being done in parallel, it is possible that more of these wide window searches will be carried out, since many processors may simultaneously have a move that is better than the first one. Consequently, there will be more true scores for the list of moves, rather than the approximations returned by the uniprocessor using a minimal window search. This can subsequently alter the search since a different ordering will result after the move list is sorted between iterations. This reordering is the reason for the change in the move selected by systems of different processor sizes and for the occasional decrease in leaf nodes searched by the larger systems [20]. For example, Table II shows an apparent anomaly in the reduction of leaf nodes searched when four processors are used. This was because the results of a single test position biased the average number of nodes searched. Owing to the size of that search, any changes had a large effect on the average, as illustrated by the decrease in the average node count from the three to four processor case.

Another factor affecting the speedup is the required synchronization of processor after the search of all the moves at a node where splitting has occurred. The problem is especially evident for searches where the principal variation changes. When a new candidate variation is found late in the search, the child that is searching this variation may be the only processor working while the others are waiting for it to finish. For our system the processor idle time is attributed to two factors: time lost waiting for a response to a shared table probe or update, and time lost waiting for other processors to terminate. Both of these losses are lumped into the general category of communication overhead, but in our experience the waiting for work or "synchronization overhead" is the more significant component. Thus, if only modest amounts of message passing are performed, the time lost while waiting for responses will be negligible [21], and so the losses may be attributed to synchronization [18], [21].

In our system, communication overhead cannot be measured directly because the message lengths are too short (only a few bytes) and the clock timer interval too long (1/60 s). However, it can be estimated as the difference between the time overhead and the search overhead (12) and (10), respectively. Much of the idle time results from the transmission of refutations and move lists after an iteration is completed. This *inter-iteration* communication occurs at 960 bytes/s for systems of up to three processors, but at half that speed with the four processor configuration. However, the largest part of the communication overhead is attributable to the short *at-node* messages, which arise when the (optional) global transposition table is used.

B. Global or Local Transposition Table

The basic performance of the system without transposition tables was shown in Table II. Transposition tables may be provided in a variety of ways and we have considered four combinations of global and local tables using both complete and partial storage of subtree results. In the latter case, only those positions that had been searched to a depth of more than two ply were saved and, similarly, a table lookup was performed only when the node was more than two ply from a leaf. In the complete system, on the other hand, a table lookup is made upon the initial visit to any tree node. For our experiments, the global transposition table held 8192 entries for each side, while local tables were half that size. For large (deep) trees this size is inadequate and overloading of the transposition table results.

When the complete storage of tree nodes is employed, there is a dramatic decrease in the number of leaf nodes visited. The results in Table III also show that with the global table fewer leaf nodes are visited than when local tables are used. This is to be expected since a global table is updated by all processors, and so will enable more cutoffs. However, the global table's effect on the leaf node count is at the expense of an increase in communication overhead. In fact, the communication delays destroy the program's performance. With a

TABLE III
COMPARISON OF PARTIAL AND FULL STORAGE OF SUBTREE RESULTS

(a) Global Transposition Table										
	all subtrees stored					only subtrees of length > 2 ply				
fanout f	leaf nodes	time secs	average speedup	% search o/head	% time o/head	leaf nodes	time secs	average speedup	% search o/head	% time o/head
1	53349	1307	1.0	0	0	52705	1275	1.0	0	0
2	53674	949	1.31	0.6	45.2	53274	684	1.84	1.1	7.3
3	53953	766	1.59	1.1	75.8	54178	498	2.5	2.8	17.2
4	56202	872	1.41	5.3	166.0	54558	420	2.91	3.5	31.8

(b) Local Transposition Table										
	all subtrees stored					only subtrees of length > 2 ply				
fanout f	leaf nodes	time secs	average speedup	% search o/head	% time o/head	leaf nodes	time secs	average speedup	% search o/head	% time o/head
1	53349	1307	1.0	0	0	52705	1275	1.0	0	0
2	53993	684	1.88	1.2	4.6	53427	670	1.89	1.4	5.1
3	55345	502	2.55	3.7	15.2	54344	485	2.59	3.1	14.1
4	57930	457	2.91	8.6	39.8	55224	402	3.1	4.8	26.1

four processor configuration the speedup is actually less than that for three processors, and this is attributed to the increased message traffic and reduced communication rate. This rate change was necessary to prevent the overloading of the communications interface unit. Fortunately, the local transposition table does not require this communication; thus, it provides superior elapsed time performance, even though it examines more leaf nodes than its global counterpart. It is not at all clear from our present set of experimental results just what the effect of using a high performance communication line would be. The components of the communication overhead are access and transfer time, and synchronization time. Reducing the access and transfer times may indirectly affect the synchronization delays. Our newer system, in which the processors are interconnected via an Ethernet, allows us to address this issue and also to improve our graphs for search and communication overhead [21].

In a parallel search system using a transposition table, there is no guarantee that the same cutoffs will always occur, especially if the number of processors changes. For example, with a global table, the order of storage and retrieval will vary for systems of different sizes, and this can even result in different moves being selected when several moves have the same best score [20]. When implementing a local table, the positions seen by other processors will not be able to produce a cutoff for the processor in question. Different sets of cutoffs will then result for various system configurations, depending on which processors search which subtrees. Consequently, the leaf node counts may even decrease when the number of processors increases, although this is not generally true.

The problem of table overloading and the high frequency of table access associated with the complete storage table can be partially alleviated by only storing the values of subtrees whose length is greater than two ply. Use of this technique decreases the communication overhead for the global table, resulting in a system which compares more favorably with the local table approach. Nevertheless, the results in Table III show that use of a local transposition table produces a better speedup than the global table case, even though the former configuration examines more leaf nodes. In both instances, the performance, based on leaf count and search time, is better when using the partial rather than the complete storage system. In the latter case, more store requests arise and these must replace older entries in the table, making a later retrieval of them impossible. A larger table should alleviate this problem.

TABLE IV
6-PLY SEARCH WITH LOCAL, DEPTH LIMITED, TRANSPOSITION TABLE

Terminal node count and CPU time (6-ply) Local transposition table for subtrees > 2 ply												
	one processor			two processors			three processors			four processors		
board	leaf nodes	time mins	move	leaf nodes	time mins	average speedup	leaf nodes	time mins	average speedup	leaf nodes	time mins	average speedup
A	(forced	mate)	d6d1!									
B	132751	45	e4e5	138063	23	1.92	140299	16	2.80	141677	12	3.58
C	141201	74	e7d8	146760	39	1.89	149580	28	2.65	153523	22	3.33
D	131304	42	e5e6!	136711	23	1.84	143206	17	2.45	144396	13	3.05
E	351867	227	f1f5	362944	125	1.82	364990	88	2.57	365387	70	3.21
F	42942	8	g5g6!	47869	5	1.69	52372	3	2.24	56051	3	2.64
G	228816	148	h5f6!	239839	78	1.89	340005	74	1.98	404481	75	1.95
H	14780	2	e2c3	15671	1	1.77	16142	1	2.30	16278	0	2.56
I	174398	111	f3e5	178305	58	1.91	178586	38	2.85	178555	30	3.66
J	436938	202	e8e6	545042	130	1.56	510633	82	2.44	509370	67	2.99
K	249238	126	g3f5	256469	66	1.89	270668	51	2.45	277254	41	3.06
L	165232	62	d7f5!	170502	33	1.89	173795	23	2.70	176636	18	3.41
M	189128	72	a1c1	192818	37	1.93	200136	25	2.82	201468	20	3.63
N	83343	39	d1d2!	86935	21	1.83	83653	14	2.70	78559	10	3.69
O	61098	30	g4g7!	64520	16	1.82	66482	12	2.44	65447	9	3.13
P	212469	100	d2e4!	187435	44	2.24	219667	38	2.62	224228	34	2.95
Q	191081	141	d7c5	199063	75	1.87	202848	53	2.64	209140	46	3.08
R	347935	191	c8g4	371298	105	1.82	379596	71	2.67	383867	56	3.41
S	200645	81	c7c5	205487	43	1.87	207509	31	2.58	208764	25	3.18
T	677076	315	c3b5	393672	95	3.32	386704	66	4.76	388266	52	6.05
U	367754	194	f5h6!	391581	103	1.88	397914	69	2.80	400983	54	3.59
V	132017	86	d7e5	130688	43	1.98	130886	39	2.19	131032	37	2.29
W	531720	204	e8g8	541273	126	1.62	540947	75	2.70	460973	63	3.20
X	176158	78	b4c5	176184	39	1.98	178088	27	2.85	179959	21	3.59
total	5239891	2590	9	5179129	1339		5334706	953		5356294	790	
mean	227821	112		225179	58	1.92	231943	41	2.66	232882	34	3.27

Based on the five-ply results, it appears that the addition of a local, depth-limited transposition table produces the best search times, and correspondingly, the best speedup for the number of processors. Therefore, a series of six-ply tests were performed and the results appear in Table IV. For the five-ply studies, a speedup of 1.89, 2.59, and 3.10, with a standard deviation of 0.10, 0.29, and 0.52, was obtained for the 2–4 processor systems, respectively. The six-ply results illustrated improved speedups of 1.92, 2.66, and 3.27 with larger standard deviations of 0.33, 0.51, and 0.75. In positions P and T, a speedup greater than the number of processors is achieved, which partly explains the increase in the six-ply average. However, this suggests that for the single processor case the transposition table was overloaded. Thus, the multiprocessor system made better use of the transposition table, leading to some good cutoffs that were not available to the uniprocessor program.

These speedups compare favorably to the 2.34 achieved with treesplitting on Arachne [22] using three processors, and the 2.4 obtained on an early version of a five-processor system, OSTRICH/P [23], in an application similar to ours.[4] Even so, the time overhead seems to be increasing exponentially (see Fig. 7), while the search overhead is reaching a plateau. This means that the communication overhead is the primary source of losses and we attribute this to the waiting synchronization that occurs as processors become idle after the search of a PV node. While this problem is less severe as the search depth increases, note that the time overhead with four processors declines from 26 percent to 22 percent when the search depth is increased from 5 to 6 ply, it is still unsatisfactorily high. Typically, these synchronization losses are not considered in the simulation studies of others [5]–[7], but in practice are a severe problem [18].

Based on 5-ply searches of all our test positions, the average value of w is 34. Thus it is possible to plot in Fig. 7 our estimate of $f/2w$ for synchronization overhead [see (14)]. However this estimate must be regarded as a lower bound, since it does not take into account the statistical nature of game trees. In practice we observe that major losses arise when $(f-1)$ processors are idle, waiting for a single processor to complete a big search. Thus, we suspect that $(f-1)/w$ may be a better estimate of the synchronization losses. Our experimental data, on the other hand, suggests that the synchronization overhead is $O(f^2)$. This inconsistency may simply reflect that a better implementation of *pvsplit* is possible.

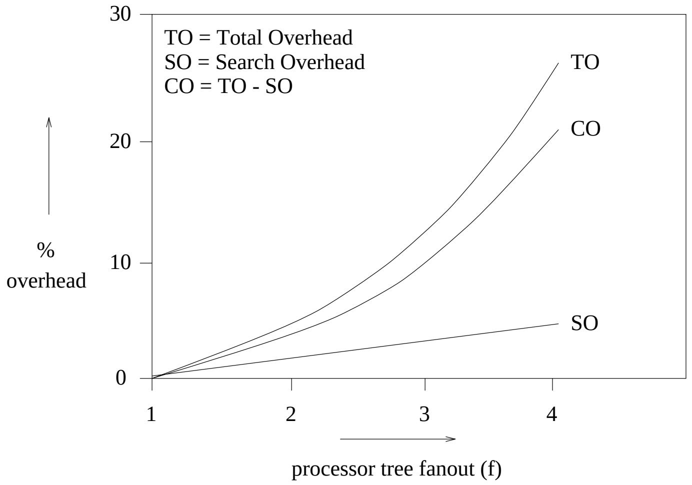

Fig. 7. Average overheads for 5-ply searches.

IX. SUMMARY

We have presented the outline of a multiprocessor-based system for use in minimax game tree searches. The use of memory tables has been examined along with the problems involved in their local and global implementations. The system development has also addressed the issues of processor management and interunit communication, which can be related to other parallel systems. Although we have designed our

[4]The designer of OSTRICH advised me that, during the summer of 1984, with eight processors his program achieved a speedup of about five.

parallel processing system for a specific application, it is expressed in general terms so that the ideas employed may be suitable for any minimax tree search application. An unsophisticated model of time, search, and communication overhead has been developed. From this the major losses have been attributed to processor synchronization (the idle time while waiting for other processors to finish their search). Unfortunately, some of the quantities that we need to develop a better insight into these losses were not directly measurable in our system. Its replacement [21] should allow us to refine these equations further.

Our experiments with the *Parabelle* system show that the PV splitting method is promising for use in multiple processor tree searching systems, and the inclusion of local, depth-limited transposition tables appears to be an effective enhancement to the basic system. Although the communication problems associated with the global transposition table could be alleviated with faster communication speeds, the savings from visiting slightly fewer leaf nodes are not likely to compensate for the wait time attributed to communication delays. Also, if the new alpha bound were made available to all processors as soon as it was determined, rather than when a child processor has finished its subtree search, more cutoffs could be obtained. A further reduction in processor idle time, and thus a corresponding improvement in performance, would be obtained by the allocation of more than one processor for searching new principal variations. By this means, the better cutoff value associated with the new variation will be used earlier in the search of all the remaining moves in the list. Another possibility for improvement lies in deferring new principal variation searches until there is more than one possible new candidate [24]. Thus, if only one such move were found, a re-search would not be necessary since it is clearly the best move, although the true score would not be available. In our experience, this approach needs a large transposition table in order to be effective.

ACKNOWLEDGMENT

The assistance of S. Sutphen and J. Rus in configuring and maintaining the system hardware was invaluable, as was the help of M.-H. Lim in typesetting the equations and figures. Also, the work of M. Olafsson in verifying the theoretical model and doing the computations to estimate the average branching factor is recognized.

REFERENCES

[1] T. A. Marsland and M. Campbell, "Parallel search of strongly ordered game trees," *Comput. Surveys,* vol. 14, pp. 533–551, 1982.

[2] T. A. Marsland, "Relative efficiency of alpha-beta implementations," in *Proc. Int. J. Conf. Artif. Intell.,* Karlsruhe, West Germany, Aug. 1983, pp. 763–766.

[3] G. Baudet, "The design and analysis of algorithms for asynchronous multiprocessors," Ph.D. dissertation, Dep. Comput. Sci., Carnegie-Mellon Univ., Pittsburgh, PA, Apr. 1978.

[4] J. Fishburn and R. Finkel, "Parallel alpha-beta search on Arachne," Dep. Comput. Sci., Univ. Wisconsin, Madison, Tech. Rep. 394, July, 1980.

[5] S. Akl, D. Barnard, and R. Doran, "Design, analysis, and implementation of a parallel tree search algorithm," *IEEE Trans. Pattern Anal. Machine Intell.,* vol. PAMI-4, pp. 192–203, Mar. 1982.

[6] G. Lindstrom, "The key node method: A highly-parallel alpha-beta algorithm," Dep. Comput. Sci., Univ. Utah, Salt Lake City, Tech. Rep. UUCS 83-101, Mar. 1983.

[7] F. W. Burton and M. Huntbach, "Virtual tree machines," *IEEE Trans. Comput.,* vol. C-33, no. 3, pp. 278–280, 1984.

[8] J. Fishburn, "Analysis of speedup in distributed algorithm," Dep. Comput. Sci., Univ. Wisconsin, Madison, Tech. Rep. 421, May 1981.

[9] B. W. Kernighan and D. M. Ritchie, *The C Programming Language.* Englewood Cliffs, NJ: Prentice-Hall, 1978.

[10] J. Pearl, "Asymptotic properties of minimax trees and game searching procedures," *Artif. Intell.,* vol. 14, pp. 113–138, 1980.

[11] M. Campbell and T. A. Marsland, "A comparison of minimax tree search algorithms," *Artif. Intell.,* vol. 20, pp. 347–367, 1983.

[12] J. Gillogly, "Performance analysis of the technology chess program," Ph.D. dissertation, Dep. Comput. Sci., Carnegie-Mellon Univ., Pittsburgh, PA, Mar. 1978.

[13] D. Slate and L. Atkin, "CHESS,4.5–The Northwestern University Chess Program," in *Chess Skill in Man and Machine,* P. Frey, Ed. New York: Springer, 1977, pp. 82–118.

[14] B. A. Bowen and R. J. A. Buhr, *The Logical Design of Multiple Microprocessor Systems.* Englewood Cliffs, NJ: Prentice-Hall, 1980.

[15] *SUN-1 System Reference Manual,* SUN Microsystems Inc., Mountain View, CA, July 1982.

[16] *Intelligent Octal Serial Interface,* Central Data Corp., Champaign, IL, 1981.

[17] D. Knuth and R. Moore, "An analysis of alpha-beta pruning," *Artif. Intell.,* vol. 6, pp. 293–326, 1975.

[18] J. Mohan, "A study of parallel computations–The traveling salesman problem," Dep. Comput. Sci., Carnegie-Mellon Univ., Pittsburgh, PA, Tech. Rep. CMU-CS-82-136, Aug. 1982.

[19] I. Bratko and D. Kopec, "A test for comparison of human and computer performance in chess," in *Advances in Computer Chess,* vol. 3, M. R. B. Clarke, Ed. New York: Pergamon, 1982, pp. 31–56.

[20] F. Popowich and T. A. Marsland, "Parabelle: Experience with a parallel chess program," Dep. Comput. Sci., Univ. Alberta, Edmonton, Alta., Canada, Tech. Rep. 83-7, Aug. 1983.

[21] T. A. Marsland, M. Olafsson, and J. Schaeffer, "Multiprocessor tree-search experiments," in *Advances in Computer Chess,* vol. 4, D. Beal, Ed. New York: Pergamon, 1985.

[22] R. Finkel and J. Fishburn, "Parallelism in alpha-beta search," *Artif. Intell.,* vol. 19, pp. 89–106, 1982.

[23] M. Newborn, "OSTRICH/P-A parallel search chess program," School Comput. Sci., McGill Univ., Montreal, P.Q., Canada, SOCS-82.3, Mar. 1982.

[24] K. Thompson, private communication, Bell Laboratories, Murray Hill, NJ, Oct. 1981.

T. A. Marsland (S'63-M'68-SM'85) received the undergraduate degree in mathematics from Nottingham University, Nottingham, England, and the graduate degree in electrical engineering from the University of Washington, Seattle.

He has been a Professor of Computing Science at the University of Alberta, Edmonton, Alta., Canada, since 1970. Prior to his teaching appointment he was a member of the Technical Staff at Bell Telephone Laboratories, Holmdel, NJ, and also worked as a Programmer Analyst at Boeing Co., Seattle, WA. His teaching and research interests are in the area of computing systems, especially distributed applications. As part of his work in experimental computer science his programs have participated over the past decade in a series of computer chess competitions.

Professor Marsland is a member of CIPS and Sigma Xi. For two years he served as a National Lecturer in distributed processing and computer chess for the Association for Computing Machinery.

Fred Popowich received the B.Sc. degree in computing science from the University of Alberta, Edmonton, Alta., Canada, in 1982.

After being employed as a researcher at the same university, he started work towards an M.Sc. degree in computing science at the University of British Columbia, Vancouver, B.C., and is currently completing his M.Sc. studies at Simon Fraser University, Burnaby, B.C.

His research interests include computational linguistics, computer chess, and programming languages.

Mr. Popowich is a member of the Association for Computational Linguistics and the Canadian Society for Computational Studies of Intelligence.