PARALLEL ALPHA-BETA SEARCH ON ARACHNE

by

John P. Fishburn
Raphael A. Finkel

Computer Sciences Technical Report #394

July 1980

TABLE OF CONTENTS

Parallel Alpha-Beta Search on Arachne

John P. Fishburn

Raphael A. Finkel

Computer Sciences Department

University of Wisconsin-Madison

Abstract

We present a distributed algorithm for implementing α-β search on a tree of processors. Each processor is an independent computer with its own memory and is connected by communication lines to each of its nearest neighbors. Measurements of the algorithm's performance on the Arachne distributed operating system are presented. A theoretical model is developed that predicts speedup with arbitrarily many processors.

1. INTRODUCTION

The α-β search algorithm is central to most programs that play games like chess. It is now well-known [1] that an important component of the playing skill of such programs is the speed at which the search is conducted. For a given amount of computing time, a faster search allows the program to "see" farther into the future. In this paper we present and analyze a parallel adaptation of the α-β algorithm. This adaptation, which we will call the tree-splitting algorithm, speeds up the search of a

large tree of potential continuations by dynamically assigning subtree searches for parallel execution.

In section 2, we summarize the α-β algorithm. Section 3 reviews a parallel implementation of the α-β algorithm suggested by Baudet [2]. Section 4 formally describes the tree-splitting algorithm. Section 5 presents performance measurements for this algorithm taken on a network of microprocessors. Section 6 discusses some possible optimizations and variations of the algorithm. Section 7 derives the obtainable speedup with k processors as k tends towards ∞.

2. THE ALPHA-BETA ALGORITHM

Consider a board position from a game like chess or checkers. All possible sequences of moves from this position may be represented by a tree of positions called the lookahead tree. The nodes of the tree represent positions; the children of a node are moves from that node. The root node of the tree represents the current position. Since lookahead trees for most games are often too large to be searched even by computer, they are usually truncated at a certain level. Since we will later be referring to a tree of processors, we reserve the following notation for nodes of lookahead trees: A node is often called a position. A node's child is its successor, and its parent is its predecessor. If each non-terminal node has n successors, we say that the tree has degree n. The level of a node or subtree is its distance from the root.

The α-β algorithm is an optimization of the minimax algorithm, which we will review first. The two players are called max and min; at the root node, it is max's turn to move. The minimax algorithm proceeds as follows: First, each leaf of the lookahead tree is assigned a static value that reflects that position's desirability. (High values are desirable to max. In a game like chess, the main component of the value is usually the material balance between the two sides.)

The interior nodes of the lookahead tree may be given minimax values recursively: If it is max's turn to move at node A, the value of A is the maximum of A's successors' values. (If the game were to proceed to node A, it would then be max's turn to move. Max, being rational, would choose the successor with the maximum value, say M. Therefore, the subtree rooted at A must have M as its value, because M is the value of the leaf node we would reach if the game reached A.) Similarly, if it is min's turn to move at a node, then the value of that node is the minimum of these values.

We will use a version of the minimax procedure called negamax: When it is max's turn to move at a terminal node, the node is assigned the same static value used in minimax. When it is min's turn to move, the static value assigned is the negative of what it would be in the minimax case. The value of a nonterminal node at any level is defined to be the maximum of the negatives of the values of its successors.

The negamax algorithm can be cast into an ad hoc Pascal-like language. The following program is adapted from Knuth [3] :

```
function negamax(p:position):integer;
var  m: integer;
     i,d : 1..MAXCHILD;
     succ : array[1..MAXCHILD] of position;
begin
     determine the successor positions
          succ[1],...,succ[d];
     if d = 0 then { terminal node }
          negamax := staticvalue(p)
     else
     begin { find maximum of child values }
          m := - ∞;
          for i := 1 to d do
               m := max(m,- negamax(succ[i]));
          negamax := m;
     end
end.
```

The α-β algorithm evaluates the lookahead tree without pursuing irrelevant branches. Suppose we are investigating the successors in a game of chess, and the first move we look at is a bishop move. After analyzing it, we decide that it will gain us a pawn. Next we consider a queen move. In considering our opponent's replies to the queen move, we discover one that can irrefutably capture the queen; she has moved to a dangerous spot. We need not investigate our opponent's remaining replies; in light of the worth of the bishop move, the queen move is already discredited.

The α-β search algorithm [3] formalizes this notion:

```
function alphabeta(p : position; α,β : integer) : integer;
label DONE;
var  i,d : 1..MAXCHILD;
     succ : array[1..MAXCHILD] of position;
begin
     determine the successor positions
          succ[1],...,succ[d];
     if d = ∅ then
          alphabeta := staticvalue(p)
     else
     begin
          for i := 1 to d do
          begin
               α := max(α, - alphabeta(succ[i], -β,-α));
               if α ≥ β then goto DONE { cutoff }
          end;
DONE:     alphabeta := α
     end
end.
```

The function alphabeta obeys the accuracy property: For a given position p, and for values of α and β such that $\alpha < \beta$,

if negamax(p) $\leq \alpha$, then alphabeta(p,α,β) $\leq \alpha$

if negamax(p) $\geq \beta$, then alphabeta(p,α,β) $\geq \beta$

if $\alpha <$ negamax(p) $< \beta$, then alphabeta(p,α,β) = negamax(p)

The first and second cases above are called failing low and failing high respectively. In the third case, success, alphabeta accurately reports the negamax value of the tree. Success is assured if $\alpha = -\infty$ and $\beta = \infty$. The pair (α,β) is called the window for the search.

To return to our example: When alphabeta is called with p representing the queen move, it is min's move. β is the cutoff value generated by the bishop move. The better the bishop move was for max, the lower is β. (Within the routine alphabeta, high values for α and β are good for the player whose move it is. A high value for α indicates that a good alternative for that

player exists somewhere in the tree. A low value for β indicates that a good alternative exists for the other player somewhere else in the tree.) When the successor that captures the queen is evaluated, α becomes larger than β, and a cutoff occurs.

α-β pruning serves to reduce the branching factor, which is the ratio between the number of nodes searched in a tree of height N and one of height N-1, as N tends to ∞. Both theory [3] and practice [4] agree that with good move ordering (investigating best moves first), α-β pruning reduces the branching factor from the degree of the lookahead tree nearly to the square root of that degree. For a given amount of computing time, this reduction nearly doubles the depth of the lookahead tree.

When the algorithm is performed on a serial computer, the value of one successor can be used to save work in evaluating its siblings later on. Nevertheless, greater speed can be obtained by conducting α-β search in a parallel fashion. We define the speedup of a parallel algorithm over a serial one to be the time required by the serial algorithm divided by the time for the parallel algorithm. We will restrict our attention to parallel computers built as a tree of serial computers. A node in this tree is a processor, a parent is a master, and a child is a slave.

3. PARALLEL ASPIRATION SEARCH

In order to introduce parallelism, Baudet [2] rejects decomposition of the lookahead tree in favor of a parallel aspiration search, in which all slave processors search the entire lookahead tree, but with different initial α-β windows. These windows are disjoint, and in the simplest variant their union covers the range from $-\infty$ to $+\infty$. Since each window is considerably smaller than $(-\infty, +\infty)$, each processor can conduct its search more quickly. When the processor whose window contains the true minimax value of the tree finishes, it reports this value, and move selection is complete. Baudet analyzes several variants of this algorithm under the assumption of randomly distributed terminal values, and concludes that the obtainable speedup is limited by a constant independent of the number of processors available. This maximum is established to be approximately 5 or 6. Surprisingly, for k equal to 2 or 3, Baudet's method yields more than k-way speedup with k processors. Baudet infers that the serial α-β search algorithm is not optimal, and estimates that a 15 to 25 percent speedup may be gained by starting the search with a narrow window.

Since a narrow window does not speed up a successful search when moves are ordered best-first, Baudet's method yields no speedup under best-first move ordering.

4. THE TREE-SPLITTING ALGORITHM

Another natural way to implement the α-β algorithm on parallel processors divides the lookahead tree into its subtrees at the top level, and queues them for parallel assignment to a pool of slave processors. The master processor, as in the serial algorithm, maintains the variable α as the maximum of the negative of all subtree values. Each slave processor computes the value of its assigned subtree. The slave may use either serial α-β search or parallel α-β search if it has slaves of its own. When it finishes, it reports the value computed to its master. As the master receives responses from slaves, it narrows its window, and possibly tells working slaves about the improved window. When all subtrees have been evaluated, the master is able to compute the value of its position. A similar approach is discussed in [5].

4.1 The Slave Algorithm

The slave algorithm runs at terminal nodes of the processor tree. We will describe its interactions with its master by means of messages. The algorithm is equally easily expressed in a shared-memory or call-return form. The slave receives EVALUATE messages from its master, followed by any number of associated UPDATE messages that narrow its window. When an UPDATE message arrives, the slave adjusts its recursive values of α and β to

what they would have been had the search been started with the smaller window. When the slave has performed the search specified by the EVALUATE command, it sends a VALUE message back to its master and then waits for another EVALUATE message.

The algorithm calls five functions:

Staticvalue(position)

returns the static value of "position".

Send(message)

sends the data in buffer "message" to process message.dest.

Receive(message)

receives a message sent to this process, and places it in buffer "message".

Catch(kind,message,catcher)

arranges for all future messages with message.kind = "kind" to be immediately routed to buffer "message", bypassing any receive. Catch returns immediately, allowing the caller to proceed. Thereafter, when a message with the indicated kind arrives, the process is interrupted, and the routine "catcher" is called. When "catcher" returns, the process resumes. Slaves use catch to receive UPDATE messages without wasting time polling for them.

Alphabeta(p)

was defined in section 2. The variables α and β are global arrays, not formal parameters, in order to facilitate updating their values in each recursive call of alphabeta when an UPDATE message arrives. The global variable "depth" represents the level of p.

The slave algorithm:

```
program slave();
label DONE;
var message,updatemessage :
          record
               pos : position;
               α,β,value : integer;
               kind : (EVALUATE,UPDATE,VALUE);
               dest : process;
          end;
     pos : position;
     α,β : array[1..MAXDEPTH] of integer;
     depth : 1..MAXDEPTH;
     tmp : integer;
     succ : array[1..MAXCHILD] of position;
     i,d : 1..MAXCHILD;
     mymaster : process;

procedure catcher; { called asynchronously by UPDATE }
     var  scalα,scalβ,tmp : integer;
          k : 1..MAXDEPTH;
     begin
          scalα := updatemessage.α;
          scalβ := updatemessage.β;
          for k := 1 to MAXDEPTH do
          begin { update α,β arrays }
               α[k] := max(α[k],scalα);
               β[k] := min(β[k],scalβ);
               tmp := scalα;
               scalα := -scalβ;
               scalβ := -tmp;
          end
     end;
begin
     catch(UPDATE,updatemessage,catcher);
     while true do
     begin { 1 iteration per EVALUATE }
          receive(message); { receive EVALUATE }
          pos := message.pos;
```

```
          depth := 1;
          α[depth] := message.α;
          β[depth] := message.β;
          determine the children of pos
               succ[1],...,succ[d];
          if d = ∅ then { evaluate terminal position }
               message.value := staticvalue(pos);
          else begin
               for i := 1 to d do
               begin { evaluate each successor }
                    α[depth+1] := - β[depth];
                    β[depth+1] := - α[depth];
                    depth := depth+1;
                    tmp := - alphabeta(succ[i]);
                    depth := depth-1;
                    if tmp > α[depth] then
                         α[depth] := tmp;
                    if α[depth] ≥ β[depth] then
                    begin message.value := α[depth];
                         goto DONE; { cutoff occurs }
                    end
               end { for i := 1 to d do }
          end;
DONE:     message.kind := VALUE;
          message.dest := mymaster;
          send(message);
     end { while TRUE do }
end. { program slave }
```

4.2 The Master Algorithm

The master algorithm runs on non-terminal nodes of the processor tree. It receives EVALUATE and UPDATE messages from its master and VALUE messages from its slave nodes. After an EVALUATE message is received, the master generates all successors of the position to be evaluated. Each slave is requested to EVALUATE one of these positions; the remaining positions are queued for service by slaves. Any UPDATE messages are relayed to active slaves.

The master may take various actions when it receives a VALUE message from a slave. First, if the VALUE message causes the current α value to increase, then $-\alpha$ is sent as an updated β value to all active slaves. Second, if α has been increased so that it becomes greater than or equal to β, then an α-β cutoff occurs. The nonpositive-width window is sent to all active slaves, quickly terminating them. Meanwhile, the master empties its queue of waiting successor positions. Third, if the queue of unevaluated successor positions is non-empty, the reporting slave is assigned the next position from the queue.

When all successors have been evaluated, the master sends a VALUE message to its master. In a game situation, the algorithm at the root node might serve as the user interface, and would remember which move has the maximum value.

Here is the master algorithm:

```
program master();
label INIT;
var  message :
     record
          pos : position;
          α,β,value : integer;
          kind : (EVALUATE,UPDATE,VALUE);
          dest : process;
     end;
     pos : position;
     succ : array[1..MAXCHILD] of position;
     succstat : array[1..MAXCHILD] of (ASSIGNED,UNASSIGNED);
     i,d : 1..MAXCHILD;
     slave : array[1..MAXSLAVE] of process;
     slavestat : array[1..MAXSLAVE] of (BUSY,FREE);
     j : 1..MAXSLAVE;
     mymaster : process;
     α,β,tmp : integer;
begin
     while true do
     begin { 1 iteration per EVALUATE }
```

```
INIT:     repeat { flush outdated UPDATEs }
               receive(message);
          until message.kind = EVALUATE;
          pos := message.pos;
          α := message.α;
          β := message.β;
          determine the successor positions
               succ[1],...,succ[d];
          if d = ∅ then
          begin { terminal node }
               message.value := staticvalue(pos);
               message.kind := VALUE;
               message.dest := mymaster;
               send(message);
               goto INIT;
          end;
          for j:= 1 to MAXSLAVE do
               slavestat[j] := FREE;
          for i := 1 to d do
               succstat[i] := UNASSIGNED;
          while there exists a FREE slave j
               and an UNASSIGNED successor i do
          begin { give initial assignments }
               message.pos := succ[i];
               message.α := -β;
               message.β := -α;
               message.kind := EVALUATE;
               message.dest := slave[j];
               send(message);
               slavestat[j] := BUSY;
               succstat[i] := ASSIGNED;
          end;
          while there exist BUSY slaves do
          begin
               receive(message);
               if message.kind = UPDATE then
               begin { forward UPDATE message }
                    if (message.α > α) or
                       (message.β < β) then
                    begin
                         α := max(α,message.α);
                         β := min(β,message.β);
                         message.α := -β;
                         message.β := -α;
                         message.kind := UPDATE;
                         send(message) to all slaves;
                    end
                    if α ≥ β then { cutoff }
                         for i:=1 to d do
                              succstat[i] := ASSIGNED;
               end
               else { message.kind = VALUE }
               begin
```

```
                    j := answering slave;
                    slavestat[j] := FREE;
                    tmp := -message.value;
                    if tmp > α then
                    begin { send new α-β window }
                         α := tmp;
                         message.α := -β;
                         message.β := -α;
                         message.kind := UPDATE;
                         send(message) to all slaves;
                    end;
                    if α ≥ β then { cutoff }
                         for i:=1 to d do
                              succstat[i] := ASSIGNED;
                    if there remains a successor,
                         i, yet to be evaluated then
                    begin { reassign slave }
                         slavestat[j] := BUSY;
                         succstat[i] := ASSIGNED;
                         message.pos := succ[i];
                         message.α := -β;
                         message.β := -α;
                         message.kind := EVALUATE;
                         message.dest := slave[j];
                         send(message);
                    end
               end{ else message.kind = VALUE }
          end; { while there are BUSY slaves }
          message.value := α;
          message.kind := VALUE;
          message.dest := mymaster;
          send(message);
     end{ while TRUE do }
end. { program master }
```

4.3 Alpha Raising

As an optimization of the master algorithm, the master running on the root node may send a special α-β window to a slave working on the last unevaluated successor. This window is (-α-1,-α) instead of the usual (-β,-α). If that successor is not the best, then the slave's search will fail high as usual, but the minimal window speeds its search. If that successor is best, then the smaller window causes the search to fail low, again ter-

minating faster. In either case, the root master determines which successor is the best move, even though its value may not be calculated. By speeding the search of the last successor, the idle time of the other slaves is reduced. (This narrow window given to the root's last subtree search can also be used in serial α-β search, as discussed in the Appendix.)

We can generalize this technique in the following way, called alpha raising: Suppose that, among slaves evaluating successors of the root, $slave_1$'s current α value, α_1, is lower than any other, and that $slave_2$ has the second lowest α value, say α_2. Update α_1 to α_2-1, speeding up $slave_1$. If this update causes $slave_1$'s otherwise successful search to fail low, then the reported value is still lower than all others, and that move is still discovered to be best.

5. MEASUREMENTS OF THE ALGORITHM

Measurements of the performance of the tree-splitting algorithm have been taken on a network of LSI-11 microcomputers running under the Arachne* [6] operating system.

---

*We have been forced to change the name of the Roscoe distributed operating system, since Roscoe is a registered trademark of Applied Data Research, Incorporated. The new name we have chosen is Arachne; the operating system and research continue unchanged.

The game of checkers was used to generate lookahead trees. Static evaluation was based on the difference in a combination of material, central board position for kings and advancement for men. Moves were ordered best-first according to their static values. General α-raising was not employed, except for the special case for the last successor. A single LSI-11 machine searches lookahead trees at a rate of about 100 unpruned nodes per second. Inter-machine messages can be sent at a rate of about 70 per second.

Since only 5 processors are currently available in Arachne, it was not possible to test processor trees of depth greater than one directly. Instead, a depth-one processor tree was used to measure the speedup gained by replacing all leaf processors with depth-one processor trees. When these slaves are at level n, we call the measured speedup γ_n. γ_0 and γ_1 were measured.

The procedure for measuring γ_1 made one simplifying assumption: Both a slave processor and a master processor below level zero can normally receive UPDATE messages from their masters. Due to the difficulty of duplicating the arrival times of these messages, they were not included in either the slave or the master-and-slaves case. (The master still gave its terminal slaves UPDATE messages.)

Ten board positions, B_1, ..., B_{10}, were chosen for use in these experiments. These positions actually arose during a human-machine game; they span the entire game. All lookahead trees from these positions were expanded to a depth of 8.

Two sets of experiments were performed. The two differed only in that the first set used one master and two slaves, while the second set used one master and three slaves. Within each experiment, $\gamma_{\emptyset}$ was measured directly for each B_i by evaluating the tree both serially and with the parallel algorithm running on a depth-one processor tree. Table 1 summarizes measurements of $\gamma_{\emptyset}$.

The ten board positions gave rise to 84 successors, so 84 EVALUATE commands were given to slaves while $\gamma_{\emptyset}$ was being measured. Times for both parallel and serial evaluation were measured for each command. The aggregate speedup for a group of commands is the total time required to execute them serially divided by the total time required to execute them in parallel. For each B_i, the aggregate speedup γ_1 for its subtree evaluations was computed. Table 2 summarizes measurements of γ_1.

Table 1: $\gamma_{\emptyset}$ for each B_i, i=1,...,1Ø

	2 slaves	3 slaves
minimum	1.37	1.37
average	1.81	2.34
maximum	2.36	3.15
standard deviation	Ø.31	Ø.56

Table 2: γ_1 for each B_i, i=1,...,1Ø

	2 slaves	3 slaves
minimum	1.Ø3	1.38
average	1.46	1.96
maximum	1.77	2.6Ø
standard deviation	Ø.22	Ø.38

Surprisingly, more than k-way speedup was occasionally achieved with k slaves: Three out of the ten B_i were sped up by more than 2 with 2 slaves, and two of those three were sped up by more than 3 with 3 slaves. Of the 84 subtrees of the B_is, 4 were sped up by more than 2 with 2 slaves, and 9 were sped up by more than 3 with 3 slaves; 2 of those achieved 6-way speedup. In each such case, subtree evaluations finished in a different order than they were assigned. While one large subtree was being evaluated by one slave, another smaller subtree was assigned and finished. The large subtree's evaluation then received an UPDATE message that sped it up or even terminated it. In fact, time-consuming searches are more likely than short ones to receive these messages. In particular, the search that receives the final $(-\alpha-1,-\alpha)$ window is likely to be larger than average.

6. OPTIMIZATIONS

Since the tree-splitting algorithm can be optimized in several ways, it should be considered the simplest variant of a family of tree-decomposing algorithms for α-β search. As a first optimization, since most of a master's time is spent waiting for messages, that time could be spent profitably doing subtree searches. However, only the deepest masters could hope to compete with their slaves in conducting searches. All other masters are by themselves slower than their slaves because their slaves

have slaves below them to help. However, more than half of all masters control terminal slaves, and greater speedup should be achieved by running a slave algorithm along with these masters on the same processors. We might expect an additional 1.5-way speedup from this technique.

A second optimization groups several higher-level masters onto a single processor. For example, the 3 highest processors in a binary processor tree could be replaced by 3 processes running on a single processor.

Third, a master might evaluate a position by assigning that position's successor's successors to slaves, rather than that position's successors. Although this technique involves more message-passing, some advantage might result, because all of a master's slaves would work on finishing the position's first subtree before going on to the second. The evaluation of the second subtree would then receive the full benefit of the beta value generated by the first subtree. Furthermore, when slaves become idle as one subtree is finished, they can immediately be set to work on the next subtree.

Since most game-playing programs must make their move within a certain time limit, any speedup in tree search ability will generally be used to search a deeper lookahead tree. If we have an unlimited supply of processors to form into a binary tree, we can obtain an unlimited speedup only if the search is not limited in time. Otherwise we cannot, because we would eventually violate our premise that the lookahead tree is at least as deep as the processor tree. A new layer on the processor tree does

not buy another full ply in the lookahead tree. For example, several speedups of 1.5 would be needed to search a 6-times larger chess lookahead tree, or about one additional ply. The depth of the processor tree would grow faster than the depth of the tree it searches and eventually would catch up. The only way to avoid this limit is to increase the fan-out of the processor tree. If the fan-out is high enough that no successor need ever be queued for evaluation by a slave, then the size of the maximum lookahead tree that can be evaluated within the time limit is limited only by the time required for EVALUATE commands to propagate from the root to the leaves. Long before this limitation is reached, we would run out of silicon for making the processors.

7. ANALYSIS OF SPEEDUP

We will now formally analyze the speedup that can be gained in searching large lookahead trees as the number of available processors grows without bound. For this purpose we introduce Palphabeta, a simplified version of the tree-splitting algorithm. This algorithm is less efficient than the version already discussed, but is more amenable to analysis. Much of the analysis in this section is a "parallelization" of results of Knuth [3]. Indeed, when $q = \emptyset$ and $f = 1$, Theorem 1 and Corollary 1 reduce to Knuth's results.

As before, the processors will be arranged in a uniform tree. Let $f \geq 1$ be the fan-out of the processor tree (uniform for all non-terminal nodes), and let $q \geq 1$ be its depth (uniform for all terminal nodes). Let $q + s$ be the depth of the lookahead tree, where $s \geq 1$. We assume that the lookahead tree has a uniform degree and that this degree, df, is a multiple of f, where d is ≥ 2. Here is Palphabeta:

```
function Palphabeta(p : position ; α, β : integer) : integer ;
var i : integer;
function g : integer;
begin
     determine the successors p1, ... ,pdf.
     begin
          if depth(p1) < q then
               g := Palphabeta
          else g := alphabeta;
          for i := 1 to d do
          begin
               α:=max(α,   max    -g(pj,-β,-α));
                        (i-1)f<j≤if
               if α ≥ β then go to DONE;
          end;
     DONE: Palphabeta:= α;
     end;
end;
```

The f function calls specified in the first line of the for-loop are intended to occur in parallel, activating functions existing on each of the f slaves. Unlike the tree-splitting algorithm, Palphabeta waits until all slaves finish before assigning additional tasks. Serial α-β search is activated on leaf slaves; Palphabeta is activated on all others.

7.1 Worst-first ordering

α-β search produces no cutoffs if, whenever the call alphabeta(p,α,β) is made, the following relation holds among the successors $p_1, \ldots, p_d$:

$$\alpha < -\text{negamax}(p_1) < \ldots < -\text{negamax}(p_d) < \beta.$$

We call this ordering worst first. If no cutoffs occur, it is easy to calculate the time necessary for Palphabeta to finish. Assume that a processor can generate f successors, send messages to all of its f slaves and receive replies in time ρ. (This figure counts message overhead time but does not include computation time at the slaves.) Assume also that the serial α-β algorithm takes time n to search a lookahead tree with n terminal positions. Let a_n be the time necessary for a processor at distance n from the leaves to evaluate its assigned position. A leaf processor executes the serial algorithm to depth s. Thus we have $a_\emptyset = (df)^s$. An interior processor gives d batches of assignments to its slaves, and each batch takes time ρ plus the time for the slave processor to complete its calculation. Thus we have $a_{n+1} = d(\rho+a_n)$. The solution to this recurrence relation is

$$a_q = \rho\left(\frac{d^{q+1} - d}{d - 1}\right) + d^{q+s}f^s,$$

which is the total time for Palphabeta to complete. Since the time for the serial algorithm to examine the same tree is $(df)^{q+s}$, the speedup for large s is f^q. There are $(f^{q+1}-1)/(f-1)$ processors, roughly f^q, so when no pruning occurs the parallel

algorithm yields speedup that is roughly equal to the number of processors used.

7.2 Best-first ordering

We will now investigate what happens when the lookahead tree is ordered best-first.

Definition: We will use the Dewey decimal system to name nodes in both processor trees and lookahead trees. The root is named by the null string. The j successors of a node whose name is $a_1 \ldots a_k$ are named by $a_1 \ldots a_k 1$ through $a_1 \ldots a_k j$.

Definition: We say that the successors of a position $a_1 \ldots a_n$ are in best-first order if

$$\text{negamax}(a_1 \ldots a_n) = -\text{negamax}(a_1 \ldots a_n 1).$$

Definition: We say a position $a_1 \ldots a_n$ in the lookahead tree is (q,f)-critical if a_i is (q,f)-restricted for all even values of i or for all odd values of i. An entry a_i is (q,f)-restricted if

$$1 \leq i \leq q \quad \text{and} \quad 1 \leq a_i \leq f$$

or $$q < i \quad \text{and} \quad a_i = 1.$$

Theorem 1: Consider a lookahead tree for which the value of the root position is not $\pm \infty$ and for which the successors of every position are in best-first order. The parallel α-β procedure Palphabeta examines exactly the (q,f)-critical positions of this lookahead tree.

Proof: We will call a (q,f)-critical position $a_1 \ldots a_n$ a type 1 position if all the a_i are (q,f)-restricted; it is of type 2 if

a_j is its first entry not (q,f)-restricted and n-j is even; otherwise (that is, when n-j is odd), it is of type 3. Type 3 nodes have a_n (q,f)-restricted. The following statements can be established by induction on the depth of the position p. (Text in brackets refers to positions of depth < q.)

(1) A type 1 position is examined by calling [P]alphabeta(p,+ ∞,- ∞). If it is not terminal, its successor position[s] p_1[, p_2,...,p_f] is [are] of type 1, and $F(p) = -F(p_1) \neq \pm\infty$. This [These] successor position[s] is [are] examined by calling [P]alphabeta(p_i,- ∞,+ ∞). The other successor positions p_2, ... ,p_{df} [p_{f+1}, ... ,p_{df}] are of type 2, and are all examined by calling [P]alphabeta(p_i,- ∞,$F(p_1)$).

(2) A type 2 position p is examined by calling [P]alphabeta(p,- ∞,β), where $-\infty < \beta \leq F(p)$. If it is not terminal, its successor[s] p_1[, p2, ..., p_f] is [are] of type 3, and $F(p) = -F(p_1)$. This [These] successor position[s] is [are] examined by calling [P]alphabeta(p_i,-β,+ ∞). Since $F(p) = -F(p_1) \geq \beta$, cutoff occurs, and [P]alphabeta does not examine the other successors p_2,...,p_{df} [p_{f+1},...,p_{df}].

(3) A type 3 position p is examined by calling [P]alphabeta(p,α,+ ∞) where $F(p) \leq \alpha < +\infty$. If it is not terminal, each of its successors p_i is of type 2, and they are all examined by calling [P]alphabeta(p_i,- ∞,-α). All of these searches fail high.

It follows by induction on the depth of p that the (q,f)-critical positions, and no others, are examined.

Q.E.D.

Corollary 1: If every position on levels $\emptyset,1,\ldots,q+s-1$ of a lookahead tree of depth q+s satisfying the conditions of Theorem 1 has exactly df successors, for d some fixed constant, and for f the constant appearing in Palphabeta, then the parallel procedure Palphabeta (along with alphabeta, which it calls), running on a processor tree of fan-out f and height q, examines exactly

$$f^{\lfloor q/2\rfloor}(df)^{\lceil (q+s)/2\rceil}+f^{\lceil q/2\rceil}(df)^{\lfloor (q+s)/2\rfloor} - f^q$$

terminal positions.

Proof: There are $f^{\lfloor q/2\rfloor}(df)^{\lceil (q+s)/2\rceil}$ sequences $a_1\ldots a_{q+s}$, with $1\le a_i\le df$ for all i, such that a_i is (q,f)-restricted for all even values of i; there are $f^{\lceil q/2\rceil}(df)^{\lfloor (q+s)/2\rfloor}$ such sequences with a_i (q,f)-restricted for all odd values of i; and we subtract f^q for the sequences $\{1,\ldots,f\}^q 1^s$, that we counted twice.

Q.E.D.

Lemma 1: Given positive constants a, b, c, d, and ρ, the relations

$$a_\emptyset = a; \quad a_{n+1} = \rho d + a_n + (d-1)b_n;$$
$$b_\emptyset = b; \quad b_{n+1} = \rho + c_n;$$
$$c_\emptyset = c; \quad c_{n+1} = d(\rho + b_n).$$

are satisfied by the sequences

$$a_n = \begin{cases} a + h(n)[d(3\rho+b+c)+\rho-b-c]-n\rho, & \text{if } n \text{ is even,} \\ a + h(n-1)[d(3\rho+b+c)+\rho-b-c]-n\rho \\ \quad + d^{(n-1)/2}(d(\rho+b)+\rho-b), & \text{if } n \text{ is odd;} \end{cases}$$

$$b_n = \begin{cases} \rho + 2\rho g(n) + (\rho+b)d^{n/2}, & \text{if } n \text{ is even,} \\ \rho + 2\rho g(n+1) + cd^{(n-1)/2}, & \text{if } n \text{ is odd;} \end{cases}$$

$$c_n = \begin{cases} 2\rho g(n+2) + cd^{n/2}, & \text{if } n \text{ is even,} \\ 2\rho g(n+1) + (\rho+b)d^{(n+1)/2}, & \text{if } n \text{ is odd;} \end{cases}$$

where the function g is defined by

$$g(n) = (d^{n/2} - d)/(d - 1),$$

and the function h is defined by

$$h(n) = (d^{n/2} - 1)/(d - 1).$$

Proof: It is easily seen that $a_\emptyset=a$, $b_\emptyset=b$, and $c_\emptyset=c$.

For the inductive step, first assume that n is odd. Then

$$a_{n+1} = \rho d + a_n + (d-1)b_n \text{ (by definition)}$$
$$= \rho d + a + h(n-1)[d(3\rho d+b+c)+\rho-b-c] - \rho n + d^{(n-1)/2}(d(\rho+b)+\rho-b)$$
$$+ (d-1)[\rho+2\rho g(n+1)+cd^{(n-1)/2}].$$

Simplifying, we get

$$= a + h(n-1)[d(3\rho d+b+c)+\rho-b-c] - \rho(n+1) + d^{(n+1)/2}(3\rho+b+c)$$
$$+ d^{(n-1)/2}(\rho-b-c)$$
$$= a + h(n-1)[d(3\rho d+b+c)+\rho-b-c] - \rho(n+1)$$
$$+ d^{(n-1)/2}[d(3\rho d+b+c)+\rho-b-c]$$
$$= a + h(n+1)[d(3\rho d+b+c)+\rho-b-c] - \rho(n+1),$$

which is the closed form we gave for a_{n+1} when n+1 is even. Furthermore,

$$b_{n+1} = \rho + c_n \text{ (by definition)}$$
$$= \rho + 2\rho g(n+1) + (\rho+b)d^{(n+1)/2},$$

which is the closed form we gave for b_{n+1} for n+1 even. Finally,

$$c_{n+1} = d(\rho+b_n) \quad \text{(by definition)}$$
$$= 2\rho d + 2\rho dg(n+1) + cd^{(n+1)/2}$$
$$= 2\rho g(n+3) + cd^{(n+1)/2},$$

which is the closed form we gave for c_{n+1} when n+1 is even.

Now consider the case when n is even.

$$a_{n+1} = \rho d + a_n + (d-1)b_n \quad \text{(by definition)}$$
$$= \rho d + a + h(n)[d(3\rho+b+c)+\rho-b-c] - \rho n$$
$$+ (d-1)[\rho+2\rho g(n)+(\rho+b)d^{n/2}]$$
$$= a + h(n)[d(3\rho+b+c)+\rho-b-c] - \rho(n+1)$$
$$+ 2\rho d^{n/2} + d^{n/2}(d(\rho+b)-\rho-b)$$
$$= a + h(n)[d(3\rho+b+c)+\rho-b-c] - \rho(n+1) + d^{n/2}(d(\rho+b)+\rho-b),$$

which is the closed form we gave for a_{n+1} with n+1 odd. Furthermore,

$$b_{n+1} = \rho + c_n, \quad \text{(by definition)}$$
$$= \rho + 2\rho g(n+2) + cd^{n/2},$$

which is the closed form we gave for b_{n+1} with n+1 odd. Finally,

$$c_{n+1} = d(\rho+b_n), \quad \text{(by definition)}$$
$$= d[\rho + \rho + 2\rho g(n) + (\rho+b)d^{n/2}]$$
$$= 2\rho[d + dg(n)] + (\rho+b)d^{n/2+1}$$
$$= 2\rho g(n+2) + (\rho+b)d^{n/2+1},$$

which is the closed form we gave for c_{n+1} with n+1 odd.

Q.E.D.

Theorem 2: Under the conditions of Corollary 1, and assuming also that (1) serial α-β search is performed in time equal to the number of leaves visited, and (2) in ρ units of time, a processor can generate f successors of a position, send a message to each

of its f slaves, and receive the f replies, then the total time for Palphabeta to complete is

$$(df)^{\lfloor s/2 \rfloor} + (df)^{\lceil s/2 \rceil} - 1 + h(q)[d(3\rho+(df)^{\lfloor s/2 \rfloor}+(df)^{\lceil s/2 \rceil})+\rho - (df)^{\lfloor s/2 \rfloor}-(df)^{\lceil s/2 \rceil}] - \rho q, \text{ if } q \text{ is even}$$

$$(df)^{\lfloor s/2 \rfloor} + (df)^{\lceil s/2 \rceil} -1 + h(q-1)[d(3\rho+(df)^{\lfloor s/2 \rfloor}+(df)^{\lceil s/2 \rceil})+\rho - (df)^{\lfloor s/2 \rfloor}-(df)^{\lceil s/2 \rceil}] - \rho q + d^{(q-1)/2}[d(\rho+(df)^{\lfloor s/2 \rfloor})+\rho-(df)^{\lfloor s/2 \rfloor}), \text{ if } q \text{ is odd}$$

Proof: Let a_n, b_n, and c_n represent the time required for a processor at distance n from the leaves of the processor tree to search type 1, 2, and 3 positions, respectively. Then these sequences satisfy the relations

$$a_\emptyset = (df)^{\lfloor s/2 \rfloor} + (df)^{\lceil s/2 \rceil} - 1, \quad a_{n+1} = \rho d + a_n + (d-1)b_n;$$

$$b_\emptyset = (df)^{\lfloor s/2 \rfloor}, \quad b_{n+1} = \rho + c_n;$$

$$c_\emptyset = (df)^{\lceil s/2 \rceil}, \quad c_{n+1} = d(\rho + b_n).$$

By substituting the constant expressions for $a_\emptyset$, $b_\emptyset$, and $c_\emptyset$ to find a_q by the formulas given by Lemma 1, we obtain the desired formula.

Q.E.D.

Under conditions of best-first search, the parallel α-β algorithm gives $O(k^{1/2})$ speedup with k processors for searching large lookahead trees. Theorem 3 formalizes this result:

Theorem 3: Suppose that Palphabeta runs on a processor tree of depth $q \geq 1$ and fan-out $f > 1$. Suppose that the lookahead tree to be searched is arranged in best-first order and is of degree df and depth q+s, where $d \geq 1$. Denote by R the time for alphabe-

ta to search this tree, and by P the time for Palphabeta to search the tree. Then

$$\lim_{s \to \infty} R/P = f^{q/2}$$

Proof: The time for the serial algorithm is

$$(df)^{\lfloor (s+q)/2 \rfloor} + (df)^{\lceil (s+q)/2 \rceil} - 1,$$

from Corollary 1. If we divide this quantity by the expression given by Theorem 2 for P, and take the limit as s goes to ∞, we obtain the desired result.

Q.E.D.

7.3 Discussion

The measurements presented in section 5 fall within the range bounded by the theoretically predicted best-first and worst-first speedups. If we take $\gamma_\emptyset \gamma_1$ to be the speedup that would be given by a processor tree of depth two, then the measured speedup for two, three, four, and nine terminal processors is 1.81, 2.34, 2.64, and 4.59 respectively. Theory predicts speedup equal to the number of terminal processors for worst-first ordering. Best-first speedup is predicted to be the square root of the number of terminal processors, or 1.41, 1.73, 2, and 3 respectively.

7.4 Random Order

Under best-first and worst-first ordering of uniform lookahead trees, sibling slaves finish simultaneously because each slave's pruned lookahead tree has the same size and shape. This fact makes it possible to calculate, for a given processor tree and lookahead tree, the exact finishing time for the algorithm Palphabeta. In this section, we analyze the behavior of the slightly weaker algorithm Pbound (no deep cutoffs) under the assumption that terminal values are independent, identically distributed random variables. Restated, this assumption says that no two terminal values are equal, and that any one of the n! orderings of the terminal values is as likely as any other. Although the expected finishing times for sibling slaves are identical, the finishing times themselves may be unequal. Pbound must therefore wait for the last busy slave to finish before assigning the next batch of tasks. For this reason, we will not attempt to calculate the expected finishing time for the parallel algorithm under conditions of random ordering of terminal nodes. We will, however, present a "parallel" version of Knuth's [3] analysis of the serial algorithm under conditions of random order. The analyses of the parallel and serial cases both yield estimates of the expected number of terminal positions examined. Only in the serial case, however, does this estimate yield a direct estimate of the finishing time of the algorithm.

Here is parallel α-β search without deep cutoffs:

```
integer procedure Pbound(position p; integer limit):
begin integer m,i,t,d;
     determine the successors p1, ... ,pdf;
     m := - ∞;
     if depth(p1) < q then fn := Pbound else fn := bound;
     for i := 1 step 1 until d do
     begin t :=     max     -fn(pj,-m);
                (i-1)f<j≤if
          if t > m then m = t;
          if m ≥ limit then goto done;
     end;
     done: Pbound := m;
end;
```

On terminal slaves, Pbound activates the serial algorithm without deep cutoffs:

```
integer procedure bound(position p; integer limit):
begin integer m,i,t,d;
     determine the successors p1, ... ,pd;
     if d = ∅ then bound = staticvalue(p) else
     begin m := - ∞;
          for i := 1 step 1 until d do
          begin t :=  - bound(pi,-m);
               if t > m then m = t;
               if m ≥ limit then goto done;
          end;
     done: bound := m;
     end;
end;
```

Let T(d,h) be the number of terminal positions examined by bound in a tree of depth h and degree d with randomly distributed terminal values. Knuth [3] establishes that T(d,h) satisfies

$$c_1(d)\,r_1^h \le T(d,h) \le c_2(d)\,r_2^h,$$

where c_1 and c_2 depend on d but not h, and r_1 and r_2 satisfy $c_3 d/\ln d \le r_1$ and $r_2 \le c_4 d/\ln d$, for certain constants c_3 and c_4. As part of the proof of this result, the inequality

$$(1)\quad \Big(\sum_{1\le i\le d} \Big(\sum_{1\le j\le d} i^{-t((j-1)/2d)} \Big)^{s/t} \Big)^{1/s} \le c_4 d/\ln d$$

is established for a certain choice of s, t satisfying $1/s + 1/t = 1$.

We begin by presenting a lemma due to Knuth and then adapting it to our own use.

Lemma 2: Suppose that $Y_{1,1}, \ldots, Y_{i-1,d}$ and $Z_1, \ldots, Z_{j-1}$ are independent sequences of $(i-1)d$ and $(j-1)$ independent identically distributed random variables.

Then

$$\frac{1}{\binom{i-1 + (j-1)/d}{i-1}}$$

is the probability that

$$(2)\qquad \max_{1\le k<i} (\min(Y_{k,1}, \ldots, Y_{k,d})) < \min_{1\le k<j} Z_k$$

Proof: If $i = 1$, the left hand side is $-\infty$. If $j = 1$, the right hand side is $+\infty$. In both cases, the probability that the relation holds is 1.

Assume then that $i,j > 1$. Consider the minimum element Y_{k_1,t_1}, over all $1\le k_1<i$ and $1\le t_1\le d$. The probability that it is less than $\min_{1\le k<j} Z_k$ is

$$\frac{(i-1)d}{((i-1)d + j-1)} .$$

Removing the elements $Y_{k_1,1}, \ldots, Y_{k_1,d}$ from consideration, we consider the minimum of the remaining Ys on the left of (2), say Y_{k_2,t_2}. The probability that Y_{k_2,t_2} is less than the right-hand side of (2) is

$$\frac{(i-2)d}{((i-2)d + j-1)},$$

and so on. Hence (2) happens exactly when $Y_{k_1,t_1} <$ RHS and $Y_{k_2,t_2} <$ RHS and ... and $Y_{k_{i-1},t_{i-1}} <$ RHS, so (2) has probability

$$\frac{(i-1)d(i-2)d \ldots 1d}{((i-1)d + j-1)((i-2)d + j-1)\ldots(d + j - 1)}$$

$$= \frac{(i-1)!((j-1)/d)!}{(i-1 + (j-1)/d)!}$$

$$= \frac{1}{\binom{i-1 + (j-1)/d}{i-1}}$$

Q.E.D.

Lemma 3 (Corollary to Lemma 2): If $Y_{1,1}, \ldots, Y_{(i-1)f,df}$ and $Z_1, \ldots, Z_{(j-1)f}$ are independent sequences of $((i-1)f)df$ and $(j-1)f$ independent identically distributed random variables, then the probability p_{ij} that

$$\max_{1\leq k\leq (i-1)f} \min_{1\leq m\leq df} Y_{k,m} < \min_{1\leq k\leq (j-1)f} Z_k$$

is

$$p_{ij} = \frac{1}{\binom{(i-1)f + (j-1)/d}{(i-1)f}}$$

Proof: This Lemma is simply Lemma 2 with a change of variables.

Substitute: df for d,
$(i-1)f + 1$ for i,
$(j-1)f + 1$ for j.

Q.E.D.

Since the simple formula k^x is always within 12% of

$$\binom{k-1+x}{k-1},$$

for $\emptyset \leq x \leq 1$ and k a positive integer [3], we will approximate p_{ij} by

(3) $p_{ij} = ((i-1)f + 1)^{-(j-1)/d}$.

Theorem 4: Let T(d,f,h) be the expected number of terminal positions examined by the parallel α-β procedure without deep cutoffs on a processor tree of degree f and height h in a random uniform lookahead tree of degree df and height h. Then

$$T(d,f,h) < f^h \, c(d,f) \, r(d,f)^h,$$

where r(d,f) is the largest eigenvalue of the matrix

$$M_{d,f} = \begin{pmatrix} \sqrt{p_{11}} & \sqrt{p_{12}} & \cdots & \sqrt{p_{1d}} \\ \sqrt{p_{21}} & \sqrt{p_{22}} & \cdots & \sqrt{p_{2d}} \\ \cdot & \cdot & & \cdot \\ \cdot & \cdot & & \cdot \\ \cdot & \cdot & & \cdot \\ \sqrt{p_{d1}} & \sqrt{p_{d2}} & \cdots & \sqrt{p_{dd}} \end{pmatrix}$$

and c(d,f) is a constant. The quantities p_{ij} in $M_{d,f}$ were defined in Lemma 3.

Proof: As before, assign Dewey decimal names to the positions of the lookahead tree. Define the functions

$$G(n) = \lfloor (n-1)/f \rfloor + 1$$

and

$$H(n) = \lfloor (n-1)/f \rfloor f + 1.$$

The nth successor position is a member of the G(n)th batch of successor positions to be assigned to slaves. The first member of that batch is the H(n)th successor position.

When Pbound examines position $a_1 \ldots a_{m-1}$, "limit" is

$$\min_{1 \le k < H(a_{m-1})} \text{negamax}(a_1 \ldots a_{m-2} k),$$

so its successor $a_1 \ldots a_m$ is examined if and only if $a_1 \ldots a_{m-1}$ is examined and

$$-\min_{1 \le k < H(a_m)} \text{negamax}(a_1 \ldots a_{m-1} k)$$

$$< \min_{1 \le k < H(a_{m-1})} \text{negamax}(a_1 \ldots a_{m-2} k)$$

Abbreviate this inequality by P_m. Then $a_1 \ldots a_h$ is examined if and only if P_1, P_2, ..., and P_h hold. P_m holds with probability p_{ij}, where $i = G(a_{m-1})$ and $j = G(a_m)$. Furthermore, P_m is a function of the terminal values

$$\text{staticvalue}(a_1 \ldots a_{m-2} j k b_{m+1} \ldots b_n)$$

for

$1 \le j < H(a_{m-1})$ and all $\emptyset \le k, b \le df$

or

$j = H(a_{m-1})$ and $1 \le k < H(a_m)$ and all $\emptyset \le b \le df$.

Therefore P_m is independent of $P_1, \ldots, P_{m-2}$. Let x be the probability that $a_1 \ldots a_h$ is examined. Then we have (assuming, without loss of generality, that h is odd)

$$x < p_{G(a_1)G(a_2)} p_{G(a_3)G(a_4)} \cdots p_{G(a_{h-2})G(a_{h-1})}$$

and

$$x < p_{G(a_2)G(a_3)} p_{G(a_4)G(a_5)} \cdots p_{G(a_{h-1})G(a_h)}.$$

Thus

$$x < \sqrt{p_{G(a_1)G(a_2)}} \sqrt{p_{G(a_2)G(a_3)}} \cdots \sqrt{p_{G(a_{h-1})G(a_h)}}$$

(for even or odd h).

Hence the expected number of terminal positions examined is less than

$$\sum_{1\le a_1,\dots,a_h\le d} f^{\sqrt{p}_{G(a_1)G(a_2)}\sqrt{p}_{G(a_2)G(a_3)}\cdots\sqrt{p}_{G(a_{h-1})G(a_h)}}$$
$$= f^h \sum_{1\le a_1,\dots,a_h\le d} \sqrt{p}_{a_1a_2}\sqrt{p}_{a_2a_3}\cdots\sqrt{p}_{a_{h-1}a_h}$$
$$= f^h \sum_{1\le a_1\le d}\ \sum_{1\le a_2\le d} \sqrt{p}_{a_1a_2} \sum_{1\le a_3\le d} \sqrt{p}_{a_2a_3}\cdots \sum_{1\le a_h\le d} \sqrt{p}_{a_{h-1}a_h},$$

which is $f^h c_{1,h}$, where the sequences $c_{i,n}$, $1 \le i \le d$, are defined by

$$c_{i,\emptyset} = 1 \text{ for } 1 \le i \le d$$
$$(4)\quad c_{i,n+1} = \sum_{1\le j\le d} \sqrt{p}_{ij} c_{j,n}, \quad \text{for } 1 \le i \le d.$$

Now define generating functions C_i, for $1 \le i \le d$, as follows:

$$C_i(z) = \sum_{n\ge\emptyset} c_{i,n} z^n$$

Then (4) is equivalent to

$$C_i(z) - 1 = \sum_{1\le j\le d} \sqrt{p}_{ij} z C_j(z) \text{ for } 1 \le i \le d.$$

Set $C(z) = (C_1(z)\dots C_d(z))^T$, and define the matrix

$$Z = \begin{pmatrix} z\sqrt{p}_{11} & z\sqrt{p}_{12} & \cdots & z\sqrt{p}_{1d} \\ z\sqrt{p}_{21} & z\sqrt{p}_{22} & \cdots & z\sqrt{p}_{2d} \\ \vdots & \vdots & & \vdots \\ z\sqrt{p}_{d1} & z\sqrt{p}_{d2} & \cdots & z\sqrt{p}_{dd} \end{pmatrix}.$$

Then $(-1\ -1\ \dots\ -1)^T = (Z-I)C$, where I is the identity matrix. By Cramer's rule, $C_1(z) = U(z)/V(z)$, where U and V are polynomials defined by

$$U(z) = \det \begin{pmatrix} -1 & z\sqrt{p_{12}} & \cdots & z\sqrt{p_{1d}} \\ -1 & z\sqrt{p_{22}} & -1 \cdots & z\sqrt{p_{2d}} \\ \cdot & \cdot & & \cdot \\ \cdot & \cdot & & \cdot \\ \cdot & \cdot & & \cdot \\ -1 & z\sqrt{p_{d2}} & \cdots & z\sqrt{p_{dd}} \; -1 \end{pmatrix}$$

and $V(z) = \det(Z - I)$.

Note that r is an eigenvalue of $M_{d,f}$ if and only if $1/r$ is a root of $V(z)$. Since $C_1(z)$ is a quotient of polynomials, it can be represented [7] as

$$C_1(z) = \sum_{1 \le k \le n} G_k(1/(z-B_k)),$$

where $B_1, \ldots, B_n$ are the distinct roots of V, and $G_1, \ldots, G_n$ are polynomials such that the degree of G_i is the multiplicity of B_i.

Every matrix of real, positive elements possesses one positive eigenvalue of multiplicity one that is strictly larger, in absolute value, than all the other eigenvalues [8]. $M_{d,f}$ is positive; let r_i, $i = 1, \ldots, n$, be its eigenvalues, with r_1 the largest. If the eigenvalues $r_1 = 1/B_1, \ldots, r_n = 1/B_n$ of $M_{d,f}$ are distinct, we have

$$C_1(z) = E + \sum_{1 \le i \le d} e_i/(z-1/r_i) = E + \sum_{1 \le i \le d} -e_i r_i/(1-zr_i),$$

$$= E + \sum_{n \ge \emptyset} \sum_{1 \le i \le d} -e_i r_i r_i^n z^n.$$

Since r_1 is the largest of the r_i, $c_{1,h} = O(r_1^h)$. If the eigenvalues of $M_{d,f}$ are not distinct, the representation of $C_1(z)$ involves polynomials of degree higher than one. Even so, the linear term containing r_1 still dominates.

Q.E.D.

Lemma 4: Suppose the real-valued sequence $a_1, a_2, a_3, \ldots$ obeys

the rule

$$a_{m+n} \leq a_m + a_n \qquad m,n = 1,2,3,\ldots$$

Then the sequence $a_1/1$, $a_2/2$, $a_3/3$, ... either diverges to $-\infty$ or converges.

Proof: It suffices to consider the case where the lim inf, α, of the second sequence is finite. Let $\epsilon > 0$, and choose m such that $a_m/m < \alpha + \epsilon$. Since every integer n can be expressed as $n = qm + r$ with $0 \leq r < m$, we have

$$a_n = a_{qm+r} \leq qa_m + a_r.$$

hence

$$\frac{a_n}{n} = \frac{a_{qm+r}}{qm+r} \leq \frac{qa_m+a_r}{qm+r} = \frac{a_m}{m}\,\frac{qm}{qm+r} + \frac{a_r}{n}$$

hence

$$\frac{a_n}{n} < (\alpha+\epsilon)\left(\frac{qm}{qm+r}\right) + \frac{a_r}{n}$$

hence $\limsup_{n \to +\infty} a_n/n = \alpha$

and so $\lim_{n \to +\infty} a_n/n = \alpha$.

Q.E.D.

Definition: Let T(d,h) be the number of terminal positions examined by a given algorithm in a lookahead tree of degree d and height h. The branching factor of T is

$$\lim_{h \to \infty} T(d,h)^{1/h},$$

if the limit exists.

Theorem 5: Let T(d,f,h) be as defined in Theorem 4. Then the branching factor of T,

(5) $$B = \lim_{h \to \infty} T(d,h,f)^{1/h},$$

satisfies

$$\frac{dfc_3}{\log df} \leq B \leq \frac{dfc_4}{\log d}$$

for certain constants $c_3, c_4 > 0$ independent of d and f.

Proof: Since $T(d,h_1,f)T(d,h_2,f)$ is the number of positions that would be examined by Pbound if "limit" were set to $+\infty$ for all positions at height h_1 in a lookahead tree of depth $h_1 + h_2$, we have $T(d,h_1+h_2,f) \leq T(d,h_1,f)T(d,h_2,f)$. Hence by Lemma 4 applied to log T(d,h,f), the limit in (5) exists.

Lower bound: The parallel α-β routine without deep cutoffs, Pbound, examines at least as many nodes as its serial counterpart, bound, since each "limit" in the parallel case is greater than its counterpart in the serial case. As mentioned above, Knuth has proven that the branching factor of the number of terminal positions examined by bound in a tree of depth h and degree df is greater than or equal to $dfc_3/\log(df)$.

Upper bound: Let s and t be positive real numbers with $1/s + 1/t = 1$, and let E be an eigenvalue of the matrix $A = (a_{ij})$. Suppose $Ax = Ex$. Then

$$|E| \left(\sum_i |x_i^s| \right)^{1/s} = \left(\sum_i \left| \sum_j a_{ij} x_j \right|^s \right)^{1/s}$$

$$\leq \left(\sum_i \left(\sum_j |a_{ij}^t| \right)^{s/t} \right)^{1/s} \left(\sum_j |x_j^s| \right)^{1/s}, \text{ by Hölder's inequality;}$$

hence $$|E| \leq \left(\sum_i \left(\sum_j |a_{ij}^t| \right)^{s/t} \right)^{1/s}.$$

We will use this inequality to show that $r(d,f) \leq c_4 d/\log d$, for a certain constant c_4 and for $r(d,f)$ as defined in Theorem 4. Let $a_{ij} = \sqrt{p_{ij}}$, $E = r(d,f)$, and use approximation (3) for p_{ij}. For all s and t such that $1/s + 1/t = 1$, we have

$$r(d,f) \leq \Big(\sum_{1\leq i\leq d} \Big(\sum_{1\leq j\leq d} ((i-1)f+1)^{-t((j-1)/2d)} \Big)^{s/t} \Big)^{1/s}$$

$$\leq \Big(\sum_{1\leq i\leq d} \Big(\sum_{1\leq j\leq d} i^{-t((j-1)/2d)} \Big)^{s/t} \Big)^{1/s}$$

$\leq c_4 d/\ln d$, for a suitable s, t, and c_4, by (1).

Theorem 4 and this upper bound for $r(d,f)$ give us the desired upper bound on the branching factor.

Q.E.D.

Theorem 5 deals with lookahead trees that are the same depth as the processor tree that searches them. In Theorem 6, we extend the analysis to the more general situation in which the lookahead tree can be deeper than the processor tree.

Theorem 6: The expected number of terminal positions examined by Pbound in a random uniform game tree of degree df and height q+s, evaluated by a processor tree of degree d and height q, where $d \geq 2$, $q \geq 0$ and $f \geq 1$, is asymptotically less than

$$c_5(d,f)\ f^q\ r(d,f)^q\ r_1(df)^s,$$

where $r(d,f)$ was given upper and lower bounds in Theorem 5, and r_1 satisfies

$$\frac{dfc_3}{\log(df)} \leq r_1(df) \leq \frac{dfc_4}{\log(df)}$$

for the constants c_3 and c_4 appearing in Theorem 5, and where $c_5(d,f)$ is a constant independent of q and s.

Proof: Since the values of the positions assigned for evaluation to leaf processors have random values, Theorem 4 implies that the number of these positions P satisfies

$$P < c(d,f)\ f^q\ r(d,f)^q.$$

Theorem 5 tells us that r(d,f) satisfies

$$\frac{dc_3}{\log(df)} \leq r(d,f) \leq \frac{dc_4}{\log d}$$

If we set "limit" at level q of the lookahead tree to $+\infty$, then each leaf processor evaluating one position at level q would examine less than $c_2(df)\,r_1(df)^s$ terminal positions [3], where $r_1(df)$ satisfies

$$\frac{dfc_3}{\log(df)} \leq r_1(df) \leq \frac{dfc_4}{\log(df)}$$

and $c_2(df)$ is a constant independent of s.

The result follows with $c_5(d,f)$ set to $c(d,f)c_2(df)$.

Q.E.D.

7.5 Discussion of Theorem 6

In searching a lookahead tree of degree df and height q + s, the serial algorithm examines, on the average, at least

$$c_1 \left(\frac{dfc_3}{\log(df)} \right)^{q+s}$$

terminal nodes, where c_1 depends only on df and c_3 is a constant. The parallel algorithm examines less than

$$(6) \qquad c_5 \, f^q \left(\frac{dc_4}{\log d} \right)^q \left(\frac{dfc_4}{\log(df)} \right)^s$$

terminal nodes on the average.

Under best-first and worst-first ordering, the finishing time for Palphabeta can be accurately estimated by dividing the amount of work to be done by the number of workers (terminal processors). This method of estimation is somewhat optimistic when applied to Pbound or the Tree-Splitting Algorithm under random ordering, because in Pbound a master waits until all successors in a batch of f have been evaluated before assigning the next batch, and in both Pbound and the Tree-Splitting Algorithm a master waits until the last successor is evaluated before receiving another position.

While we await more powerful methods, let us make the estimate anyway. Dividing (6) by the number of terminal processors, f^q, gives us

$$c_5 \left(\frac{dc_4}{\log d} \right)^q \left(\frac{dfc_4}{\log(df)} \right)^s$$

as the finishing time, and so the speedup would be at least

$$\frac{c_1}{c_5}\left(\frac{c_3}{c_4}\right)^{s+q} f^q \left(\frac{\log d}{\log(df)}\right)^q .$$

The factor $(c_3/c_4)^{s+q}$ appears in this expression because we used an optimistic bound for the serial algorithm and a pessimistic bound for the parallel algorithm. We can most likely remove it. The resulting expression is of order

$$\left(\frac{f \log d}{\log d + \log f}\right)^q$$

Recall that speedup under worst-first ordering is of order

$$f^q,$$

and by Theorem 3, speedup under best-first ordering is of order

$$f^{q/2}.$$

Speedup under both random and best-first ordering is clearly less than speedup under worst-first ordering. Speedup under random ordering is asymptotically greater than speedup under best-first ordering whenever

$$\frac{f \log d}{\log d + \log f} > \sqrt{f} ;$$

i.e. whenever

$$d > f^{\left(\frac{1}{\sqrt{f} - 1}\right)} .$$

8. APPENDIX - SOME OPTIMIZATIONS OF SERIAL ALPHA-BETA SEARCH

In this appendix we propose three optimizations of the serial α-β algorithm.

8.1 Falphabeta

The first optimization, called falphabeta for "fail-soft alpha-beta search", is completely riskless in the sense that it never searches more nodes than alphabeta. Although it requires a slight constant overhead, it results in a slight expected speedup whenever an initial window other than $(-\infty,+\infty)$ is used. Here is falphabeta:

```
integer procedure falphabeta(position p,integer α,integer β):
begin integer m,i,t,d;
     determine the successor positions p1,...,pd;
     if d = ∅ then falphabeta := staticvalue(p) else
     begin
          m := - ∞;
          for i := 1 to d do
          begin t := -falphabeta(pi,-β,-max(m,α));
               if t > m then m := t;
               if m ≥ β then go to done;
          end;
     done: falphabeta := m;
     end;
end;
```

Falphabeta differs from alphabeta only in that m has been initialized to $-\infty$ instead of α. In order to keep this change from affecting the third actual parameter to the recursive call to falphabeta, "-m" is changed to "-max(m,α)". The computational overhead of repeatedly computing the maximum of m and α is the only added expense of falphabeta. The value returned by the call

to the original α-β procedure, alphabeta(p,α,β), obeys the following relation with respect to the true negamax value of a search tree:

If alphabeta $\leq \alpha$, then negamax(p) $\leq \alpha$,
if alphabeta $\geq \beta$, then negamax(p) $\geq \beta$,
if $\alpha <$ alphabeta $< \beta$ then negamax(p) = alphabeta.

Falphabeta obeys a stronger relation:

Theorem 1: If p is the root node of a lookahead tree, and if α and β are integers satisfying $\alpha < \beta$, then the value falphabeta returned by falphabeta(p,α,β) satisfies:

If falphabeta $\leq \alpha$, then negamax(p) $\leq$ falphabeta,
if falphabeta $\geq \beta$, then negamax(p) $\geq$ falphabeta,
if $\alpha <$ falphabeta $< \beta$ then negamax(p) = falphabeta.

Proof: The relations clearly hold if p is a terminal node. Assume for the induction step that the relations hold for any tree of height k or less. Let p be the root of a tree of height $k+1$. Let $p_1, \ldots, p_d$ be the successors of p. Each p_i is the root of a tree of height k or less.

1) If

$$\text{falphabeta}(p,\alpha,\beta) \leq \alpha,$$

then for all $1 \leq i \leq d$, we have

$$\text{falphabeta}(p_i,-\beta,-\alpha) \geq -\alpha.$$

By the induction hypothesis, we have

$$\text{negamax}(p_i) \geq \text{falphabeta}(p_i,-\beta,-\alpha).$$

Hence

$\max_i$ -negamax(p_i) $\leq$ $\max_i$ -falphabeta(p_i,-β,-α).

Hence negamax(p) $\leq$ falphabeta(p,α,β).

2) If falphabeta(p,α,β) $\geq$ β, then there exists i such that

-falphabeta(p_i,-β,-α') = falphabeta(p,α,β) $\geq$ β,

for some α' such that $\alpha \leq \alpha'$. By the induction hypothesis, we may conclude that

negamax(p_i) $\leq$ falphabeta(p_i,-β,-α').

Hence negamax(p) = $\max_i$ -negamax(p_i) $\geq$ falphabeta(p,α,β).

3) If α < falphabeta(p,α,β) < β, then let i be the smallest integer such that

-falphabeta(p_i,-β,-α') = falphabeta(p,α,β),

for some α' such that falphabeta(p,α,β) > $\alpha' \geq \alpha$. Hence

-β < falphabeta(p_i,-β,-α') < -α'.

Therefore, by the induction hypothesis,

negamax(p_i) = falphabeta(p_i,-β,-α') = -falphabeta(p,α,β).

Since negamax(p) = negamax(p_i), we have

negamax(p) = falphabeta(p,α,β);

Q.E.D.

Theorem 1 implies that falphabeta can give a tighter bound than alphabeta on the true value of the tree when it fails high or low. Falphabeta "fails softer" than alphabeta. The extra information that falphabeta gives can be used in two ways. First, this information is useful whenever the common wisdom "start with a tight window" is followed. If the tight window (α,β) causes the search to fail, the penalty of doing the entire search over again must be paid. With normal α-β search, this second search

must be done with the window $(-\infty,\alpha)$ (if the original search failed low) or $(\beta,+\infty)$ (if the original search failed high). Falphabeta reduces this penalty: A low fail will sometimes return a number $k < \alpha$, and the second search can be started with the tighter window $(-\infty,k)$. We can expect a similar saving when a high fail occurs.

We need two definitions to explain the second use of falphabeta. Staged iteration evaluates a lookahead tree to depth N by first searching to depths 2, 3, ..., N-1. After each stage, the principal line (the path the game would take if each player played optimally) is saved. The next stage begins its depth-first search by descending to the end of this path; whenever a node on the principal line is visited, its principal child is examined first. Staged iteration provides very reliable best-first move ordering at type-one nodes, so it actually decreases the number of nodes searched in chess programs.

Forward pruning, as opposed to α-β pruning, which is a form of backward pruning, cuts off a node of a tree before fully investigating any of its siblings. It is obvious that forward pruning can provide enormous savings in tree search. Unfortunately, forward pruning is very risky. No one has yet discovered how to perform forward pruning without occasionally pruning away the best move. (The very best chess programs do not perform forward pruning.) One of the reasons that forward pruning has not been successfully implemented is that when a poor move is evaluated after a better move, alphabeta assigns both the same score (except when the poor move is with two moves of the

terminal node that produces the poor score). Falphabeta sometimes gives the poor move a more appropriate value, so it may provide a basis for reliably pruning the move during the next stage of a staged iteration.

8.2 Lalphabeta

When alphabeta is recursively called on the last successor p_d, of the root of the entire tree, p, the current value $-\beta$ $(-\infty)$ is passed as formal parameter α. Suppose that -m-1 is passed instead. If p_d is not the best move, then negamax(p_d) $\geq$ -m, and alphabeta(p_d,-m-1,-m) fails high as before. If p_d is the best move, then negamax(p_d) $\leq$ -m-1, and so alphabeta(p_d,-m-1,-m) fails low instead of succeeding. Nevertheless, the algorithm can still conclude that p_d is the best move, since its negamax value has been established to be lower than any other. The modified algorithm does not discover the value of the best move when that move is evaluated last. However, it still determines which move is best. This slight reduction in information can buy a time savings, since the evaluation of p_d has a very narrow window.

The new algorithm will be called lalphabeta, short for "last-move-with-minimal-window alpha-beta search".

```
integer procedure lalphabeta (position p,
                  integer α, integer β):
    begin integer m,i,t,d;
        determine the successor positions p1,...,pd;
        if d = 0 then lalphabeta := staticvalue(p) else
        begin m := α;
            for i := 1 to d-1 do
            begin t := -alphabeta(pi,-β,-m);
                if t > m then m := t;
                if m ≥ β then go to done;
            end;
```

```
            t := -alphabeta(p_d,-m-1,-m);
            if t > m then m := t;
        done: lalphabeta := m;
        end;
    end;
```

Lalphabeta provides an elegant solution to the forced-move problem: Programmers writing their first game-playing program often find to their amusement that alphabeta conducts a full-scale search even though only one move is available to the computer. Lalphabeta searches the one available move with the window $(\infty - 1,\infty)$. Besides greatly speeding up the search, lalphabeta actually performs useful work in this case: It decides if it should resign!

8.3 Palphabeta

The third optimization, called palphabeta because it is called only on nodes along the principal line, is a generalization of lalphabeta, and profits from falphabeta, but carries with it the risk that in certain cases more nodes will be examined.

```
integer procedure palphabeta(position p)
begin integer m,i,t,d;
    generate the successors p_1,...,p_d.
    if d = ∅ then palphabeta := staticvalue(p) else
    begin
        m = -palphabeta(p_1);
        for i := 2 to d do
        begin t = -falphabeta(p_i,-m-1,-m);
            if t > m then m := -falphabeta(p_i,- ∞,-t)
        end;
        palphabeta := m;
    end;
end.
```

If palphabeta evaluates the best move first at type one nodes, then all of the other subtrees are searched with a minimal window. On the other hand, every subtree that is better than its older siblings must be searched twice, resulting in more work. The first search, conducted with the minimal window, discovers that the subtree is the new best one, and really should not have been searched with the minimal window after all. The second search discovers the true value. It is important that the best move be evaluated first with high enough probability that the savings outweigh the penalties. Staged iteration can generate the best move first with high probability. If the principal line established for the (N-1)th stage is a prefix of the principal line for the Nth stage, then at the Nth stage virtually the entire tree is searched with a minimal window.

8.4 Measurements

To measure the improvement due to lalphabeta and palphabeta, four checkers games were played, during which the program made 46 moves. Each move selection was repeated six times, one for each of the six algorithms: alphabeta, lalphabeta, palphabeta, salphabeta, slalphabeta, and spalphabeta. Alphabeta, lalphabeta, and palphabeta have already been defined, and were done without staging. Salphabeta, slalphabeta, and spalphabeta are the staged versions of these three algorithms. During each of the 46*6 move selections, the number of nodes visited was counted, providing 46 values for alphabeta, lalphabeta, palphabeta, salphabeta, slalphabeta, and spalphabeta, and hence 46 values for the five

derived quantities alphabeta/salphabeta, lalphabeta/alphabeta, palphabeta/alphabeta, slalphabeta/salphabeta, and spalphabeta/salphabeta.

Table 1 shows statistics for alphabeta/salphabeta. Checkers, unlike chess, does not profit from staging, possibly due to checker's smaller branching factor. On the average, ab searched only 81% as many nodes as salphabeta.

Table 1: alphabeta/salphabeta

Minimum	Ø.Ø19
Maximum	2.768
Average	Ø.8Ø8
Standard Deviation	Ø.462

Table 2 gives statistics for lalphabeta/alphabeta, palphabeta/alphabeta, slalphabeta/salphabeta, and spalphabeta/salphabeta.

Table 2:

	lalphabeta/alphabeta	palphabeta/alphabeta
Minimum	Ø.881	Ø.666
Maximum	1.ØØØ	5.75Ø
Average	Ø.987	1.163
Standard Deviation	Ø.Ø24	Ø.868

	slalphabeta/salphabeta	spalphabeta/salphabeta
Minimum	Ø.899	Ø.696
Maximum	1.ØØØ	2.174
Average	Ø.988	Ø.96Ø
Standard Deviation	Ø.Ø23	Ø.227

As expected, staged iteration was crucial to making palphabeta work at all; without staging, palphabeta actually searched more nodes than alphabeta. However, the measurements of spalphabeta (palphabeta with staging) are disappointing. Spalphabeta searched only four percent fewer nodes than salphabeta. Since savings from starting with a narrow window (an optimization that could be used in place of palphabeta or spalphabeta) are on the order of 20 percent [2], palphabeta and spalphabeta are probably not to be recommended.

Lalphabeta and slalphabeta, on the other hand, are unqualified (albeit small) successes. On the average, each searches about one percent fewer nodes than the corresponding standard algorithm. Although this improvement is not great, the optimization is clearly a good bargain, since its space overhead is insignificant and its time overhead is zero. Lalphabeta is never slower than alphabeta and slalphabeta is never slower than salphabeta. Therefore, every game-playing program that uses α-β search should use some form of lalphabeta.

9. ACKNOWLEDGMENTS

The authors gratefully acknowledge the help and ideas offered by Karl Anderson, Sharon Lawless, Will Leland, Marvin Solomon, and Larry Travis.

1Ø. REFERENCES

[1] H. J. Berliner, "A chronology of computer chess and its literature," Artificial Intelligence 1Ø, pp. 2Ø1-214 (April 1978).

[2] G. M. Baudet, The Design and Analysis of Algorithms for Asynchronous Multiprocessors, Department of Computer Science, Carnegie-Mellon University (April 1978).

[3] D. E. Knuth and R. W. Moore, "An Analysis of Alpha-Beta Pruning," Artificial Intelligence 6, 4, pp. 293-326 (Winter 1975).

[4] A.L. Samuel, "Some Studies in Machine Learning Using the Game of Checkers, II - Recent Progress," IBM Journal of Research and Development, pp. 6Ø1-617 (November 1967).

[5] S.G. Akl, D.T. Barnard, and R.J. Doran, "Searching Game Trees in Parallel," Technical Report No. 79-87, Queen's University Department of Computing and Information Science (November 1979).

[6] M. Solomon and R. Finkel, "The Roscoe Distributed Operating System," Proceedings of the Seventh Symposium on Operating Systems Principles, pp. 1Ø8-114 (December 1979).

[7] R. Nevanlinna and V. Paatero, Introduction to Complex

Analysis, Addison-Wesley (1969).

[8] Oskar Perron, "Zur Theorie der Matrices," Math. Ann. 64, pp. 248-263 (1907).