Searching Game Trees in Parallel*

Igor Steinberg
igor@cs.wisc.edu

and
Marvin Solomon
solomon@cs.wisc.edu

Abstract

We present a new parallel game-tree search algorithm. Our approach classifies a processor's available work as either mandatory (necessary for the solution) or speculative (may be necessary for the solution). Due to the nature of parallel game tree search, it is not possible to keep all processors busy with mandatory work. Our algorithm *ER* allows potential speculative work to be dynamically ordered, thereby reducing starvation without incurring an equivalent increase in speculative loss. Measurements of ER's performance on both random trees and trees from an actual game show that at least 16 processors can be applied profitably to a single search. These results contrast with previously published studies, which report a rapid drop-off of efficiency as the number of processors increases.

1. Introduction

Game playing programs require a great deal of computation and would thus appear to be an ideal candidate for parallel implementation. In fact, current champion chess-playing programs generate alternative moves and evaluate board positions using parallel algorithms, but the central part of game planning—game tree search—has thus far proved resistant to efficient parallel implementation. Berliner and Ebeling [Berliner1988] claim that it is difficult to achieve speedups greater than the square root of the number of processors for chess.

The alpha-beta algorithm, a very effective serial algorithm for game tree search, uses information obtained during search to reduce the amount of search that must be performed. A parallel algorithm that simply partitions the tree amongst the available processors will search a much greater portion of the tree than serial alpha-beta, resulting in low efficiency. A reasonable alternative approach is to perform work only when it becomes known to be necessary. Unfortunately, this scheme leaves many processors idle; a situation known as starvation.

In order to achieve high efficiency, a parallel algorithm must perform *speculative work*—work that is performed before it is proved necessary. Our key finding is that the alpha beta algorithm is ill suited to gather the information necessary for an effective selection of the most promising speculative work. We present a new algorithm *ER* which appears better suited to parallel implementation. We describe the algorithm and an experimental implementation, and present experimental results on its effectiveness.

Section 2 describes game-tree search in more detail and presents the alpha-beta algorithm and an analysis of its effectiveness. Readers familiar with literature on game-playing programs may skip this

*This work supported in part by the National Science Foundation under grant DCR-8521228.

section. Section 3 discusses the problem of searching game trees in parallel. In section 4, we briefly survey previous work on parallel game-tree search. Section 5 presents $\mathcal{ER}$. While $\mathcal{ER}$ is not necessarily better than alpha-beta on a serial processor, it appears must better suited for parallel implementation. Section 6 describes an experimental implementation of parallel $\mathcal{ER}$ on a Sequent multiprocessor. Section 7 presents experimental results from this implementation. We conclude in section 8 with a summary of our results and directions for future research.

2. Game Tree Search

To select an optimal move in a two-person game, the computer constructs a *game tree*, in which each node represents a configuration (such as a board position). The root of the tree corresponds to the current position, and the children of a node represent configurations reachable in one move. Usually, the entire tree is too large to search, so the analysis is truncated at some level (or *ply*) and the resulting *terminal nodes* are evaluated by a heuristic procedure called a *static evaluator*. This procedure assigns a numeric value to the position, where larger values represent positions more favorable to the player whose turn it is to play. We will assume that the value of a position from the point of view of one player is the negative of its value from the point of view of the other. The *principal variation* is defined as the path from the root on which each player plays optimally. It may be found as follows: Label each interior node with the maximum of the negatives of the values of its children. Then the principal variation is the path that goes from each node to its lowest-valued child, and the value at the root is the value of terminal position reached by this path (from the point of view of the first player). This labeling procedure is called the *negmax procedure* [Knuth1975]. The algorithm may be programmed as

```
function negmax(p:position):integer;
var m,i,d:integer;
begin
  determine the child positions p1,...,pd
  if d=0 then return(static_evaluator(p))
  else begin
    m:= -∞;
    for i:= 1 to d do
      m:= max(m,-negmax(pi));
    return(m);
  end;
end.
```

Figure 1 shows a game tree for tic-tac-toe. We have labeled terminal nodes with -1, 0, or +1 according to whether the position represents a loss, tie, or win for the current player. The negmax values for the other nodes are also indicated. The value 0 at the root indicates that the game will end in a draw if each player plays optimally. The principal variation, which leads to this draw, is indicated by bold lines.

2.1. Alpha-beta Search

The negmax procedure performs depth-first search of the game tree and assigns a value to each node. Alpha-beta pruning is a technique that determines the root's negmax value while avoiding examination of a

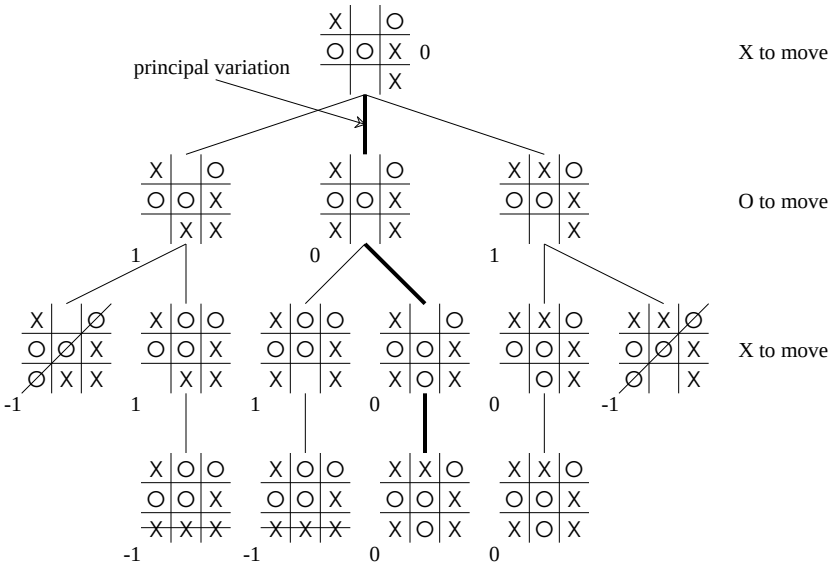

Figure 1. A Tic-tac-toe game tree.

substantial portion of the tree. Alpha-beta pruning requires two bounds (α,β) that are used to restrict the search. Taken together, these bounds are known as the *alpha-beta window*. If the initial window is $(-\infty,\infty)$, the alpha-beta procedure will determine the correct root value [Knuth1975].

The *alpha-beta* procedure avoids many nodes in the tree by achieving *cutoffs*. Figure 2 shows two types of cutoffs known as *shallow* and *deep* cutoffs. In Figure 2(a), the value of node *A* must be at least 7, since one of its children has been found to have value −7. The value of *B* can affect *A* only if *B*'s value is less than −7. But since *B*'s left child has value 5, *B* must be at least -5. Therefore, *B*'s subtree need not be searched any further if only the negmax value of *A* is desired. Examining more children of *B* may increase its value, but will have no effect higher in the tree.

Figure 2(b) illustrates a deep cutoff. The negmax procedure ensures that $D \geq -5$, $C \geq -D$, and $B \geq -C$. If the negmax value of *C* is determined by some child other than *D*, then there is no point in

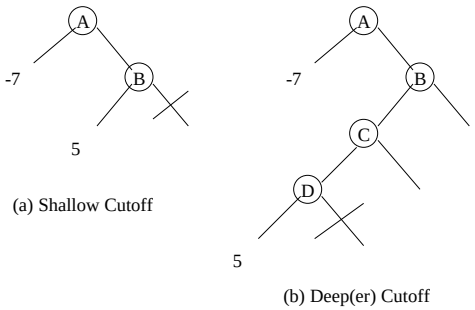

Figure 2. Alpha-beta cutoffs (adapted from [Baudet1978]).

exploring *D*'s children further. But if the value of *C* depends on *D*, then $-C = D \geq 5$, so $B \geq -C \geq -5$, and further search below *D*, which can only increase the value of *B*, will have no effect on the value of *A*. In either case, *D*'s remaining children need not be explored.

The alpha-beta algorithm may be programmed as

```
function alpha_beta(p:position;α,β:integer):integer;
var m,i,t,d:integer;
begin
  determine the child positions p1,...,pd
  if d=0 then return(static_evaluator(p))
  else begin
    m:= α;
    for i:= 1 to d do begin
      t:= -alpha_beta(pi,-β,-m);
      if t > m then m:=t;
      if m≥β then return(m);
    end;
    return(m);
  end;
end.
```

For any node, *P*, β represents an upper bound that is used to restrict search below *P*. A cutoff occurs when it is determined that *P*'s negmax value is greater than or equal to β. In such a situation, the opponent can already choose a move that avoids *P* with a value no greater than β. From the opponent's standpoint, the alternate move is no worse than *P*, so continued search below *P* is not necessary. In game playing terminology, *P* is *refuted*.

2.2. Analysis of Alpha-beta Search

The performance of alpha-beta depends critically on the order in which children of a node are expanded. The greatest number of cutoffs is achieved if the children of each node are examined in increasing order of their negmax values (*best-first order*). Of course, if the negmax values of the children were known *a priori*, there would be no need to search. However, if the static evaluation function yields reasonable estimates of the values of the children, they can be searched in nearly best-first order.

Knuth and Moore [Knuth1975] show that, for any game tree, there is a *minimal (sub)tree* that will always be examined by alpha-beta regardless of the values of the terminal nodes. Moreover, if the tree is searched in best-first order, only the minimal subtree will be searched. A node in the minimal tree is called a *critical node*. Figure 3 shows a game tree with the minimal tree outlined in bold.

The following rules can be used to determine the critical nodes and, hence, the minimal tree for any game tree:

i. The root of the game tree is a type 1 node.
ii. The first child of a type 1 node is also type 1. Remaining children are type 2.
iii. The first child of a type 2 node is a type 3 node.
iv. All children of a type 3 node are type 2.
v. A node is critical iff it is assigned a number by the above rules.

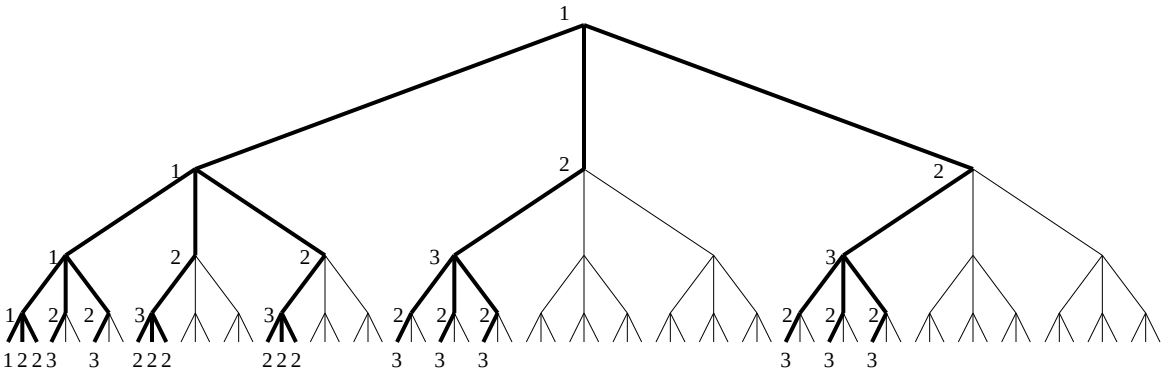

Figure 3. A game tree and its minimal subtree (adapted from [Knuth1975]).

The number of terminal nodes on the minimal subtree of a complete *d*-ary tree of height, *h* is [Slagle1969, Knuth1975]

$$d^{\lceil h/2 \rceil} + d^{\lfloor h/2 \rfloor} + 1 \approx 2\sqrt{N} \ ,$$

where *N* is the number of leaves in the whole tree. Baudet [Baudet1978a] has shown that the effect of deep cutoffs is second-order, and they are not implemented by several algorithms. The minimal tree for these algorithms contains only 1-nodes and 2-nodes. The rules for determining this minimal tree are:

i. The root of the game tree is a type 1 node.

ii. The first child of a type 1 node is also type 1. Remaining children are type 2.

iii. The first child of a type 2 node is a type 1 node.

3. Parallel Game Tree Search

The success of a parallel algorithm may be measured by the the extent to which it takes advantage of available processors. Fishburn [Fishburn1981] defines *speedup* as

$$speedup = \frac{time\ required\ by\ best\ serial\ algorithm}{time\ required\ by\ parallel\ algorithm}.$$

Speedup can be more perspicuously displayed in terms of *efficiency*, defined as

$$efficiency = \frac{speedup}{number\ of\ processors\ used}.$$

"Perfect" efficiency is 1.0; current parallel game tree algorithms fall far short of this ideal. We will examine the reasons for poor efficiency in existing parallel alpha-beta algorithms.

Many parallel algorithms may be described as *problem-heap* algorithms [Møller-Nielsen1984]. In this model, the problem heap is a set of unfinished subproblems. All processors asynchronously execute the same program. When a processor becomes idle, it takes a unit of work (a subproblem) from the heap and solves it, possibly generating new subproblems. Any subproblems generated are placed back on the heap.

3.1. Sources of Efficiency Loss

A problem heap algorithm may fail to achieve perfect efficiency due to various kinds of "loss." *Starvation loss* occurs when processes are blocked waiting for the heap to become nonempty. *Interference loss* is caused by contention for exclusive access to shared resources, including the problem heap. One way to decrease starvation is to split the work into smaller units. Unfortunately, this approach tends to increase interference. Another approach is to introduce speculative work.

Speculative work is work performed by a parallel algorithm before it is possible to determine that the work is necessary. Speculative work introduces another potential source of loss, *speculative loss*, which occurs when processes perform fruitless speculative work. Speculative loss is particularly severe for some parallel alpha-beta algorithms. A parallel game tree algorithm must evaluate different nodes simultaneously. Information gained by searching one node may indicate that search at others could have been avoided. The amount of extra work performed due to these missed cutoffs can be substantial.

It is possible to design parallel game tree algorithms that perform little or no speculative work, but such algorithms usually starve due to the limited size of the pool of known *mandatory work*. For any parallel algorithm $\mathcal{A}$ we define *mandatory work* with respect to a reference algorithm $\mathcal{B}$ as all work that would be performed by $\mathcal{B}$ on the same input. It is possible that $\mathcal{A}$ might terminate successfully on some inputs without performing all the mandatory work. For example, a parallel alpha-beta algorithm may achieve cutoffs not achieved by serial alpha-beta. (This phenomenon can occasionally lead to the anomalous result of greater than "perfect" efficiency.) Experience suggests, however, that for most trees it is likely that a parallel game-tree algorithm will examine significantly more nodes than deemed mandatory (with respect to serial alpha-beta), so our definition is useful in practice.

4. Previous Work

4.1. Parallel Aspiration Search

One approach to parallel alpha-beta search is to divide the alpha-beta window into disjoint intervals. Each processor searches using its own window, and only one processor succeeds. This approach, suggested by Baudet [Baudet1978], has the advantage that processors need not communicate with each other until one finds the solution. When only 2 or 3 processors are used, Baudet actually observed an efficiency greater than 1. Unfortunately, he also found the speedup to be limited to a maximum of 5 or 6 regardless of the number of processors used. Since each processor must examine the minimal tree, parallel aspiration search achieves no speedup if nodes are visited in best-first order.

4.2. Mandatory Work First

Akl, Barnard, and Doran [Akl1982] propose the mandatory work first (MWF)[1] algorithm, which exploits the minimal tree for alpha-beta search without deep cutoffs. The goal of the algorithm is to reduce speculative loss. MWF initially searches the entire minimal tree in parallel. If more than the minimal tree must be examined, MWF performs speculative work in a restricted manner.

---

[1]This name for the algorithm was coined by Finkel and Fishburn.

Recall that the minimal tree for alpha-beta without deep cutoffs has (critical) 1-nodes and 2-nodes. In the first phase of the MWF algorithm, the minimal tree is examined. In subsequent phases, right (i.e. non-critical) children of 2-nodes are searched as follows.[2] Consider a 2-node P with children $\{s_1,...,s_n\}$, where s_1 is the leftmost child. MWF will not search the subtree rooted at a right child s_i until the search of P's left sibling and the search of all siblings s_j for $j<i$ have completed. When the subtree rooted at s_i is searched, it is searched by the serial alpha-beta algorithm. These restrictions ensure that reasonable cutoff information is available at node P prior to committing to the (possibly) speculative work at s_i.

Consider the game tree in Figure 4. In the first phase of the algorithm, the minimal tree, shown in bold, is searched in parallel. In the next phase, which we call the *first speculative phase*, the first right children of nodes *D* and *E* (both labelled *a*) will be searched in parallel if they cannot already be cut off. Search of either *a* node may result in the cutoff of its *b* sibling. If necessary, a *b* node will be searched only after search of its *a* sibling completes. The first speculative phase terminates when search below node *B* completes. During the next (and in this case, final) speculative phase, search of right subtrees below nodes *F* and *H* proceeds in a manner analogous to that described for nodes *D* and *E*. Each interior 2-node in Figure 4 is enclosed by a rectangle that indicates the speculative phase during which its right children will be searched.

Fishburn [Fishburn1981] shows that for best-first game trees, MWF is almost optimal in the sense that efficiency is very close to 1 when large numbers of processors are used to search suitably large game trees. Simulation results for MWF on four-ply random game trees of various fixed degrees [Akl1982] show that the number of nodes examined by MWF increases moderately, but rapidly reaches a plateau as the number of processors is increased. Speedup increases rapidly for the first few processors, but it too quickly reaches a plateau near six. Increasing the number of processors beyond 10 seems to have negligible effect on both speedup and the number of nodes examined. At this point it appears that each additional processor contributes only to starvation.

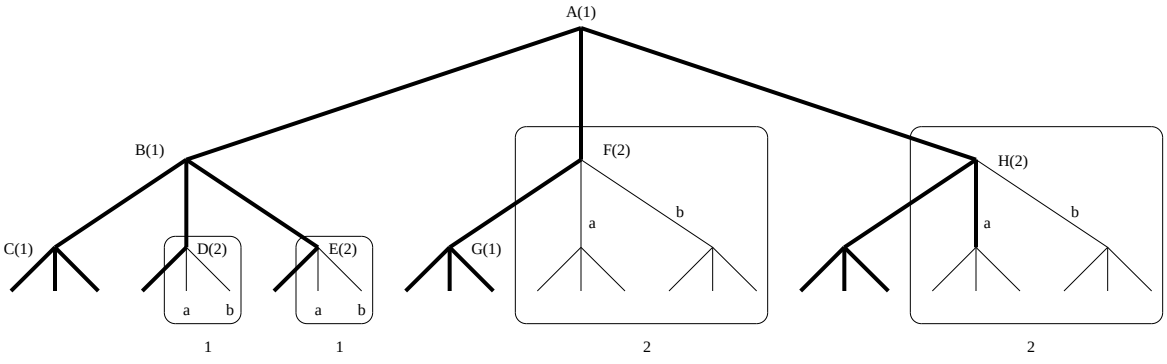

Figure 4. The Mandatory Work First algorithm.

[2]We use the term *right child* to refer to any child but the first.

4.3. Tree-Splitting

Fishburn [Fishburn1981] describes a general *tree-splitting algorithm* for parallel alpha-beta search that runs on a tree of processors. In this tree, a node's parent is termed its *master*, and its children are its *slaves*. Figure 5 shows the initial mapping of a binary processor tree of height two onto a ternary game tree of height four. Game tree nodes are labelled by letters and processors by numbers. To avoid ambiguity, we shall reserve the terms *child* and *parent* for nodes in the game tree and use *master* and *slave* to refer to the processor tree.

The purpose of a master processor is to keep its slaves busy and to keep them informed of the most up-to-date alpha-beta window. A master processor searches its assigned subtree by generating all children of the subtree's root and assigning each child to a slave. If there are more children than slaves, the extra children are queued and assigned to a slave when one becomes available. With the exception of the root processor, all interior processors are both masters and slaves. Leaf processors are only slaves. A leaf processor executes the serial alpha-beta algorithm on its assigned subtree and reports the result to its master. In Figure 5, nodes 4, 5, 6, and 7 are leaf processors. When a processor completes its assigned search, the value returned to its master may allow the alpha-beta window for its master's other slaves to be narrowed. In the extreme case all remaining search below the master can be cut off.

Consider the configuration in Figure 5. Processor 2 has assigned its two slaves to the subtrees rooted at nodes *E* and *F*. When one of the slaves finishes, the result may allow the alpha-beta window for the other to be improved. The free processor can be assigned the subtree with root *G*. Note that search at node *D* will not be initiated until search is completed for either *B* or *C*.

Fishburn has determined expected speedup for the tree-splitting algorithm for two extreme orderings of the game tree. The greatest speedup occurs when the game tree is ordered such that alpha-beta achieves no cutoffs. In this case, the entire tree must be searched and tree-splitting achieves speedup almost equal to the number of processors used. In the case where the game tree is ordered optimally (i.e. best-first order), tree-splitting only achieves and efficiency of $O(1/\sqrt{k})$ relative to alpha-beta, where k is the number of processors.

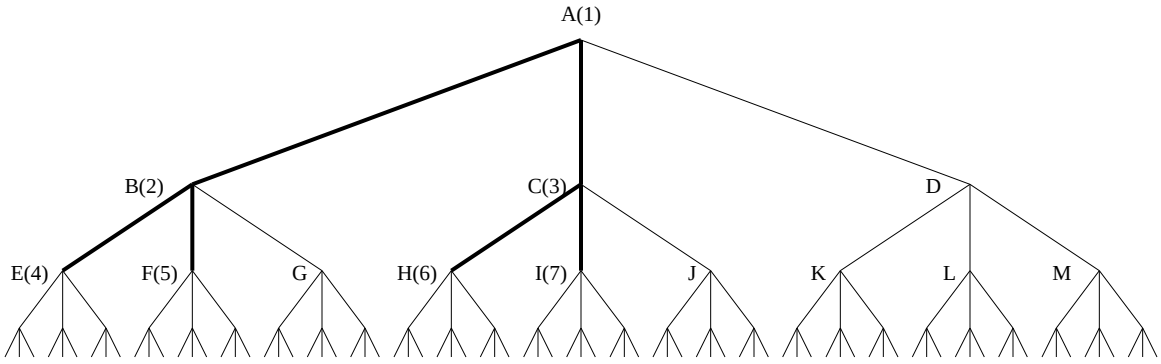

Figure 5. The Tree-splitting algorithm.

4.4. Principal Variation Splitting

The tree-splitting algorithm misses many cutoffs, resulting in significant speculative loss [Finkel1982]. The *principal variation-splitting* (pv-splitting) algorithm [Marsland1983], a variant of the tree-splitting algorithm intended for strongly-ordered game trees, attempts to reduce the number of missed cutoffs and thereby increase efficiency. Marsland calls a tree *strongly ordered* if the first branch from each node is best at least 70 percent of the time, and if the best move is in the first quarter of the branches 90 percent of the time. These numbers are, of course, arbitrary, but meant to capture the notion that the static evaluation function is reasonably accurate—the search is "nearly" in best-first order.

The pv-splitting algorithm traverses the candidate principal variation (i.e. the leftmost branch of the game tree) until the remaining depth of the game tree is equal to the height of the processor tree. At this depth, the tree-splitting algorithm is invoked.[3] When the tree-splitting algorithm terminates, the negmax value for the node is backed up to its parent and the tree-splitting algorithm invoked on the remaining siblings. This process is repeated all the way back to the root of the game tree.

Consider the game tree in Figure 5. When invoked with a binary processor tree of height 2, the pv-splitting algorithm traverses the leftmost branch of the game tree to node *E*. The tree-splitting algorithm is used to determine the negmax value for *E*, which is then backed up to node *B*. The tree-splitting algorithm is then run on nodes *F* and *G* simultaneously. *B*'s value is backed up to the root, and the tree-splitting algorithm is invoked on nodes *C* and *D*. For every node along the candidate principal variation, pv-splitting determines the negmax value of the leftmost child before initiating search on the remaining children. This strategy allows most nodes in the tree to be searched with a bound that allows for cutoffs.

Experiments with 6-ply chess game trees demonstrate that pv-splitting with 4 processors examines only 5% more nodes (on average) than with 1 processor, and achieves a speedup of 3.27 [Marsland1985]. These results compare favorably with Fishburn's results for the tree splitting algorithm using checkers game trees [Fishburn1981]. Although pv-splitting appears effective in limiting speculative loss, Marsland and Popowich note that efficiency appears to drop exponentially as the number of processors is increased: Efficiency with four processors is approximately half that with three processors.

5. A Better Parallel Algorithm

In order to develop a better parallel game tree search algorithm we must understand the limitations of the existing algorithms. For large numbers of processors, the tree-splitting algorithm has low efficiency primarily due to speculative loss, while pv-splitting and MWF suffer primarily from starvation. These efforts show that there is a tradeoff between speculative loss and starvation. Speculative work is necessary to avoid starvation, but speculative work need not incur speculative loss. Our goal is to perform speculative work that is likely to prove useful. This approach is not novel, as both pv-splitting and to some extent MWF perform carefully selected speculative work. Unfortunately, it does not appear to be possible to increase speculative work performed by parallel alpha-beta algorithms without incurring substantial speculative loss. We believe that this is due to the nature of the serial alpha-beta algorithm.

---

[3] Marsland and Popowich [Marsland1985] describe a version of pv-splitting that uses parallel minimal window search rather than the tree-splitting algorithm to verify the rightmost children nodes along the candidate principal variation.

If we examine the requirements of algorithms that evaluate game trees, it becomes clear that alpha-beta examines the tree in a somewhat arbitrary order. Consider an algorithm $\mathcal{A}$ that must evaluate (i.e. determine the negmax value of) some node E in a game tree. By definition, the negmax value of E is equal to the negmax value of its best (lowest-valued) child, so $\mathcal{A}$ must evaluate at least one child of E. $\mathcal{A}$ must also refute each of the remaining children of E.

Let R be a child of E that must be refuted. To refute R, $\mathcal{A}$ must show that $-R<E$, which requires finding a child of R with a lower value than E. $\mathcal{A}$ must, therefore, evaluate one of R's children.[4] If the value of this child is not sufficient to refute R, $\mathcal{A}$ must examine another of R's children. This process is repeated until either R is refuted or until all children of R have been examined. If the value of R is still less than $-E$ after all of R's children have been examined, the refutation is said to have *failed* and E's value is increased to $-R$.

If evaluation of R's first child does not refute R, $\mathcal{A}$ need only try to refute (not evaluate) R's remaining children. If $\mathcal{A}$ were to evaluate another of R's children, it might miss cutoffs below R. Consider the tree in Figure 6. Suppose that $\mathcal{A}$ is attempting to evaluate node I and that the value of J is known to be -7. $\mathcal{A}$ attempts to refute K, so L is evaluated and its value is 9. K is, therefore, not refuted. In an attempt to refute M, $\mathcal{A}$ evaluates N. The value of N is -11, so M is refuted and the value of K is -9. Node O need not be examined. If $\mathcal{A}$ were to evaluate (rather than refute) M, O would also need to be examined.

Thus game-tree search may be viewed as a process of evaluating some nodes, which we call *e-nodes* and refuting others, called *r-nodes*. The tree in Figure 7 highlights the differences between e-nodes and r-nodes. R-nodes are "cheaper" than e-nodes, since all children of e-nodes, but as few as one child of an r-node need be examined. Each (non-leaf) node has exactly one e-node child (*e-child*).

Any child may be chosen to be the e-child of a node, but children with lower negmax values are better. If E is chosen to be the e-child of N, but some other child R has a lower negmax value, the initial

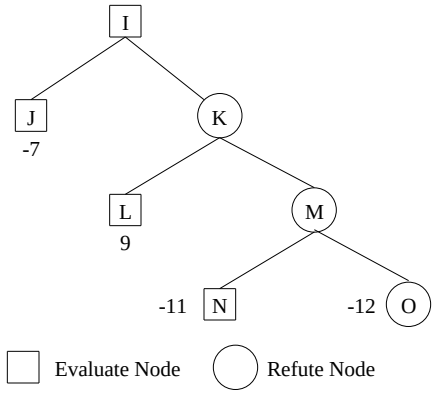

Figure 6. Evaluate and refute nodes.

[4]We are ignoring the possibility of deep cutoffs.

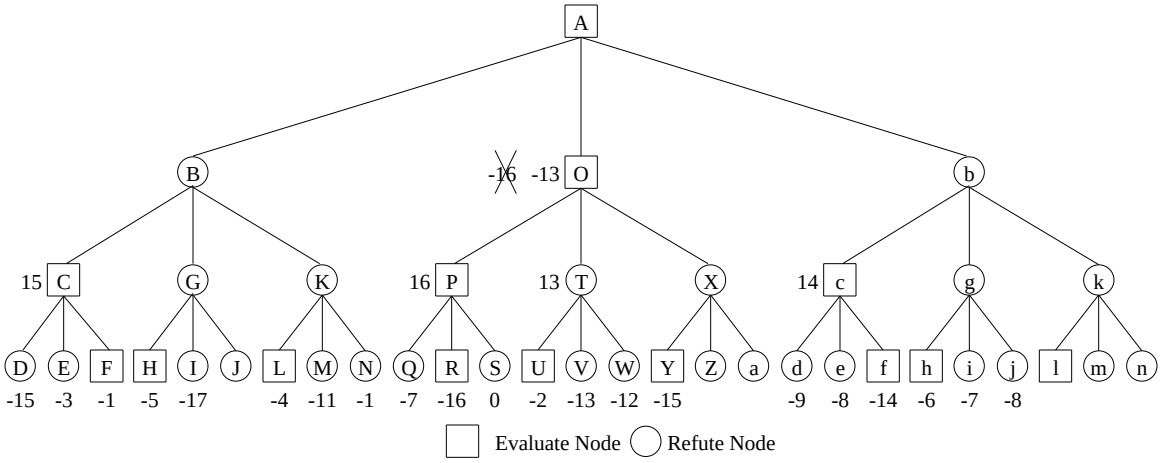

Figure 7. A more complex example of evaluate and refute nodes.

value of *N* derived from evaluating *E* will not be sufficiently high to refute *R*. Refutation of *R* will fail, and the algorithm will end up evaluating *R* as well. On the other hand, a child cannot be refuted until at least one of its siblings as been completely evaluated. In Figure 6, if 9 were only a tentative value for *L* based on partial search of its descendents, we could not refute *M* based on its tentative value of 11. Further search of *L*'s descendents might increase its value above 11, making *M* the minimal child and necessitating complete evaluation of *M*.

In a serial algorithm these considerations lead to a depth-first strategy. One child is chosen on the basis of some intrinsic considerations, such as its static value, to be the e-child. This child is completely evaluated before any of its siblings is considered. In a parallel setting, however, it may be beneficial to delay committing to a choice of e-child. Let *E* be an e-node in a game tree and let $\{E_1, \cdots, E_k\}$ be its children in increasing order of static value. Similarly, let $E_{i.j}$ denote the j^{th} child of E_i in order of increasing static value. Regardless of which E_i is chosen to be an e-node, each E_i will have one e-child. Suppose that instead of choosing E_1 as the e-child of *E*, we choose $E_{i.1}$ to be the e-child of E_i, for each *i*, and evaluate all of these grandchildren before committing to a choice of e-child for *E*. We will have done no more work than under the depth-first strategy, but the information gained from evaluation of the $E_{i.1}$ may permit a better choice of e-child for *E*. We shall refer to the $E_{i.1}$ as *elder grandchildren* of *E*. In summary, our strategy is to *evaluate* the elder grandchildren (concurrently, if possible), choose the child with largest elder grandchild to be the e-child, *evaluate* this child, and finally *refute* the remaining children in order.

As an example, consider the game tree in Figure 7. Prior to selecting an e-child for A, nodes *C*, *P*, and *c* are evaluated. Of these, *P* has the greatest value so *O* is selected to be the e-child of the root. Since the e-child of *O* has already been selected and evaluated, the remaining children of *O* are refuted, left-to-right. Since all children of *T* have values greater than -16, refutation of *T* fails. *T* becomes the best child of *O*, and *O*'s value is updated to -13. With this updated value for *O*, *X* is refuted as soon as *Y* evaluates to -15. The true value of *O* is therefore -13. The values of *B* and *b* are tentatively set to -15 and -14 respectively, and refuted. Node *B* is to the left of *b*, so it is refuted first. Node *G* is refuted after examining node *I*. Node *K* has value 11 and is not refuted. The value of *B* is therefore updated to -11 and it is refuted. To

refute *b*, the algorithm examines *g*. Node *g* has value 8 and cannot be refuted. However, the updated value of -8 for *b* is sufficient to refute it.

$\mathcal{ER}$, an algorithm based on these ideas, is presented in Figure 8. $\mathcal{ER}$ is a better candidate for parallelization than alpha-beta. A parallel implementation of $\mathcal{ER}$ may evaluate the elder grandchildren of an e-node in parallel since they represent mandatory work. These grandchildren are themselves e-nodes, so their elder grandchildren may also be evaluated in parallel. Unfortunately, if $\mathcal{ER}$ is to perform only mandatory work, the remaining work at an e-node must be performed in sequence. As a result $\mathcal{ER}$ suffers from starvation. Just as parallel alpha-beta algorithms must perform speculative work to avoid starvation, so must $\mathcal{ER}$. We have investigated three ways to add speculative work to $\mathcal{ER}$. In all of the descriptions below, suppose that *E* is an e-node.

```
type node =
  record
    value: integer;
    done: boolean;
  end;

function ER(P:node; α,integer): integer;
var  i,d,t: integer;
with P do begin
  value:= α;
  determine the child nodes P1,...,Pd;
  if d=0 then return(static_evaluator(P));
  for i:=1 to d do
    begin
       t:= -Eval_first(Pi,-β,-value);
       if Pi.done then
         begin
           if t>value then value:= t;
           if value≥β then
             return(value);
         end;
    end;
  sort(P); /* sort* children of P */
  for i:=1 to d do
    if not Pi.done then
      begin
         t:= -Refute_rest(Pi,-β,-value);
         if t>value then value:= t;
         if value≥β then
           return(value);
       end;
  return value;
end;
```

*The sort procedure places the children of P in ascending order according to their tentative values.

```
function Eval_first(P:node; α,β:integer)
            :integer;
var  d,t: integer;
with P do begin
  value:= α;
  determine the child nodes P1,...,Pd;
  if d=0 then
    begin
      done:= TRUE;
      return(static_evaluator(P));
    end;
  else
    begin
       t:= -ER(P1,-β,-value);
       if t>value then value:= t;
       done := value≥β or d=1;
       return(value);
     end;
end;

function Refute_rest(P:node; α,β:integer)
            :integer;
var  i,d,t: integer;
with P do begin
  value:= α;
  for i:=2 to d do begin
    t:= -Eval_first(Pi,-β,-value);
    if (not Pi.done) then
      t:= -Refute_rest(Pi,-β,-value);
    if t>value then value:= t;
    if value≥β then return(value);
  end;
  return(value)
end;
```

Figure 8. A serial version of algorithm $\mathcal{ER}$.

Parallel refutation.

After the e-child of *E* is evaluated, refute the remaining children of *E* in parallel. For example, in Figure 7 after node *O* is evaluated both *B* and *b* can be refuted in parallel. The children of each of these nodes must still be examined sequentially. For example, although nodes *b* and *B* may attempt to refute their remaining children simultaneously, node *B* must attempt to refute *G* before attempting to refute *K*, and node *b* must attempt to refute *g* before *k*. Parallel refutation may do unnecessary work if the refutation of one child fails because information from the failure may allow more cutoffs when refuting the other children. Nonetheless, the bound provided by the evaluation of *E*'s e-child is often sufficient to refute all of *E*'s remaining children.

Multiple e-nodes.

After the e-child of *E* has been evaluated, select the next best child as a second e-child. In Figure 7 after node *O* is evaluated, node *B* can be selected as a second e-child. If the first e-child is not able to immediately refute the remaining children of *E*, some other node may prove to be the best child, in which case all of that node's children must be examined. Rather than waiting for evaluation of the best child to complete, we can start to evaluate a second child immediately, thereby avoiding missed cutoffs below *E*'s other children. This method can be generalized as follows: Once the elder grandchildren of *E* have been evaluated, ensure that *E* always has at least one active e-child. When it is necessary to select an e-child, select the node with the most optimistic bound on the value of *E*.

Early choice of e-nodes.

Once several elder grandchildren have been evaluated, a reasonable estimate of the correct e-child is available, so it is wasteful to have processors waiting for the bound on the remaining children. Therefore, we might immediately select the child with the best known optimistic bound to be *E*'s e-child.

Parallel refutation is similar in spirit to the speculative work performed by MWF and pv-splitting. The other methods, however, are unique to *ER*. Both these methods aggressively select e-nodes, increasing the pool of work that may be performed in parallel. These methods use the known bound information to select work that is likely to prove useful, in contrast with parallel alpha-beta algorithms, which have no such information available.

6. Implementation

ER has been implemented as a problem-heap algorithm on a Sequent Symmetry multiprocessor. The program executed by each processor has the following outline.

```
repeat
  take a node from the problem-heap;
  if node is a leaf then
    begin
      node.value:= static_evaluator(node);
      combine(node);
    end;
  else
    generate children as specified in Table 1;
until done;


procedure combine(node);
begin
  repeat
    if node = root then
      halt;
    p:= parent(node);
    if -node.value > p.value then
      p.value:= -node.value;
    node:= p;
  until node has active children AND node can't be cut off;
  last_node:= node;
  perform action specified in Table 2;
end
```

Our implementation exploits all three sources of speculative work described in section 5. In the case of "early choice of e-nodes," we select the e-child of an e-node as soon as all but one of the elder grandchildren have been evaluated.

The problem-heap is represented by a pair of priority queues: the *speculative priority queue*, which contains potential speculative work, and the *primary priority queue*, which contains *scheduled work*—both mandatory and selected speculative work. A processor that needs work first attempts to remove a *scheduled node* from the primary priority queue. If the primary queue is empty, it attempts to remove a node from the speculative queue. A node removed from the primary queue has one or more children generated depending on the node's *type* (e-node, r-node, or undecided). Table 1 outlines the actions taken when a node is removed from the primary queue. The primary queue is ordered by node depth, with the deepest nodes first.

All nodes on the speculative queue are e-nodes and must have at least one potential e-child. A node cannot be placed on the speculative queue until all but one of its elder grandchildren have been evaluated. When a node *E* is removed from the speculative queue, its best child becomes an e-child, and *E* is returned to the speculative queue. Nodes on the speculative queue are ranked by number of e-children (fewer e-children first), with ties broken in favor of shallower nodes.

Each processor repeatedly removes a node from the problem-heap, generates children, and places the children back on the heap. When a leaf node is removed from the heap, its value is determined by the static evaluator. The value of the leaf is then "backed-up" the tree as far as possible by the `combine`

Node Type	State	Action(s)
E-node		Generate all children. Assign each child "undecided" type. Place each child on primary queue.
Undecided		Generate first child (an "e-node") and place on primary queue.
R-node	No children generated	Generate first child (an "e-node") and place on primary queue.
R-node	One or more children generated	Generate next child (an "r-node") and place on primary queue.

Table 1. Rules for generating nodes.

procedure. When all the children of an interior node *N* have been combined, *N* may also be combined. $\mathcal{ER}$ terminates when all children of the root have been combined.

The `combine` procedure proceeds until a node with active children is encountered. This node is called `last_node`. Depending on the type of `last_node`, one or more nodes may be placed on the problem-heap. Table 2 outlines the actions taken depending on the type of `last_node`.

The programming interface to our implementation of $\mathcal{ER}$ is similar to DIB [Finkel1987]. The caller supplies a procedure for generating nodes of the game tree, a static evaluation function, and integers specifying the number of processors to use, the search depth, and the depth below which serial $\mathcal{ER}$ is to be used.

7. Experimental Results

We tested algorithm $\mathcal{ER}$ on synthetically generated random game trees, as well as on trees generated by a program by Steven Scott that plays the game of Othello (see [Rosenbloom1982] for more information about this game). The root configuration for the Othello trees is shown in Figure 9. It is WHITE's turn to

Node Type	State	Action(s)
E-Node	All but one of the elder grandchildren are evaluated.	Place node on speculative queue.
E-node	All elder grandchildren are evaluated, but an e-child has not been selected.	Select the e-child and place it on the primary queue.
E-node	The first e-child has been evaluated and remaining children are "undecided."	Assign each active child type "r-node" and place it on the primary queue. (All children may be refuted in parallel.)
Undecided	All but one of the parent's elder grandchildren have been evaluated.	Place the parent on the speculative queue.
Undecided	All of the parent's elder grandchildren have been evaluated, but the first e-child of the parent has not been selected.	Select the parent's e-child and place it on the primary queue.

Table 2. Rules for combining nodes.

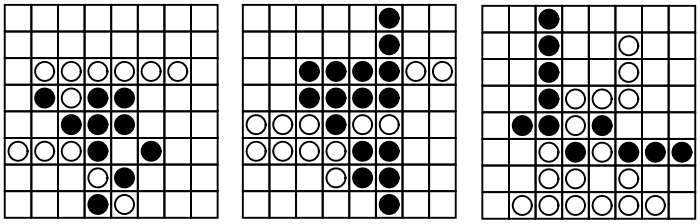

Figure 9. Three Othello configurations.

move in each configuration. Each Othello tree was searched to a fixed depth of seven ply. In order to improve search order, children were sorted according to values returned by the static evaluator. Sorting was not performed below ply five. Successors of e-nodes were also not sorted. For the random trees, each leaf was assigned an independent pseudo-random value drawn from a uniform distribution. Table 3 outlines the differences among the trees.

Figures 10 and 11 plot efficiency for *ER* on Othello and random game trees, respectively. The

Name	Type	Degree	Search Depth	Serial Depth
R1	Random	4	10 ply	7
R2	Random	4	11 ply	7
R3	Random	8	7 ply	5
O1	Othello	varying	7 ply	5
O2	Othello	varying	7 ply	5
O3	Othello	varying	7 ply	5

Table 3. Descriptions of the game trees used in the experiments.

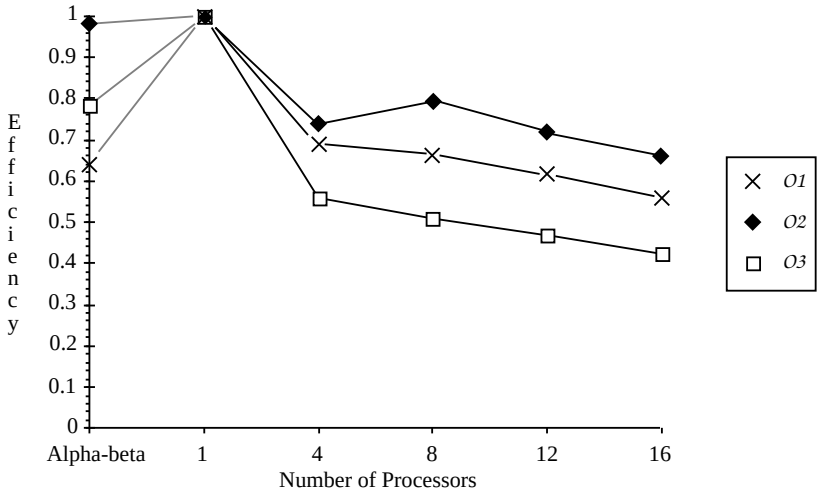

Figure 10. Efficiency of *ER* for Othello game trees.

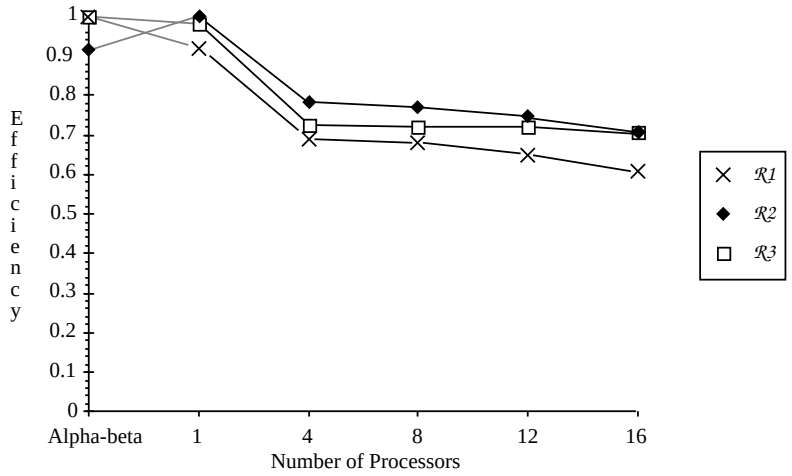

Figure 11. Efficiency of *ER* for random game trees.

"efficiency" of serial alpha-beta[5] search is shown for comparison. Since efficiency is computed relative to the *fastest* serial algorithm, serial alpha-beta has efficiency less than unity for those trees on which serial *ER* performs better (all of the Othello tress and the second random tree).

With 16 processors, we achieve speedups ranging from 9.8 to 11.2 (representing efficiencies of 0.61 to 0.70) on random trees. The low efficiency of 0.61 for *R1* is partly due to the performance of alpha-beta over serial *ER*. Nevertheless, efficiency appears to be reasonably consistent for random trees. For Othello trees, speedup ranges from 6.7 to 10.6 (efficiency 0.42 to 0.66) with 16 processors.

Figures 12 and 13 compare the number of nodes examined by each of the experiments. For every tree, the number of nodes examined by 4-processor *ER* is substantially more than the number of nodes examined by serial *ER*. With more than 4 processors the number of nodes examined tends to grow slowly. Speculative loss, therefore, only increases moderately between 4 and 16 processors. This result is especially interesting since *ER* does not greatly restrict speculative work. Efficiency does decrease, however, between 4 and 16 processors for all trees, primarily due to increased contention for the shared tree data structure and to starvation. It would be possible to reduce contention by decreasing the serial depth, but decreasing the depth would only increase starvation. Perhaps a better distribution of work could reduce this contention/starvation efficiency loss.

Although alpha-beta examined fewer nodes than serial *ER* for *O1*, *ER* was faster. This apparent anomaly is due to the fact that *ER* need not sort the children of e-nodes. Sorting the children of a node incurs not only the sorting overhead, but the application of the static evaluator on each child. As Figure 12 shows, the increased overhead can be substantial. It is possible to reduce the sorting overhead for alpha-beta, since the children of critical 1-nodes and 3-nodes need not be sorted. It is not clear that *ER* would still be the faster algorithm on *O1* if our alpha-beta implementation included this optimization.

---

[5]Alpha-beta with deep cutoffs

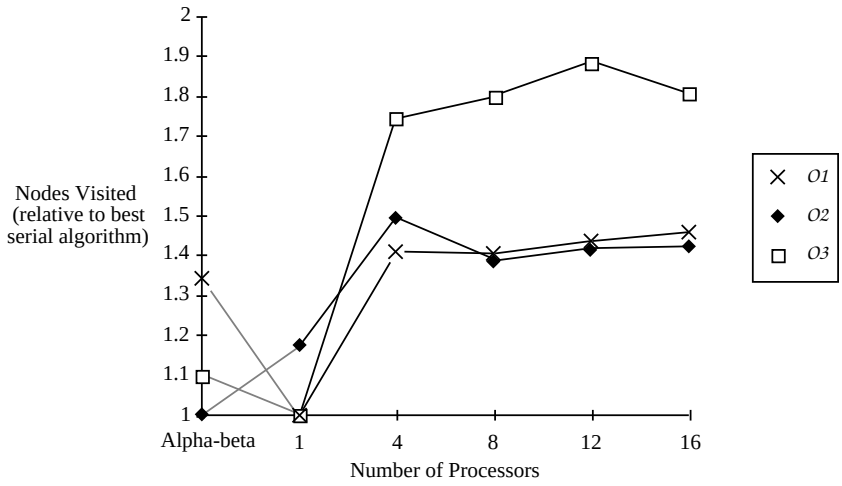

Figure 12. Number of nodes generated for Othello game trees.

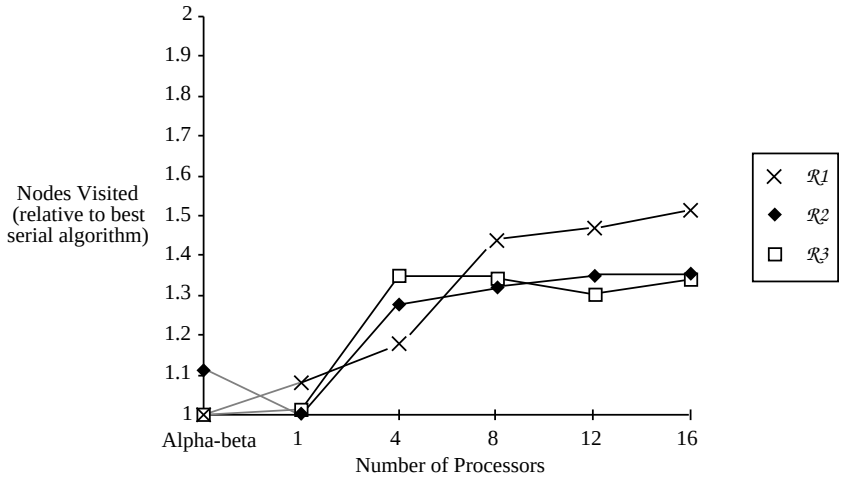

Figure 13. Number of nodes generated for random game trees.

8. Summary and Future Work

We have presented *ER*, a new parallel algorithm for game tree search. Unlike most parallel game tree search algorithms, *ER* is not a "parallel extension" of alpha-beta. *ER*'s ability to dynamically rank potential speculative work distinguishes it from other parallel game tree search algorithms. As a result, speculative loss for *ER* does not increase rapidly as the number of processors is increased and *ER* achieves impressive performance on both random and real game trees. Nonetheless, although speculative loss doesn't increase much, it is still the major contributor of efficiency loss between 4 and 16 processors.

ER's ability to dynamically rank speculative work is local to a specific e-node. It cannot, for example, dynamically compare two e-nodes and decide which represents the best speculative work. (Currently, e-nodes are ranked on the speculative queue according to depth; a rather naive ordering.) In order to reduce speculative loss and improve efficiency a better mechanism for globally ranking speculative work

must be found.

As we have already noted, contention in the game tree becomes increasingly significant as the number of processors is increased. We expect that this efficiency loss can be reduced by distributing work in a manner that reduces processor interaction.

Finally, perhaps the most obvious gap in our experimental results is the absence of direct quantitative comparison with previous techniques. Such comparisons are difficult, since previous algorithms were implemented in a variety of hardware and software configurations, making absolute comparisons meaningless, and previous papers tend to report results only for very small processor pools, making trends difficult to see. We are currently working on reimplementing some of the more important existing algorithms, which will allow direct comparison.

Acknowledgements

We would like to thank Steven Scott for providing the algorithms for generating and evaluating Othello game trees.

References

Akl1982.
S. G. Akl, D. T. Barnard, and R. J Doran, "Design, analysis, and implementation of a parallel tree search algorithm," *IEEE Transactions on Pattern Analysis and Machine Intelligence* **4** pp. 192-203 (1982).

Baudet1978.
G. Baudet, "The design and analysis of algorithms for asynchronous multiprocessors," CMU-CS-78-116, Computer Science Department, Carnegie-Mellon University (1978).

Baudet1978a.
G. Baudet, "On the branching factor of the alpha-beta pruning algorithm," *Artificial Intelligence* **10** pp. 173-199 (1978).

Berliner1988.
Hans Berliner and Carl Ebeling, "Pattern Knowledge and Search: The SUPREM Architecture," CMU-CS-88-109, Computer Science Department, Carnegie-Mellon University (1988).

Finkel1982.
R. A. Finkel and J. P. Fishburn, "Parallelism in alpha-beta search," *Artificial Intelligence* **19** pp. 89-106 (1982).

Finkel1987.
R. A. Finkel and U. Manber, "DIB: A distributed implementation of backtracking.," *TOPLAS* **9**(2) pp. 235-256 (1987).

Fishburn1981.
J. P. Fishburn, "Analysis of speedup in distributed algorithms," TR #431, Computer Sciences Department, University of Wisconsin—Madison (1981).

Knuth1975.
D. E. Knuth and R.W. Moore, "An analysis of alpha-beta pruning," *Artificial Intelligence* **6** pp. 293-326 (1975).

Marsland1983.
T. A. Marsland and M. Campbell, "Parallel search of strongly ordered game trees," *ACM Computing Surveys* **14** pp. 347-367 (1983).

Marsland1985.
T. A. Marsland and F. Popowich, "Parallel game tree search," *IEEE PAMI* **7** pp. 442-452 (1985).

Møller-Nielsen1984.
P. Møller-Nielsen and J. Staunstrup, *Problem-heap: A paradigm for multiprocessor algorithms.*, Department of Computer Science, University of Washington (Nov 13, 1984). Unpublished report

Rosenbloom1982.
P. S. Rosenbloom, "A world-championship-level othello program," *Artificial Intelligence* **19** pp. 279-320 (1982).

Slagle1969.
J. R. Slagle and J. K. Dixon, "Experiments with some programs that search game trees.," *JACM* **16** pp. 189-207 (1969).