A comparison of some parallel game-tree search algorithms (Revised version)

Jaleh Rezaie (jrezaie@ms.uky.edu)
Raphael Finkel (raphael@ms.uky.edu)

Department of Computer Science
University of Kentucky
Lexington, KY 40506-0027

Abstract

This paper experimentally compares several sequential and parallel game-tree search methods: alpha-beta, mandatory work first, principal-variation splitting, tree splitting, ER, and delay splitting. All have been implemented in a common environment provided by the DIB package.

Key words: game trees, heuristic search, alpha-beta

1. Introduction

In this paper we compare some of the parallel methods for searching large game trees. These trees arise in the area of artificial intelligence and are closely related to trees searched in other application areas. Exhaustive search of a tree is prohibitively expensive. There are several ways to ameliorate the problem.

- Search only to a given depth.
- Apply heuristics, such as the alpha-beta method, to cut off fruitless search.
- Apply many computers simultaneously in pursuing the search.

We concentrate on distributed variants of the alpha-beta heuristic that try to avoid searching unnecessary parts of the tree while keeping many processors fruitfully busy.

The algorithms we compare are alpha-beta, mandatory work first, principal-variation splitting, tree splitting, ER, and delay splitting. To be able to make a fair comparison between the above algorithms, we have extended the DIB package [1] to use it as a framework for implementing all the algorithms we compare.

Section 2 describes the DIB package. Section 3 introduces the alpha-beta pruning and briefly describes the algorithms used in the experiment. Section 4 presents experimental results that compare the algorithms. Section 5 compares the effects of several sorting strategies on the above algorithms. Section 6 illustrates the new results achieved by adjustments made to MWF algorithm. Section 7 summarizes the results, and details remaining parts of this experiment.

2. DIB — A distributed implementation of backtracking

In this section we describe how DIB works and how we use it to implement different tree-search algorithms.

2.1. Description of DIB

DIB is a multi-purpose package developed by Finkel and Manber for tree-traversal problems [1]. It allows applications such as backtrack and branch-and-bound to be implemented on a multicomputer. DIB's requirements from the distributed operating system are minimal. The machines must be connected by a network that supports a message-passing mechanism; each machine must be able to communicate, not necessarily directly, with all other machines. Our implementation of DIB is programmed in C and runs in the Unix environment across machines connected by an internet or on a Unix multiprocessor.

The application program must specify the root of the problem tree, how to generate children, and calculations needed at each node. It can also optionally specify how to generate values of a tree node from combining its children's values and how to spread information either globally or locally throughout the tree.

DIB divides the problem into subproblems and assigns them to any number of processors (potentially nonhomogeneous machines in a network) dynamically. Each processor maintains a table of explicit work, recording all the problems that have been sent to the processor, have been generated by the processor itself, and/or have been sent to other processors. Each processor is responsible for the work in its table. Each item of work (represented by a node in the backtrack tree, which stands as well for all its descendents) is labeled by which processor, if any, has been assigned that work.

When a processor A is finished with a problem and has reported its result to the processor that gave it that problem, it will take the first (in an inorder traversal of the tree) unassigned problem from its table. If no unassigned problem is available, A sends a **work request** message to another processor (or processors), selected at random from A's peers, repeatedly (with some delay) until new work arrives.

A processor B that receives a work request message interrupts its own search and trys to respond by sending some work to the requesting processor from its table. If no unassigned problem is available in the table, then the problem B is working on is subdivided and its children are put in the table. Until work is subdivided, DIB maintains a fast representation of the current search (just a recursion stack; we call it the ''implicit'' representation); subdivided work is explicit in the table. After subdivision, B can usually send some unassigned work to the requesting processor. Subdivision may have to be repeated several times before an unassigned problem is generated, but if it reaches a trivial problem (not worth subdividing), or if it reaches the depth at which B itself is searching, the request is not granted. B resumes its search after dealing with the incoming request.

DIB is fault tolerant, in that work that B has given to A can still be accomplished by B if there is nothing else worth doing and if A has not yet reported the result of that work. This mechanism does not need timeouts or ''heartbeats'' to detect failure.

We have enhanced the DIB package so that it can achieve high efficiency for game tree search. The principal enhancement is added flexibility given to the application level for delaying evaluation of a game-tree node. That is, the application can refuse to generate additional children for a node but indicate that in the future it may again be willing to do so. DIB does not attempt to generate children of such a node again until some other child of that node has completed or a data update message has arrived at that node.

To experiment with game playing, we have designed a two-level application structure. The **game** level is game-specific, knowing the rules for tic-tac-toe, Othello, or checkers. The **control** level communicates both with DIB and the game level. It knows the pattern of evaluation for one of the algorithms we compared, namely, alpha-beta, mandatory work first, principal-variation splitting, tree splitting, ER, or delay splitting. Any of the game modules we implemented can be used with any of the control modules; any such combination can be used with our enhanced DIB.

Since DIB distributes work, collects and reports results, and passes messages between processors in a similar way for all the combinations, we can compare different control modules in a fairly implementation-independent fashion. Previous comparisons are questionable because each algorithm was implemented in a different parallel environment.

3. Parallel tree search algorithms

The best way to evaluate a parallel algorithm for a given problem is to measure the extent in which it takes advantage of available processors. This idea can be formulated as follows:

$$\textit{speedup } S = \frac{\textit{time required by best sequential algorithm}}{\textit{time required by parallel algorithm}}$$

$$\textit{efficiency } E = \frac{S}{\textit{number of processors used}}$$

It is not easy to achieve a ‘‘perfect’’ efficiency of 1.0. For a given sized problem, efficiency tends to decrease as the number of processors increases. This relationship is explained by Kumar and Rao [2] as resulting from an increase in the communication time (sum of the time spent by all processors in communicating with neighboring processors, waiting for messages, time in starvation, and so forth), while there is no change in computation time (sum of the time spent by all the processors in useful computation). The relationship between communication time (T_{cm}), computation time (T_{cp}), and efficiency (E) is described as follows:

$$E = \frac{T_{cp}}{T_{cp} + T_{cm}}$$

Kumar and Rao [2] define an **isoefficiency function** that shows how the problem must grow with number of processors to achieve the same efficiency. They also mention that since most problems have a sequential component (in depth-first search, it is one node expansion), problem size must grow at least linearly to maintain a particular efficiency.

Steinberg and Solomon [3] blame the failure to achieve perfect efficiency on three types of ‘‘loss’’.

- **Starvation loss**: processors sitting idle while awaiting work to be given to them.
- **Interference loss**: time spent waiting for access to shared resources such as the set of unfinished subproblems.
- **Speculative loss**: time spent performing unnecessary work, such as that performed by a parallel algorithm before it is possible to determine that the work is necessary.

Because a parallel algorithm must evaluate different nodes simultaneously, information gained by evaluation of one node could come too late to cut off evaluation of other nodes.

3.1. Alpha-beta

The alpha-beta algorithm is a sequential technique used to evaluate a game tree efficiently. The nodes corresponding to the first player's moves are called **max** nodes, and the other nodes are called **min** nodes. The value of a max node is the maximum of the value of its children, where as the value of a min node is the minimum of the value of its children. The value of a leaf is determined by a game-specific static evaluator. Alpha-beta ignores branches that are certain not to contribute to the value of the current node. Figure 1 shows a sample game tree with a cutoff. In this figure, node z, which is a max node, has two children, and its first child is evaluated to 9. Therefore,

$$\text{value}(z) = \max\{9, \text{value}(y)\}$$

where y is the other child of z. Now if the first child (we will often call it the **eldest** child) of y is evaluated to 7 then

$$\text{value}(y) = \min\{7, ...\}$$

so the value of z is 9 regardless of the value of y. It follows that the remaining children of the node y need not be evaluated. Ignoring those children is called **shallow cutoff**.

Figure 2 illustrates another type of cutoff. After the eldest child of node z is evaluated, we see that z's value will be greater than or equal to 9. This value is the current lower bound in the alpha-beta algorithm. The value of a **min** node in the subtree rooted at node y must be greater than 9 in order for the lower bound to change. Therefore, when the algorithm reaches node w (a min node) and its first child is evaluated to 7, the evaluation of the remaining children can be avoided. This cutoff is called a **deep cutoff** because the node w is more than one ply below the node z.

Following Fishburn [4], we present the following Pascal-like code of the alpha-beta algorithm, as adapted from Knuth and Moore [5]:

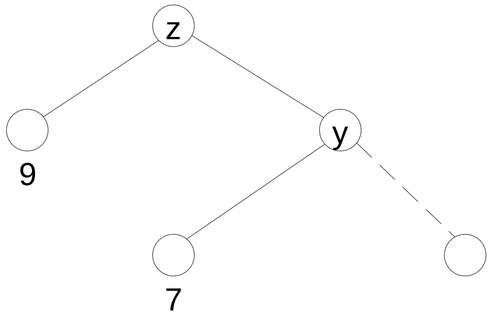

Figure 1: Shallow cutoff

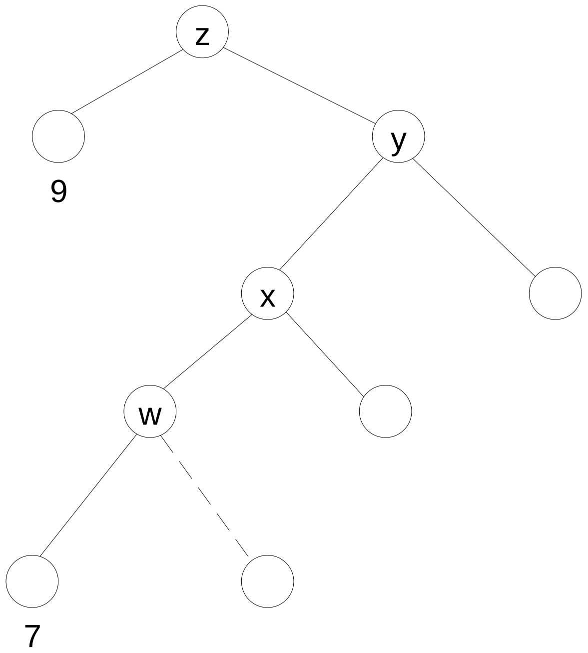

Figure 2: Deep cutoff

```
function alphabeta(z : position; α, β : integer):integer;
        var
                Answer, Child, t, d : integer;
        begin
                determine the child positions z_1,..., z_d
                if d = 0 then
                        return(StaticValue(z))
                else
                        begin
                        Answer := α ;
                        for Child := 1 to d do
                                begin
                                t := -alphabeta(z_Child , -β, -Answer);
                                if t > Answer then
                                        Answer := t;
                                if Answer ≥ β then
                                        return(Answer); {cutoff}
                                end;
                        return(Answer);
                        end;
        end.
```

The alpha-beta algorithm satisfies the following conditions [5]:

if negamax(z) $\le \alpha$	then alphabeta(z, α, β) $\le \alpha$,
if $\alpha <$ negamax(z) $< \beta$	then alphabeta(z, α, β) = negamax(z),
if negamax(z) $\ge \beta$	then alphabeta(z, α, β) $\ge \beta$.

These conditions imply that

$$\text{alphabeta}(z, -\infty, \infty) = \text{negamax}(z),$$

which means that if the initial window [alpha, beta] is $(-\infty, \infty)$ then the alpha-beta algorithm returns the same value as the negamax algorithm (straightforward tree-evaluation algorithm that never cuts work off) [5].

The performance of the alpha-beta algorithm depends a great deal on the order in which children of a node are expanded. If the children of each node in the game tree are expanded in increasing order of their negamax values, then the largest number of cutoffs will occur.

Knuth and Moore [5] introduced the idea of **critical nodes** in their analysis of the best case of the alpha-beta algorithm, and Steinberg and Solomon [3] use the following rules to determine the critical nodes:

- The root of the game tree is a type-1 node.
- The eldest child of a type-1 node is also type-1. The remaining children are type-2.
- The eldest child of a type-2 node is a type-3 node.
- All children of a type-3 node are type-2.
- A node is critical iff it is assigned a number by the above rules.

The critical nodes form a minimal subtree [3] of the game tree which, regardless of the values of the terminal nodes, will always be examined by the alpha-beta algorithm [5]. The number of terminal nodes in the minimal subtree of a complete d-ary tree of height h is

$$d^{\lceil h/2 \rceil} + d^{\lfloor h/2 \rfloor} - 1$$

If the tree is examined in increasing order of value, the alpha-beta procedure examines precisely the minimal subtree of the game tree. In short, alpha-beta examines about $2n^{1/2}$ nodes, where negamax would examine n nodes.

3.2. Mandatory work first (MWF)

This algorithm was proposed by Akl, Barnard, and Doran as a parallel implementation of alpha-beta without deep cutoffs. The name MWF was coined by Fishburn and Finkel. MWF evaluates critical nodes concurrently and then returns to evaluate other nodes if needed [6, 7]. When deep cutoffs are not considered in the search algorithm, only type-1 and type-2 nodes are critical, as shown in Figure 3.

MWF evaluates type-1 nodes completely, but only evaluates type-2 nodes partially. After the eldest child of a type-1 node (also type-1) has been evaluated, the remaining children (all type-2) are completely evaluated only if the result of the partial evaluation is not sufficient to cut them off. All evaluations currently allowed by MWF may be undertaken simultaneously.

Akl, Barnard, and Doran [6] tested MWF with game trees of depth 4 and branching factors of 5, 10, 15 and 20. They noticed that MWF has a better efficiency when the game tree has a larger fanout, but found that the speedup reaches a plateau around six. The total number of nodes visited as well as the number of terminal nodes examined showed an increase with increasing number of processors, but the plateau was reached much faster.

Fishburn [4] analyzed MWF for best-first and worst-first ordering of the game tree. In the best-first ordering, MWF is almost optimal, since its efficiency is very close to 1

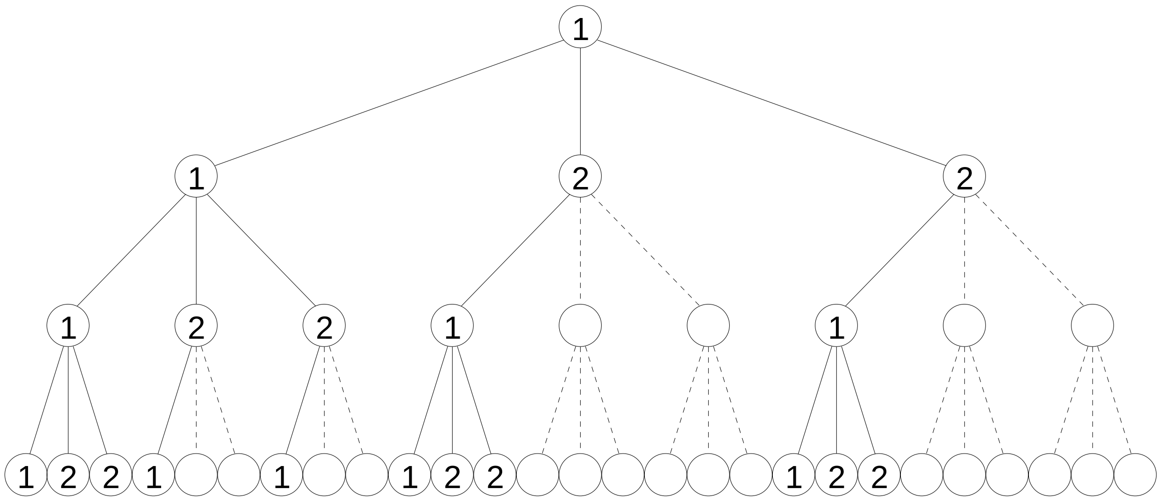

Figure 3: Minimal subtree when deep cutoffs are not considered

when a large number of processors is used. For the worst-first ordering, Fishburn used an example game tree of degree 38 and processor tree of fanout 2 to predict that speedup for MWF will satisfy

$$p^{0.93} \leq S \leq p^{0.96}$$

where p is the number of processors. This result is almost as good as tree-splitting.

3.3. The tree-splitting algorithm

Fishburn proposed this method as a natural parallel way to implement the alpha-beta algorithm. The tree-splitting algorithm splits the game tree into its subtrees at the root node, and each subtree is assigned to a pool of processors for evaluation [4]. The pool will evaluate the subtree in parallel if it has more than one processor. In other words, the game tree is mapped to a processor tree. as shown in Figure 4. Here we have a binary tree of processors with height two (connected by heavy arcs) mapped onto a ternary game tree. When there are more branches in the game tree than there are in the processor tree, the extra game tree branches are queued and assigned to a processor when one becomes available.

All interior processors in the tree of processors are both masters and slaves except the root processor, which is only a master. All the leaf processors are slaves. When a slave processor finishes the search of its assigned subtree, it reports the value computed to its master. When a master processor receives a response from one of its slaves, it updates its alpha-beta window and informs the other working slaves of this new window. The new window may allow the remaining work under the master to be cutoff. When all the slaves have finished, either by cutoff or by reporting their values, the master processor can compute the value of its own position.

Fishburn [4] calculates the speedup for the tree-splitting algorithm for two different orderings of the game tree. Worst-first ordering produces no alpha-beta cutoffs. It is achieved by sorting all children of all nodes so that whenever the call alphabeta(z, α, β)

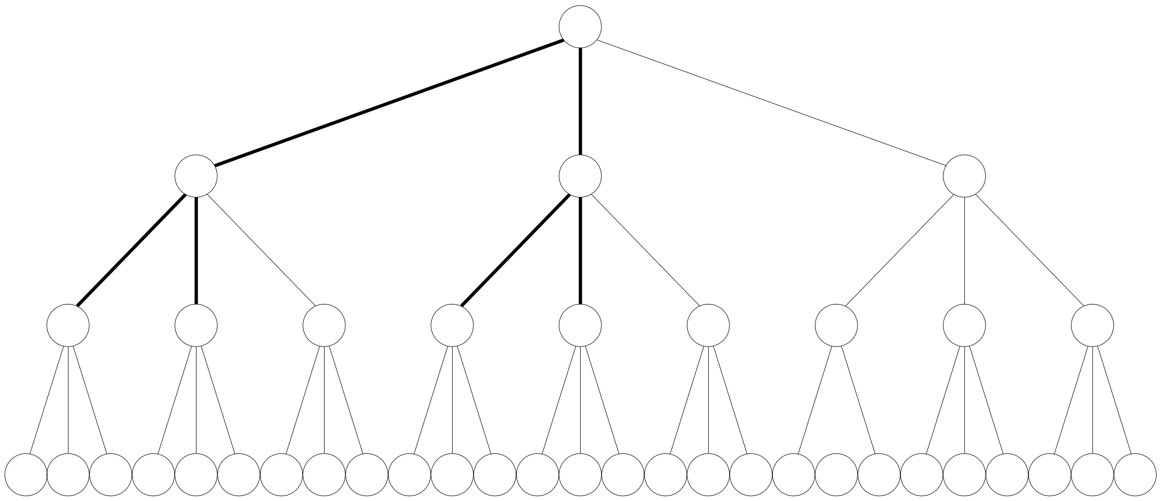

Figure 4: Processor tree mapped onto game tree

is made, the following relation holds among the children $z_1, ..., z_d$:

$$\alpha < -\text{negamax}(z_1) < ... < -\text{negamax}(z_d) < \beta$$

Since there are no cutoffs, there is no speculative loss, so tree splitting achieves practically perfect speedup.

Best-first ordering produces the maximum number of alpha-beta cutoffs. It is achieved by sorting all children of all nodes so that:

$$\text{negamax}(z) = -\text{negamax}(z_1) \text{ for all nodes } z \text{ in the game tree.}$$

Using this ordering, the tree-splitting algorithm gives $S = O(p^{1/2})$ with p processors.

The tree-splitting algorithm gives $S = O(p^{1/2})$ with p processors when best-first ordering [4] of the game tree is used.

3.4. Principal-variation (PV) splitting

PV splitting is a refinement of the tree-splitting algorithm [8]. It assumes that the search tree is mapped onto an underlying tree of processors and that the game tree is strongly ordered, that is, the first branch of each node is the best branch at least 70 percent of the time and that the best move is in the first quarter of the branches being searched 90 percent of the time.

The type-1 nodes are recursively evaluated until a given ply is reached, at which point tree splitting is used. After the value of the principal variation (type-1) node is backed up the tree, tree splitting is used to evaluate the remaining siblings if they can not be cut off.

There are two differences between PV splitting and MWF. First, PV splitting requires a particular underlying processor structure, in contrast with the pool of processors used in MWF. Second, it waits for the search of type-1 nodes to end before it starts evaluating the other nodes. This aspect of PV splitting ensures that the best available value of α is passed to the other nodes of the tree.

PV splitting was compared experimentally with the tree splitting algorithm using trees of depth 4 and width 24. Experimental results show that PV splitting outperforms tree splitting, especially when a wider processor tree is used [8]. For example, when a processor tree with both depth and width of 2 was used, tree-splitting examined 912 nodes, and PV splitting examined 648 nodes. But when the width of the processor tree was changed to 8, tree-splitting and PV splitting examined 772 and 277 nodes respectively.

3.5. The ER algorithm

This algorithm was developed by Steinberg and Solomon for parallel evaluation of game trees. It is a sequential algorithm with a parallel implementation [3]. The nodes in the game tree are divided into two groups, e-nodes and r-nodes. E-nodes will be fully evaluated, and r-nodes will be refuted, that is, will have an estimated value. All children of an e-node are evaluated, but as few as one child of an r-node needs to be examined. Therefore e-nodes are more ‘‘costly’’ than r-nodes. Every internal node has exactly one e-node child (e-child).

Any child of a node can be chosen as the e-child, but the child with the lowest negamax value is the best choice [3].

To choose the e-child of a node z, ER evaluates the elder grandchildren (eldest children of z's children) in parallel, and chooses the child whose elder child has the largest value. The e-child is then evaluated while avoiding re-evaluation of its oldest child, since we just got its value. The remaining children are refuted in order.

In the parallel implementation of ER, the elder grandchildren can be evaluated in parallel because they represent mandatory work. Since these grandchildren are themselves e-nodes, their elder grandchildren can also be recursively evaluated in parallel. These parallel evaluations are mandatory work, but if ER is to perform only the mandatory work, the remaining siblings of an e-node must be examined sequentially. To avoid these sequential evaluations and thus starvation loss, ER uses the following two methods:

- Parallel refutation: After an e-child y of an e-node z is evaluated, refute y's siblings in parallel. This parallel evaluation is likely to encounter lots of speculative loss. This work is similar to the speculative work performed by MWF and PV splitting algorithms.
- Multiple e-children: After an e-child of an e-node z is evaluated, choose the next best child of z as a second e-child. If it happens that the first e-child is not actually the best child of z (other children cannot be immediately refuted), we will have another e-child that will hopefully help us cut off z's other children. In general, after the first e-child has been evaluated, ensure that z always has one active e-child.

Steinberg and Solomon compared ER to PV splitting. Sequential ER evaluates more nodes than alpha-beta, but sequential PV splitting is identical to alpha-beta. For this reason Steinberg and Solomon [3] used relative efficiency and relative speedup as shown below to compare the two algorithms.

$$relative\ efficiency = \frac{relative\ speedup}{no.\ of\ processors\ used}$$

$$relative\ speedup = \frac{time\ required\ by\ parallel\ algorithm\ with\ 1\ processor}{time\ required\ by\ parallel\ algorithm}$$

Experiments show that ER achieves twice the efficiency and speedup of the PV splitting algorithm when used on sufficiently deep trees [3]. The average efficiency achieved by ER using 16 processors for 7, 8, 9, and 10 ply trees are respectively 0.44, 0.52, 0.68, and 0.58. The corresponding efficiencies for PV splitting are 0.28, 0.31, 0.31, and 0.31. The range of speedup for 7 to 9 ply trees using 16 processors are 7.1 to 10.9 for the ER algorithm and 4.5 to 5.0 for the PV splitting algorithm. Steinberg and Solomon contribute ER's higher efficiency to low starvation loss.

3.6. Delay splitting

This algorithm delays the evaluation of each node until its eldest sibling is completely evaluated. Starvation loss is accepted in order to increase the number of cutoffs. Evaluation delay occurs at every level for each node, thus making delay splitting different from PV splitting, in which delay of evaluation occurs only along the principle variation route.

The following is Pascal-like code for delay splitting:

```
function DelaySplit(z : position; α, β): integer;
    var d, i : integer;
        value : array[1..MAXWIDTH] of integer;
    begin
        if (I am a leaf processor) then
            return(alphabeta(z, α, β));
        determine the child positions z1, ..., zd
        α = -DelaySplit(z1, −β, −α);
        if α ≥ β then
            return(α);
        for i := 2 to d do in parallel
            begin
            value[i] := -DelaySplit(zi, −β, −α);
            begincrit {critical region }
                if value[i] > α then
                    α := value[i];
            endcrit;
            if α ≥ β then
                return(α); { cutoff }
            end;
    end. { DelaySplit }
```

4. Experimental results

We have tested the above algorithms on a Sequent Symmetry with 26 cpus using an unsorted tree of depth 9 and fanout of at most 15 (the fanout decreases by one at each level) generated by the game tic-tac-toe. The algorithms tend to have more cutoffs with a sorted tree, but are likely to have a higher efficiency with a worst-first sorted tree. Not sorting at all gives us a comparison with a reasonable amount of cutoff and a reasonable amount of parallelism. The relative efficiency comparisons are shown in Figure 5. (Relative efficiency compares the parallel execution time to the sequential execution time of the same algorithm in the same environment. The sequential execution times of all algorithms we tested are quite similar.) For this particular test, the MWF algorithm achieved an almost perfect efficiency, followed by delay splitting and ER algorithms with speedup of over 12 and 9 respectively, using 20 processors. We attribute our ability to exceed a speedup of 6 with MWF to DIB's parallelization environment and the tree structure used in this experiment.

Figure 6 shows the ratio of the number of nodes examined by the algorithms using different numbers of machines verses using one machine (sequential algorithm).

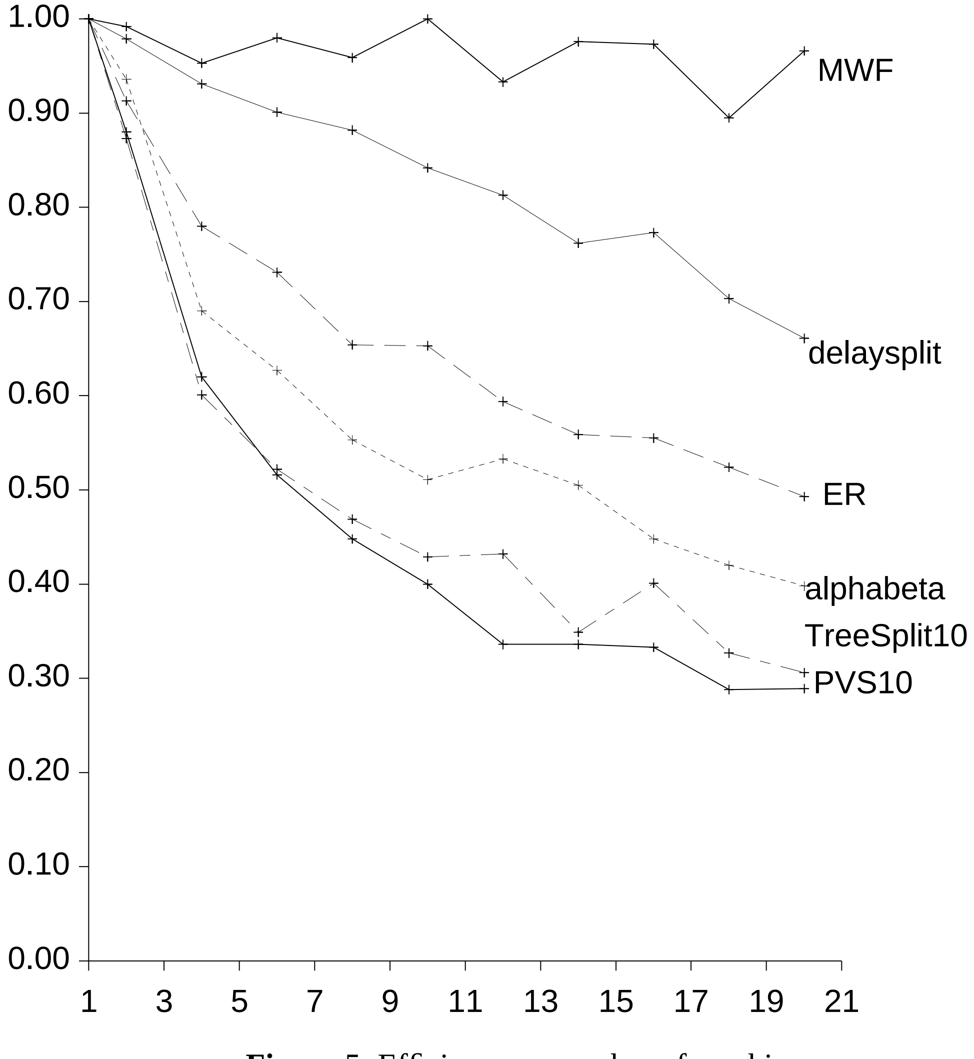

Figure 5: Efficiency vs number of machines

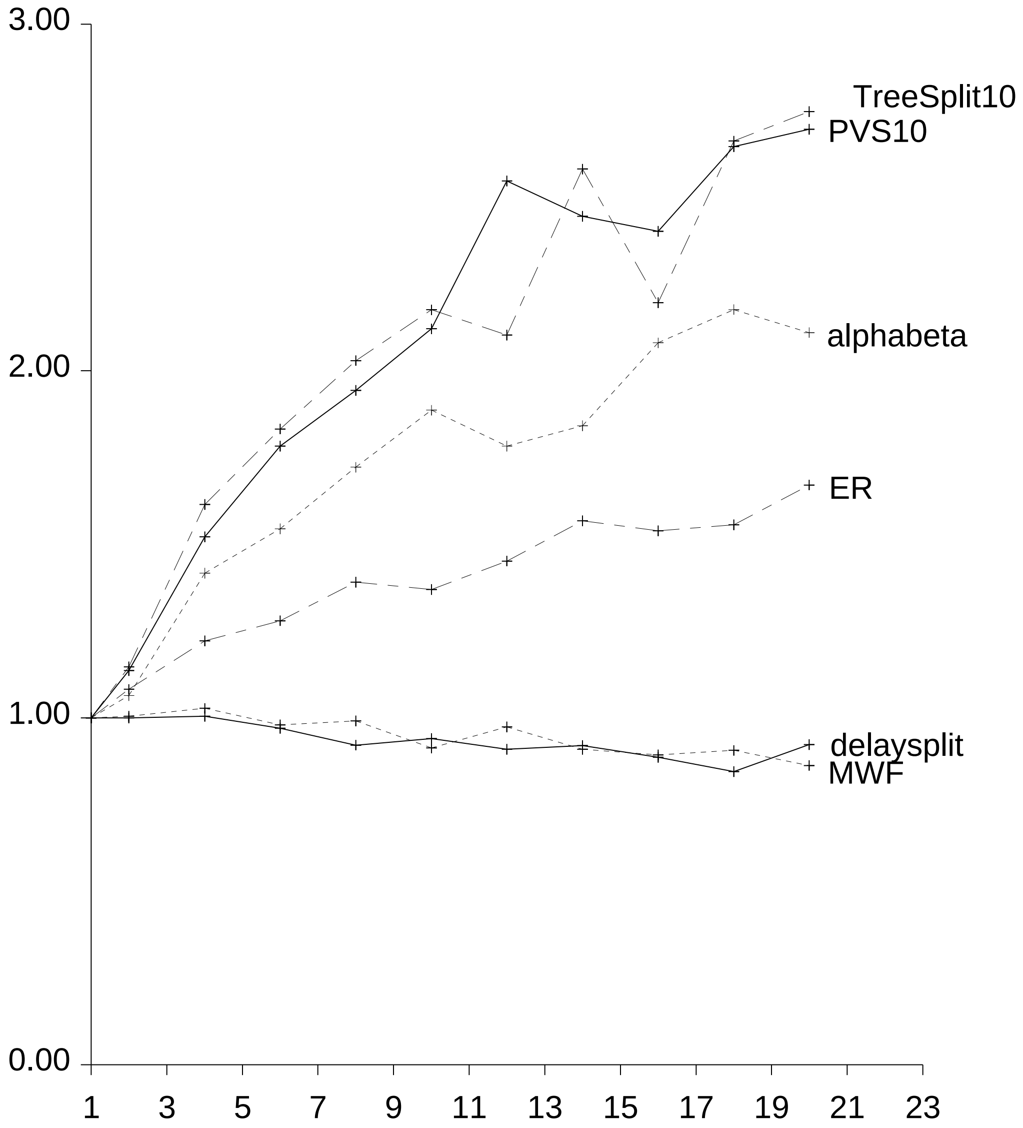

Figure 6: Relative no. of nodes examined vs. number of machines

We have also tested the algorithms using Othello with a 6×6 board. All algorithms have almost the same relative efficiency, with delay splitting leading when fewer than six processors are used. In this experiment, the speedup did not exceed 7 (Figure 7).

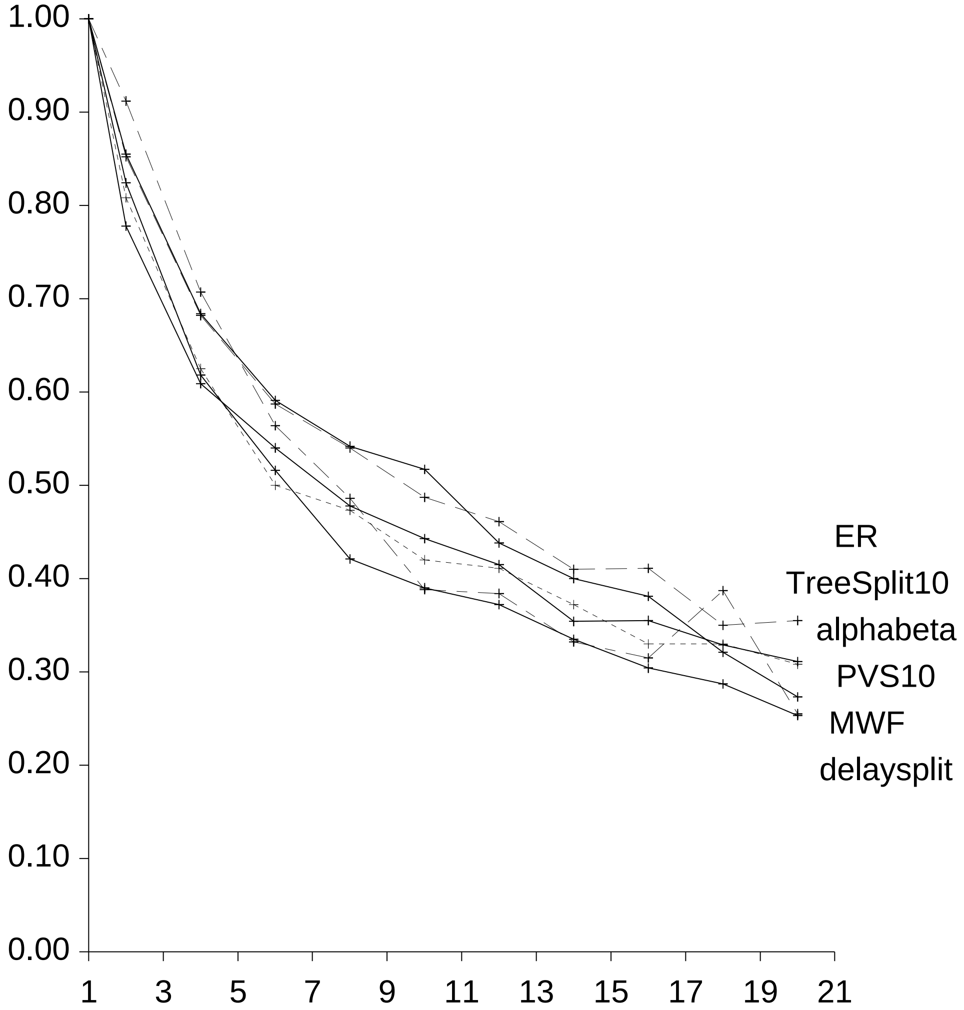

Figure 7: Efficiency vs number of machines

5. Sorting methods

We have compared the effects of four different sorting strategies on the search algorithms. In all the strategies, an **expansion depth parameter** specifies how many levels below a node are expanded in order to sort that node's children. We set the expansion depth parameter to 3 for the experiments reported here. That is, to sort a node z, we expand the tree to three levels below z, evaluate the leaves statically, apply alpha-beta to those values to back them up to node z's children, then sort those children accordingly.

Another adjustable parameter is the **sorting depth parameter**, which specifies the maximum depth of a node to which sorting may be applied. For some applications, like tic-tac-toe with a 4×4 board, every level of the search tree may be profitably sorted, but in other applications, sorting nodes beyond some level increases the total computation time; sorting time outweighs cutoff benefits. For example, in Othello with a 6×6 board and a search tree of nine levels, the best overall results are achieved when the sorting depth parameter is set to six levels.

Our first attempt at sorting was to sort only the children of the top node. This regime, called ‘‘TN (top-node) sorting’’, applies to all the algorithms except ER, which has its own internal sorting mechanism. There is no noticeable difference in number of nodes, total time, or efficiency for any of the algorithms between TN and no sorting at all.

Next we extended the sorting to include all nodes on the principal variation route. We call this sorting regime ‘‘PVR sorting’’. Our tests of PVR sorting for PV spitting, MWF, and delay splitting show no significant improvement in total computation time.

In the third sorting regime, we sort at the top node and at every eldest child. We call this sorting regime ‘‘EC (eldest child) sorting’’. EC sorting applies to MWF and delay splitting, since only in these two algorithms do we suspend evaluation of all nodes that are not eldest children until their eldest sibling is fully evaluated. Therefore having the best child evaluated first should result in more cutoffs.

As expected, the results are much better for EC sorting than PRV sorting, as evidenced by tests with our tic-tac-toe and Othello applications. The total computation time is almost reduced by half when fewer machines are used. The reduction in the efficiency, expected due to the improvement in the sequential performance of the algorithms, is not too bad. Figures 8, 9 and 10 show the number of nodes examined (in multiples of 1000), total time and efficiency comparisons for MWF using the 4×4 tic-tac-toe game.

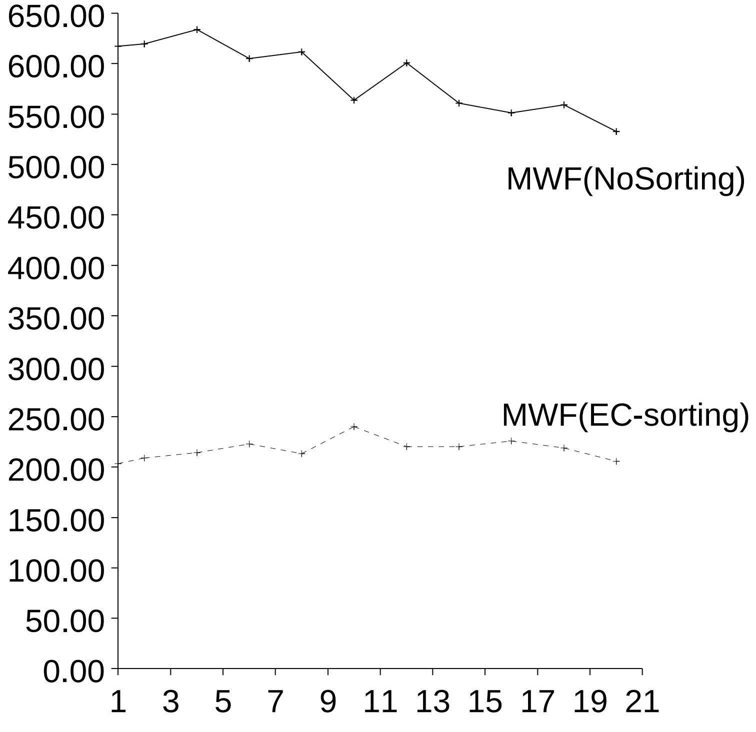

Figure 8: Nodes examined vs. number of machines

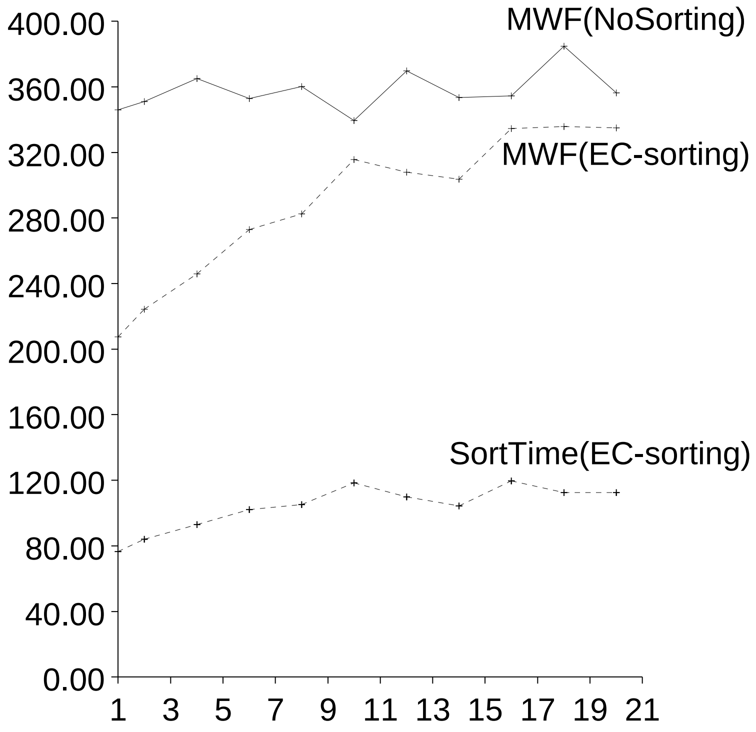

Figure 9: Time vs. number of machines

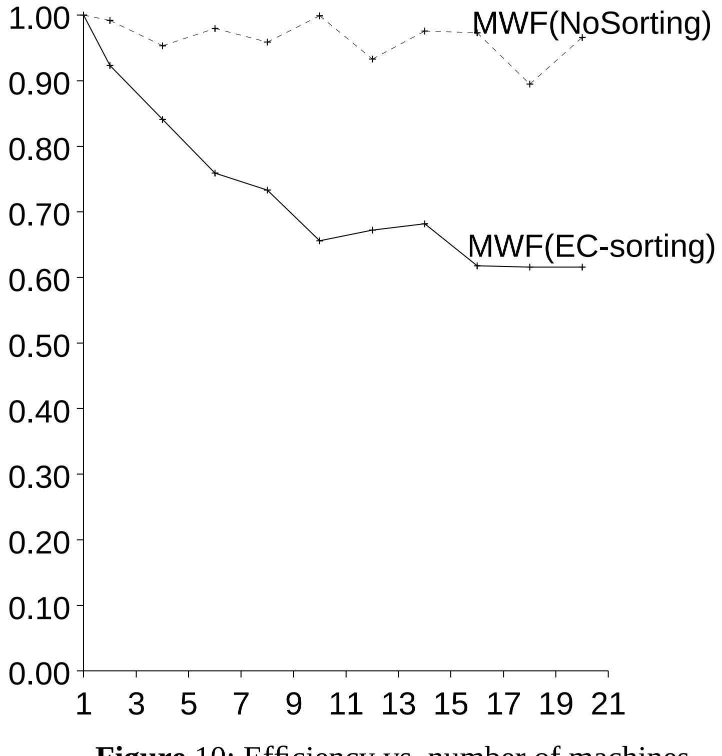

Figure 10: Efficiency vs. number of machines

The last sorting regime is motivated by considering the type-2 nodes in MWF. The eldest child of a type-2 node is evaluated before its other children are generated. Therefore, if we also sort the children of a type-2 node there should be even more cutoffs. In this sorting regime, in addition to sorting children of every eldest child, we also sort at every type-2 node. We call this regime ‘‘TT (type-two) sorting’’.

TT sorting resulted in a vast improvement in total computation time and the number of nodes generated. **Unfortunately, a subtle bug in our implementation (since fixed) renders all our conclusions about this sorting method inaccurate; previous versions of this report should not be trusted in this regard. In fact, what we implemented did not evaluate type-2 nodes as deeply as type-1 nodes, so far fewer nodes were evaluated. So TT sorting (in this report) implies a different evaluation strategy as well.** The computation time is almost 30 times faster than the computation time using EC sorting for a 4×4 tic-tac-toe game, even though about 95% of the time is spent on sorting in the 1-machine (sequential) case. These results are shown in Figures 11 and 12. The efficiency is drastically reduced because the work does not seem to be divided evenly between the machines. Our assumption is that a better distribution of work can be achieved by some adjustments to DIB itself so that this great speed can be accompanied by a better efficiency.

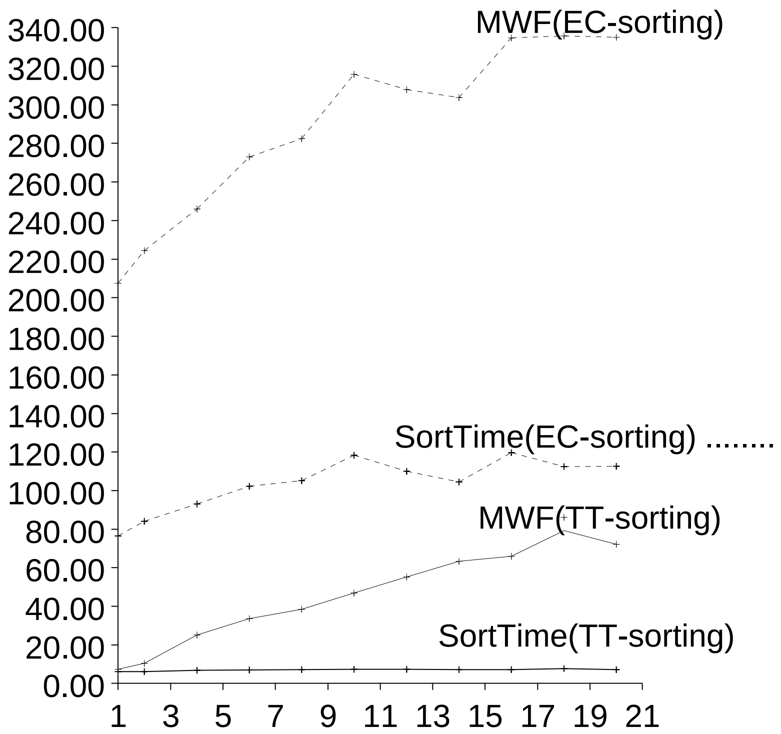

Figure 11: Time vs. number of machines

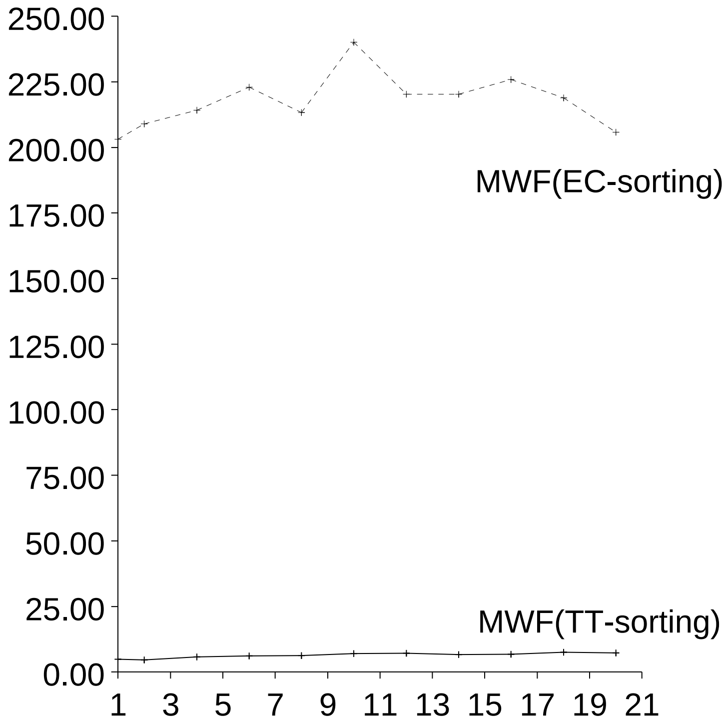

Figure 12: Nodes examined vs. number of machines

We also used three-level sort for TT sorting. The great speed in this method allowed us to build a complete game tree for our experiments. In previous experiments with tic-tac-toe, we built a search tree with nine levels; the complete search tree for a 4×4 board has 16 levels.

6. New results

We have new results based on adjustments made to MWF concerning the selection of type-1 nodes. In dynamic MWF (DMWF), we decide which children of a type-1 node to fully investigate not by taking the first (as in MWF), but by taking all those whose static value is above the 90th percentile of its siblings. To our knowledge, no one has tried algorithms that dynamically adjust their width of full evaluation based on evidence provided in the tree. This adjustment can also be made to other algorithms like delay splitting and ER.

We compared MWF and DMWF under EC sorting for a tic-tac-toe tree with a 4×4 board. Figure 13 shows the comparison of the total time (including communication and idle time) between MWF and DMWF. DMWF is about one-third faster than MWF. This speed is achieved with even less sorting time because it generates fewer nodes, as demonstrated in Figure 14 (number of nodes are in multiples of 1000). There is also an improvement in the efficiency (Figure 15).

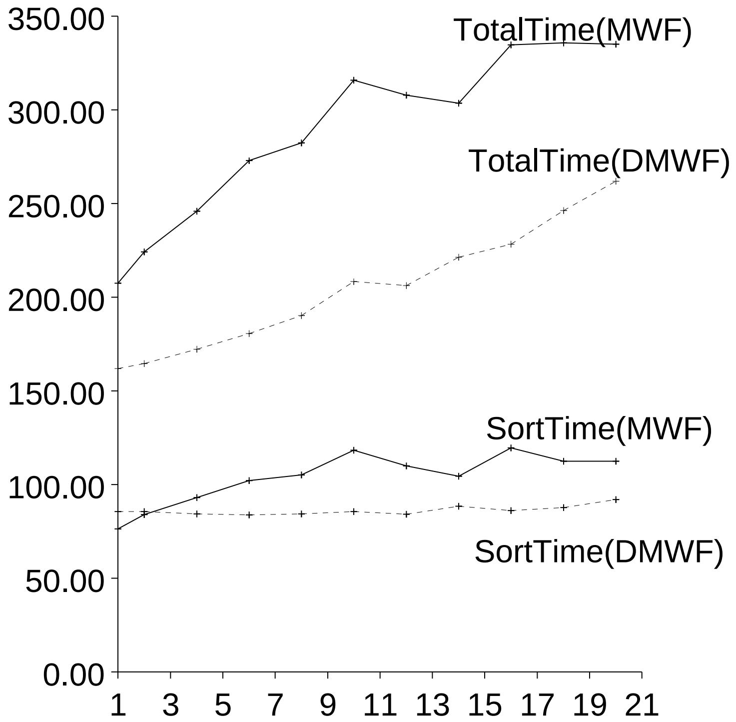

Figure 13: Time vs. number of machines

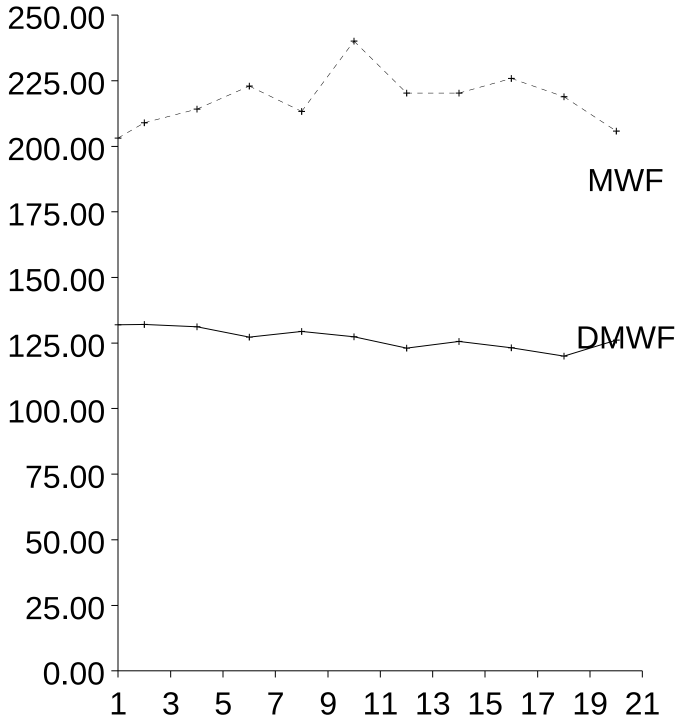

Figure 14: Nodes examined vs. number of machines

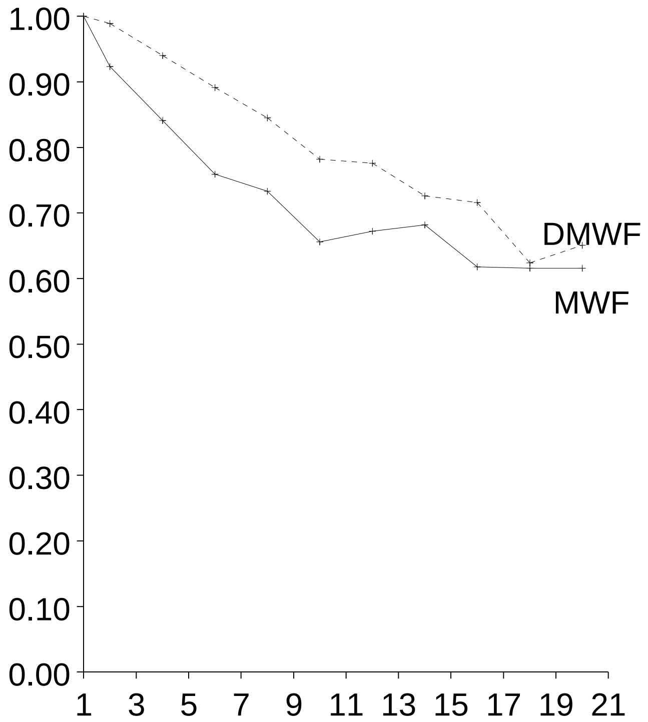

Figure 15: Efficiency vs number of machines

We also compared MWF and DMWF under EC sorting for Othello with a 6×6 board. Since Othello is a more complicated game than tic-tac-toe, the sorting depth parameter was set to 6 levels, as mentioned above. The results of this experiment, shown in Figures 16, 17 and 18, are similar to the results from tic-tac-toe.

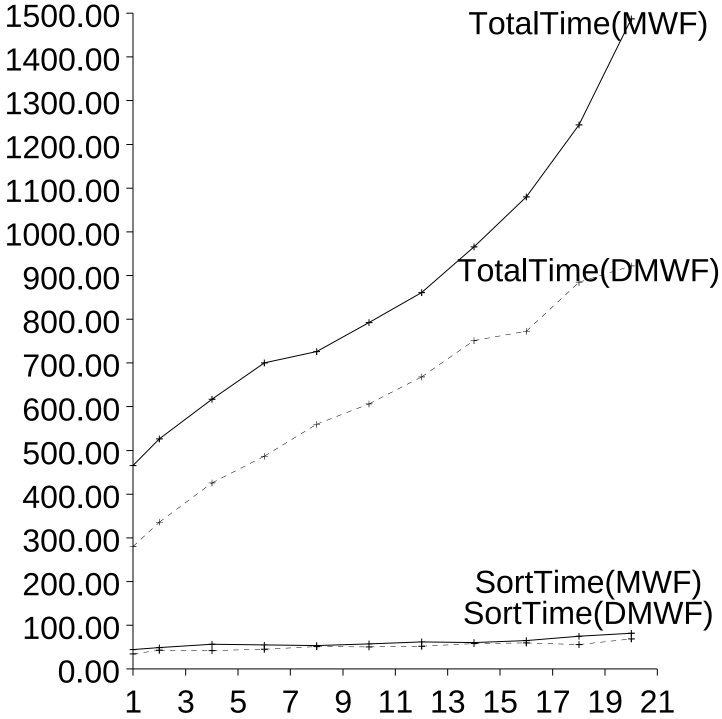

Figure 16: Time vs. number of machines

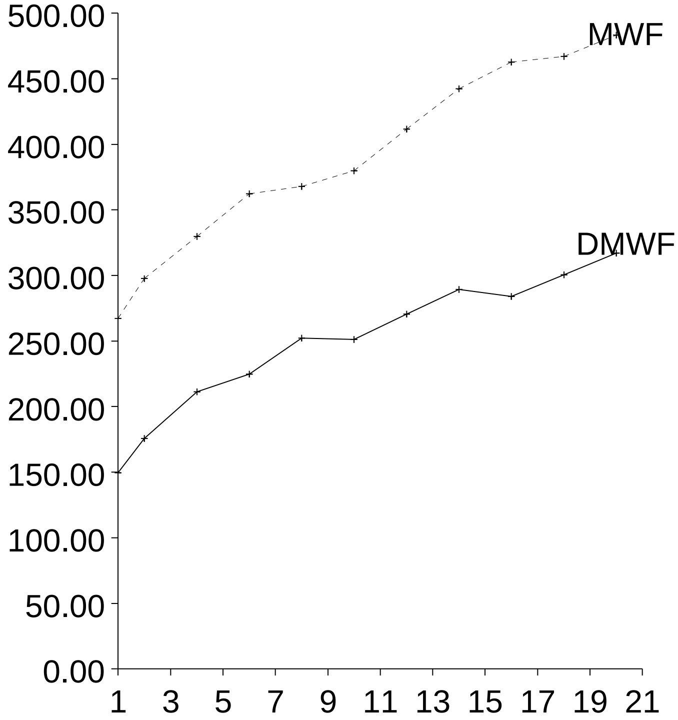

Figure 17: Nodes examined vs. number of machines

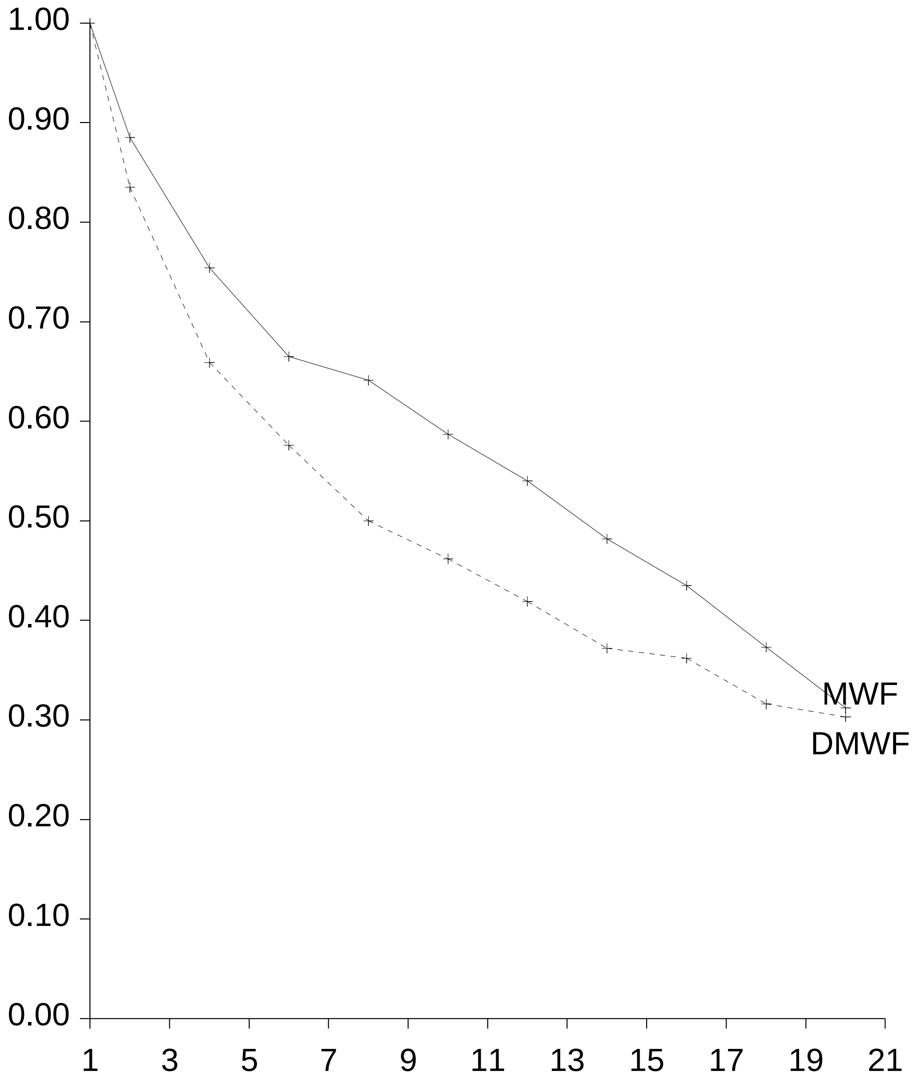

Figure 18: Efficiency vs number of machines

7. Future work

With enhancements made to DIB for achieving high efficiency in game tree search, we have developed an environment in which we can test different algorithms in a consistent way. These algorithms have been examined using a test suite of problems taken from game trees for checkers, tic-tac-toe, and Othello. These games are coded independently from the search algorithms, thus contributing to the consistency of the experiment.

We are currently testing these algorithms on a 26-processor Sequent Symmetry machine. Our future plans include the use of a KSR multicomputer with 64 cpus to examine the search algorithms with a larger number of processors.

We will also experiment with worst-case sorting, not because it would be used in practice, but to see how each algorithm is sensitive to sorting.

References

1. Raphael Finkel and Udi Manber, ‘‘DIB — A Distributed Implementation of Backtracking,’’ *ACM Transactions on Programming Languages and Systems* **9**(2) pp. 235-256 (April 1987).
2. Vipin Kumar and V. Nageshwara Rao, *Scalable Parallel Formation of Depth-First Search*.
3. Igor Steinberg and Marvin Solomon, *Searching Game tree in Parallel*.
4. John Philip Fishburn, ‘‘Analysis of speed up in Distributed Algorithms,’’ Ph.D. Thesis, Department of Computer Science, University of Wisconsin-Madison (1981).
5. D. V. Knuth and R. W. Moore, ‘‘An analysis of alpha-beta prunning,’’ *Artificial Intelligence* **6** pp. 293-326 (1975).
6. Selim G. Akl, David T. Barnard, and Ralph J. Doran, ‘‘Desing, Analysis, and Implementation of a Parallel Tree Search Algorithm,’’ *IEEE* **PAMI-4**(2)(March 1982).
7. R. A. Finkel and J. P. Fishburn, ‘‘Parallelism in alpha-beta search,’’ *Artificial Intelligence* **19** pp. 89-106 (1982).
8. T. A. Marsland and M. Campbell, ‘‘Parallel Search of Strongly Ordered Game Trees,’’ *Computing Surveys* **14**(4) pp. 533-551 (December 1982).