Abstract

This paper describes a high-performance parallel tree search algorithm that uses Dynamic Tree Splitting (DTS) to search alpha/beta minimax game trees, specifically those for the game of chess. This algorithm divides the search tree among several processors on a shared memory parallel machine. This paper discusses the following topics: (1) the DTS algorithm, (2) how the tree is divided into parallel pieces that can be searched in parallel; (3) performance results of the algorithm and (4) analysis of the results to see where further improvements might occur..

1. Introduction.

Over the past ten years, high performance computer architectural designers have turned to parallel processing to push the performance envelope to higher and higher levels. Unfortunately, for many algorithms in general, and the alpha/beta algorithm in particular, parallel processing offers a substantial hurdle to extracting maximum performance from parallel machines.

The alpha/beta algorithm is based on a highly sequential property of searching that depends on prior knowledge to prevent searching parts of the game tree that have no bearing on the final results [Knuth75,Hyatt89]. Since a parallel search traverses parts of the tree "simultaneously" such a priori knowledge is not always available [Campbell81,Hyatt88,Hyatt89,Hsu90,Lindstrom83,Marsland85, Popowich83,Schaeffer89,many others].

The best-known early attempt at searching such trees in parallel was the Principal Variation Splitting (PVS) algorithm [Hyatt86,Marsland80,Marsland81, Marsland82,Marsland86,Newborn85,Schaeffer89]. This was both simple to understand and easy to implement. When starting an N-ply search, one processor generates the moves at the root position, makes the first move (leading to what is often referred to as the left-most descendent position), then generates the moves at ply=2, makes the first move again, and continues this until reaching ply=N. At this point, the processor pool searches all of the moves at this ply (N) in parallel, and the best value is backed up to ply N-1. Now that the lower bound for ply N-1 is known, the rest of the moves at N-1 are searched in parallel, and the best value again backed up to N-2. This continues until the first root move has been searched and the value is known. The remainder of the root moves are searched in parallel, until none are left. The next iteration is started and the process repeats for depth N+1.

Performance analysis with this algorithm (PVS) produced speedups given below in table 1 [Hyatt88] (others have produced results very similar to this). Clearly the performance curve is flattened badly on the upper end, and the C90 with 16 processors and the T90 with 32 are not going to perform as well as might be expected, although they are certainly very fast. An upper bound of 5X (maybe) does not look very attractive, however [Hyatt88].

# processors	1	2	4	8	16
speedup	1.0	1.9	3.4	5.4	6.0

Table 1 PVS performance results

This approach has a couple of fairly obvious drawbacks. First, notice that all of the processors work together at a single node, searching descendent positions in parallel. If the number of possible moves is small, or the number of processors is large, some have nothing to do. Second, every branch from a given position does not produce a tree of equal size, since some branches may grow into complicated positions with lots of checks and search extensions that make the tree very large, while other branches grow into simple positions that are searched quickly. This leads to a load balancing problem where one processor begins searching a very large tree and the others finish the easy moves and have to wait for the remaining processor to slowly traverse the tree.

This second problem is a serious drawback, because with a reasonable number of processors, the speedup can look very bad if most of the time many of the processors are waiting on one last node to be completed before they can back up to ply N-1 and start to work there.

Because of this, when we first moved Cray Blitz to the eight processor Cray YMP, we discovered that the performance was only marginally better than it was on the four processor XMP, if you discount the improved clock speed of each YMP processor. The first approach we developed was called Enhanced Principal Variation Splitting (EPVS) and is also pretty easy to understand and implement [Hyatt88,Hyatt89]. EPVS simply notices whenever a processor at a node runs out of work. When this happens, assuming there are M processors working, we know that there are now M-1 (or less) active branches remaining. EPVS immediately stops all processors (which are working at ply P, the current position) and follows the first remaining ply=P branch two plies and then splits the tree at that point with a parallel search exactly like the old PVS idea. The benefit is that now all processors are working together on a single branch at a node where there is little work, and have stepped down into the tree to a node with more moves to search, and also a node where the descendent trees will be smaller since this node is two plies deeper in the tree. It should be noted that this is effective because the transposition table stores the partial results from the searches that are stopped, so that when these branches are searched later, the work is not repeated because the information is found in the table. In case you might wonder why the search steps further into the tree two plies, rather than one, recall that with alpha/beta, if one ply (P) requires that all moves be searched, then the next ply (P+1) only requires that one move be searched to produce a refutation (except for the case of PV nodes where no moves have yet been searched.) Therefore, since we know that we are going to examine all nodes at P (since we just backed up the PVS algorithm to ply P), we know that all nodes at ply P+2 will most likely have to be searched as well, so long as we choose a reasonable move at ply=P+1.

Performance analysis with this algorithm (EPVS) produced speedups that were modest improvements over PVS, as given in table 2 below. This is roughly 20% faster than PVS on larger numbers of processors, from the results below, but the 8 and 16 processor times still are disappointing, and the 32 processor results would be abysmal. Clearly, a better algorithm had to be found, or we were going to see a vast amount of unused processing power on future Cray computers [Hyatt88].

# processors	1	2	4	8	16
speedup	1.0	1.9	3.4	5.4	6.0

Table 2 EPVS performance results

2. Dynamic Tree Splitting algorithm (DTS)

Before describing the DTS algorithm in detail, one important point needs explanation. This algorithm is specifically designed for a shared memory multiprocessor architecture. As a result, issues that other programs and programmers have to face (distributed transposition tables, killer moves, history information and such) are not an issue here at all. Some of the decisions in the DTS design would likely cause significant problems on a message-passing architecture, no matter how fast the communication channel might be. On the Cray, which has an absolutely astounding cpu-to-memory bandwidth, moving a megabyte of data takes very little time at all, which simply means that in this algorithm, communication costs are always assumed to be zero. An assumption that would fail on other architectures of course, but one which certainly affected decisions made while designing the DTS algorithm as well. It turns out that loading a "flag word" (which indicates that there is some parallel-processing related task to perform) and testing it can be buried in the instruction stream so that they cause no delays at all.j

2.1 The DTS approach

In analyzing the previous two algorithms, they both suffer from two distinct problems: (1) all processors work together at a single node, which is acceptable for middle-games with 35+ legal moves (average), but which does not look attractive in end-games where the number of legal moves is much smaller; (2) if one processor selects a move that leads to a very bushy tree, the other processors might end up waiting for an extended period before that tree is completed, producing long periods where most processors are waiting, and greatly degrading parallel performance. The DTS algorithm was specifically developed to avoid both of these problems.

The first design goal in developing the DTS algorithm was to completely eliminate the cases where a processor was idle, with no moves left to search, while other processors were busy. In accomplishing this, it became obvious that the first step was to disassociate all of the processors, so there was no requirement that they all congregate at a single node and work together searching moves until all were completed.

Note that there are alternative ways to keep processors busy, besides trying to have them all work together. One such approach was used in Phoenix [Schaeffer89], and used two separate search engines, one a full chess program, the other (called Minix) was a material-only searcher that searched deeper to confirm/reject moves proposed by the full chess program. Obviously there is duplicate work done in this approach, as well as other problems related to choosing a move to play when the positional search says play X, but the tactical search says "no." This sort of speculative computing, which is an admission that additional processors won't speed up the basic search, was deemed too inefficient and ineffective to consider, and something better was desired.

The second design goal was to avoid completely ripping Cray Blitz apart, effectively starting from scratch; rather, we wanted to use the same move generator code, tree search code, move ordering code, and so forth for both a one-processor sequential program and the multi-processing version.

A simple explanation of DTS is fairly easy to understand, but it also hides a great deal of complexity that has to be overcome when writing and debugging the code. At the beginning of a new iteration, DTS behaves just like PVS, searching down the left-hand side of the tree from ply=1 up to ply=N, with only one processor. At ply=N, all processors "join the party" and search the ply=N move list in parallel, just like PVS/EPVS. However, when a processor completes the move it is searching, and discovers there are no more to be searched, DTS reacts to this.

This idle processor broadcasts (using shared memory) that it is idle, and is available to "help" any other processor finish searching its tree. The busy processors collect the "state of the tree" data, and store it in shared memory for the idle processor to examine. This idle processor analyzes this data, and decides which (if any) of the busy processors seems to have a tree that is complicated enough that it would be efficient to help with the search. If such a position is found, the idle processor informs the processor which owns that node of this and they "join" forces.

To accomplish this, the idle processor specifically chooses a ply=S position and tells the owner that S has been chosen as the split-point ply. The processor with that subtree in progress then copies the complete tree state to a shared memory area called a "split block" (this tree state includes the various search bounds, move lists for each ply under analysis, current board position and other related search data such as the repetition list and so forth). Both processors can now extract moves from this shared data and search in parallel. Whenever one runs out of work, it simply repeats this process. In some cases, we might see two processors split the tree at ply=8, then one runs out of work, and decides that the other one has the best split candidate, and they then split at ply=10. The first might run out of work at that split point, and help the other at a new split point at ply=12. Or it might bail out completely and choose to help another processor, since going deeper and deeper splitting the tree means that each subtree searched is smaller and smaller, and eventually the two processors might spend as much time splitting and sharing as they do searching (more about this possible condition later.)

The overall effect of this is that the processors start together on a single node, searching in parallel, but they quickly spread out in groups over the active tree, helping at points where there is work to be done.

Now the next issue. Notice that as a processor becomes idle, it can re-split the same tree it was sharing with another processor, but so far, only at deeper plies. Why is this? Because it is inefficient to back up to ply N-1 and search branches there before ply=N has been completely searched (remember the first ply=N split node is a PV node just like PVS, and we don't want to start working on ply N-1, until we have the actual score for ply=N). However, eventually, the trees can't be further sub-divided any further because the cost of doing so would be more than the cost of searching the subtree. Now a critical decision has to be made, "what to do next?"

Since we are using the best-known move ordering heuristics in the alpha/ beta tree, it is likely that by the time the first few moves at ply=N have been searched, we know the actual score for this node, as moves further down the list should be worse and not improve the score further. Then, allowing idle processors to back up to ply=N-1 and start searching there before ply=N is completed is probably safe, and this is what DTS does. On the rare occasions when the ply=N search completes, and the last branch produces an even better score, the processors that have backed up to ply=N-1 already are searching with a less efficient bound. DTS notices this and each processor already searching at N-1 is given the new (correct) bound as soon as it is known. This sounds easy and clean, but it can cause some interesting problems, because this new bound might mean most of what a processor has been busy doing can be thrown away since the new bound would have caused a cutoff much sooner than the original bound used.

The best metaphor for describing this might be "ants at a picnic" where the chess tree is the picnic, and the ants are the search engines (processors). The ants start eating on a large crumb, but as it is eaten away, it becomes too small and the ants that are displaced move to another crumb and start working there.. Eventually lots of ants are congregated around a steadily diminishing supply of crumbs, which brings up the final phase of the DTS algorithm development.

In testing this code, we found, on occasion, a position that would blow up and produce search times much longer than even the single-processor search. For example, 1 processor would take 1 minute to do a 10 ply search, but with DTS, out of 5,000 test runs, one run would take 1 minute to do a 5 ply search with 16 processors. What we found was a condition we later termed a "feeding frenzy" to maintain the ants analogy. What was happening was that as the tree was nearly completed, the search reached a point where there were no good "split points". DTS was quite good at recognizing when it was appropriate to split a tree into parallel slices and when it was not. Unfortunately, it was not so good at determining when it was time to quit checking to see if there were any good split points.

Imagine a case with 16 processors, with 15 idle, and one working trying to search the last 15 nodes left in the entire tree. This one processor keeps getting interrupted with "may I help you?", "can I help you?", "could you use some help there?" and so forth, so that rather than searching, it stays busy copying its tree state to shared memory so that the idle processors can attempt to find a split point. They fail, and ask again, and again. And in a very few pathological cases, this almost hung the search. To solve this problem, which only occurred right at the end of an iteration, we chose to add a "thrashing" counter, which was nothing more than a counter that was set to some value when a processor handled one of those "may I help you?" queries. Until that processor searched N nodes (N was the value set in the thrashing counter) it would refuse to answer any more "may I help you?" queries, which eliminated the thrashing right at the end of an iteration.

That is a basic explanation of DTS. However, to appreciate exactly how complex and sophisticated this eventually became, the next step is to look at some of the specific components and data structures need to make this algorithm perform. First, we need a couple of definitions to make sure terminology does not obscure the algorithm.

2.2 Definitions

Split-point is a node within the tree where it appears that the search must examine all of the successor nodes. The quality of a split point is related both to its depth (nodes closer to the root of the tree represent larger subtrees and thus are better split points) and the confidence that all of the branches must be searched must be high. This confidence is described in section three which discusses choosing split points.

Branching factor is a measure of the complexity of a position reached in the tree search which characterizes this complexity in terms of the average number of successor branches (legal moves) from this position. Chess seems to average around 38 according to many published papers, but the number can vary from zero (mate or stalemate or draw) to around 200 (with many queens and an open board.)

2.3 The HELP command

The HELP command is the primary signaling mechanism within DTS. Whenever a processor is "out of work" it sends the HELP command to what is effectively the entire group of active processors.

The HELP command simply requests that any processors that are actively searching subtrees temporarily stop, copy the "tree state" to shared memory, and then continue searching.

As these "tree states" become available, the idle processor that initiated the HELP command analyzes each state to determine if it can find a satisfactory split point. If not, it simply re-broadcasts the HELP command.

As will be shown later, finding a split point is non-trivial, because it is a key step in the DTS algorithm. If an incorrect split point is chosen, then it is likely that the processors that start to help at that point are going to search nodes that are not necessary, which does nothing but increase the parallel search overhead without making the search progress any faster through the tree.

An interesting feature here, is that once a processor finds a viable split-point, and the Split() operation is performed, whenever any other processor becomes idle, they check for active split-points before broadcasting a HELP command. If a processor locates any valid split points, and there is work remaining at any of them, it simply attaches to the split point with the most work remaining, and does not broadcast a help command. This tends to keep the processors working together in related parts of the tree, which makes the transposition table and scoring hash tables more effective since the information is more useful among closely related positions. However, if a processor becomes idle, and finds no split-point with work remaining, it does initiate a HELP command and goes about finding a new split-point. Other processors will likely join it at that split point as they become idle. Another reason for doing this is that finding a split point takes time, and once a good one has been located, there's no point in locating another until the first one has been completed, or at least until there are no more moves left at that point that can be searched by idle processors. In the very worst case, with N processors each split point could have two processors working, if the tree is very narrow. In the optimal case, which is fairly common in the middlegame, there are rarely more than one or two split points, the first is nearly completed, and the second (new) split point is where each processor goes when it runs out of work at the original split point.

To process the HELP command, a simple test was added to the sequential search code. After any node is processed, HELP(i) is checked, and if set, the tree state is copied to shared memory, the flag is cleared, and the search continues normally. This reduces the idle time for a processor to the time required for a processor to search one node and return to the top of Search() and then store the "tree state" for the idle processor to examine. (About 30 microseconds on a Cray C90.) After this time, an idle processor will have at least one tree state to examine for split points, making this reasonably efficient.

While it sounds like a lot of wait time could accumulate, many tests on a C90 have confirmed that this wait time is negligible. In searching for 5-6 minutes of wall-clock time, typical idle times (per processor) average between .01 seconds and .1 seconds. On rare occasions (notably very simple endgame positions where the branching factor is quite low) this has reached one second per processor, still keeping the idle time per processor down to less than .3% of the total search time. One major reason for this is that the code which selects split points is very careful, and favors nodes near the root of the tree, because the subtrees below those nodes represent a significant amount of work that must be completed before another split point must be found. The only time that processors are idle is while they are waiting on other processors to provide tree state information to one idle processor, or while they are waiting on one idle processor to select a split point. If this event is relatively rare, as it typically is in DTS, then the opportunities for accumulating excessive idle time are few and far between.

2.4 The Split() operation.

Split() is called whenever an idle processor has one or more "tree state" blocks to examine when attempting to find a split-point. Split() first attempts to find a good split point (section three) and then sets things up so that the Select() operation described later can function correctly.

The first operation performed after a good split-point is located is to allocate a data structure known as a SPLIT BLOCK. This data structure is simply an exact duplicate of the tree search data structures that already exist in each processor's local memory area. It contains the various move lists, board information, scoring information, etc.

The Split() operation then copies ALL of the current processor's local tree state information to the SPLIT BLOCK. (ALL here means that if the current ply is five (5), then all of the data for plies 1-5 are copied, but not all of the useless data below ply 5. Note that EVERYTHING is copied, so that this global (shared) memory area has an exact copy of the current processor's search data structures. It would now be possible for this processor to exit, and another processor could copy this global data to its private memory area and resume the search just as if it were the original processor since it now would have access to everything the original processor had. (Note: Cray Blitz was written before Fortran supported recursion, and as a result, it did not use the now in vogue "negamax" recursive alpha/beta search implementation. As a result, there was no problem in implementing the parallel search as explained. Negamax would have made it much more difficult, since the call stack is inaccessible to the search code and moving the tree state around would have been much more difficult [Pearl80].)

The Split() operation then passes the address of this SPLIT BLOCK to any processors that are currently idle, so that they may begin working at this point immediately. The idle processors drop out of their busy wait loop since they now have a pointer to a valid SPLIT BLOCK rather than a value of "zero". They must first copy the global data to their private memory area, and then begin searching at that point.

The primary reason for copying ALL of the "splitting" processor's data to the shared memory region is to avoid any type of synchronization delays after searching all branches from the split-point. The specific problem addressed is that the "controlling" processor (the one splitting work for other processors to help with) might choose a branch that is very simple with a minimum number of nodes and finish this branch quite quickly. Another processor might choose a branch that has many checks and/or search extensions and requires an extended amount of time to search. The controlling processor would often find itself waiting on other processors to complete their searches and return the values for their subtrees before the controlling processor could back up the proper value.

With the current implementation, this never happens. Since ALL processors have a complete copy of the search data, any of them is capable of finishing the search of the nodes in the tree after the split-point is completed. For example, processor 1 can search from plies 1 through 5 and then another processor selects ply=5 as a split-point. Processor 2 joins the search at this split-point and selects a branch to examine. Processor 1 completes the remainder of the branches at this split-point and then sends a HELP command to processor 2 so that it can help with the remainder of the complex branch processor 2 is searching. When the search finally backs up to ply=1, EITHER processor (but not both) could be in charge of the search at that point; it depends on which one finishes first. In simple terms, this is a "peer-to-peer" design, rather than a "master-slave" design, so that all processors are equal.

The nice feature of this approach is that whenever ANY processor runs out of work at a split-point, it then assists one of the busy processors by first sending a HELP command to force the creation of a new split-point where this idle processor can "join in." This feature is implemented by a simple rule that prevents any processor from backing the search up past a split point, unless (if and only if) only one processor is working on that split point. The best visual analogy is the game of "leap-frog" where one processor starts on a position, another helps it, the first finishes and "leap-frogs" the second to a position further into the tree and helps the second processor search at that point, and so forth.

2.5 The Select() operation.

one of the two goals mentioned when designing the DTS algorithm was to avoid completely re-writing the search code for Cray Blitz. This code (approximately 10,000 lines of FORTRAN, which are replaced by about 20,000 lines of Cray Assembly Language [CAL] when running on the C90) has been developed and debugged over a period of years, and altering it significantly was something the authors wanted to avoid if possible since debugging the parallel processing code already loomed as a large obstacle.

For simplicity, the design retained the original programming methodology that a processor maintains the tree (subtree, actually) it is searching by keeping all of the related data in local (private, task common, etc.) memory so that other processors can not "change" this information directly.

However, since a group of processors must work together at some point within the tree, shared memory is also required to allow this communication. The simple implementation mechanism we chose was to modify the procedure Select() so that it could work in this environment.

Select() is a procedure called to select the next move at the current node in the tree. Normally, it examines the move list in the private memory for the current processor and removes one of the moves for examination. At a split-point, this is slightly more complicated since multiple processors are "sharing" this move list.

Select() is unchanged, the only code that was modified was the code that actually calls Select() within Search(). Search() simply tests to see if the current ply is a split point, and if so it copies the shared move list to the local move list, calls Select() to choose one for searching and then removes it from the move list. Search() then copies the local move list back to the shared one (all protected by a semaphore so that no race conditions arise). If the current ply is not a split point, Select() is simply called normally. the following code shows how this was implemented. The major benefit here is that Select() is a very large block of code, that generates moves, orders the list, selects moves from the list based on things like captures and the expected gain, killer moves, and so forth. It was considered very desirable to leave this completely alone, since the assembly version of this module is several thousand lines long.

```
call Select()
```

was replaced by the following lines of code:

```
if(ply .eq. splitl(taskid)) then
   call Selget()
   call Select()
   call Selput()
else
   call Select()
endif
```

In the above code, Selget() sets a semaphore to lock this move list (there are multiple semaphores so that multiple move lists can be shared without unnecessary interaction) and then copies the global move list to the local memory. Select() then operates normally. Selput() then copies the move list back (after a move has been removed for searching) and finally clears the semaphore.

The vector splitl(taskid) is set up by the split operation and identifies the current split point for this processor (taskid). It should be noted that this implementation is quite good in terms of efficiency, because the Selget() (copy shared memory to local memory) and the Selput() (back to shared memory) operations are rarely used, and on a Cray they vectorize quite well and cost very little. Note that this same code works well without parallel processing since Selget() and Selput() are never called if a Split() operation has not been performed.

2.6 The Merge() operation.

Whenever a processor calls Select() and is told that the moves list is empty, the first test made by the DTS search algorithm is "is this a split-point?" If the answer is yes, this processor has computed a value that represents only a subset of the branches from this node. The DTS Search now calls procedure Merge() to process this partial value. Merge() simply compares the search value from the current partial search with the best search value for this node yet backed up, and remembers the best result. It also notes that one less processor is working at this split point by decrementing the processor count. If this count is now exactly one, the split-point is no longer needed; in this case, Merge() sends an UNSPLIT command to the remaining processor (described below.)

After the cleanup is completed via Merge(), the DTS algorithm then returns this processor to its idle loop where it immediately checks for another split-point to join, or else it generates a HELP command to create a split-point.

2.7 The Unsplit() operation.

The next-to-last processor working on a split point sends an UNSPLIT command to the remaining processor, whenever it finds no more work to do at this split point. The split-point has been completely searched except for the branch being analyzed by the remaining processor.

When a processor receives an UNSPLIT command, it calls procedure Unsplit() to cleanup the split-point(S) that have been "almost completed" (recall that the current processor is STILL searching a branch at one or more of these exhausted split-points.)

Unsplit() performs two actions. The first action it takes is to compare the partial score from the split-point with its own partial score for that ply and remember the best one. It then deallocates the SPLIT BLOCK so that this block can be used whenever another Split() operation is required. After an Unsplit() operation, all signs of the previous Split() operation are gone.

2.8 The Share() operation.

Since the efficiency of the alpha/beta algorithm directly depends on knowing the best scores backed up through the tree, whenever backing up a new best score to a split-point ply (by any of the processors working at that split-point), special action is required.

The regular search procedure Backup() was modified to check for this condition in the same manner as Select() described previously, it checks the SPLITL(i) flag to determine if it is "working together" with another processor. If so, it sends a SHARE command to all processors working at this split- point. These processors then call Share() to determine if this newly backed up value is better than the value in this processor's search. If so, some quick tests for alpha/beta cutoffs are made to determine if this newly available value would prune away part of the tree currently being searched by this processor.

This procedure was added to address two problems. (1) When a better value is found at a split-point, efficiency demands that all processors be made aware of it as soon as possible in order to maximize the alpha/beta efficiency. (2) We anticipated the case where the search might create a split-point at a location within the tree where either alpha or beta might be unknown (this is the idea of speculative searching to keep processors busy described earlier.) Share() determines these values for the current processor whenever they become available.

3. Choosing where to split the tree.

The most important decision that the DTS algorithm frequently addresses is where to split the tree into parallel subtrees. If it chooses a good split point, performance is good; if it chooses a poor split point, performance suffers.

3.1 Node types PV, CUT and ALL.

Knuth and Moore clearly defined three node classes of nodes within the alpha/beta minimax tree [Knut75]. While their analysis was centered on a minimal game-tree with perfect move ordering, the concepts they presented also apply to "real" alpha/beta trees, even though it is impossible to produce perfect move ordering [Hyatt89]. In this context we use the terminology developed in [Marsland85], which uses the terms PV, CUT and ALL instead of type one, two and three.

> Type one (PV) nodes. The root position is a type one node. The first successor of a PV node is also a PV node while all other successors of a PV node are CUT nodes. PV nodes require examination of all their successors. A PV node is easy to recognize, because both alpha and beta are at their original values since nothing has yet been searched.
>
> Type two (CUT) nodes. A CUT node is a successor of either a PV node (as given above) or an ALL node. A CUT node only requires examination of one successor (for perfectly ordered game trees.) This is the node type that we must recognize and avoid selecting as a split-point, because with best move ordering, only one branch needs to be searched, which leaves no work for additional processors, other than work that is completely unnecessary.
>
> Type three (ALL) nodes. An ALL node is a successor of a CUT node and requires examination of all its successor branches. Even more interesting, move ordering within a type three node is completely unimportant and has no effect on the total nodes searched. This is an important node type in a parallel search because every move must be searched, offering plenty of work that can be done in parallel.

From these definitions, several things become apparent. (1) Type three (ALL) nodes are perfect candidates for parallel searching since all successors must be searched, and the order of traversal for these successors is unimportant. (2) Type two (CUT) nodes must be avoided as split points since only one successor must be examined. If such a node is chosen as a split point, extra branches will be searched, resulting in wasted work. (3) Type one (PV) nodes appear to be good candidates for parallel search until careful study uncovers the fact that the first successor of a type one node must be completely examined before any of the other successors. This is required since the first branch establishes a search bound for the remainder of the successors, and if they are searched before this bound is known, extra work might be done.

3.2 Classifying node types.

When a processor generates a HELP command, and obtains tree-state data from busy processors, it must (if possible) establish a split point so that it (and other idle processors) can "join in" and help. From section 3.1, it becomes obvious that type ALL nodes make desirable split points, type PV nodes make desirable split points AFTER the first successor of the node has been completely searched, and type CUT nodes must be avoided at all costs.

Cray Blitz contains a function TypeNode(), that types any node from ply one to the current ply. this function is called by function Split() to make an initial "guess" of the node types for each ply in the current processor's search space.

It makes the following assumptions. If, for the current node being tested, the values of alpha and beta are equal to the initial search window, then this node is a PV node. Otherwise, if the current node is at an odd ply and alpha is equal to the lower initial search bound, or the current node is at an even ply and beta is equal to the upper initial search bound, then the node type is CUT. For all other cases, it is type ALL.

Split() uses the above algorithm to set its initial guess for each node type from ply 1 through the current ply. Next it enters an "override" phase. Split() starts at ply=2 and checks the number of moves that have been zeroed by the search (the number of moves that have actually been searched.) For an ALL node, many moves already searched increases the "confidence" that this is truly an ALL node. For CUT nodes, if more than one move has been searched, then the confidence for this CUT node is lowered, since it should not be necessary to search more than one move at a real CUT node. In fact, if more than some limit of moves has been examined (currently=3) then the type for this node is overridden and set to ALL, since it appears that move ordering has somehow failed to search the best move first at some previous ply.

After this override phase, a final simple check is made since it is now possible to have two ALL nodes on successive plies. If this happens, the second ALL node probably means that the successor to this node is really a CUT node and we are resetting the upper/lower search bounds after searching a wrong first move somewhere. The final override phase will note the second ALL node, and then force the successor of this node to be type CUT since yet another ALL node can't follow this one unless move ordering is hopelessly bad. This phase of the override code simply allows only two ALL nodes to be consecutive in the tree. After the second ALL node, the next node must be CUT, the next ALL, etc. For all of these overrides the confidence is very "low" and, again, the number of moves searched at each ply is used to improve this confidence.

This has proven to be a critical step in the DTS algorithm. A mistake here produces severe problems later because the search space is going to increase due to searching branches that the sequential search would be able to avoid. The initial estimate was quite good, but the override phase further improved the reliability of choosing a good node for a Split() operation. This has been modified more than any other part of the DTS code, because Cray Blitz still (on occasion) searches trees in parallel that are much larger than the same tree searched by only one processor. If this were 100% accurate, then Cray Blitz would produce almost linear speedup as additional processors are added to the search.

Others have tackled this "where to split the tree" in different ways, and have produced interesting results [Akl82,Awerbuch85,Baudet78,Feldman90, Feldman93,and others]. Most all were designed for message-passing systems, which made choosing split-points more complex since it becomes a serious issue when communication costs become a major design consideration. It should be noted that the only serious attempts to control tree search overhead all do so at the expense of synchronization/wait penalties. In a distributed environment, this makes a great deal of sense, since the cost of dividing a tree into pieces is very high. On the Cray architecture, the opposite is true.

3.3 Choosing a SPLIT ply

The first goal of the Split() procedure is to select an ALL node for splitting since this is the only reasonable type of node that won't add any search overhead (extra nodes.) Since the overhead for choosing a SPLIT point is costly, a major consideration for selecting a split point is to choose a split point that will take a reasonable amount of time to search in parallel so Split() won't be invoked again. As a general rule, nodes at shallow depths represent more work than nodes at deep plies, making shallow nodes desirable split points. This can be complicated by at least two features of the nodes; (1) the confidence of an ALL node is low, making it risky to search it in parallel and possibly introduce extra nodes into the tree search, and (2) the node has very few branches remaining, so that it will only supply work for a small number of processors which will force yet another Split() when the remaining processors become idle.

As can be seen, this is a somewhat subjective decision, and tuning this code will continue for some time. Other recent modifications to this include questions like "is the king in check at this node?" since this can result in a misleading "work estimate." Cray Blitz generates pseudo-legal moves, and when the king is in check, most of these moves are, in fact, illegal. If these moves are used to estimate the work to be done at this ply, a poor SPLIT point will be chosen since many moves are available, but most will be immediately recognized as illegal with almost no work required to determine this. Several such features can affect the "estimated workload" of a potential split point to make it less attractive. Another example is that the search extensions used by Cray Blitz at ply=N are affected by the extensions at plies before N in the tree. The idea is that if something is causing extensions at shallow plies, the search should be careful and extend at deeper plies too, to find out what is going on. Therefore, if there are many extensions before a potential split-point, the subtrees below that possible split-point will likely be larger than normal, a fact that should be considered when choosing from several candidate nodes for a split-point.

Split() is the single most important function in the DTS algorithm. When it inadvertently chooses a CUT node as a split-point, search overhead increases dramatically. Choosing a PV node potentially increases the search overhead until the first branch is examined and Share() correctly establishes the upper/lower search bounds.

3.4 Splitting at the root

Splitting the search at the root (or not) uncovers some interesting problems. First, most computer chess programs use the so-called iterated search where a one ply search is done, then that information is used to order the moves for a two ply search, and the information from that is used to order a three ply search and so forth. This process is continued until the time allotted for this move runs out.

It should be obvious that the reason for doing a "depth+1" search is to find a better move than the one found from the "depth" search. Often, the program does not change its mind from iteration to iteration, and, in such circumstances, splitting the tree at the root is an excellent idea. After searching the first root branch completely, searching the remainder in parallel introduces almost no overhead (some arises due to transposition table interaction.) However, when the first move searched is not the best, and the program ultimately chooses a different root move as best, searching root branches in parallel causes a problem in timed tournament chess games.

From prior analysis [Knuth75,Hyatt88,Hyatt89,Hsu90] the first branch from the root produces a subtree much larger than the remainder of the branches (when the first branch is best.) If the program then must "change its mind" and select a different move as best, this move will also produce a much larger subtree than the other moves. This "much larger subtree" causes an interesting problem in timed events.

Consider a case where the second branch is really the one that the "depth+1" search will ultimately select. After searching the first branch (using parallel processing) one processor selects the second move in the list (that is really the best move) and starts searching the very large subtree it produces. Other processors examine other root branches that are unimportant. If time runs out before the second branch is completely examined, the first move will be chosen, resulting in the program making a worse move than it really has to. An alternative is to have all processors work on the first move, then all processors work on the second move, and so forth. Then, when a new best move is searched, all processors search it and complete the search before time runs out.

Some might be quick to suggest that "the search should notice that the unfinished move has produced a large tree, and is likely about to become a new best, so don't give up until it has been completed." However, there are two issues with this: (1) just because a root move produces a tree several orders of magnitude larger than any other (than the first) root move does not imply that this move is better. It might be better, or it might simply be a complicated move that leads to positions which stimulate lots of search extensions and make the tree quite large. Therefore, it is not safe to simply say the tree so far is big, wait, because that might take a significant amount of time to finish. And note the major problem here is that only one processor is searching that branch should we choose to wait for a result. (2) "Just waiting" is not a particularly good plan in a timed event. Since time is a factor, using it wisely is a major part of any chess program, and having to burn many minutes just because the search can't resolve whether a move further down the ply=1 move list is better or not can lead to timing difficulties.

The drawback to solving this is that we "know" that all branches at the root should be searched (time permitting) and that searching them will cause no extra overhead (after the first branch establishes a lower search bound, assuming that the first move is actually the best move. This is true for a large majority of positions,). The root is therefore a highly favorable (from a search overhead point of view) place to search in parallel. If the entire ply=1 move list was searched, then splitting at ply=1 would work well. However, since time can run out and stop the search, examining the first few moves on the list (searching each in succession with all processors working together on each move) ensures that the first few moves are COMPLETELY examined before time runs out. (Note: in Cray Blitz and Crafty, an iteration is not completed after time has run out. Rather, the search stops, the move is made, and then the ponder search picks up and continues searching.) We chose to accept this inefficiency (searching extra nodes) in order to let the program change its mind on the last iteration, if the first move is not best.

This often leads to a lively debate about (a) whether or not a parallel search should split at the root and (b) whether or not the program should completely search the root move list before stopping for a time limit. We have experimented with possible alternative algorithms, but none have proven better to date.

4. DTS performance results

The DTS algorithm was tested using a Cray C916/1024 computer. This machine has 16 processors with a cycle time of 4.166 nanoseconds, and also has 1024 million words of memory (eight gigabytes.)

4.1 Testing methodology

In producing these results, all testing used the machine in a dedicated mode so that all of the machine's memory was used, regardless of the number of processors utilized in each test (except for the one-processor test, which is explained later). Often, particularly when using distributed machines like the Hypercubes, adding additional processors also adds additional memory [Feldmann90,Feldmann93,Kuszmaul94,Schaeffer89,Yang93], effectively changing two search parameters at the same time (number of processors or total computation power and total memory available.) It is then difficult to attribute the performance improvement to additional processor power alone as the transposition table is extremely important to search performance and making it larger often dramatically speeds up search times.

The testing done to produce the results given herein differs from the testing used in previous parallel tree search algorithms [Hyatt88, Hyatt89, Schaeffer89, Feldmann90,others]. Rather than use a group of unrelated chess problems, we elected to take a segment of a real chess game and have the program play through it with varying numbers of processors. There are two reasons for choosing this approach: (1) this is what a chess program is designed to do, "play a complete game", not "search random positions" and (2) it is well-known that parallel algorithms perform better when searching deeper trees [Hyatt88,Hyatt89, Schaeffer89,others]. When searching unrelated problems, there is no "continuity" between problems. When searching a series of moves from the same game, the transposition table, the killer move list, the dynamic scoring parameters, all "tie things together" and allow deeper searches. Note also, that if you are iterested in results on a traditional test like the Kopek/Bratko positions, they are available in [Hyatt88].

The testing methodology was to take the 16-processor log produced during the actual game, and then "contrive" things so that the "lesser" configurations would do the same amount of work (roughly). In these tests, if the 16 processor search reached 11 plies and searched the first 10 root moves before the search timed out, then all lessor configurations had to also do exactly this same search, which led to some embarrassingly long search times for one processor as will be seen. One other minor note is that the single processor times appear to search slightly slower than the equivalent parallel searches, which would seem to be counter-intuitive. However, due to the enormous time required to play through this game without stopping (which would have cleared the transposition table and so forth) we ran this test on a production machine, and competed with other processes. As a result, memory conflicts were much higher (bank conflicts to those that are Cray-savvy) as well as swapping overhead which gets charged to the user. While this is well below the .1% level of noise, it is noticeable, and should be remembered.

The test positions came from the game Mchess Pro vs Cray Blitz at the 1993 ACM International Computer Chess Championship. (This game is included as Appendix A.) This game was chosen after looking at analysis produced by Cray Blitz during the tournament. The opening was a King's Gambit Accepted where white sacrificed a piece for some pawns and a strong attack. C-B saw the evaluation steadily drop as it discovered just how strong the attack was, then it leveled off and followed a "tight-rope" for several moves, making the only possible move that would not lose, then it started failing high repeatedly as it finally survived the attack and started a counter-attack that ultimately won the game.

The first position occurs after the sequence of moves 1. e4 e5 2. Nc3 Nc6 3. f4 exf4 4. Nf3 g5 5. d4 g4 6. Bc4 gxf3 7. o-o d5 8. exd5 Bg4 9. Qd2 ... At this point, C-B was "out of book" and quickly discovered that its king was exposed in the center although it was a piece up and a couple of pawns down. The evaluation steadily dropped for the next few moves as C-B "saw" how exposed its king was. The evaluation dropped to a point where C-B was about 1/4 pawn "down" at the low point in the game. It then began a climb to roughly 1/3 of a pawn "ahead" before discovering that Mchess could force a perpetual draw. (This occurred at position 17 where C-B took a long time trying to find any way out of the very deep perpetual check. The perpetual was some 19 plies deep.) For the next two moves, it was resigned to a draw, but Mchess apparently did not search deeply enough to detect the repetition and varied, giving C-B another chance. From this point on the evaluation climbed steadily.

This particular series of positions offered both "good," "bad" and "normal" positions that C-B searched in parallel.

A "good" position is one where the program finds the correct move at a shallow search and "sticks with it" through later searches and ultimately makes that move. This means that ordering at the root of the tree is perfect, and often means that ordering farther down in the tree is also quite good, making the search reasonably efficient. These positions produce good speedups regardless of the number of processors used. Positions 20 and 21 are examples of such positions.

A "normal" position is one where the program occasionally changes its mind at the root, but that move ordering is still good. This lets C-B find the new best move at the root faster than it would if it could split the work at the root (previous discussion.)

A "bad" position is one where every iteration unveils some new threat that the program must then find a way to defend against. At the beginning of each new iteration, the best move from the previous iteration often fails low, and with little help in ordering moves, the parallel version of the search begins to search significantly more nodes than the sequential version. These positions typically have poor speedup results when compared with a sequential search. An excellent example of this is position 17. This is the position where C-B discovers that it can't avoid a draw by repetition (with best play by the opponent.) Move ordering is poor since prior knowledge of good moves is quickly refuted by the drawing lines found by the constantly increasing search depth.

The results include two sets of numbers for each test, where a test used either one, two, four, eight or sixteen processors. The numbers are the clock time required for the search (wall clock time) and the number of nodes searched (which shows how much extra work is added by using additional processors.

These results are summarized below in table 3, which can be compared to the results for the two previous algorithms.

# processors	1	2	4	8	16
speedup	1.0	2.0	3.7	6.6	11.1

Table 3 DTS performance results

5. Performance analysis

The performance results for DTS are presented in table 4 (search time for each test position in the game), table 5 (nodes searched for each test position) and table 6 (the performance improvement for each test position).

The simplest way to analyze the performance is to examine Table 3 which gives the speedup for each position. This table presents the average speedup for two, four, eight and sixteen processors, and is a summary of the timing results from table 6, giving each position a weight of one and dividing the sum of the column by 24. It is interesting to note that there is an inverse relationship between speedup and total nodes searched, which is exactly as expected with DTS. Recall that DTS does not allow processors to remain idle for any significant amount of time, so that poor speedup results has to be the result of searching nodes that are unnecessary, exactly as the raw data given shows.

For the older Cray XMP machines with a maximum of four processors, an average speedup of two (two processors) and 3.7 (four processors) looks extremely efficient. Moving up to an eight processor machine (the old YMP line is an example) produces a speedup of 6.6 which, again, is a reasonable result. Here doubling the number of processors from four to eight produces a performance improvement factor of 1.8. However, pushing this further begins to show just how difficult parallelizing an alpha/beta tree search really is. The speedup of 11.1 for 16 processors means that over one quarter of the C90's cycles were wasted. For these tests, processor idle time averaged under two seconds for each position (per processor), showing that the parallel search examined a significant number of nodes that the sequential search didn't (examine Table 2 and compare the 1cpu node counts column to any of the others. For position 17 it is obvious that the program searched almost three times as many nodes with 16 processors as it did with one, so that, even though the entire machine (16 processors) was busy, the majority of the processors were doing unnecessary work.

Cray Blitz counts the number of split operations that it attempts, as part of its built-in performance analysis. This count is affected by two parameters; (1) the extra nodes searched by the program, and (2) the total search time. In effect, the number of splits per unit of time (inversely) follows the extra nodes searched by the program. For these test positions, the SPLIT count ranged from a low of 307 to a high of over 120,000! As a result, this data has not been presented since the speedup table and time tables accurately predict the split counts obtained from the tests. The best speedup (and the fewest extra nodes) occurs on positions where few split operations are attempted. To avoid a situation of "information overflow" additional statistics gathered by C-B have been omitted since statistical analysis shows strong correlation among all of the metrics. One example is that C-B counts the number of times a processor starts a parallel task only to be stopped by another because move ordering was not optimal. These "early stops" correspond with the "split operations attempted" and with the "extra nodes" quite well.

The most immediate problem in the current algorithm is that the parallel search examines a significant number of extra nodes that the sequential algorithm does not. Much work remains to determine just how far this can be reduced. Intuitively, it is not possible to reduce the number of extra nodes to zero since this would require nearly perfect move ordering. The relevant question is whether or not it is possible to choose a split point more accurately than the present algorithm does. If some information were available that might be used to predict how good move ordering is at a particular node, better split points might be chosen.

For current machines with sixteen processors (and even for future potential supercomputers with up to 64 processors) the processor idle time in the current algorithm does not pose a significant problem. The current idle times of two to three seconds over a move that takes 200 to 500 seconds does not affect the search times significantly for small numbers of processors (2-16), but for really large numbers of processors, something must be done to avoid running into Amdahl's Law.

This result compares favorably with the DPVS (Dynamic PVS) algorithm developed for Phoenix (Schaeffer89]. The overall algorithms seem quite similar, but again, DTS is designed for a much different communication mechanism (shared memory) and can overlook some things that Phoenix [Schaeffer89], Zugzwang [Feldman93], *Socrates [Kuszmaul], and others have to be very concerned about because the communication costs are so significant in those message passing architectures used. In fact, perhaps the Deep Thought/Deep Blue hardware comes closest to the architecture used by Cray Blitz, but even this chess-specific architecture still does not have a uniform shared memory system that is global across all processors and chess processors [Hsu90].

It's unfortunate that shared memory is an architectural design feature that doesn't scale to large numbers of processors, because CB might produce some interesting performance results with really large numbers of processors. However, this is not the case, and it is doubtful that a Cray-class machine will support shared memory on more than 64 processors, using a high-speed interconnect like the Cray machines use. As a result, this algorithm really will only scale as far as a shared memory architecture scales. Any delays (such as those in some architectures that use a hierarchical memory organization with varying delays depending on how far the memory is from the requesting processor) will certainly have an adverse effect on DTS.

	processors				
pos	1	2	4	8	16
1	2,830	1,415	832	435	311
2	2,849	1,424	791	438	274
3	3,274	1,637	884	467	239
4	2,308	1,154	591	349	208
5	1,584	792	440	243	178
6	4,294	2,147	1,160	670	452
7	1,888	993	524	273	187
8	7,275	3,637	1,966	1,039	680
9	3,940	1,970	1,094	635	398
10	2,431	1,215	639	333	187
11	3,062	1,531	827	425	247
12	2,518	1,325	662	364	219
13	2,131	1,121	560	313	192
14	1,871	935	534	296	191
15	2,648	1,324	715	378	243
16	2,347	1,235	601	321	182
17	4,884	2,872	1,878	1,085	814
18	646	358	222	124	84
19	2,983	1,491	785	426	226
20	7,473	3,736	1,916	1,083	530
21	3,626	1,813	906	489	237
22	2,560	1,347	691	412	264
23	2,039	1,019	536	323	206
24	2,563	1,281	657	337	178

Table 4 Search Time in Seconds

	processors				
pos	1	2	4	8	16
1	87,735,974	89,052,012	105,025,123	109,467,495	155,514,410
2	88,954,757	90,289,077	100,568,301	110,988,161	137,965,406
3	101,302,792	102,822,332	111,433,074	117,366,515	119,271,093
4	71,726,853	72,802,754	74,853,409	88,137,085	104,230,094
5	49,386,616	50,127,414	55,834,316	61,619,298	89,506,306
6	133,238,718	135,237,296	146,562,594	168,838,428	226,225,307
7	58,593,747	62,602,792	66,243,490	68,868,878	93,575,946
8	225,906,282	229,294,872	248,496,917	261,728,552	340,548,431
9	122,264,617	124,098,584	138,226,951	159,930,005	199,204,874
10	75,301,353	76,430,872	80,651,716	83,656,702	93,431,597
11	95,321,494	96,751,315	104,853,646	107,369,070	123,994,812
12	79,975,416	85,447,418	85,657,884	94,000,085	112,174,209
13	66,100,160	70,622,802	70,796,754	78,834,155	96,053,649
14	58,099,574	58,971,066	67,561,507	74,791,668	95,627,150
15	84,143,340	85,405,488	92,557,676	97,486,065	124,516,703
16	75,738,094	80,920,173	79,039,499	84,141,904	94,701,972
17	154,901,225	184,970,278	242,480,013	279,166,418	416,426,105
18	20,266,629	22,856,254	28,443,165	31,608,146	42,454,639
19	93,858,903	95,266,785	100,527,830	108,742,238	114,692,731
20	231,206,390	234,674,482	241,284,621	271,751,263	264,493,531
21	112,457,464	114,144,324	114,425,474	123,247,294	118,558,091
22	81,302,340	86,865,131	89,432,576	106,348,704	135,196,568
23	63,598,940	64,552,923	68,117,815	81,871,010	103,621,303
24	80,413,971	81,620,179	83,919,196	85,810,169	90,074,814

Table 5 Total Nodes Searched

	processors				
pos	1	2	4	8	16
1	1	2.0	3.4	6.5	9.1
2	1	2.0	3.6	6.5	10.4
3	1	2.0	3.7	7.0	13.7
4	1	2.0	3.9	6.6	11.1
5	1	2.0	3.6	6.5	8.9
6	1	2.0	3.7	6.4	9.5
7	1	1.9	3.6	6.9	10.1
8	1	2.0	3.7	7.0	10.7
9	1	2.0	3.6	6.2	9.9
10	1	2.0	3.8	7.3	13.0
11	1	2.0	3.7	7.2	12.4
12	1	1.9	3.8	6.9	11.5
13	1	1.9	3.8	6.8	11.1
14	1	2.0	3.5	6.3	9.8
15	1	2.0	3.7	7.0	10.9
16	1	1.9	3.9	7.3	12.9
17	1	1.7	2.6	4.5	6.0
18	1	1.8	2.9	5.2	7.7
19	1	2.0	3.8	7.0	13.2
20	1	2.0	3.9	6.9	14.1
21	1	2.0	4.0	7.4	15.3
22	1	1.9	3.7	6.2	9.7
23	1	2.0	3.8	6.3	9.9
24	1	2.0	3.9	7.6	14.4
avg	1	2.0	3.7	6.6	11.1

Table 6 Parallel Processing Speedup

6. Future work

Perhaps the most significant future work for this algorithm lies in selecting split points more accurately. For some positions, the current Split() algorithm performs admirably. For others, it performs poorly. Anything that improves the "poor" cases has a positive effect on overall performance.

There are at least two approaches to this problem. One is to more accurately find good split points (something that might be impossible in a significant number of positions). Another is to develop some sort of confidence in a list of split points, somehow eliminating those that are potentially bad. After eliminating such split points, the overhead should drop significantly. Unfortunately, detecting such split points is not easy. The first step will likely be more detailed analysis of the trees searched by C-B. The problem with this is that these trees regularly exceed one billion nodes on a T90, making the volume of data somewhat large! Such large trees require recognizing a specific performance feature and then adding instructions to the code to measure the feature "in situ" since analyzing such a large tree by visual inspection is impractical.

The current algorithm depends on rapid information sharing and uses shared memory to implement this sharing. The next logical step in algorithm development is to move toward a distributed architecture. With HIPPI and/or other high-performance processor interconnects, a distributed version of this algorithm should deliver high performance if care is taken to control task granularity (which is not an issue on machines like the C90/T90 with shared memory.)

7. Bibliography

S. Akl, D. Barnard, and R. Doran, "The Design, Analysis and Implementation of a Parallel Alpha-Beta Algorithm", IEEE Transactions on Pattern Analysis and Machine Intelligence, PAMI-4, (2) (1982), (192-203).

B. Awerbuch, "A New Distributed Depth-First Search Algorithm", Information Processing Letters (20) (1985) 147-150.

G. Baudet, "The Design and Analysis of Algorithms for Asynchronous Multiprocessors", Ph. D Dissertation, Carnegie-Mellon University, Pittsburgh, Pa. (1978).

M. Campbell, "Algorithms for the Parallel Search of Game Trees", M. Sc. Thesis, Technical Report TR 81-8, Computer Science Department, University of Alberta, Edmonton (1981).

R Feldmann, B. Monieni, P. Mysliwietz and O. Vornberger, "Distributed game tree search," in Parallel algorithms for machine intelligence and pattern recognition (1990).

R Feldmann, "Game Tree Search on Massively Parallel Systems," Ph.D. Theses, University of Paderborn, August, 1993.

R. Finkel and J. Fishburn, "Parallelism in Alpha-Beta Search", Artificial Intelligence (1982) 89-106.

F-h Hsu, "Large scale parallelism of alpha-beta search: an algorithmic and architectural study," Ph.D. Thesis, Carnegie-Mellon University (1990).

R. Hyatt, B. Suter, and H. Nelson, "A Parallel Alpha/Beta Tree Searching Algorithm," Parallel Computing 10 (1989) 299-308.

R. Hyatt, "A High-Performance Parallel Algorithm to Search Depth-First Game Trees," Ph.D. Dissertation, University of Alabama at Birmingham, 1988.

R. Hyatt, A. Gower, and H. Nelson, "Cray Blitz", Advances in Computer Chess 4, Pergammon Press (1986) (8-18).

R. Hyatt, H. Nelson, A. Gower, "Cray Blitz - 1984 Chess Champion", Telematics and Informatics (2) (4), Pergammon Press Ltd. (1986) (299-305).

D. Knuth and R. Moore, "An Analysis of Alpha-Beta Pruning", Artificial Intelligence 6 (1975) (293-326).

B. Kuszmaul, "Synchronized MIMD Computing," Ph.D. Thesis, MIT, 1994.

G. Lindstrom, "The Key Node Method: A Highly Parallel Alpha-Beta Algorithm", Technical Report UUCS 83-101, Department of Computer Science, University of Utah, 1983.

T. Marsland and J. Schaeffer, Computers, Chess and Cognition, Springer-Verlag, 1990.

T. Marsland and M. Campbell, "Parallel Search of Strongly Ordered Game Trees", ACM Computing Surveys (4) (1982) (533-551).

T. A. Marsland and F. Popowich, "Parallel Game-tree Search", IEEE Transactions on Pattern Analysis and Machine Intelligence, PAMI-7, (1985) (442-452).

T. Marsland, M. Campbell, and A. Rivera, "Parallel Search of Game Trees", Technical Report TR 80-7, Computing Science Department, University of Alberta, Edmonton (1980).

T. Marsland and M. Campbell, "Methods for Parallel Search of Game Trees", Proceedings of the 1981 International Joint Conference on Artificial Intelligence.

T. Marsland, M. Olafsson, and J. Schaeffer, "Multiprocessor Tree-Search Experiments", Advances in Computer Chess 4, Pergammon Press (1986) (37-51).

M. Newborn, "A Parallel Search Chess Program", Proceedings, ACM Annual Conference, (1985), (272-277).

J. Pearl, "Scout: A Simple Game-Searching Algorithm with Proven Optimal Properties", Proceedings of the First Annual National Conference on Artificial Intelligence, Stanford, (1980).

F. Popowich and T. Marsland, "Parabelle: Experiments with a Parallel Chess Program", Technical Report TR 83-7, Computing Science Department, University of Alberta, Edmonton (1983).

J. Schaeffer, "Distributed game-tree search," Journal of Parallel and Distributed Computing 6 (2) (1989), 90-114.

I. Steinberg and M. Solomon, "Searching game trees in parallel," Proceedings of the International Conference on Parallel Processing (Vol 3) (1990), 9-17.

Z. Yang and T. Marsland, "Global States and Time in Distributed Systems," IEEE Computer Society Press, November 1993.

Appendix A

[Event "23rd ACM International Computer Chess Championship"]
[Site "Indianapolis, MD"]
[Date "02.14.1993"]
[Round "?"]
[White "Mchess Pro"]
[Black "Cray Blitz"]
[Result "0-1"]
1. e4 e5 2. Nc3 Nc6 3. f4 exf4 4. Nf3 g5 5. d4 g4 6. Bc4 gxf3 7. o-o d5 8. exd5 Bg4 9. Qd2 Na5 10. Bb5+ c6 11. Qxf4 Nf6 12. Re1+ Kd7 13. dxc6+ bxc6 14. Ne4 Nxe4 15. Qxg4+ Kc7 16. Rxe4 cxb5 17. Qxf3 Qd7 18. Rf4 Be7 19. Rxf7 Raf8 20. Bf4+ Kb6 21. Be5 Rhg8 22. c3 Nc6 23. Re1 Qe8 24. Rxf8 Qxf8 25. Qd5 Rg5 26. a4 bxa4 27. c4 Nxe5 28. Rxe5 Rxe5 29. Qxe5 Qd8 30. c5+ Kb5 31. Qe2+ Kb4 32. Qd3 Bf6 33. Qc3+ Kb5 34. Qd3+ Kc6 35. Qf3+ Kd7 36. Qe4 Qe7 37. Qb7+ Ke8 38. Qa8+ Qd8 39. Qc6+ Kf7 40. Qb7+ Kf8 41. d5 Qe7 42. Qb8+ Kf7 43. Kf1 Bxb2 44. g4 Bc3 45. Kf2 Qxc5+ 46. Kf1 Qc4+ 47. Kf2 Qd4+ 48. Kf3 Qd1+ 49. Ke3 Bd2+ 50. Kf2 Qe1+ 51. Kg2 Qe2+