Distributed Game-Tree Search Using Transposition Table Driven Work Scheduling

Akihiro Kishimoto and Jonathan Schaeffer
Department of Computing Science, University of Alberta,
Edmonton, Alberta, Canada T6G 2E8
{kishi@cs.ualberta.ca, jonathan@cs.ualberta.ca}

Abstract

The $\alpha\beta$ algorithm for two-player game-tree search has a notorious reputation as being a challenging algorithm for achieving reasonable parallel performance. MTD(f), a new $\alpha\beta$ variant, has become the sequential algorithm of choice for practitioners. Unfortunately, MTD(f) inherits most of the parallel obstacles of $\alpha\beta$, as well as creating new performance hurdles. Transposition-table-driven scheduling (TDS) is a new parallel search algorithm that has proven to be effective in the single-agent (one-player) domain. This paper presents TDSAB, the first time TDS parallelism has been applied to two-player search (the MTD(f) algorithm). Results show that TDSAB gives comparable speedups to that achieved by conventional parallel $\alpha\beta$ algorithms. However, since this is a parallelization of a superior sequential algorithm, the results in fact are better. This paper shows that the TDS idea can be extended to more challenging search domains.

Keywords: *$\alpha\beta$ search, transposition-table-driven scheduling, single-agent search, transposition table.*

1. Introduction

The development of high-performance game-playing programs has been the subject of over 50 years of artificial intelligence research. At the heart of these programs is the $\alpha\beta$ tree search algorithm. The strong correlation between the depth of search and the resulting program's performance prompted researchers to quickly move to parallel solutions, including multi-processor systems (e.g., WAYCOOL using a 256-processor HyperCube [6], CRAY BLITZ using a 16-processor Cray [8], SUN PHOENIX using a network of 32 workstations [19], ZUGZWANG using 1,024 Transputers [4]) and special-purpose hardware (including DEEP BLUE [7]). Of course, the most vivid demonstration of this technology was the 1996 and 1997 matches between DEEP BLUE and World Chess Champion Garry Kasparov.

Many artificial intelligence applications are search based and require real-time responses. Clearly faster hardware enables more computations to be performed in a fixed amount of time, generally allowing for a better quality answer. Dual-processor machines and clusters of inexpensive processors are ubiquitous and are the *de facto* research computing platforms used today.

Single-agent domains (puzzles) and two-player games have been popular test-beds for experimenting with new ideas in sequential and parallel search. This work transfers naturally to many real-world problem domains, for example planning, path-finding, theorem proving, and DNA sequence alignment. There are many similarities in the approaches used to solve single-agent domains (A* and IDA*) and two-player games ($\alpha\beta$). Many of the sequential enhancements developed in one domain can be applied (with modifications) to the other.

Two recent developments have changed the way that researchers look at two-player search. First, MTD(f) has emerged as the new standard framework for the $\alpha\beta$ algorithm preferred by practitioners [15]. Second, TDS is a new, powerful parallel search paradigm for distributed-memory hardware that has been applied to single-agent search [17, 18]. Given that there is a new standard for sequential $\alpha\beta$ search (MTD(f)) and a new standard for parallel single-agent search (TDS), the obvious question is what happens when both ideas are combined.

In MTD(f), all searches are done with a so-called minimal window $[\alpha, \alpha+1]$. Each search answers a binary question: is the result $\leq \alpha$ or is it $> \alpha$? At the root of the tree, a series of minimal window searches are performed until the result converges on the value of the search tree. MTD(f) has been shown to empirically out-perform other $\alpha\beta$ variants. It has the nice property of searching using a single bound, an important consideration in a parallel search.

TDS is an elegant idea that reverses the traditional view of parallel search. Instead of sending data to the work that needs it, TDS sends the work to the data. This simple reversal of the relationship between computation and data simplifies the parallelism, reduces parallel overhead, and yields impressive results for single-agent search applications.

This paper introduces TDSAB, TDS parallelism adapted to $\alpha\beta$ search (specifically, MTD(f)) [9]. This is the first attempt to integrate TDS parallelism into two-player search. The speedups in two application domains (the game of Awari, small branching factor; Amazons, large branching factor) average roughly 23 on a network of 64 workstations, a result that is comparable to what others have achieved using conventional parallel $\alpha\beta$ algorithms. However $\alpha\beta$ is not the best sequential algorithm for searching game trees; MTD(f) has been shown to be 5-15% better [15]. Parallel performance must always be compared to that of the best sequential algorithm. Given that the game-development community is moving to MTD(f) as their sequential standard, the results in this paper confirm that this algorithm is also suitable for high-performance parallel applications.

Section 2 discusses sequential and parallel game-tree search algorithms. Section 3 introduces TDSAB, while Section 4 presents experimental data on its performance. Section 5 discusses future work on enhancing this algorithm.

2. Game-tree Search

This section gives a quick survey of $\alpha\beta$ searching. Good surveys of the literature are available for sequential [12] and parallel [1] game-tree search.

2.1. Sequential Search

For more than 30 years the $\alpha\beta$ algorithm has been the most popular algorithm for two-player games. The algorithm eliminates provable irrelevant nodes from the search. Two bounds are maintained, α and β, representing the lower and upper bounds respectively on the minimax value of the search tree (the search window). The savings of $\alpha\beta$ come from the observation that once the search proves that the score of the node is outside the search window, then further effort at that node is irrelevant.

Many enhancements have been added to $\alpha\beta$ to (dramatically) improve the search efficiency. The most important of these is the transposition table, a cache of the results from previously searched sub-trees [20]. Its effectiveness is application dependent. For chess, for example, it is worth a factor of 10 in performance. Thus any high-performance implementation must have a transposition table.

MTD(f) is recognized as the most efficient variant of sequential $\alpha\beta$. Figure 1 shows that MTD(f) is just a sequence of minimal window $\alpha\beta$ calls, searching node n to depth d. The initial search is centered around the value f (usually the value returned by the previous iteration in an iterative-deepening search). This result is then monotonically increased or decreased to the correct minimax value. The transposition table is critical to the performance of MTD(f), since the tree is repeatedly traversed, albeit with a different search window. The table prevents nodes that have been proven to be inferior from being searched repeatedly.

```
int MTD(node_t n, int d, int f) {
  int score;
  int lowerbound = −∞; upperbound = ∞;
  if (f == -∞) bound = f + 1;
  else bound = f;
  do {
    /* Minimal window search */
(+) score = AlphaBeta(n, d, bound-1, bound);
    if (score < bound) upperbound = score;
    else lowerbound = score;
    /* Re-set the bound */
    if (lowerbound == score) bound = score + 1;
    else bound = score;
  } while (lowerbound ≠ upperbound);
}
```

Figure 1. MTD(f)

2.2. Obstacles to Parallel Search

$\alpha\beta$ has proven to be a notoriously difficult algorithm to get good parallel performance with. To achieve a sufficient speedup, we must cope with the following obstacles:

- *Search overhead* is the (usually) larger tree built by the parallel algorithm as compared to the sequential algorithm. This cost is estimated by:

  $$Search\ overhead = \frac{parallel\ tree\ size - sequential\ tree\ size}{sequential\ tree\ size}.$$

  Note that this overhead could be negative (super-linear speedups are occasionally observed).
- *Synchronization overhead* occurs when processors have to sit idle waiting for the results from other searches. A good approximation for this is to measure the average percent of idle time per processor. This overhead can be reduced by increasing the amount of concurrency (for example, speculative search), usually at the cost of increasing the search overhead.
- *Communication overhead* is the consequence of sharing information between processors. In a shared-memory environment, this is effectively zero (although there may be some locking overhead). In a distributed-memory environment, communication is done via message passing. Since all messages are small (less than 100 bytes), the overhead can be estimated by counting messages and multiplying this by the average cost of a message send. Note that fast communication is critical to performance; a late message can result in unnecessary work being spawned in parallel. Hence, a faster network can improve the efficiency of the search.
- *Load balancing* reflects how evenly the work has been distributed between the processors. Ideally, each processor should be given work that takes exactly the same amount of time. In practice, this does not happen, and some processors are forced to wait for others to complete their tasks. Note that load balancing is an important influence on synchronization overhead,

None of these overheads is independent of each other. A high-performance $\alpha\beta$ searcher needs to be finely tuned to choose the right mix of parameters and algorithm enhancements to balance the performance trade-offs.

2.3. Parallel Search Algorithms

Numerous parallel $\alpha\beta$ algorithms have been proposed. The PV-Split algorithm invoked parallelism down the left-most branch of the tree (the *principal variation*) [13]. In the late 1980s, several algorithms appeared that initiated parallelism throughout the tree (Enhanced PV-Split [8], Dynamic PV-Split [19], Young Brothers Wait Concept [5]). Further concurrency could be achieved using speculative search (UIDPABS [14], APHID [3]).

The Young Brothers Wait Concept (YBWC) serves as a suitable framework for describing most of the commonly used parallel $\alpha\beta$ implementations [4]. There are many variants of YBWC, but they only differ in the implementation details (e.g., [10, 21]). Highly optimized sequential $\alpha\beta$ search algorithms usually do a good job of ordering moves from best to worst. A well-ordered $\alpha\beta$ search tree has the property that if a cut-off is to occur at a node, then the first move considered has a high probability of achieving it. YBWC states that the left-most branch at a node has to be searched before the other branches at the node are searched. This observation reduces the search overhead (by ensuring that the move with the highest probability of causing a cut-off is investigated before initiating the parallelism) at the price of increasing the synchronization overhead (waiting for the first branch to return).

To minimize synchronization overhead and improve load balancing, many algorithms (including YBWC) use *work stealing* to offload work from a busy processor to an otherwise idle processor. When a processor is starved for work, it randomly chooses a processor and then "steals" a piece of work from its work queue. Although work stealing improves the load balancing, on distributed-memory machines this can have an adverse impact because of the lack of information available in the transposition table.

2.4. Parallel Transposition Tables

Practical $\alpha\beta$ search algorithms have transposition tables. In a shared-memory environment, transposition tables can be easily shared, making results from one processor available to others as soon as it has been computed. In a distributed memory environment, things are not so simple; the efficient implementation of transposition tables becomes a serious problem. Since processors do not share memory, they cannot access the table entries of other processors without incurring communication overhead.

There are three naive ways to implement transposition tables on distributed-memory machines. With *local transposition tables*, each processor has its own table. No table entries are shared among processors. Therefore, looking up and updating an entry can be done without communication. However, local transposition tables usually result in a large search overhead, because a processor may end up repeating a piece of work done by another processor.

With *partitioned transposition tables*, each processor keeps a disjoint subset of the table entries. This can be seen as a large transposition table divided among all the processors (e.g., [4]). Let L be the total number of table entries with p processors, then each processor usually has $\frac{L}{p}$ entries. When a processor P needs a table entry, it sends a message to ask the processor Q which keeps the corresponding entry to return the information to P (communication overhead). P has to wait for Q to send back the information on the table entry to P (synchronization overhead). When processor P updates a table entry, it sends a message to the processor that "owns" that entry to update its table entry. Updating messages can be sent asynchronously.

Using *replicated transposition tables* results in each processor having a copy of the same transposition table. Looking up a table entry can be done by a local access, but updating an entry requires a broadcast to *all* the other processors to update their tables with the new information (excessive communication overhead). Even if messages for updates can be sent asynchronously and multiple messages can be sent at a time by combining them as a single message, the communication overhead increases as the number of processors increases. As well, replicated tables have fewer aggregate entries than a partitioned table.

All three approaches may do redundant search in the case of a DAG (Directed Acyclic Graph). A search result is stored in the transposition table *after* having finished searching. If identical nodes are allocated to different processors, then duplicate search may occur, which increases the search overhead. Because the efficient implementation of transposition tables in a distributed environment is a challenging problem, researchers have been looking for better solutions [2, 19].

2.5. TDS

Transposition-table Driven Scheduling (TDS) flips the idea of work-stealing to solve the transposition table problem [17, 18]. Work-stealing moves the data to where the work is located; TDS moves the work to the data. A similar idea also appears in [11].

In TDS, transposition tables are partitioned over the processors like a partitioned transposition table. Whenever a node is expanded, its children are scattered to the processors (called *home processors*) which keep their transposition table entries. Once the work is sent to a processor, that processor accesses the appropriate transposition table information locally. All communication is asynchronous. Once a processor sends a piece of work, it can immediately work on another task. Processors periodically check to see if new work has arrived.

The idea of TDS seems to be easily applied to two-player games. However, there are important differences between single-agent search (IDA*) and two-player search ($\alpha\beta$) that complicate the issue:

(1) **Pruning:** $\alpha\beta$'s scheme for pruning is different from that of IDA*. IDA* uses a single bound; $\alpha\beta$ uses a pair of bounds (the search window).
(2) **Cut-offs:** IDA* only aborts parallel activity when the final search result has been determined. $\alpha\beta$ cut-offs occur throughout the search. When a cut-off occurs at a node, there has to be a way to abort all work spawned from that node.
(3) **Search window:** $\alpha\beta$ can search identical nodes with different search windows.
(4) **Priority of nodes:** The order in which nodes are considered is more important in $\alpha\beta$ than in IDA*. In IDA*, move ordering only affects the efficiency of the last iteration, while in $\alpha\beta$ it impacts every node in the tree. For IDA*, a stack is sufficient to prioritize nodes; for $\alpha\beta$, this is insufficient.
(5) **Saving the tree:** An IDA* tree node does not need to know its parent. In $\alpha\beta$, values must be passed back from child nodes to parent nodes. This requires an $\alpha\beta$ implementation that guarantees the search tree is saved in the transposition table.

However, TDS has several important advantages that can facilitate the parallel performance of $\alpha\beta$:

(1) All transposition table accesses are done locally. All communication is asynchronous.
(2) DAGs are not a problem, since identical positions are guaranteed to be assigned to the same processor.
(3) Given that positions are mapped to random numbers to be used for transposition table processor assignments, statistically good load balancing happens for free.

Since TDS has proven to be so successful in single-agent search, the obvious question to ask is how it would fare in two-player search.

3. TDSAB

This section presents a new parallel $\alpha\beta$ algorithm, TDSAB (Transposition-table Driven Scheduling Alpha-Beta), the first attempt to parallelize MTD(f) using the TDS algorithm. Transposition tables are critical to the performance of MTD(f), and a TDS-like search addresses the problem of an efficient implementation of this data structure in a distributed-memory environment.

3.1. The TDSAB Algorithm

MTD(f) has an important advantage over classical $\alpha\beta$ for parallel search; since all searches use a minimal window, the problem of disjoint and overlapping search windows will not occur (a serious problem with conventional parallel $\alpha\beta$ implementations). The disadvantage is that for each iteration of MTD(f), there may be multiple calls to $\alpha\beta$, each of which incurs a synchronization point (at line "+" in Figure 1). Each call to $\alpha\beta$ has the parallelism restricted to adhere to the YBWC restriction to reduce the search overhead. The distribution of nodes to processors is done as in TDS.

Since TDSAB follows the TDS philosophy of moving work to the data, the issues explained in the last section have to be resolved. The following new techniques are used for TDSAB:

(1) **Search order:** The parallel search must preserve the good move ordering (best to worst) that is seen in sequential $\alpha\beta$. Our solution to this issue is similar to that used in APHID [2]. Each node is given a priority based on how "left-sided" it is. To compute the priority of a node, the path from the root to that node is considered. Each move along that path contributes a score based on whether the move is the left-most in the search tree, left-most in that sub-tree, or none of the above. These scores are added together to give a priority, and nodes are sorted to determine the order in which to consider work (see [9] for details).
(2) **Signatures:** When searching the children of a node in parallel and a cut-off score is returned, further work at this node is not necessary; all outstanding work must be stopped. However, since TDS does not have all the descendants of a node on the same processor, we have to consider an efficient way of tracking down this work (and any work that has been spawned by it) and terminating it. Cut-offs can be elegantly handled by the idea of giving each node a *signature*. When a cut-off happens at a node P, TDSAB broadcasts the signature of P to all the processors. A processor receiving a cut-off signature examines its local priority queue and deletes all the nodes whose signature *prefix* is the same as the signature of P (see Section 3.2).
(3) **Synchronization of nodes:** The search order of identical nodes must be considered carefully in the case of cyclic graphs, in order to avoid deadlock. We have developed a strategy for synchronizing identical nodes that is deadlock-free (see Section 3.3).

Figure 2 gives pseudo code for TDSAB. For simplicity, we just explain TDSAB without YBWC and also do not use the negamax form. The function $ParallelMWS$ does one iteration of a minimal window search $[\alpha, \alpha+1]$ in parallel. The end of the search is checked by the function $FinishedSearchingRoot$, which can be implemented by broadcasting a message when the score for the root has been decided. The function $RecvNode$ checks regularly if new information comes to a processor. $RecvNode$ receives three kinds of information:

1. NEW_WORK: a processor has examined a node and spawned the node's children to be evaluated in parallel.

The new work arrives, assigned a priority, and inserted in the priority queue (lines marked "="). Eventually this work will reach the head of the priority queue (if it is not cut-off). If the work to be searched is terminal or a small piece of work then it is immediately searched locally (the cost of doing it in parallel out-weighs the benefits) and sent back to its parent node (lines marked "-" in the figure). Otherwise, the children of the node are generated and sent to their $HomeProcessors$ to be searched (lines marked "!").

2. CUT-OFF: a signature is received and used to remove work from the priority queue. If a processor receives a signature, the function $CutAllDescendants$ examines its local queue and discards all nodes with a matching signature prefix (see the pseudo-code at "*").
3. SEARCH_RESULT: the minimax score of a node is being returned to its parent node (lines marked "+").

If new information arrives at a processor, $GetNode$, $GetSignature$, and $GetSearchResult$ get information on a node, signature, and score for a node respectively. $GetLocalJob$ determines a node to be expanded from its local priority queue, and $DeleteLocalJob$ deletes a node from the queue. We note that TDSAB keeps information on nodes being searched unlike the IDA* version of TDS. $SendNode$ sends a node to the processor chosen by the function $HomeProcessor$, which returns the id of the processor having the table entry for the node (a function of the transposition table key *TTKey*).

When receiving a search result, TDSAB has to consider two cases ($StoreSearchResult$). If a score proves a fail high (result $> \alpha$), TDSAB does not need to search the rest of the branches. The fail-high score is saved in the transposition table ($TransFailHighStore$), the node is dequeued from the priority queue, and the score is sent to the processor having the parent of the node ($SendScore$). Only after a processor has completed searching a node is it discarded. Because searching the rest of the branches has already started, the processor broadcasts a signature to abort useless search, then deletes the node. When a fail low happens (result $\leq \alpha$), a processor stores the maximum score of the branches. If all the branches of a node are searched, the fail-low score for the node is stored in the transposition table ($TransFailLowStore$), and the score is reported back to its parent.

3.2. Signatures

Let P be a node and Q be a child of P. When searching the children of P in parallel, if Q returns a score that causes a cut-off at P, searching other children of P is not necessary. If any child of P is currently being searched, then it must be stopped. TDSAB, therefore, has to stop any useless searches in order to avoid increasing the search overhead. However, because all the descendants of P are not always on the same processor in the TDS framework, we

```
int α; /* A search window is set to [α, α + 1]. */
/* Granularity depends on machines, networks, and so on. */
const int granularity;

void ParallelMWS() {
   int type, v;
   node_t p;
   signature_t sig;
   do {
      /* Check if new information arrives. */
      if (RecvNode(&type) == TRUE) {
         switch(type) {
(=)         /* New work is stored in its priority queue. */
(=)         case NEW_WORK:
(=)            GetNode(&p); Enqueue(p); break;
(*)         /* Obsolete nodes are deleted from its priority queue. */
(*)         case CUT_OFF:
(*)            GetSignature(&sig); CutAllDescendents(sig); break;
(+)         /* A search result is saved in the transposition table. */
(+)         case SEARCH_RESULT:
(+)            GetSearchResult(&p,&v); StoreSearchResult(p,v); break;
         }
      }
      GetLocalJob(&p);
      if (p == FOUND) {
(-)      if (p == terminal || p.depth ≤ granularity) {
(-)         /* Local search is done for small work. */
(-)         v = AlphaBeta(p,p.depth,α,α + 1);
(-)         SendScore(p.parent,v);
            DeleteLocalJob(p);
         } else {  /* Do one-ply search in parallel. */
(!)         for (int i = 0; i < p.num_of_children; i++) {
(!)            int pe = HomeProcessor(TTKey(p.child_node[i]));
(!)            p.child_node[i].depth = p.depth - 1;
(!)            SendNode(p.child_node[i],pe);
            }
         }
      }
   } while (!FinishedSearchingRoot());
}

void StoreSearchResult(node_t p, int value) {
   if (value > α) { /* Fail high */
      TransFailHighStore(p,value);
      SendScore(p.parent,value);
      SendPruningMessage(p.signature); DeleteLocalJob(p);
   } else { /* Fail low */
      p.score = MAX(p.score,value); p.num_received ++;
      if (p.num_received == p.num_of_children) {
         /* All the scores for its children are received. */
         TransFailLowStore(p,p.score);
         SendScore(p.parent,p.score);
         DeleteLocalJob(p);
      }
   }
}
```

Figure 2. Simplified Pseudo Code for TDSAB

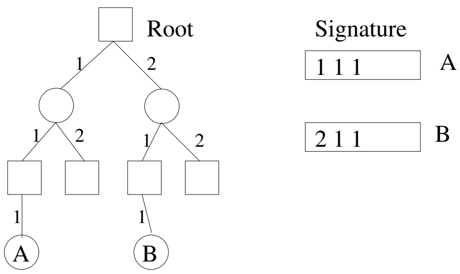

Figure 3. Signatures

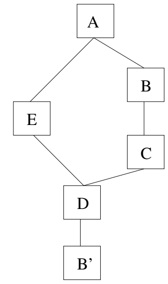

Figure 4. Deadlock with Cycles

have to consider an efficient implementation for cut-offs in TDSAB. In a naive implementation, when a cut-off happens at a node, a message has to be sent to all processors searching that node's children asking them to remove the child node from their priority queue (and, in turn, messages to their children to stop searching, and so on). This approach clearly results in the exchange of many messages which can lead to a large increase in communication overhead, and also a delay in killing unnecessary work (which, in turn, results in more search overhead).

In TDSAB, when a cut-off occurs, we reduce the number of messages exchanged by using a *signature*. Intuitively, the signature for P is the path traversed from the root node to P. Every branch of a node has a tag which differentiates it from other branches at that node; a signature of P is seen as a sequence of these tags from the root to P. Figure 3 illustrates an example of signatures. The decimal number on each branch between two nodes is the tag. The signature of A is 111 derived from the path from the root to A; the signature of B is 211.

When a cut-off happens at a node P, TDSAB broadcasts the signature of P to all the processors. When a processor receives a cut-off signature, it examines its local priority queue and deletes all the nodes which have the same paths from the root to P. For example, in Figure 3, if TDSAB wants to prune all the children of A, the signature 111 is broadcast and each processor prunes all the nodes that begin with the signature "$111\cdots$".

3.3. Deadlock

The search order of identical nodes has to be carefully handled in order to avoid deadlock. Assume that a processor has two identical nodes. If searching the second node is *always* delayed until after the completion of the first node, then a deadlock may occur. Figure 4 illustrates this problem. Suppose that B and B' are identical nodes. If these nodes are searched in the following order, a deadlock will occur: (1) A is expanded, and B and E are sent to their home processors. (2) B is expanded, and C is sent. If B's processor receives a node B' identical to B, searching B' is delayed until it receives a score for B. (3) E is expanded, and D is sent. (4) D is expanded, and B' is sent. Searching B' is done after finishing B. (5) C is expanded, and D is sent. In this case, B waits for the score for C, C waits for D, D waits for B', and B' waits for B. Therefore a cyclic wait has been created and a deadlock ensures.

To eliminate the possibility of deadlock, if two identical nodes are encountered and neither of the nodes has been searched yet, then TDSAB searches the shallower one first. When a node n_1 whose search depth is shallower or equivalent to an identical node n_2 whose search has already begun, then n_2 waits until n_1's search completes. When a deeper search has already started, the shallower search of an identical node is also started. This strategy avoids a deadlock by preventing a shallower node from waiting for a deeper node to return its score, which happened in Figure 4. However, some nodes can be searched more than once even if it does not cause a deadlock, when a deeper node is expanded before a shallower identical node. In practice, this additional overhead is small. The correctness of this approach is proven in [9].

3.4. Implementation Details

TDSAB has been implemented for the games of Awari and Amazons (see `www.cs.ualberta.ca/~games`). The African game of Awari is characterized by a low branching factor (less than 6) and an inexpensive evaluation function. Amazons is a new game that has grown in popularity since it seems to be intermediate in difficulty between chess and Go. It has a very large branching factor (2,176 at the start of the game) and an expensive evaluation function. These games have different properties that exhibit themselves by different characteristics of a parallel search.

Historically, chess has been used to benchmark the performance of parallel $\alpha\beta$ algorithms. Chess is no longer in vogue, and researchers have moved on to other games with interesting research problems. Both Awari and Amazons are the subject of active research efforts and thus are of greater interest.

For Amazons the YBWC strategy was modified. Be-

Number of Processors	Execution Time (seconds)	Speedup	Search Overhead (%)	Synch. Overhead (%)	Comm. Overhead (%)
1	2177.2	-	-	-	-
8	463.81	4.69	18.5	20.3	0.8
16	253.48	8.59	15.3	30.1	1.1
32	152.67	14.26	15.6	40.5	1.5
64	99.80	21.82	15.6	55.0	3.6

Table 1. Awari Performance, 24-ply

Number of Processors	Execution Time (seconds)	Speedup	Search Overhead (%)	Synch. Overhead (%)	Comm. Overhead (%)
1	1604.1	-	-	-	-
8	343.58	4.67	55.8	11.1	0.9
16	180.11	8.90	55.2	15.1	1.0
32	116.11	13.81	79.2	25.4	1.3
64	68.25	23.50	66.4	35.5	3.6

Table 2. Amazons Performance, 5-ply

cause of the large branching factor the basic YBWC strategy distributes too many nodes to the processors, resulting in a lot of search overhead. Therefore, if the first branch of a node does not cause a cut-off, a smaller number of children (P where P is the number of processors) are searched in parallel at a time. If none of these branches causes a cut-off then the next P nodes are searched in parallel, and so on.

For Awari, the search was modified to consider all the children of the root node in parallel. Although this is a search overhead versus synchronization overhead trade-off, it solves a serious problem for any domain with a small branching factor: insufficient work to keep the processors busy (*starvation*).

4. Experiments

The Awari and Amazons programs were written in C and used PVM. Tables 1 and 2 show the experimental results. All results were obtained using Pentium IIIs at 933 Mhz, connected by a 100Mb Ethernet. Each processor had its own 200 MB transposition table. Each data point is the average of 20 test positions. The search depths were chosen so that a test position would take 1-2 minutes on 64 processors (i.e., the typical speed seen in tournaments). Awari, with its low branching factor and inexpensive evaluation function, can search 24-ply deep in roughly the time it takes to search Amazons (and its large branching factor and expensive evaluation function) 5-ply deep (one ply is one move by one player).

To measure synchronization and communication overheads, we used an instrumented version of the programs. Therefore, we note that the theoretical speedups calculated by these overheads do not always reflect the observed speedups in each game.

The Awari results can be compared to previous work us-

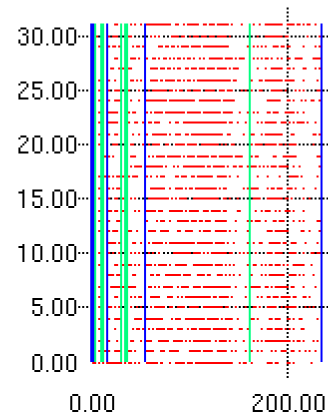

Figure 5. Awari Idle Times

ing checkers, which has a similarly small branching factor. The TDSAB speedup of 21.8 on 64 processors easily beats the APHID speedup of 14.35 using comparable hardware [2]. Analysis of the overheads shows that synchronization is the major culprit. This is not surprising, given that there are 12 iterations (the program iterated in steps of two ply at a time), and an average of 3 synchronization points per iteration. Figure 5 shows a graph of processor idle time (white space) for a typical search. The Y-axis is the processor number (0-31) and the X-axis is time. The vertical lines show where a synchronization point occurred. The last few synchronization points resulted in lots of idle time, limiting the speedup.

Amazons has only slightly better performance (23.5-fold speedup), which may seem surprising given the large branching factor (and, hence, no shortage of work to be done). The very large branching factor turns out to be a liability. At nodes where parallelism can be initiated, many pieces of work are generated, creating lots of concurrent activity (which is good). If a cut-off occurs, many of these pieces of work may have been unnecessary resulting in increased search overhead (which is bad). In this case, search overhead limits the performance, suggesting that the program should be more prudent than it currently is in initiating parallel work. Other parallel implementations have adopted a similar policy of searching subsets of the possible moves at a node, precisely to limit the impact of unexpected cut-offs (for example, [21]).

Multigame is the only previous attempt to parallelize MTD(f) [16] (conventional parallel $\alpha\beta$ was used). Multigame's performance at checkers (21.54-fold speedup) is comparable to TDSAB's result in Awari. For chess, Multigame achieved a 28.42-fold speedup using partitioned transposition tables; better than TDSAB's results in Amazons. However, comparing these numbers is not fair. The Multigame results were obtained using slower machines (Pentium Pros at 200 Mhz versus Pentium IIIs at 933 Mhz), a faster network (Myrinet 1.2Gb/s duplex network versus 100Mb/s Ethernet), longer execution times (roughly 33% larger), and different games.

When comparing TDSAB's parallel performance to that of other implementations (including ZUGZWANG), one must take into account that "standard" $\alpha\beta$ is now an inferior sequential algorithm. MTD(f) builds trees that are 5-15% smaller on average [15]. All speedups should be computed relative to the *best sequential algorithm*. Now that a new standard for sequential $\alpha\beta$ performance has been set, previously published parallel algorithms and results should be re-evaluated.

5. Conclusions

The results of our work on TDSAB are encouraging. Clearly, the TDS framework offers important advantages for a high-performance search application, including asynchronous communication and effective use of memory. However, these advantages are partially offset by the increased synchronization overhead of MTD(f). The end result of this work are speedups that are comparable to what others have reported, perhaps even better when one takes into account the differences between MTD(f) and $\alpha\beta$. This is the first attempt to apply TDS to the two-player domain, and undoubtedly improvements will be found to further enhance performance.

Moving TDS from the single-agent domain to the two-player domain proved challenging. New techniques had to be invented to accommodate the needs of $\alpha\beta$. In many ways, $\alpha\beta$ is a worst-case scenario; most artificial intelligence search algorithms do not need all the parallel capabilities required by $\alpha\beta$. This work generalizes TDS and shows that it can be a powerful parallel paradigm for a wide class of search algorithms.

There are numerous ideas yet to explore with TDSAB including: better priority queue node ordering, reducing MTD(f) synchronization, controlling the amount of parallelism initiated at a node, and speculative search. As well, a TDS implementation of $\alpha\beta$ (not MTD(f)) would be useful for comparison purposes. All these ideas are topics of current research.

6. Acknowledgments

Financial support was provided by the Natural Sciences and Engineering Research Council of Canada (NSERC) and Alberta's Informatics Circle of Research Excellence (iCORE).

References

[1] M. Brockington. A taxonomy of parallel game-tree search algorithms. *International Computer Chess Association Journal*, 19(3):162–174, 1996.

[2] M. Brockington. *Asynchronous Parallel Game-Tree Search*. PhD thesis, Dept. of Computing Science, Univ. of Alberta, 1998.

[3] M. Brockington and J. Schaeffer. Aphid: Asynchronous parallel game-tree search. *Journal of Parallel and Distributed Computing*, 60:247–273, 2000.

[4] R. Feldmann. *Game Tree Search on Massively Parallel Systems*. PhD thesis, Univ. of Paderborn, August 1993.

[5] R. Feldmann, B. Monien, P. Mysliwietz, and O. Vornberger. Distributed game tree search. *Journal of the International Computer Chess Association*, 12(2):65–73, 1989.

[6] E. Felten and S. Otto. Chess on a hypercube. In G. Fox, editor, *Third Conference on Hypercube Concurrent Computers and Applications*, volume II-Applications, pages 1329–1341, 1988.

[7] F. Hsu. IBM's Deep Blue chess grandmaster chips. *IEEE Micro*, (March-April):70–81, 1999.

[8] R. Hyatt and B. Suter. A parallel alpha/beta tree searching algorithm. *Parallel Computing*, 10:299–308, 1989.

[9] A. Kishimoto. Transposition Table Driven Scheduling for Two-Player Games. Master's thesis, Dept. of Computing Science, Univ. of Alberta, 2002.

[10] B. Kuszmaul. *Synchronized MIMD Computing*. PhD thesis, Massachusetts Institute of Technology, 1994.

[11] U. Lorenz. Parallel controlled conspiracy number search. In *13th Annual Symposium on Parallel Algorithms and Architectures (SPAA2001)*, pages 320–321, 2001.

[12] T. Marsland. Relative performance of alpha-beta implementations. In *IJCAI*, pages 763–766, 1983.

[13] T. Marsland and F. Popowich. Parallel game-tree search. *IEEE PAMI*, 7(4):442–452, 1985.

[14] M. Newborn. Unsynchronized iteratively deepening parallel alpha-beta search. *IEEE PAMI*, 10(5):687–694, 1988.

[15] A. Plaat, J. Schaeffer, W. Pijls, and A. de Bruin. Best-first fixed-depth minimax algorithms. *Artificial Intelligence*, 87(1–2):1–38, 1996.

[16] J. Romein. *Multigame - An Environment for Distributed Game-Tree Search*. PhD thesis, Vrije Universitat Amsterdam, 2001.

[17] J. Romein, H. Bal, J. Schaeffer, and A. Plaat. A performance analysis of transposition-table-driven scheduling. *IEEE PDS*, 2002. To appear.

[18] J. Romein, A. Plaat, H. Bal, and J. Schaeffer. Transposition table driven work scheduling in distributed search. *AAAI National Conference*, pages 725–731, 1999.

[19] J. Schaeffer. Distributed game-tree searching. *Journal of Parallel and Distributed Computing*, 6:90–114, 1989.

[20] D. Slate and L. Atkin. CHESS 4.5 - The Northwestern University Chess Program. In P. Frey, editor, *Chess Skill in Man and Machine*, pages 82–118. Springer-Verlag, 1977.

[21] J.-C. Weill. The ABDADA distributed minmax-search algorithm. *International Computer Chess Association Journal*, 19(1):3–16, 1996.