Transposition Table Driven Work Scheduling in Distributed Game-Tree Search

Akihiro Kishimoto and Jonathan Schaeffer

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

Abstract. MTD(f) is a new variant of the $\alpha\beta$ algorithm that has become popular amongst practitioners. TDS is a new parallel search algorithm that has proven to be effective in the single-agent domain. This paper presents TDSAB, applying the ideas behind TDS parallelism to the MTD(f) algorithm. Results show that TDSAB gives comparable performance to that achieved by conventional parallel $\alpha\beta$ algorithms. This result is very encouraging, given that traditional parallel $\alpha\beta$ approaches appear to be exhausted, while TDSAB opens up new opportunities for further performance improvements.

1 Introduction

Many artificial intelligence applications require real-time responses. Achieving this can be achieved by a combination of software and hardware. On the software side, *anytime* algorithms have been developed to ensure that a quality response is available as a function of the resources consumed [3]. 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 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 [8]. Second, TDS is a new, powerful parallel search paradigm for distributed-memory hardware that has been applied to single-agent search [11, 10]. 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. This paper investigates the issues of using TDS to parallelize MTD(f).

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 value, 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 the overhead of parallelism, and produces impressive results for single-agent search applications.

This paper introduces TDSAB, TDS parallelism adapted to $\alpha\beta$ search (specifically, MTD(f)) [5]. This is the first implementation of TDS in the two-player domain, and the results are very encouraging. The speedups in two application domains (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 algorithms. Given that this is the first implementation of TDSAB, and that there are numerous ideas for further enhancing performance, this is a successful result. 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 [7] and parallel [1] 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.

A large number of enhancements have been added to $\alpha\beta$ to (dramatically) improve the search efficiency. The most important of these is the transposition table, a large cache containing the results from previously searched sub-trees [13]. 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 s;
   int lowerbound = −∞; upperbound = ∞;
   if (f == -∞) b = f + 1;
   else b = f;
   do {
      /* Null window search */
(+)   s = AlphaBeta(n, d, b-1, b);
      if (s < b) upperbound = s;
      else lowerbound = s;
      /* Re-set the bound */
      if (lowerbound == s) b = s + 1;
      else b = s;
   } while (lowerbound ≠ upperbound);
}
```

Fig. 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. There are numerous obstacles to overcome:

1. *Search overhead* is the (usually) larger tree built by the parallel algorithm as compared to the sequential algorithm.
2. *Synchronization overhead* occurs when processors have to sit idle waiting for the results from other searches.
3. *Communication overhead* is the result of processors sending messages between each other.
4. *Load balancing* reflects how evenly the work has been distributed between the processors.

None of these overheads is independent of each other. For example, increased communication can reduce search overhead; reducing synchronization time usually increases search overhead. A high-performance $\alpha\beta$ searcher needs to be finely tuned to choose the right mix of parameters to balance the performance trade-offs.

To minimize synchronization overhead, many algorithms 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 adverse impact because of the lack of information available in the transposition table.

2.3 Parallel Search Algorithms

Numerous parallel $\alpha\beta$ search algorithms have been proposed. The Young Brothers Wait Concept (YBWC) [4] is a well-known representative of this class of algorithms. There are many variants of YBWC, but they only differ in the implementation details (e.g., [6, 14]). 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 cutoff 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 cutoff is investigated before initiating the parallelism) at the price of increasing the synchronization overhead (waiting for the first branch to return).

Practical $\alpha\beta$ search algorithms have transposition tables. When parallelizing $\alpha\beta$ search with the work-stealing framework on distributed-memory machines, 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 the processor P updates a table entry, P sends a message to the corresponding processor to update the entry. Updating messages can be done 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, 12].

ABDADA [14] does not use work-stealing, preferring a shared transposition table to control the parallel search. All the processors start searching the root node simultaneously. Each transposition table entry has a field for the number of processors entering a node, which is used to determine the order in which to search children of that node. ABDADA achieved better speedups than YBWC in chess and Othello on a shared memory machine. However, it is hard to implement ABDADA on distributed-memory machines because of the necessity of sharing the transposition tables.

2.4 TDS

Transposition-table Driven Scheduling (TDS) flips the idea of work-stealing to solve the transposition table problem [11, 10]. While work-stealing moves the data to where the work is located, TDS moves the work to where the data is located.

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 $\alpha\beta$ that complicate the issue. First, $\alpha\beta$ has a search window which IDA* does not have. This makes the implementation of TDS complicated because (a) the window may be narrowed after searching a node, and (b) a node reached through more than one path may be searched with different windows. Second, the order in which nodes are considered in $\alpha\beta$ is much more important than in IDA*. In IDA*, the search order is managed using a stack to let TDS behave in a depth-first manner. For parallel $\alpha\beta$ search, we need a more complicated scheme to let the left-most and shallowest nodes be searched first. Finally, IDA* does not have $\alpha\beta$-like pruning. The results of the children in IDA* are not reported back to its parent. On the other hand, $\alpha\beta$ checks if a cutoff happens, after searching a branch of a node. When a cutoff occurs, we need a mechanism to not only receive the scores reported from the children, but also to tell the other processors to stop searching other branches to avoid unnecessary search.

TDS has several important advantages:

1. Transposition table access always involve only local communication. 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 $\alpha\beta$ search.

3 TDSAB

This section presents a new parallel $\alpha\beta$ algorithm, TDSAB (Transposition-table Driven Scheduling Alpha-Beta), combining parallel search (TDS) with MTD(f). 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:

- **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 [5] for more details).

– **Pruning:** When searching the children of a node in parallel and a cutoff score is returned, further work at this node is not necessary; all outstanding work must be stopped. However, because in TDS all the descendants of a node are not always 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. Cutoffs can be elegantly handled by the idea of giving each node a *signature*. Intuitively the signature for a node P is a function of the path traversed from the root node to P; hence every node has a unique signature. When a cutoff happens at a node P, TDSAB broadcasts the signature of P to all the processors. A processor receiving a cutoff signature examines its local priority queue and deletes all the nodes whose *prefix* of the signatures are the same as the signature of P.

Figure 2 gives pseudo code for TDSAB. For simplicity, we just explain TDSAB without YBWC. 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 done a one-ply search and has spawned the children to be evaluated in parallel. The new work arrives, assigned a priority, and inserted in the priority queue. If the new piece of work is terminal or a small piece of work then it is immediately searched locally (the cost of doing it in parallel out-weights the benefits) and sent to its parent node (lines marked "-" in the figure).
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.

If new information arrives at a processor, $GetNode$, $GetSignature$, and $Get-SearchResult$ 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 processor having a transposition entry to the node, decided by the transposition table key ($TTKey$).

When receiving a search result, TDSAB has to consider two cases ($Store-SearchResult$). 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 ($SendScore$).

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

    void ParallelMWS() {
      int type, value;
      node_t p;
      signature_t signature;
      do {
        if (RecevNode(&type) == TRUE) { /* Check if new information arrives. */
          switch(type) {
            case NEW_WORK:  /* New work is stored in its priority queue. */
              GetNode(&p); Enqueue(p); break;
(*)         case CUT_OFF:  /* Obsolete nodes are deleted from its priority queue. */
(*)           GetSignature(&signature); CutAllDescendants(signature); break;
(+)         case SEARCH_RESULT:
(+)           GetSearchResult(&p,&value);
(+)           StoreSearchResult(p,value); break;
          }
        }
        GetLocalJob(&p);
        if (p ≠ NOT_FOUND) {
(-)       if (p == terminal || p.depth ≤ granularity) {
(-)         value = AlphaBeta(p,p.depth,α,α + 1); /* Local search is done for small work. */
(-)         SendScore(p.parent,value);
            DeleteLocalJob(p);
          } else {  /* Do one-ply search in parallel. */
            for (int i = 0; i < p.num_of_children; i++) {
              p.child_node[i].depth = p.depth - 1;
              SendNode(p.child_node[i],HomeProcessor(TTKey(p.child_node[i])));
            }
          }
        }
      } 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);
        }
      }
    }
```

Fig. 2. Simplified Pseudo Code for TDSAB (without YBWC)

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 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 recently-invented 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 charateristics of a parallel search.

For Amazons the YBWC strategy was modified. Because 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 cutoff, 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.

Two enhancements were added to our Awari implementation. First, all the children of the root node are searched 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*). Second, 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 a node is *always* delayed until after the completion of the other node, a deadlock may occur. Figure 3 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 a processor of B 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 ensues.

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

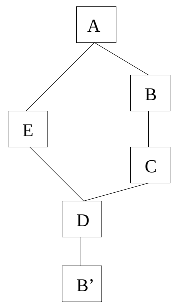

Fig. 3. Deadlock with Cycles

preventing a shallower node from waiting for a deeper node to return its score, which happened in Figure 3. However, some nodes are searched more than once even if it does not cause a deadlock, when a deeper node is expanded before a shallower indentical node.

4 Experiments

Tables 1 and 2 show the experimental results for Awari and Amazons. All results were obtained using Pentium IIIs at 933 Mhz, connected by a 100 Mb/s Ethernet. Each processor had its own 200 MB transposition table. The search depths were chosen so that the typical 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.

To measure synchronization and communication overheads, we used different programs, which have extra operations than those used to measure speedups and search overhead. 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 using 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 a a time), which contained an average of 34 synchronization points in the experiments. Figure 4 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. Clearly, the last few synchronization points resulted in a large amount of idle time, limiting the speedup.

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. Performance and Overheads (Awari, 24-ply Search)

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. Performance and Overheads (Amazons, 5-ply Search)

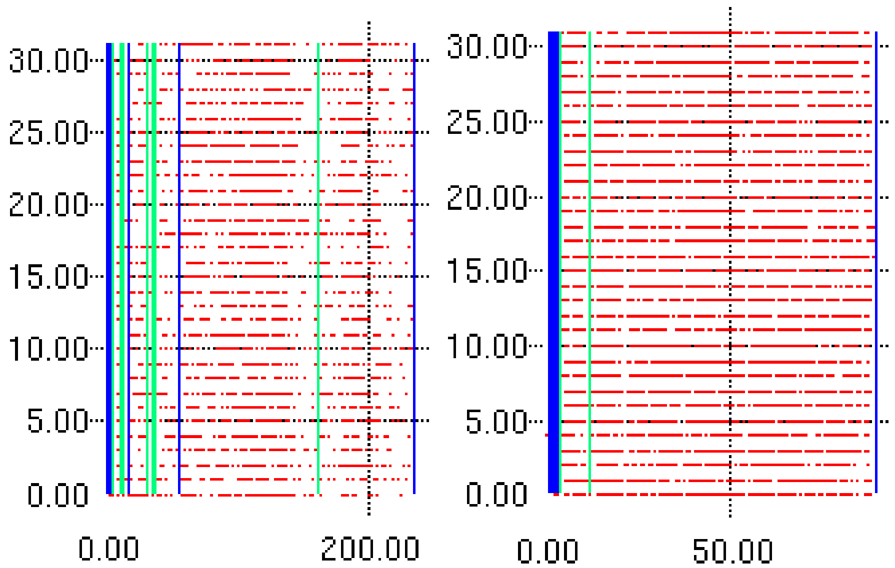

Fig. 4. Awari (left) and Amazons (Right) Idle Times

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 cutoff 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 cutoffs (for example, [14]).

In some sense, implementing a new parallel idea using MTD(f) was a mistake. When comparing results with previous work, one has to realize that two parameters have changed: the sequential algorithm (MTD(f) versus $\alpha\beta$) and the parallel algorithm (TDS versus something like YBWC). Thus, it is not obvious which component is responsible for any parallel inefficiencies. As the synchronization overhead for Awari demonstrates, MTD(f) has more synchronization points at the root and, hence, more synchronization overhead.

Multigame is the only previous attempt to parallelize MTD(f) [9]. Multigame's performance at checkers (21.54-fold speedup) is comparable to TDSAB's result in Awari. In 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 1.2 Gb/s duplex network versus 100 Mb/s Ethernet), longer execution times (roughly 33% larger), and different games. Chess and checkers could have been used for our TDSAB implementations, allowing for a fairer comparison between our work and the existing literature. However, chess and checkers no longer seem to interest the research community. Both Awari and Amazons are the subject of active research efforts and thus are of greater interest.

5 Conclusions

The results of our work on TDSAB are both encouraging and discouraging. 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 achieved. This is disappointing, since given the obvious advantages of transposition table driven scheduling, one would hope for a better result. On the other hand, this is the first attempt to apply TDS to the two-player domain, and undoubtedly improvements will appear.

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. 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. Mark Brockington. A taxonomy of parallel game-tree search algorithms. *International Computer Chess Association Journal*, 19(3):162–174, 1996.
2. Mark Brockington. *Asynchronous Parallel Game-Tree Search*. PhD thesis, Department of Computing Science, University of Alberta, 1998.
3. T. Dean and M. Boddy. An analysis of time-dependent planning. In *AAAI National Conference*, pages 49–54, 1988.
4. Rainer Feldmann. *Game Tree Search on Massively Parallel Systems*. PhD thesis, University of Paderborn, August 1993.
5. Akihiro Kishimoto. Transposition Table Driven Scheduling for Two-Player Games. Master's thesis, Department of Computing Science, University of Alberta, 2002.
6. Bradley Kuszmaul. *Synchronized MIMD Computing*. PhD thesis, Massachusetts Institute of Technology, 1994.
7. Tony Marsland. Relative performance of alpha-beta implementations. In *International Joint Conference on Artificial Intelligence*, pages 763–766, 1983.
8. Aske Plaat, Jonathan Schaeffer, Wim Pijls, and Arie de Bruin. Best-first fixed-depth minimax algorithms. *Artificial Intelligence*, 87(1–2):1–38, 1996.
9. John Romein. *Multigame - An Environment for Distributed Game-Tree Search*. PhD thesis, Vrije Universitat Amsterdam, 2001.
10. John Romein, Henri Bal, Jonathan Schaeffer, and Aske Plaat. A performance analysis of transposition-table-driven scheduling. *IEEE Transactions on Parallel and Distributed Systems*, 2001. To appear.
11. John Romein, Aske Plaat, Henri Bal, and Jonathan Schaeffer. Transposition table driven work scheduling in distributed search. *AAAI National Conference*, pages 725–731, 1999.
12. Jonathan Schaeffer. Distributed game-tree searching. *Journal of Parallel and Distributed Computing*, 6:90–114, 1989.
13. David Slate and Larry Atkin. CHESS 4.5 - The Northwestern University Chess Program. In Peter Frey, editor, *Chess Skill in Man and Machine*, pages 82–118. Springer-Verlag, 1977.
14. J.-C. Weill. The ABDADA distributed minmax-search algorithm. *International Computer Chess Association Journal*, 19(1):3–16, 1996.