Transposition Table Driven Work Scheduling in Distributed Search

John W. Romein, Aske Plaat and **Henri E. Bal**
john@cs.vu.nl aske@cs.vu.nl bal@cs.vu.nl
Department of Computer Science,
Vrije Universiteit,
Amsterdam, The Netherlands

Jonathan Schaeffer
jonathan@cs.ualberta.ca
Department of Computing Science,
University of Alberta,
Edmonton, Canada

Abstract

This paper introduces a new scheduling algorithm for parallel single-agent search, *transposition table driven work scheduling*, that places the transposition table at the heart of the parallel work scheduling. The scheme results in less synchronization overhead, less processor idle time, and less redundant search effort. Measurements on a 128-processor parallel machine show that the scheme achieves nearly-optimal performance and scales well. The algorithm performs a factor of 2.0 to 13.7 times better than traditional work-stealing-based schemes.

Introduction

Heuristic search is one of the cornerstones of AI. Its applications range from logic programming to pattern recognition, from theorem proving to chess playing. For many applications, such as real-time search and any-time algorithms, achieving high performance is of great importance, both for solution quality and execution speed.

Many search algorithms recursively decompose a state into successor states. If the successor states are independent of each other, then they can be searched in parallel. A typical scenario is to allocate a portion of the search space to each processor in a parallel computer. A processor is assigned a set of states to search, performs the searches, and reports back the results. During the searches, each processor maintains a list of work yet to be completed (the *work queue*). When a processor completes all its assigned work, it can be pro-active and attempt to acquire additional work from busy processors, rather than sit idle. This approach is called *work stealing*.

Often, however, application-specific heuristics and search enhancements introduce interdependencies between states, making efficient parallelization a much more challenging task. One of the most important search enhancements is the transposition table, a large cache in which newly expanded states are stored (Slate & Atkin 1977). The table has many benefits, including preventing the expansion of previously encountered states, move ordering, and tightening the search bounds. The transposition table is particularly useful when a state can have multiple predecessors (i.e., when the search space is a graph rather than a tree). The basic tree-based recursive node expansion strategy would expand states with multiple predecessors multiple times. A transposition table can result in time savings of more than a factor 10, depending on the application (Plaat *et al.* 1996).

Unfortunately, transposition tables are difficult to implement efficiently in parallel search programs that run on distributed-memory machines. Usually, the transposition table is partitioned among the local memories of the processors (for example, (Feldmann 1993)). Before a processor expands a node, it first does a remote lookup, by sending a message to the processor that manages the entry and then waiting for the reply (see Figure 1). This can result in sending many thousands of messages per second, introducing a large communication overhead. Moreover, each processor wastes much time waiting for the results of remote lookups. The communication overhead can be reduced (e.g., by sending fewer messages), but this usually increases the size of the search tree that needs to be explored. Extensive experimentation may be required to find the “right” amount of communication to maximize performance.

In this paper, we discuss a different approach for implementing distributed transposition tables, called *transposition table driven work scheduling* (or *transposition-driven scheduling*, TDS, for short). The idea is to integrate the parallel search algorithm and the transposition table mechanism: drive the work scheduling by the transposition table accesses. The state to be expanded is migrated to the processor on which the transposition for the state is stored (see Figure 2). This processor performs the local table lookup and stores the state in its work queue. Although this approach may seem counterintuitive, it has important advantages:

1. All communication is asynchronous (nonblocking). A processor expands a state and sends its children to their home processors, where they are entered into the transposition table and in the work queue. After sending the messages the processor continues with the next piece of work. Processors never have to wait for the results of remote lookups.
2. The network latency is hidden by overlapping communication and computation. This latency hiding is effective as long as there is enough bandwidth in the network to cope with all the asynchronous messages. With modern high-speed networks such bandwidth usually is more than enough available.

The idea of transposition-driven scheduling can apply to a

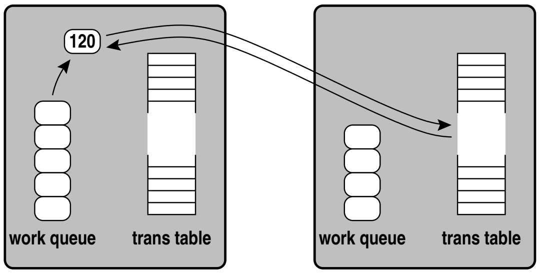


Figure 1: Work stealing with a partitioned table.

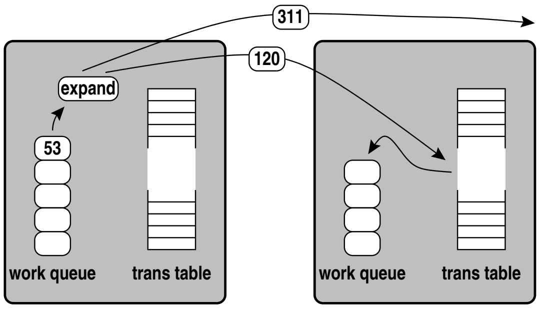


Figure 2: Transposition-driven scheduling.

variety of search algorithms. In this paper we describe the algorithm and present performance results for single-agent search (IDA* (Korf 1985)). We have implemented TDS on a large-scale cluster computer consisting of Pentium Pro PCs connected by a Myrinet network. The performance of this algorithm is compared with the traditional work stealing scheme. Performance measurements on several applications show that TDS wins a factor of 2.0 to 13.7 at the application level and thus outperforms the work stealing scheme by a large margin. Moreover, TDS scales much better to large numbers of processors. On 128 processors, TDS is 109 to 122 times faster than on a single processor, while the work stealing algorithm obtains speedups of only 8.7 to 62.

In traditional parallel single-agent search algorithms, the algorithm revolved around the work queues, with other enhancements, such as the transposition table, added in as an afterthought. With TDS, the transposition table is at the heart of the algorithm, recognizing that the search space really is a graph, not a tree. The result is a simple parallel single-agent search algorithm that achieves high performance.

The main contribution of this paper is to show how effective the new approach is for single-agent search. We discuss in detail how TDS can be implemented efficiently and we explain why it works so well compared to work stealing. The rest of this paper is organized as follows. First, we give some background information on parallel search algorithms and related work. Then, we describe the transposition-driven scheduling approach. Next, we evaluate the performance of the new approach. The last section presents conclusions.

Background and Related Work

This paper uses IDA* (Iterative Deepening A*) as the single-agent search algorithm (Korf 1985). IDA* repeatedly performs a depth-first search, using a maximum search depth that is increased after each iteration, until a solution is found. The answer is guaranteed to be optimal, assuming that the heuristic used is admissible. We parallelize IDA* in two ways, differing in the way the search space is distributed over the processors. One uses work-stealing and the other uses TDS for distributing the work. Our analysis is facilitated by the *Multigame* environment for distributed one and two player search (Romein, Bal, & Grune 1997).

Numerous parallel single-agent search algorithms have appeared in the literature. The most popular are task distribution schemes where the search tree is partitioned among all the available processors (Rao, Kumar, & Ramesh 1987). Task distribution can be simplified by expanding the tree in a breadth-first fashion until the number of states on the search frontier matches the number of processors (Kumar & Rao 1990). This can cause load balancing problems (the search effort required for a state varies widely), implying that enhancements, such as work stealing, are necessary for high performance. A different approach is Parallel Window Search, where each processor is given a different IDA* search bound for its search (Powley & Korf 1991). All processors search the same tree, albeit to different depths. Some processors may search the tree with a search bound that is too high. Since sequential IDA* stops searching after using the right search bound, PWS results in much wasted work.

All these schemes essentially considered only the basic IDA* algorithm, without consideration of important search algorithm enhancements that can significantly reduce the search tree size (such as transposition tables).

IDA* uses less space than A*. This comes at the expense of expanding additional states. The simple formulation of IDA* does not include the detection of duplicate states (such as a cycle, or transposing into a state reached by a different sequence of state transitions). The transposition table is a convenient mechanism for using space to solve these search inefficiencies, both in single-agent (Reinefeld & Marsland 1994) and two-player (Slate & Atkin 1977) search algorithms. There are other methods, such as finite state machines (Taylor & Korf 1993), but they tend to be not as generally applicable or as powerful as transposition tables.

A transposition table (Slate & Atkin 1977) is a large (possibly set-associative) cache that stores intermediate search results. Each time a state is to be searched, the table is checked to see whether it has been searched before. If the state is in the table, then, depending on the quality of the information recorded, additional search at this node may not be needed. If the state is not in the table, then the search engine examines the successors of the state recursively, storing the search results into the transposition table.

Indexing the transposition table is usually done by hashing the state to a large number (usually 64 bits or more) called the *signature* (Zobrist 1970). The information in the table depends on the search algorithm. For the IDA* algorithm, the table contains a lower bound on the number of moves required to reach the target state. In addition, each entry may contain information used by table entry replacement algorithms, such as the effort (number of nodes searched) to compute the entry.

In parallel search programs the transposition table is typically shared among all processes, because a position analyzed by one process may later be re-searched by another process. Implementing shared transposition tables efficiently on a distributed-memory system is a challenging problem, because the table is accessed frequently. Several approaches are possible. With *partitioned* transposition tables, each processor contains part of the table. The signature is used to determine the processor that manages the table entry corresponding to a given state. To read or update a table entry, a message must be sent to that processor. Hence, most table accesses will involve communication (typically $(p-1)/p$ for p processors). Lookup operations are usually implemented using synchronous communication, where requesters wait for results. Update operations can be sent asynchronously. An advantage of partitioned tables is that the size of the table increases with the number of processors (more memory becomes available). The disadvantage is that lookup operations are expensive: the delay is at least twice the network latency (for the request and the reply messages). In theory, remote lookups could be done asynchronously, where the node expansion goes ahead speculatively before the outcome of the lookup is known. However, this approach is complicated to implement efficiently and suffers from thread-switching and speculation overhead.

Another approach is to *replicate* the transposition table entries in the local memory of each machine. This has the advantage that all lookups are local, and updates are asynchronous. The disadvantage is that updates must now be broadcast to *all* machines. Even though broadcast messages are asynchronous and multiple messages can be combined into a single physical message, the overhead of processing the broadcast messages is high and increases with the number of processors. This limits the scalability of algorithms using this technique, and replicated tables are seldom used in practice. Moreover, replicated tables have fewer entries than partitioned tables, as each entry is stored on each processor. A third approach is to let each processor maintain only a *local* transposition table, independent from the other processors (Marsland & Popowich 1985). This would eliminate communication overhead, but results in a large search overhead (different processors would search the same node). For many applications, local tables are the least efficient scheme.

Also possible are hybrid combinations of the above. For example, each processor could have a local table, but replicate the "important" parts of the table by periodically broadcasting this information to all processors (Brockington 1997). Several enhancements exist to these basic schemes. One technique for decreasing the communication overhead is to not access the distributed transposition table when searching near the leaves of the tree (Schaeffer 1989). The potential gains of finding a table entry near the root of the tree are larger because a pruned subtree rooted high in the tree can save more search effort than a small subtree rooted low in the tree. Another approach is to optimize the communication software for the transposition table operations. An example is given in (Bhoedjang, Romein, & Bal 1998), which describes software for Myrinet network interface cards that is customized for transposition tables.

Despite these optimizations, for many applications the cost of accessing and updating transposition tables is still high. In practice, this overhead can negate most of the benefits of including the tables in the search algorithm, and researchers have not stopped looking for a better solution. In the next section, we will describe an alternative approach for implementing transposition tables on distributed-memory systems: using TDS instead of work stealing. By integrating transposition table access with work scheduling, this approach makes all communication asynchronous, allowing communication and computation to be overlapped. Much other research has been done on overlapping communication and computation (von Eicken *et al.* 1992). The idea of self-scheduling work dates back to research on data flow and has been studied by several other researchers (see, for a discussion, (Culler, Schauser, & von Eicken 1993)). In the field of problem solving, there are some cases in which this idea has been applied successfully. In software verification, the parallel version of the Murphi theorem prover uses its hash function to schedule the work (Stern & Dill 1997). In game playing, a parallel generator of end-game databases (based on retrograde analysis) uses the Gödel number of states to schedule work (Bal & Allis 1995). In single agent search, a parallel version of A*, PRA*, partitions its OPEN and CLOSED lists based on the state (Evett *et al.* 1995).

Interestingly, in all three papers the data-flow-like parallelization is presented as following in a natural way from the problem at hand, and, although the authors report good speedups, they do not compare their approaches to more traditional parallelizations. The paper on PRA*, for example, does discuss differences with IDA* parallelizations, but focuses on a comparison of the *number* of node expansions, without addressing the benefit of asynchronous communication for *run times*.[1] (A factor may be that PRA* was designed for the CM-2, a SIMD machine whose architecture makes a direct comparison with recent work on parallel search difficult.)

Despite its good performance, so far no in-depth performance study between work stealing and data-flow-like approaches such as TDS has been performed for distributed search algorithms.

Transposition-Driven Work Scheduling

The problem with the traditional work stealing approach is that it is difficult to combine with shared transposition tables. To overcome this problem, we investigate a different approach, in which the work scheduling and transposition table mechanisms are integrated. The traditional approach is to move the data to where the work is located. Instead, we move the work to where the data is located. Work is sent to the processor that manages the associated transposition table entry, instead of doing a remote lookup to this processor.

[1] Evett et al compare PRA* against versions of IDA* that lack a transposition table. Compared to IDA* versions with a transposition table, PRA*'s node counts would have been less favorable.

```
PROCEDURE MainLoop() IS
   WHILE NOT Finished DO
      Node := GetLocalJob();
      IF Node <> NULL THEN
         Children := ExpandNode(Node);
         FOR EACH Child IN Children DO
            IF Evaluate(Child) <= Child.SearchBound THEN
               Dest = HomeProcessor(Signature(Child));
               SendNode(Child, Dest);
            END
         END
      ELSE
         Finished := CheckGlobalTermination();
      END
   END
END


PROCEDURE ReceiveNode(Node) IS
   Entry := TransLookup(Node);
   IF NOT Entry.Hit OR
              Entry.SearchBound <= Node.SearchBound THEN
      TransStore(Node);
      PutLocalJob(Node);
   END
END
```

Figure 3: Simplified TDS algorithm.

Once this is done, the sending processor can process additional work *without having to wait for any results to come back*. This makes all communication asynchronous, allowing the costs of communication to be hidden. Below we first describe the basic algorithm and then we look at various implementation issues.

The basic algorithm

Each state (search tree node) is assigned a *home processor*, which manages the transposition table entry for this node. The home processor is computed from the node's signature. Some of the signature bits indicate the processor number of the node's home, while some of the remaining bits are used as an index into the transposition table at that processor.

Figure 3 shows the simplified pseudo code for a transposition-driven scheduling algorithm, which is executed by every processor. The function *MainLoop* repeatedly tries to get a node from its local work queue. If the queue is not empty, it expands the node on the head of the queue by generating the children. Then it checks for each child whether the lower bound on the solution length (*Evaluate*) causes a cutoff (the lower bound exceeds the IDA* search bound). If not, the child is sent to its home processor (see also Figure 2). When the local work queue is empty, the algorithm checks whether all other processors have finished their work and no work messages are in transit. If not, it waits for new work to arrive.

The function *ReceiveNode* is invoked for each node that is received by a processor. The function first does a transposition table lookup to see whether the node has been searched before. If not, or if the node has been searched to an inadequate depth (e.g., from a previous iteration of IDA*), the node is stored into the transposition table and put into the local work queue; otherwise it is discarded because it has transposed into a state that has already been searched adequately.

The values stored in the transposition table are used differently for work stealing and TDS. With work stealing, a table entry stores a lower bound on the minimal distance to the target, derived by searching the subtree below it. Finding a transposition table hit with a suitably high table value indicates that the node has been previously searched adequately for the current iteration. With TDS, an entry contains a search *bound*. It indicates that the subtree below the node has either been previously searched adequately (as above) or is currently being searched with the given bound. Note that this latter point represents a major improvement on previous distributed transposition table mechanisms in that it prevents two processors from ever working on the same subtree concurrently.

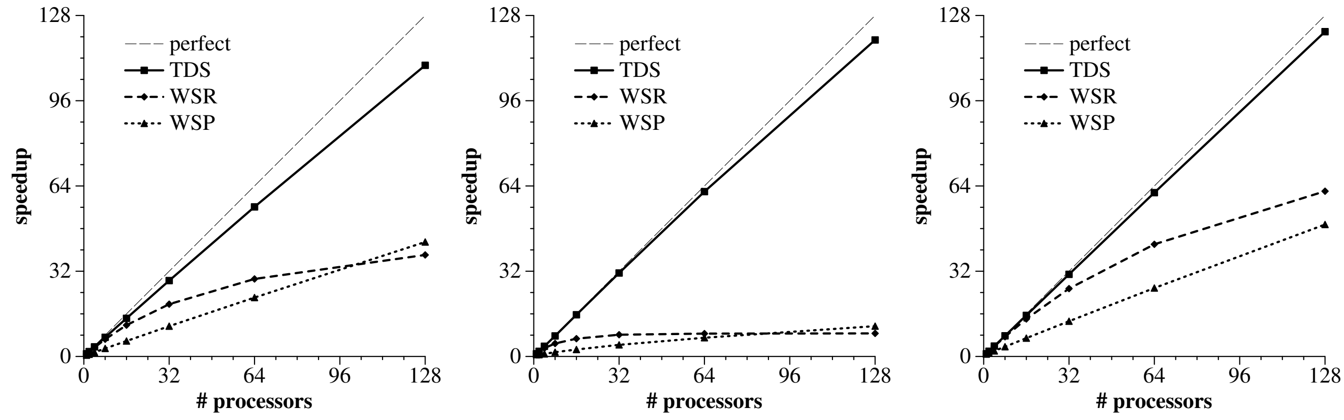
Implementation issues

We now discuss some implementation issues of this basic algorithm. An important property in our TDS implementation of IDA* is that a child node does not report its search result to its parent. As soon as a node has forked off new work for its children, work on the node itself has completed. In some cases (for example, for two-agent search) the results of a child should be propagated to its parent. This complicates the algorithm since it requires parent nodes to leave state information behind, and may result in some work in progress having to be aborted (for example, when an alpha-beta cutoff occurs). This is the subject of ongoing research.

When no results are propagated to the parent, the TDS algorithm needs a separate mechanism to detect global termination. TDS synchronizes after each IDA* iteration, and starts a new iteration if the current iteration did not solve the problem. One of the many distributed termination detection algorithms can be used. We use the time count algorithm from (Mattern 1987). Since new iterations are started infrequently, the overhead for termination detection is negligible.

Another issue concerns the search order. It is desirable to do the parallel search in a depth-first way as much as possible, because breadth-first search will quickly run out of memory for intermediate nodes. Depth-first behavior could be achieved using priority queues, by giving work on the left-hand side of the search tree a higher priority than that on the right-hand side of the tree. However, manipulating priority queues is expensive. Instead, we implement each local work queue as a stack, at the possible expense of a larger working set. On one processor, a stack corresponds to pure depth-first search.

An interesting trade-off is when and where to invoke the node evaluation function. One option is to do the evaluation on the processor that creates a piece of work, and migrate the work to its home processor only if the evaluation did not cause a cutoff. Another option is to migrate the work immediately to its home processor, look it up in the transposition table, and then call the evaluation function only if the lookup did not cause a cutoff. The first approach will migrate less work but will always invoke the evaluation function, even if it has been searched before (on the home pro-



(a) 15-puzzle (b) double-blank puzzle (c) Rubik's cube

Figure 4: Average application speedups.

cessor). Whichever is more efficient depends on the relative costs for migrating and evaluating nodes. On our system, the first approach performs the best for most applications.

An important optimization performed by our implementation is message combining. To decrease the overhead per migrated state, several states that have the same source and the same destination processors are combined into one physical message. Each processor maintains a message buffer for every other processor. A message buffer is transmitted when it is full, or when the sending processor has no work to do (typically during the start and the end of each iteration, when there is little work).

Discussion

Transposition-driven scheduling has five advantages:

1. All transposition table accesses are local.
2. All communication is done asynchronously; processors do not wait for messages. As a result, the algorithm scales well to large numbers of processors. The total bandwidth requirements increase approximately linearly with the number of processors.
3. No duplicate searches are performed. With work stealing, multiple processors sometimes concurrently search a transposition because the transposition table update occurs *after* the subtree below it was searched. With the new scheme this cannot occur; all attempts to search a given subtree must go through the same home processor. Since it has a record of all completed and in-progress work (in the transposition table), it will not allow redundant effort.[2]
4. TDS produces more stable execution times for trees with many transpositions than the work stealing algorithm.
5. No separate load balancing scheme is needed. Previous algorithms require work stealing or some other mechanism to balance the work load. Load balancing in TDS is done implicitly, using the hash function. Most hash functions are uniformly distributed, causing the load to be distributed evenly over the machines. This works well as long as all processors are of the same speed. If this is not the case, then the stacks of the slow processors will grow and may exhaust memory. A flow control scheme can be added to keep processors from sending states too frequently. In our experiments, we have not found the need to implement such a mechanism.

Measurements

We compare the performance of TDS with that of work stealing, both with partitioned (WSP) and replicated (WSR) transposition tables. Our test suite consists of three games: the *15-puzzle*, the *double-blank puzzle*, and *Rubik's cube*. The double-blank puzzle is a modification to the 15-puzzle, where we removed the tile labeled '15'. By having two blanks, we create a game with many transpositions, because two consecutive moves involving both blanks can usually be interchanged.

The 15-puzzle evaluation function includes the Manhattan distance, linear conflict heuristic (Hansson, Mayer, & Yung 1992), last move heuristic (Korf & Taylor 1996), and corner conflict heuristic (Korf & Taylor 1996). The double-blank puzzle uses the same evaluation function, adapted for two blanks. The Rubik's cube evaluation is done using pattern databases (Korf 1997), one each for corners and edges.

The test positions used for the 15-puzzle are nine of the hardest positions known (Gasser 1995).[3] To avoid long

[2]There is one situation in which duplicate work will get done. If the transposition table is too small for the given search, some table entries will get overwritten. This loss of information can result in previously completed searches being repeated. This is a fundamental problem with fixed-size transposition tables.

[3]Most parallel 15-puzzle programs are benchmarked on the 100 test problems in (Korf 1985). Unfortunately, using a sophisticated lower bound and a fast processor means that many of these test problems are solved sequentially in a few seconds. Hence, a more challenging test suite is needed.

sequential searches, we stopped searching after a 74-ply search iteration. For the double-blank puzzle, we used the same positions with the '15'-tile removed, limited to a 66-ply search depth. Rubik's cube was tested using 5 random problems. Since a random problem requires weeks of CPU time to solve, we limited the search depth to 17. The WSP and WSR programs have been tuned to avoid remote table accesses for nodes near the leaves whenever that increases performance.

We studied the performance of each of the algorithms on a cluster of 128 Pentium Pros running at 200 MHz. Each machine has 128 Megabytes of RAM. For the 15-puzzle and the double-blank puzzle, we use 2^{22} transposition table entries per machine. For Rubik's cube we use 2^{21} entries, to leave room for pattern databases. The machines are connected through Myrinet (Boden *et al.* 1995), a 1.2 Gigabit/second switching network. Each network interface board contains a programmable network processor. WSP runs customized software on the network coprocessor to speed up remote transposition table accesses (Bhoedjang, Romein, & Bal 1998). WSR and TDS use generic network software (Bhoedjang, Rühl, & Bal 1998).

Figure 4 shows speedups with respect to TDS search on a single processor, which is virtually as fast as sequential search. TDS outperforms WSP and WSR by a factor 2.0 to 13.7 on 128 processors. TDS scales almost linearly.

Even on 128 processors, TDS only uses a small fraction of the available Myrinet bandwidth, which is about 60 MByte/s per link between user processes. The 15-puzzle requires 2.5 MByte/s, the double-blank puzzle 1.7 MByte/s, and Rubik's cube 0.38 MByte/s. Each piece of work is encoded in 32–68 bytes. For all games we combine up to 64 pieces of work into one message. The communication overhead for distributed termination detection (TDS synchronizes after each iteration) is well below 0.1% of the total communication overhead.

WSP suffers from high lookup latencies. Even with the customized network firmware, a remote lookup takes 32.5 μs. The double-blank puzzle, which does 24,000 remote lookups per second per processor, spends 78% of the time waiting for lookup replies. WSR spends most of its time handling incoming broadcast messages. For the double-blank puzzle, each processor receives and handles 11 MByte/s (680,000 updates) from all other processors. Although hard to measure exactly, each processor spends about 75–80% of the time handling broadcast messages.

	15-puzzle		double-blank puzzle		Rubik's cube	
	C_{ovh}	S_{ovh}	C_{ovh}	S_{ovh}	C_{ovh}	S_{ovh}
TDS	1.30	0.90	1.22	0.88	1.05	1.00
WSP	2.71	1.10	6.05	1.86	2.39	1.08
WSR	3.03	1.11	6.45	2.29	1.89	1.09

Table 1: Communication overheads (C_{ovh}) and search overheads (S_{ovh}) on 128 processors.

Imperfect speedups are caused by communication and search overhead. Communication overhead is due to message creation, sending, receiving, and handling. Search overhead is the number of nodes searched during parallel search divided by the number of a sequential search. Load imbalance turned out to be negligible; the processor that does the most work, does typically less than 1% more work than the processor that does the least work.

Table 1 lists the communication and search overheads for the applications, relative to a sequential run. The overheads explain the differences in speedup. For the 15-puzzle, for example, TDS has a total overhead of $1.30 \times 0.90 = 1.17$ and WSP has an overhead of $2.71 \times 1.10 = 2.98$. The difference between the overheads is $2.98/1.17 = 2.55$, which is about the same as the difference in speedups (see Figure 4(a)).

On a large number of processors, TDS usually searches fewer nodes than on a single processor, because of the larger table. This explains the search overheads smaller than 1. WSP also benefits from this behavior, but still has search overheads greater than 1. For the 15-puzzle, this can be explained by the fact that remote lookups are skipped near the leaves since otherwise communication overhead would be too large. For the double-blank puzzle, which has many transpositions, the main reason is that transpositions may be searched by multiple processors concurrently, because a table update is done *after* the search of a node completes. This phenomenon does not occur with TDS, since the table update is done *before* the node is searched.

The speedups through 64-processors for the 15-puzzle are similar to those reported by others (e.g., (Cook & Varnell 1997) reports 58.90-fold speedups). However, previous work has only looked at parallelizing the basic IDA* algorithm, usually using the 15-puzzle with Manhattan distance as the test domain. The state of the art has progressed significantly. For the 15-puzzle, the linear conflicts heuristic reduces tree size by roughly a factor of 10, and transposition tables reduce tree size by an additional factor of 2.5. These reductions result in a less well balanced search tree, increasing the difficulty of achieving good parallel performance. Still, our performance is comparable to the results in (Cook & Varnell 1997). This is a strong result, given that the search trees are *at least* 25-fold smaller (and that does not include the benefits from the last move and corner conflict heuristics).

Conclusion

Efficient parallelization of search algorithms that use transposition tables is a challenging task, due to communication overhead and search overhead. We have described a new approach, called transposition-driven scheduling (TDS), that integrates work scheduling with the transposition table. TDS makes all communication asynchronous, overlaps communication with computation, and reduces search overhead.

We performed a detailed comparison of TDS to the conventional work stealing approach on a large-scale parallel system. TDS performs significantly better, especially for large numbers of processors. On 128 processors, TDS achieves a speedup between 109 and 122, where traditional work-stealing algorithms achieve speedups between 8.7 and 62. TDS scales well to large numbers of processors, because it effectively reduces both search overhead *and* communication overhead.

TDS represents a shift in the way one views a search algorithm. The traditional view of single-agent search is that IDA* is at the heart of the implementation, and performance enhancements, such as a transposition tables, are added in afterwards. This approach makes it hard to achieve good parallel performance when one wants to compare to the best known sequential algorithm. With TDS, the transposition table becomes the heart of the algorithm, and performance improves significantly.

Acknowledgments

We thank Andreas Junghanns, Dick Grune, and the anonymous referees for their valuable comments on earlier versions of this paper.

References

Bal, H., and Allis, L. 1995. Parallel Retrograde Analysis on a Distributed System. In *Supercomputing*. San Diego.

Bhoedjang, R. A. F.; Romein, J. W.; and Bal, H. E. 1998. Optimizing Distributed Data Structures Using Application-Specific Network Interface Software. In *International Conference on Parallel Processing*, 485–492.

Bhoedjang, R. A. F.; Rühl, T.; and Bal, H. E. 1998. Efficient Multicast On Myrinet Using Link-Level Flow Control. In *International Conference on Parallel Processing*, 381–390.

Boden, N.; Cohen, D.; Felderman, R.; Kulawik, A.; Seitz, C.; Seizovic, J.; and Su, W. 1995. Myrinet: A Gigabit-per-second Local Area Network. *IEEE Micro* 15(1):29–36.

Brockington, M. 1997. *Asynchronous Parallel Game-Tree Search*. Ph.D. Dissertation, University of Alberta, Edmonton, Alberta, Canada.

Cook, D., and Varnell, R. 1997. Maximizing the Benefits of Parallel Search Using Machine Learning. In *AAAI National Conference*, 559–564.

Culler, D. E.; Schauser, K. E.; and von Eicken, T. 1993. Two Fundamental Limits on Dataflow Multiprocessing. In *Proceedings of the IFIP WG 10.3 Working Conference on Architectures and Compilation Techniques for Fine and Medium Grain Parallelism, Orlando, FL.* North-Holland.

Evett, M.; Hendler, J.; Mahanti, A.; and Nau, D. 1995. PRA*: Massively Parallel Heuristic Search. *Journal of Parallel and Distributed Computing* 25:133–143.

Feldmann, R. 1993. *Game Tree Search on Massively Parallel Systems*. Ph.D. Dissertation, University of Paderborn.

Gasser, R. 1995. *Harnessing Computational Resources for Efficient Exhaustive Search*. Ph.D. Dissertation, ETH Zürich, Switzerland.

Hansson, O.; Mayer, A.; and Yung, M. 1992. Criticizing Solutions to Relaxed Models yields Powerful Admissible Heuristics. *Information Sciences* 63(3):207–227.

Korf, R., and Taylor, L. 1996. Finding Optimal Solutions to the Twenty-Four Puzzle. In *AAAI National Conference*, 1202–1207.

Korf, R. 1985. Depth-first Iterative Deepening: an Optimal Admissible Tree Search. *Artificial Intelligence* 27(1):97–109.

Korf, R. 1997. Finding Optimal Solutions to Rubik's Cube Using Pattern Databases. In *AAAI National Conference*, 700–705.

Kumar, V., and Rao, V. 1990. Scalable Parallel Formulations of Depth-first Search. In Kumar, V.; Gopalakrishnan, P.; and Kanal, L., eds., *Parallel Algorithms for Machine Intelligence and Vision*, 1–42. Springer-Verlag.

Marsland, T., and Popowich, F. 1985. Parallel Game-Tree Search. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 7(4):442–452.

Mattern, F. 1987. Algorithms for Distributed Termination Detection. *Distributed Computing* 2:161–175.

Plaat, A.; Schaeffer, J.; Pijls, W.; and de Bruin, A. 1996. Exploiting Graph Properties of Game Trees. *AAAI National Conference* 1:234–239.

Powley, C., and Korf, R. 1991. Single-Agent Parallel Window Search. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 3(5):466–477.

Rao, V.; Kumar, V.; and Ramesh, K. 1987. A Parallel Implementation of Iterative-Deepening-A*. In *AAAI National Conference*, 178–182.

Reinefeld, A., and Marsland, T. A. 1994. Enhanced Iterative-Deepening Search. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 16(7):701–710.

Romein, J. W.; Bal, H. E.; and Grune, D. 1997. An Application Domain Specific Language for Describing Board Games. In *Parallel and Distributed Processing Techniques and Applications*, volume I, 305–314. Las Vegas, NV: CSREA.

Schaeffer, J. 1989. Distributed Game-Tree Searching. *Journal of Parallel and Distributed Computing* 6:90–114.

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

Stern, U., and Dill, D. L. 1997. Parallelizing the Murphi Verifier. In *Ninth International Conference on Computer Aided Verification*, 256–267.

Taylor, L., and Korf, R. 1993. Pruning Duplicate Nodes in Depth-First Search. In *AAAI National Conference*, 756–761.

von Eicken, T.; Culler, D. E.; Goldstein, S. C.; and Schauser, K. E. 1992. Active Messages: a Mechanism for Integrated Communication and Computation. In *Proc. of the 19th Int'l Symposium on Computer Architecture*.

Zobrist, A. 1970. A New Hashing Method with Application for Game Playing. Technical Report 88, Computer Science Department, University of Wisconsin, Madison. Reprinted in: *ICCA Journal*, 13(2):69–73, 1990.