A Performance Analysis of Transposition-Table-Driven Scheduling in Distributed Search *

John W. Romein[1], Henri E. Bal[1], Jonathan Schaeffer[2], and Aske Plaat[1]

[1] *Vrije Universiteit, Faculty of Sciences, Department of Mathematics and Computer Science, Amsterdam, The Netherlands*

[2] *University of Alberta, Department of Computing Science, Edmonton, Canada*

email: john@cs.vu.nl, bal@cs.vu.nl, jonathan@cs.ualberta.ca, aske@xs4all.nl

Abstract

This paper discusses a new work-scheduling algorithm for parallel search of single-agent state spaces, called *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 close-to-linear speedups; for large problems the speedups are even superlinear due to better memory usage. On the same machine, the algorithm is 1.6 to 12.9 times faster than traditional work-stealing-based schemes.

Keywords: *distributed search, work pushing, Transposition-Table-Driven Work Scheduling (TDS), IDA**

*The idea of Transposition-Table-Driven Scheduling and a preliminary analysis were originally presented in the Proceedings of the AAAI National Conference [34].

1 Introduction

Many applications heuristically search a state space to solve a problem. These applications range from logic programming to pattern recognition, and from theorem proving to chess playing. Achieving high performance, both in terms of solution quality and execution speed, is of great importance for many search algorithms, such as real-time search and any-time algorithms.

Often, search algorithms recursively decompose a state into successor states. If the successor states are independent of each other, 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*.

In its basic form, work stealing is a clean and simple approach. 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 [36]. 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 [27].

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, the distributed chess programs Zugzwang [13] and ⋆Socrates [17] partition the table). Before a processor expands a node, it first does a remote lookup: it sends a message to the processor that manages the entry and then waits for the reply. 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 that may contain the corresponding transposition-table entry. This processor performs the local table lookup to see whether the state has already been searched. If this is not the case, or if the state has not been searched deeply enough, the state is stored in the transposition table and in the local work queue for expansion later. The receiver thus is responsible for further expansion (search) of the state.

TDS eagerly pushes work where traditional schemes lazily steal work. Although this approach may seem counterintuitive due to the frequent migration of work, it has important advantages:

1. All communication is asynchronous (nonblocking). A processor expands a state and pushes 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 asynchronous nature of TDS allows combining multiple pieces of work into a single, large network message. This optimization reduces the communication overhead, since less time is spent in the protocol stack of the network software.

3. 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 amply available.

4. Assuming the table is large enough to cache all visited states, TDS guarantees that no redundant search effort is performed. If a state has multiple parents, the state is searched only once.

The idea of transposition-driven scheduling can apply to a variety of search algorithms. In this paper we describe the algorithm and present performance results for single-agent[1] search (IDA* [20]). 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 128 processors for several applications show that TDS is 1.6 to 12.9 times faster than work-stealing-based approaches, and thus outperforms work stealing by a large margin. Moreover, TDS scales much better to large numbers of processors. On 128 processors, TDS is 122 to 138 times faster than on a single processor, while the work stealing algorithm obtains speedups of only 10 to 79. TDS can exploit the increasing transposition table size to decrease the search effort and therefore sometimes even achieves superlinear speedups, especially for hard search problems that require large run times.

[1] Unfortunately, the term "agent" has multiple meanings. In this article, "agent" refers to the type of tree being searched, not to the processor searching the tree.

In traditional parallel 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 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) IDA* and discuss related work. Then, we describe the transposition-driven scheduling approach and discuss several of its implementation issues. Next, we evaluate the performance of the new approach, and compare TDS to traditional work-stealing based implementations of IDA*. We analyze the sensitivity to bandwidth, latency, and overhead of the network. Finally, we summarize the contributions of this work.

2 Background and related work

Although the idea of TDS is not limited to the IDA* search algorithm, we use IDA* for our experiments. Below, we will describe the IDA* search algorithm and the transposition table, and how they are traditionally implemented to run on a distributed system. People familiar with these concepts can skip the remainder of this section.

2.1 Sequential IDA*

Iterative Deepening A* (IDA*) [20] is used for searching single-agent state-spaces like those of the 15-puzzle (sliding-tile puzzle), route planners, optimizing schedulers, and Rubik's cube. The objective is to find the shortest solution path from a given problem position to

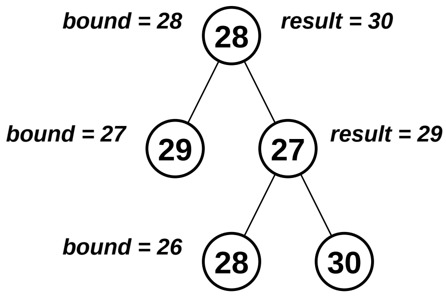


Figure 1: One IDA* iteration.

a target position (or one out of a number of target positions). IDA* is a memory-efficient variant of A* [26]. IDA* searches a *search tree*, where the nodes in the tree represent states (in practice, the terms *state*, *node*, and *position* are used interchangeably). Vertices represent possible state transitions; for example, in games, the children of a node are those positions that can be reached by a legal move, according to the rules of the game.

IDA* repeatedly descends the search tree, starting from the root position. Each iteration, the tree is searched to an increased depth, until a solution is found.

An example of an IDA* iteration is shown in Figure 1. The tree is traversed depth first, left to right. Each node is searched with a *search bound* that controls the maximum search depth. The search bound is decreased by 1; the cost to go from one state to another. Some applications (like the traveling-salesman problem) use a non-unity cost function. A node can be evaluated using an *evaluation function*. In the figure, the numbers inside the circles represent evaluation values. The evaluation function examines the position and returns a lower bound on the number of moves required to reach a target state. If the evaluation value of a node exceeds its search bound, it is not possible to reach the target state within the maximum number of moves left, and the subtree below the node is pruned. Otherwise, the node is expanded and its children are (recursively) searched.

In the example, the root is initially searched with a search bound of 28. Since its evaluation value does not exceed its search bound, the root is expanded and its first child is

```
FUNCTION IDA(Root) : INTEGER
    NewBound := Evaluate(Root);

    REPEAT
        OldBound := NewBound;
        NewBound := Search(Root, NewBound);
    UNTIL OldBound = NewBound;

    RETURN NewBound;
END

FUNCTION Search(Node, Bound) : INTEGER
    MinDist := Evaluate(Node);

    IF MinDist <= Bound AND NOT IsTarget(Node) THEN
        MinDist := INFINITY;
        Child   := FirstChild(Node);

        REPEAT
            MinDist := MIN(MinDist, Search(Child, Bound - 1) + 1);
            Child   := NextSibling(Child);
        UNTIL Child = NULL OR MinDist = Bound;
    END

    RETURN MinDist;
END
```

Figure 2: The sequential IDA* search algorithm.

searched with bound 27. Here the evaluation value (29) exceeds the search bound, and the node is pruned. The search is continued at the next child. Since the evaluation value of this node does not exceed the search bound, its children are searched too.

IDA* returns a *search result* for each node that is visited in the tree. The search result denotes the new minimum solution length, and is returned to the parent of the node. The search result of a pruned node equals its evaluation value. The search result of an expanded node is obtained by taking the minimum of its children's search results and adding 1 (accounting for the move from the parent to the child). The figure shows the search results for the expanded nodes. The search result of the root equals 30, stating that the minimal solution length is 30. The algorithm will start a new iteration with search bound 30; this tree will be deeper as the one shown in the figure. New iterations are started as long as the root's search result exceeds its search bound; this indicates that no solution was found so far. The pseudo code for the IDA* algorithm is shown in Figure 2.

The evaluation function plays an important role during the search. To guarantee that

IDA* will find a *shortest* solution, the evaluation function must not overestimate the distance to the target. Such an evaluation function is said to be *admissible*. A well-known example of an admissible evaluation function is the Manhattan distance for the sliding-tile puzzle, which sums the distances between each tile's current position and the tile's target position. To prune as much work as possible, the evaluation function should estimate the minimum solution length as accurately as possible, but must not overestimate the solution length if minimal solutions are desired.

2.2 Parallel IDA*

To decrease the search time, one can search an IDA*-tree in parallel. Numerous parallel versions of IDA* have appeared in the literature. Most algorithms use task distribution schemes that partition the search tree over the available processors [29]. 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 [23]. 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 (PWS) [28], where each processor is given a different IDA* search bound for its search. 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. Asynchronous IDA* (AIDA*) [31] uses a combination of a data partitioning scheme and work stealing, and allows processors to search to different depths concurrently.

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

IDA* uses less memory than A*. This comes at the expense of repeatedly expanding some states: a state can be expanded again in a subsequent iteration. The simple formulation of IDA* does not include the detection of duplicate states (transpositions), such as a cycle, or transposing into a state reached by a different sequence of state transitions. Treating the search space as a tree, while in fact it is a graph, leads to duplicated search of the subtree below a transposition. The transposition table is a convenient mechanism for using memory to solve these search inefficiencies, both in single-agent [30] and two-agent [36] search algorithms. There are other methods, such as finite state machines [38], but they tend to be not as generally applicable or as powerful as transposition tables.

2.3 The transposition table

A transposition table 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, the table entry contains a value that denotes a lower bound on the number of moves required to reach the target state. If the lower bound is greater than the search bound of the node, the state and the subtree below it can be pruned. If the state is not in the table, or if the lower bound in the table is not sufficient to prune the state, 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* [39]. The information in the table depends on the search algorithm. For the IDA* algorithm, the table entry contains a lower bound on the solution length. 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.

2.4 Distributed transposition tables

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. 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. Moreover, replicated tables have fewer entries than partitioned tables, as each entry is stored on each processor. These facts limit the scalability of algorithms using this technique, and replicated tables are seldomly used in practice.

A third approach is to let each processor maintain only a *local* transposition table, independent from the other processors [24]. 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 [8].

The communication overhead for the partitioned and the replicated distribution schemes is high, since each processor accesses the table tens or hundreds of thousands of times per second. 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 [35]. 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 [3, 32], which describes software for Myrinet network interface cards that is customized for transposition tables. One can also prefetch remote table entries, and make the remote lookup asynchronous [32]. This helps for many applications, but the savings are modest. Like prefetching, concurrently performing an asynchronous remote lookup and speculatively generating the node helps hiding the lookup latency [14].

2.5 Scheduling

The table distribution schemes described above are intuitive ways to implement a distributed transposition table. However, we believe that the traditional way to implement distributed search, using *work stealing*, disallows an efficient implementation of a distributed transposition table. Without a transposition table, work stealing is efficient, since work stealing itself involves little communication overhead. But if one first parallelizes the search algorithm and subsequently adds a distributed transposition table as an afterthought, it is hard to get a table entry to the place where it is needed: at the processor that processes the corresponding state.

By integrating transposition table access with work scheduling, TDS makes all communication asynchronous, allowing communication and computation to overlap. Much other research has been done on overlapping communication and computation [11]. The idea of self-scheduling work dates back to research on data flow and has been studied by several other researchers (see [10] for a discussion). 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 protocol verifier uses its hash function to schedule the work [37]. In game playing, a parallel generator of end-game databases (based on retrograde analysis) uses the Gödel numbers of states to schedule work [2]. In single-agent search, a parallel version of A*, PRA*, partitions its OPEN and CLOSED lists based on the state [12]. The parallel theorem prover Peers-mcd [7] assigns clauses to processors based on common ancestors. In this, Peers-mcd differs from the others, since it uses surrounding states to schedule the work, rather than a state itself.

Interestingly, the last four papers present the data-flow-like parallelization 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 state expansions, without addressing the benefit of asynchronous communication for *run times*.[2] (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.)

[2] 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.

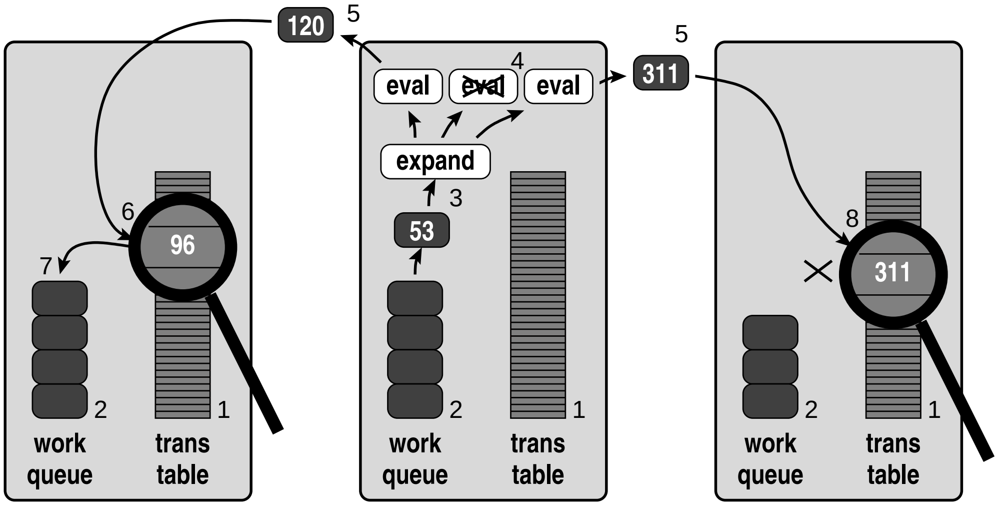


Figure 3: Transposition-Driven Scheduling for IDA*. Black numbers are referred to in the text.

Despite the good performance of data-flow-like parallelization, 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.

3 The basic algorithm

TDS is a distributed scheduling algorithm, and, like work stealing, is built on top of a search algorithm. The scheduling algorithm describes where and when states are expanded. Work stealing naturally clusters subtrees on individual processors, but TDS scatters the tree over all processors. At first sight, this seems illogical, since TDS communicates much more than work stealing does; the basic work-stealing algorithm (without transposition table) hardly communicates at all. However, distributed transposition tables are hard to implement efficiently when combined with work-stealing based scheduling algorithms. TDS avoids the problem by integrating the scheduling and the transposition table, lowering both communication and search overheads.

Figure 3 illustrates how TDS for IDA* works; the numbers in this paragraph correspond to the black numbers in the figure. Each processor stores part of the transposition table (1),

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


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

Figure 4: Simplified TDS algorithm.

and has a local work queue (2). The local work queue contains states that need to be expanded (searched). As long as there are states in the work queue, the processor takes a job, and expands it to its successor states (3). After expansion, the parent state is destroyed. Each child is evaluated, using an admissible evaluation function. States that are too far from a target (i.e., the evaluation function returns a minimum distance that is greater than the state's search bound) are pruned (4). Each of the remaining states is hashed to a transposition table entry, and pushed to the processor that owns the entry (5). Upon arrival, the state is looked up in the transposition table. If the state is not there (6), the entry is written both into the transposition table and into the local job queue (7). If the state is already in the table (8), the state is a transposition, and there is no need to search it again.

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

Figure 4 shows the pseudo code for the Transposition-Driven Scheduling algorithm, which is executed by every processor. The function *MainLoop* repeatedly tries to retrieve a state from its local work queue. If the queue is not empty, it expands the state on the head of the queue by generating the children. Then it checks for each child whether the lower bound on the solution length (obtained by *Evaluate*) exceeds the IDA* search bound, in which case it causes a cutoff. If not, the child is sent to its home processor. 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 there is still work somewhere, it waits for new work to arrive.

The function *ReceiveState* is invoked for each state that is received by a processor. The function first does a transposition table lookup to see whether the state has been searched before. If not, or if the state has been searched to an inadequate depth (e.g., by a previous iteration of IDA*), the state is stored into the transposition table and put into the local work queue; otherwise the state 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 *search result* (a lower bound on the minimal distance to the target), derived by searching the subtree below it. Finding a transposition table entry with a suitably high table value indicates that the state has been previously searched adequately. With TDS, an entry contains a *search bound*. It indicates that the subtree below the state has either been previously searched adequately (as above), or is currently being searched with the given bound, or is pending in the job queue. Note that this point represents a major improvement over previous distributed transposition table mechanisms in that it prevents two processors from ever working on the same subtree concurrently.

4 Implementation issues

We now discuss some implementation issues of this basic algorithm. Since 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 described in [25], which counts the size of the local work queues and the number of pieces of work in transit. The overhead for termination detection is negligible, because new iterations are started infrequently, and because the termination detection algorithm is active only when a work queue becomes empty.

Another issue concerns the search order. Scheduling prescribes not only on which processor a state is expanded, but also in which order. It is desirable to do the parallel search in a depth-first way as much as possible, because breadth-first search will quickly exhaust the memory for intermediate states. 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. When searching sequentially, a stack corresponds to pure depth-first search.

An interesting trade-off concerns when and where to invoke the evaluation function. One option is to do the evaluation on the processor that creates a piece of work, and to migrate the work to its home processor only if the evaluation did not cause a cutoff as in Figure 4. 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 (evaluation at the source processor) will migrate less work but will always invoke the evaluation function, even if the state has been searched before (on the home processor). However, no transposition table accesses are done for nodes that cause a cutoff after evaluation. The overhead of evaluating extra states is partly compensated by having fewer table accesses. Another effect of the latter approach (evaluation at the destination processor) is that the extra amount of table writes fills the table more quickly, increasing the chance of table conflicts and leading to increased search effort. Which approach is more efficient depends on the relative costs for migrating and evaluating states, accessing the transposition table, and on the rate at which the transposition table is filled.

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; this typically happens during the start and the end of each iteration, when there is little work.

Many applications increment the root's search bound of a new IDA* iteration by a value greater than one. Admissible evaluation functions (for example, the one used for the 15-puzzle) may return a value that underestimates the distance to the target, but always return the right parity (i.e., if the evaluation function returns an even value, the real distance is even, otherwise the real distance is odd). As a result of this, the search bound can be increased by two after each iteration that did not lead to a solution. The work-stealing IDA* algorithm, which updates the parent's search results, will discover this automatically. For TDS, we determine the root's search bound of a new IDA* iteration as follows. During an iteration we compute for each node that is pruned the difference between its evaluation value and its search bound. Each processor maintains the local minimum of the differences seen so far. If an iteration does not lead to a solution, the next iteration will be started with a search bound that is increased by the global minimum of the differences. Determining the global minimum hardly requires extra communication, since the local minima can be collected during global termination detection. In this way, TDS is, just like work-stealing, able to discover the search bound of the next iteration.

Since TDS does not backpropagate search results, it requires some effort to construct a solution path after the search has succeeded. There are several feasible ways to retrieve a solution path, neither of which require extra information in the transposition table to be stored. One option is to tag each state with the moves leading from the root to the state. Although each move can usually be represented in a few bits, it considerably enlarges the size of a state in deep, shallow search trees, and increases the communication overhead accordingly. Another option, which is the default in our implementation, is to maintain only the first few moves, and to re-search the subtree starting from the end of the partial solution path (with a cleared transposition table), until the complete solution path is retrieved. A re-search requires considerably less time than the original search, since the search tree is much smaller and the search bound of the subtree's root is known exactly; therefore no time is wasted in unsuccessful IDA* iterations.

Yet another option is to construct the solution path from the target to the root, using information that is found in the transposition table. Initially, only the target is on the partial solution path. Then, repeatedly, all possible parents of the head of the partial solution path are created and looked up in the transposition table (possibly on another machine). If the search bound (i.e., the distance to the target) of the head equals n, then there is at least one parent with a search bound $n + 1$. From the list of possible parents, we add the one with search bound $n + 1$ to the partial solution path (if there are multiple such parents, the head is a transposition and any parent will do) and repeat the process until the entire solution path is created. More effort is needed when transposition table information from the possible parents is lost, and no parent with search bound $n + 1$ can be found. If there is only one possible parent missing from the table, it must be the real parent; otherwise, we proceed with a backward search, until one of the ancestors is found in the transposition table. Such a backward search is best implemented with breadth-first search, because depth-first tends to lose its way searching for an ancestor on the solution path.

In our experience, the latter method is the most efficient method when the transposition table is sufficiently large; when the table is so small that most of the parents are already evicted from the table, one of the other solutions is preferred.

5 Discussion

Transposition-Driven Scheduling has six advantages:

1. All transposition table accesses are local.

2. All communication is asynchronous; processors do not wait for messages (except for termination detection, of which the overhead is negligible). 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 may concurrently search a transposition because the transposition-table update occurs *after* the subtree below it was searched. With TDS this cannot occur; all attempts to search a given subtree must go through the same home processor. Since TDS has a record of all completed and in-progress work in the transposition table, it will not allow redundant effort.

4. TDS uses the memory of multiple processors in an efficient way. The extra memory is used to cache more states during long searches, which decreases the likelihood that entries are evicted from the table.

5. TDS produces more stable execution times for trees with many transpositions than the work-stealing algorithm, because TDS does not randomly allocate work to processors.

6. 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, including the one we use [39], 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, 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.

An important property in our TDS implementation of IDA* is that a child state does not report its search result to its parent. As soon as a state has forked off new work for its children, work on the state itself has completed. Traditional implementations of IDA* determine a parent's search result as the minimum of the children's search results plus one. Without propagating the result back to the parent, additional search effort may be required, especially in trees where the evaluation value of a parent often differs much from those of its children. Many applications build search trees in which this scenario rarely occurs. For example, in the sliding-tile puzzle the evaluation value of a parent state is seldomly[3] off more than 1 from the minimum of the children's evaluation values. However, there are applications in which the scenario occurs frequently. For example, in Sokoban (a puzzle where a man must push barrels over a grid floor to target positions) a deadlock situation arises if a barrel is pushed into a corner [18]. When all children of a state are deadlocks, the state itself is a deadlock. To recognize such deadlocks in subsequent search iterations it is much better to include backpropagation of search results in TDS. Other search algorithms that backpropagate search results, such as Alpha-Beta search [19], also need this mechanism. However, for the applications we use, backpropagation is not necessary.

[3] With an evaluation function that only implements the Manhattan distance, the evaluation value is *never* off more than 1. Additional heuristics, such as the linear-conflict heuristic, sometimes cause greater differences.

6 Performance measurements

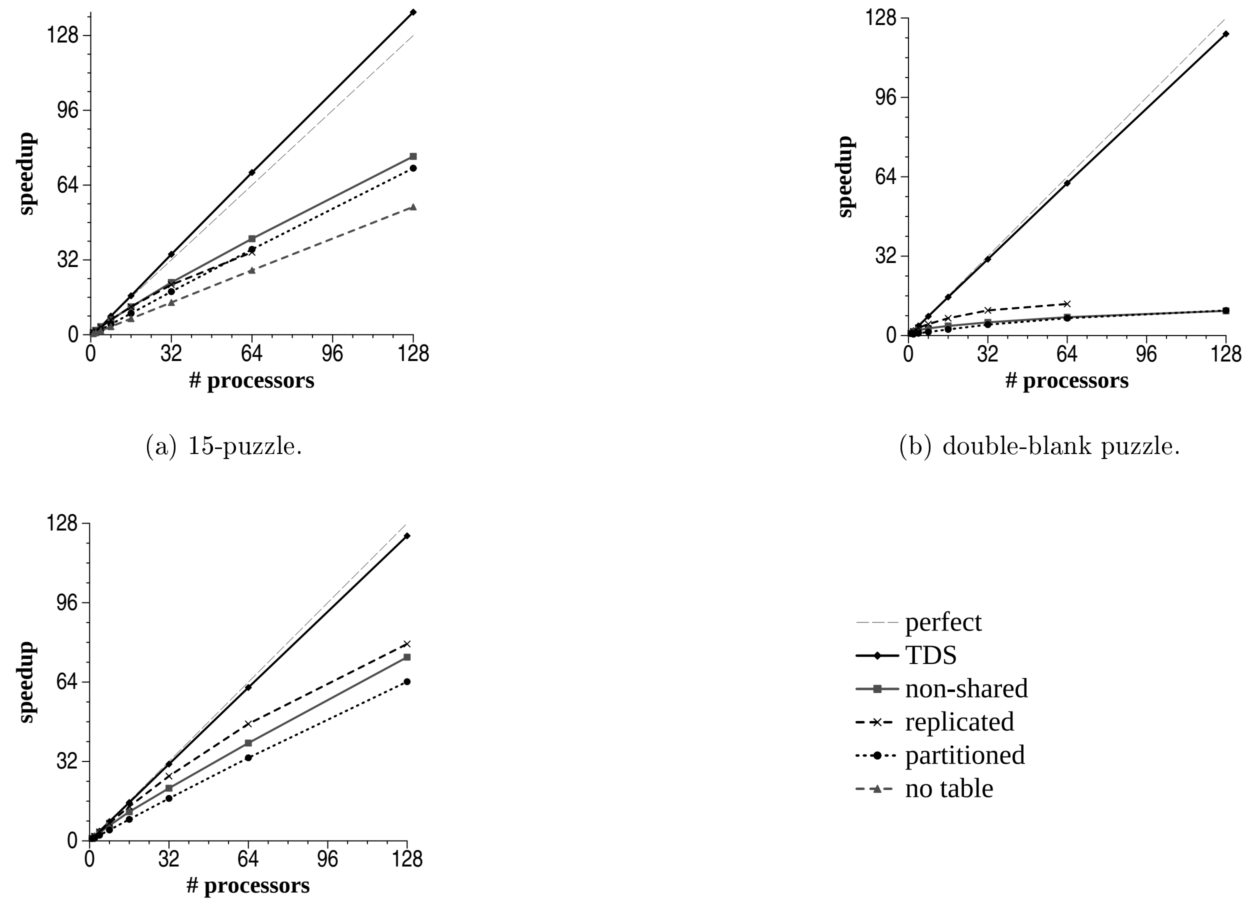
We compare the performance of TDS with that of work stealing, enhanced with *partitioned*, *replicated*, or *non-shared* transposition tables. Our test suite consists of three games: the *15-puzzle*, the *double-blank puzzle*, and *Rubik's cube*. The double-blank puzzle is a modification of the 15-puzzle, 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. All three games were implemented in Multigame [32, 33], a high-performance environment for distributed game-tree search.

The 15-puzzle uses a state-of-the-art evaluation function. It includes the Manhattan distance, linear conflict heuristic [16], last move heuristic [22], and corner conflict heuristic [22]. The double-blank puzzle uses the same evaluation function, adapted for two blanks. The Rubik's cube evaluation is done using pattern databases [21], one each for corners and edges.

Both the *replicated* and *partitioned* variants of the 15-puzzle reduce the amount of transposition-table communication by avoiding remote accesses near the leaves. *Replicated* performs an update for a node when it searched at least 64 nodes in the subtree below it. For *partitioned* such an approach to reduce lookups is not possible, because the lookup occurs before the subtree below it has been searched, and at the time of the lookup the size of the subtree is not known. We therefore use the following heuristic: a lookup for a node is done if the lookup for the parent or the lookup for the grandparent was successful. If neither lookup was successful, the node probably has not been visited by a previous iteration of IDA*, and it is likely that the node is somewhere near the leaves. Using this heuristic increases the number of visited nodes by 23%, but reduces the communication costs by 76%.

The test positions used for the 15-puzzle are nine of the hardest positions known [15].[4] To avoid long sequential searches, we stopped searching after the search iteration with a 76-

[4] Most parallel 15-puzzle programs are benchmarked on the 100 test problems in [20]. Unfortunately, using a sophisticated lower bound means that many of these test problems are solved sequentially in a few seconds. Hence, a more challenging test suite is needed.



(a) 15-puzzle.

(b) double-blank puzzle.

(c) Rubik's cube.

Figure 5: Average application speedups.

move search bound. By stopping searching before a solution is found, we circumvent another problem: the last iteration (in which a solution is found) needs an unpredictable amount of search time, since a solution can appear anywhere in the tree. All previous iterations build the same trees, which require the same search effort. By not searching the last iteration, we obtain reproducible execution times. For the double-blank puzzle, we used the same positions with the '15'-tile removed, limited to a 66-move search bound. Rubik's cube was tested using five random problems. Since a random problem requires weeks of CPU time to solve, we limited the search bound to 17 moves.

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. All machines run the

RedHat 6.2 Linux operating system. The machines are connected through Myrinet [6], a 1.2 Gigabit/second switching network. For the 15-puzzle and the double-blank puzzle, we use 2^{23} transposition table entries (64 MB) per machine. The transposition table is organized as a four-way associative cache and always stores a new result, evicting the least valuable entry in the cache line when the cache line is full. Since the 15-puzzle has relatively few transpositions, we include numbers for a variant that uses no transposition table at all. For Rubik's cube we use 2^{21} entries, to leave room in the memory for pattern databases.

The algorithms against which we compare TDS have been heavily optimized. Each Myrinet network interface board contains a programmable network processor. *Partitioned* runs customized software on the network processor to speed up remote transposition table accesses [3, 32]. Moreover, *partitioned* prefetches remote accesses whenever possible [32]. *Replicated* relies on the high broadcast bandwidth provided by the Panda communication library [4] and the LFC Myrinet control program [5], but does not run customized network software. *TDS* runs directly on top of LFC (without specialized firmware) since it does not need Panda's flow control and message fragmentation capabilities.

Figure 5 shows speedups with respect to *TDS* searching on a single processor, the fastest variant for sequential searches for all applications. On 128 processors, *TDS* is 1.6 to 12.9 times faster than the work-stealing based variants. *TDS* scales almost linearly. For the 15-puzzle, we even obtain superlinear speedups. The overhead for communication is more than compensated by the decrease of node expansions when more processors are added, because the transposition table caches more states when more memory is added. The double-blank puzzle and Rubik's cube do not achieve superlinear speedups, because the problems in their test sets did not search enough states to fill the entire table on 128 processors. The speedup for *TDS* increases for larger problem sizes. The hardest 15-puzzle problem even yielded a speedup of 154.

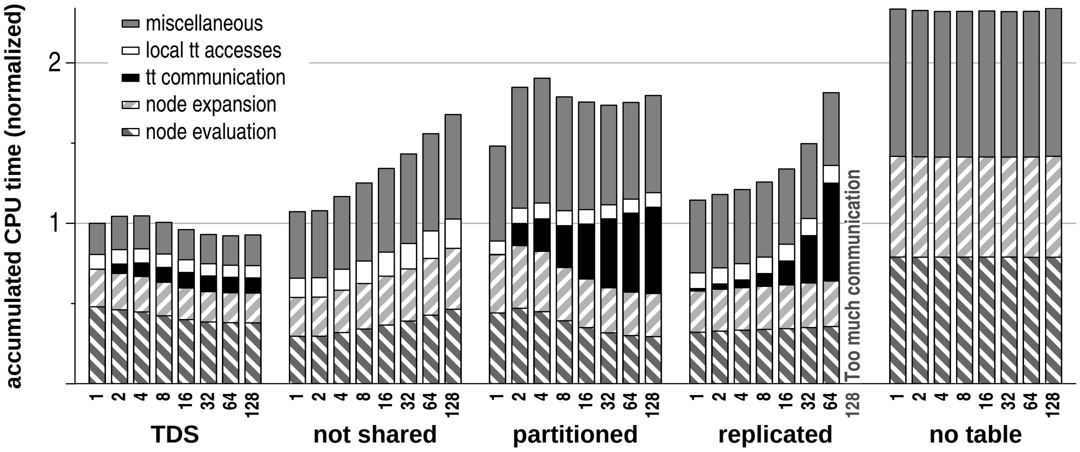
We were not able to perform measurements for the *replicated* variants of the 15-puzzle and the double-blank puzzle on 128 processors, because LFC cannot handle the communication load when all machines broadcast data too frequently.

Figure 6 shows a performance breakdown for the applications. We measured how much CPU time is spent in several program parts. We distinguish the following program parts:

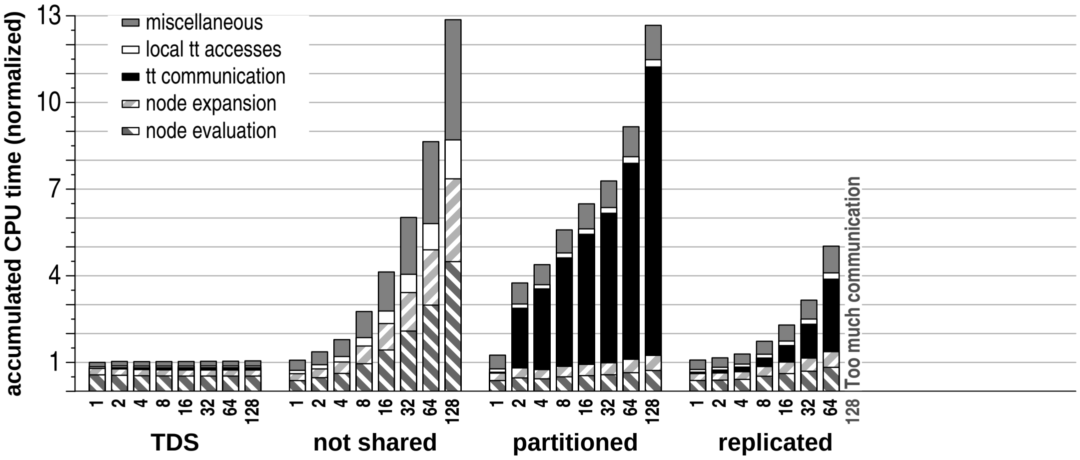
- *Node evaluation* denotes the amount of time spent in the evaluation function. For Rubik's cube, this time includes the time for doing (local) pattern database lookups.
- *Node expansion* specifies how much time is needed to generate new states.
- *Transposition table communication* is the time needed for doing remote transposition table lookups and updates, and includes both the time to issue requests and to handle incoming messages. For *TDS*, the black areas represent the time to communicate the work to other processors, rather than the time to communicate remote transposition-table entries.
- *Miscellaneous* is the time spent in the remaining program parts. These include the search engine, position repetition detection (in the work-stealing variants), node allocation and deallocation, and local job queue overhead.

We accumulate the time that all processors spend in a particular program part and average these times for several test positions (by taking the geometric mean). The height of each bar reflects how much time the processors spend in a particular program part; the total height reflects the (average) total amount of CPU time needed to solve a problem. The y-axes of the graphs are normalized to the average single-processor *TDS* run time, which is the fastest single-processor variant for all applications. Thus, if the total height of the bar equals 2 and if 128 processors are used, the application requires twice as much (accumulated) CPU time as on a single processor; consequently the speedup is 64.

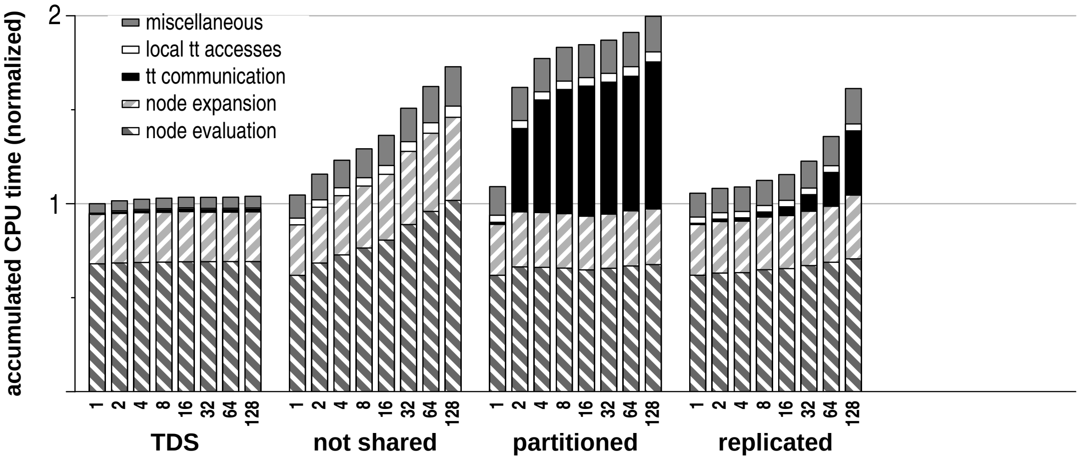
The *TDS* graphs show that the algorithm scales nearly perfectly. The graphs also show that as more processors are added, the time spent in the different program parts vary. The



(a) 15-puzzle.



(b) Double-blank puzzle.



(c) Rubik's cube.

Figure 6: Performance breakdown for the 15-puzzle, the double-blank puzzle, and Rubik's cube. The horizontal axes lists for each variant the number of processors.

number of node expansions decreases, since the increase in transposition table size reduces the number of transposition table conflicts. Since less work is generated, other program parts benefit as well.

The gray shaded areas (which represent the time spent in remaining program parts) for *TDS* are smaller than for the other variants. Due to its simplicity, the search engine of *TDS* is considerably faster than the other search engines. *TDS* does not require a separate mechanism to detect forward and backward moves or other cycles in the directed search graph. *TDS* detects repetition of positions through the transposition table, because *TDS* updates the transposition table *before* a state is searched.

Figure 6 also illustrates that *TDS* performs well on large-scale systems. The increase in transposition table size and the resulting decrease in search effort largely compensate the increase in communication overhead. Load imbalance turned out to be negligible; the busiest processor does typically less than 1% more work than the least busy processor.

TDS uses only a small fraction of the available Myrinet bandwidth, which is about 70 MByte/s per link between user processes, and about 33 MByte/s under high contention, when 64 processors send messages to random destinations as fast as they can. The 15-puzzle requires 2.3 MByte/s, the double-blank puzzle 1.9 MByte/s, and Rubik's cube 0.39 MByte/s. Each job is encoded in 32–68 bytes. For the 15-puzzle and the double-blank puzzle, we combine up to 31 pieces of work into one message, and for Rubik's cube, we combine up to 14 pieces of work. The communication overhead for distributed termination detection (*TDS* synchronizes after each iteration) is well below 0.1% of the total communication overhead. The local work queue (implemented as a stack) remains small: even for the largest 15-puzzle problem (searching 2.5 billion positions on 128 processors in 2 minutes) the stack does not exceed 1 MB in size.

Partitioned suffers from high lookup latencies. Even with the customized network firmware, a remote lookup takes 12–35 μs, including the overhead for prefetching. The double-

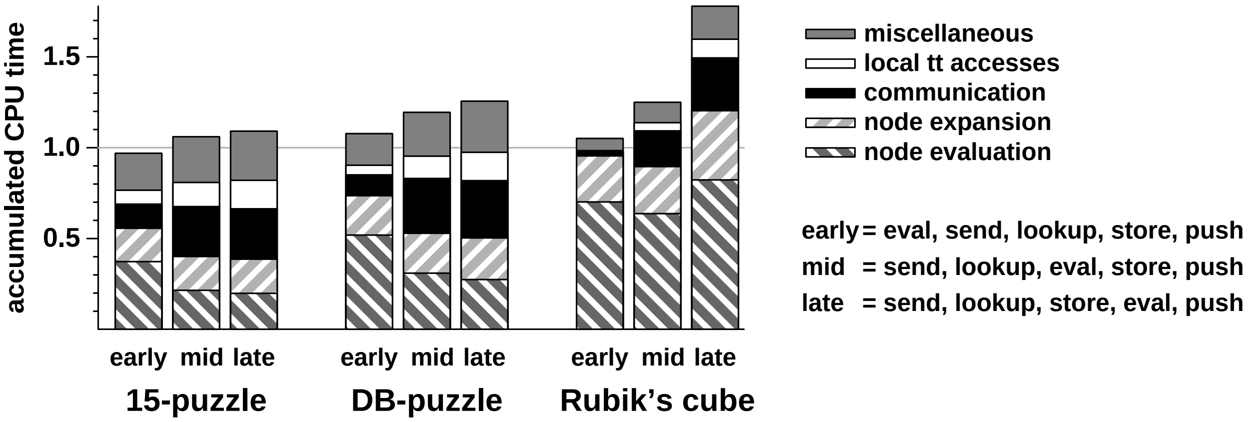


Figure 7: Evaluation on sending vs. receiving machine, on 64 processors.

blank puzzle spends 79% of the time communicating table entries, at a rate of 27,500 remote table accesses per second per processor.

Like *TDS*, *partitioned* benefits from the increase in table size when more processors are added. Yet the performance graph for the double-blank puzzle, which has many transpositions, shows that the application searches 96% more nodes on 128 processors than on a single processor. We explain this as follows. *Partitioned* (as well as *replicated* and *non-shared*) updates the transposition table *after* the search of a state completes. A transposition is not recognized as such before the update is performed, thus *partitioned* may search a transposition multiple times by multiple processors to the same depth concurrently. This phenomenon does not occur with *TDS*, where the table update is done *before* the state is searched.

Replicated passes most of its time handling incoming broadcast messages when many processors are used. On 128 processors, LFC collapses under the high communication load of the 15-puzzle and the double-blank puzzle. The other measurements on many processors show a significant communication overhead. The double-blank puzzle on 64 processors performs 14,500 transposition table stores per processor per second. These stores are buffered (64 entries per buffer, 12 bytes per entry) and broadcast to all processors. This means that each processor receives 14,500 messages per second, spending 50% of the total time communicating transposition table entries.

Section 4 argued that the evaluation function can be invoked on either the sending or the receiving processor. We studied the effects on execution times for the three different execution orders:

1. Evaluate a node on the sending processor (potentially pruning the node), send the node to its destination, lookup the node in the transposition table (cutting of the node if it is a transposition), store the node in the table, and push the node onto the work queue. This execution order is called *early*.

2. Send the node, look it up, evaluate it, store it, and push it (called *mid*).

3. Send the node, look it up, store it, evaluate it, and push it (called *late*).

The latter method stores a node in the transposition table, even if it is pruned, where method *mid* potentially evaluates a node multiple times if the node causes a cutoff. Figure 7 shows the effects on execution times for the three applications on 64 processors. Each bar is composed the same way as in Figure 6. *Early* clearly evaluates more nodes than the others, but spends less time accessing the transposition table and communicating. *Late* spends more time accessing the transposition table, since it stores information about all nodes in the table. In the case of Rubik's cube, this is even counter-productive, since the many stores of cutoff nodes thrash the table, increasing the total amount of work done. In conclusion, all three applications perform best when the evaluation function is invoked on the sending processor.

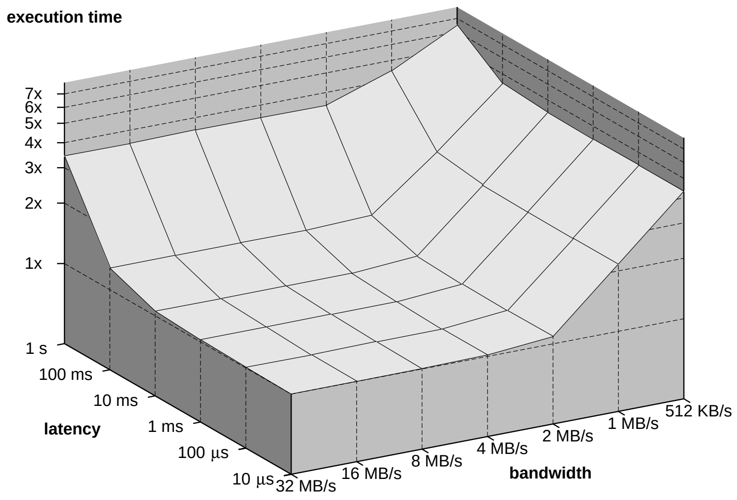
The speedups of *TDS* on 64 processors for the 15-puzzle are higher than those reported by others (e.g., [9] reports 58.90-fold speedups). Moreover, previous work has only looked at parallelizing the basic IDA* algorithm, usually using the 15-puzzle with Manhattan distance as the test domain. The state-of-the-art has progressed significantly. For the 15-puzzle, the linear conflicts heuristic [16] reduces tree size by roughly a factor of 10; transposition tables reduce tree size by an additional factor of 2.6; and the last move and corner conflict heuristics [22] reduce the tree size even more. These reductions result in a less well balanced search tree, increasing the difficulty of achieving good parallel performance. Still, our performance is at least as good as the results in [9]. This is a strong result, given that the search trees are tens of times smaller.

7 Latency, bandwidth, and overhead analysis

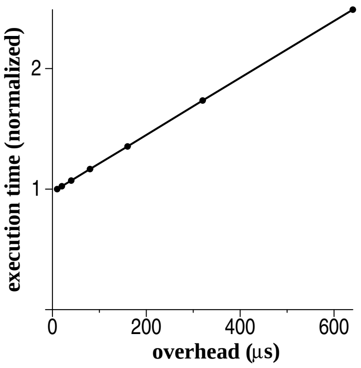
To predict the performance of TDS on other types of parallel systems, we analyzed the behavior of the 15-puzzle (the most communication-intensive among the applications) under varying latency, bandwidth, and overhead constraints, using a model that resembles the LogGP model [1]. The model characterizes the communication behavior of an application using different parameters. The message latency is the delay between the sending and the arrival of a message. The overhead is the sum of the send overhead needed to hand off a message to the communication substrate and the receive overhead needed to deliver a message to the application. The bandwidth is the number of bytes that the application can send and receive each second.

We performed the latency, bandwidth, and overhead analysis as follows. The latency and bandwidth are varied by delayed delivery to the application. Each incoming message is tagged with a delivery time and buffered until the application may consume it. During that time, the processor may receive new messages and may expand new states. The send and receive overheads are increased by having the processor spin in a tight loop until it can proceed. During that time, the processor neither receives new messages nor expands new states.

To better handle large latencies and low bandwidths we made a few modifications to the code of our basic TDS implementation. The original global termination detection algorithm [25] orders the processors in a ring and sends messages along the ring. The time



(a) Sensitivity to bandwidth and latency.



(b) Sensitivity to overhead.

Figure 8: Bandwidth, latency, and overhead sensitivity for the 15-puzzle on 64 processors.

needed for (successful) global termination detection is the number of processors times the message latency. During the first iterations of the search, when there is little work, the global termination detection dominates the search times when the latency is high. To tolerate higher latencies, we changed the global termination detection algorithm by not sending the messages in a ring, but by having one processor broadcast a termination detection request message. Each processor then replies with a unicast message. Since broadcast and unicast messages are unordered, it is necessary to include a timestamp in each message. The broadcast version requires twice the latency for global termination detection, independent of the number of processors.

Another problem occurs when the bandwidth becomes lower than what the application requires. In this situation the rate at which new jobs are received is too low to keep a processor busy. When a processor immediately flushes all outgoing message buffers with new states when it becomes idle, the buffers are almost empty; most of the time a message contains a single job only. Sending many small messages not only increases the total overhead (which is not that bad because the processors are underloaded anyway), but also increases the amount of bytes sent since more message headers are sent. A simple and effective solution is to wait a short period to give the next message a chance to arrive before the message buffers are flushed.

During the low bandwidth and high latency experiment, one of the 15-puzzle test positions buffered so many messages that flow control became necessary. We use a credit-based scheme and stop a sender when its destination has buffered 250 undelivered messages from the same source. This limits the buffer size to at most 16 MB per processor. In practice, less than 1% of the sends stall.

Figure 8(a) shows the effects of increased message latencies and decreased network bandwidth for the 15-puzzle on 64 processors. We used the same test set as in Section 6. All axes are in log-scale. The execution times are normalized; the norm is the average execution time with maximum LFC bandwidth (over 32 MByte/s under contention) and minimum LFC latency and overhead (together about 10 μs). The graph shows two interesting results. First, TDS is tolerant to high latencies: latencies of up to 10 ms are hardly noticeable and even latencies of 100 ms still give reasonable speedups. This is an expected result, since each IDA* iteration is inherently asynchronous. Second, TDS is intolerant to low bandwidths. The execution time increases inversely proportional with the bandwidth when the bandwidth drops below the required bandwidth (for the 15-puzzle, this is 2.3 MByte/s per link).

Figure 8(b) shows the sensitivity to the send and receive overhead for the 15-puzzle on 64 processors. We already learned that the application spends a relatively small amount of time communicating, despite the high bandwidth requirements. This is due to the low overhead of LFC: we measured a send overhead of 7.35 μs and a receive overhead of 1.90 μs (LFC achieves such a low receive overhead because the network processor on the Myrinet interface does most of the work to receive a message). The figure shows the application behavior for increasing overheads. For this experiment we use equal send and receive overheads; the sum is shown on the X-axis of the figure. The figure shows that the application is moderately sensitive to overhead: increasing the overhead results in a significant performance loss, but the performance does not drop as fast as in the bandwidth experiment.

In summary, TDS can tolerate latencies up to 10–100 ms, bandwidths down to a few MByte/s, and overheads up to 100 μs. 100 Mbit/s Ethernet, used via a kernel-level socket interface (either TCP or UDP), operates within these limits, provided that the network is switched and the switch can handle the aggregate bandwidth demands. We expect that the applications will run a few tens of percents slower than over Myrinet using LFC, because the overhead of the socket interface will be higher. The latency and bandwidth provided by 100 Mbit/s Ethernet will be sufficient and will not influence the run times at all.

8 Conclusions

Efficient parallelization of search algorithms that use transposition tables is a challenging task, due to communication overhead and duplicate search of subtrees. We have described a new approach, called Transposition-Driven Scheduling (TDS), which integrates work scheduling with the transposition table. TDS pushes work eagerly to the processor that caches intermediate search results. It makes all communication asynchronous, overlaps communication with computation, and reduces search overhead. TDS is applicable to any search algorithm that searches graphs, such as game-tree search algorithms, retrograde analysis, constraint satisfaction algorithms, optimization algorithms, and data-flow algorithms.

We implemented parallel IDA* using TDS, and 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 122 and 138, where traditional work-stealing algorithms achieve speedups between 10 and 79. TDS scales well to large numbers of processors, because it effectively reduces both search overhead *and* communication overhead. TDS' beneficial use of memory can even lead to superlinear speedups, especially for large search problems. We also performed a latency, bandwidth, and overhead analysis for the 15-puzzle, the most communication-intensive application in the test set. TDS is tolerant to high latencies, somewhat sensitive to high overhead, but performs poorly on low-bandwidth networks. However, modern networks like Myrinet amply provide the required bandwidth.

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 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.

References

[1] A. Alexandrov, M.F. Ionescu, K.E. Schauser, and C. Scheiman. LogGP: Incorporating Long Messages into the LogP Model — One Step Closer Towards a Realistic Model for Parallel Computation. In *ACM Symposium on Parallel Algorithms and Architectures*, pages 95–105, 1995.

[2] H.E. Bal and L.V. Allis. Parallel Retrograde Analysis on a Distributed System. In *Supercomputing '95*, San Diego, CA, December 1995.

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

[4] R.A.F. Bhoedjang, T. Rühl, R. Hofman, K. Langendoen, H.E. Bal, and M.F. Kaashoek. Panda: A Portable Platform to Support Parallel Programming Languages. *Symposium on Experiences with Distributed and Multiprocessor Systems*, pages 213–226, September 1993.

[5] R.A.F. Bhoedjang, K. Verstoep, T. Ruhl, H.E. Bal, and R.F.H. Hofman. Evaluating Design Alternatives for Reliable Communication on High-Speed Networks. In *Architectural Support for Programming Languages and Operating Systems*, Cambridge, MA, November 2000.

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

[7] M.P. Bonacina. The Clause-Diffusion Theorem Prover Peers-mcd. In *Conference on Automated Deduction (LNCS 1249)*, pages 53–56. 1997.

[8] M.G. Brockington. *Asynchronous Parallel Game-Tree Search.* PhD thesis, University of Alberta, Edmonton, Alberta, Canada, November 1997.

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

[10] D.E. Culler, K.E. Schauser, and T. von Eicken. 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, January 1993. North-Holland.

[11] T. von Eicken, D.E. Culler, S.C. Goldstein, and K.E. Schausser. Active Messages: A Mechanism for Integrated Communication and Computation. In *International Symposium on Computer Architecture*, pages 256–266, Gold Coast, Australia, May 1992.

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

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

[14] R. Feldmann, P. Mysliwietz, and B. Monien. Studying Overheads in Massively Parallel MIN/MAX-Tree Evaluation. In *ACM Symposium on Parallel Algorithms and Architectures*, pages 94–103, 1994.

[15] R. Gasser. *Harnessing Computational Resources for Efficient Exhaustive Search*. PhD thesis, ETH Zürich, Switzerland, 1995.

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

[17] C.F. Joerg and B.C. Kuszmaul. Massively Parallel Chess. In *Third DIMACS Parallel Implementation Challenge*, October 1994.

[18] A. Junghanns and J. Schaeffer. Single-Agent Search in the Presence of Deadlocks. In *AAAI National Conference*, pages 419–424, July 1998.

[19] D.E. Knuth and R.W. Moore. An Analysis of Alpha-Beta Pruning. *Artificial Intelligence*, 6(4):293–326, 1975.

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

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

[22] R.E. Korf and L.A. Taylor. Finding Optimal Solutions to the Twenty-Four Puzzle. In *AAAI National Conference*, pages 1202–1207, August 1996.

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

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

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

[26] N.J. Nilsson. *Problem-Solving Methods in Artificial Intelligence*. McGraw–Hill, New York, NY, 1971.

[27] A. Plaat, J. Schaeffer, W. Pijls, and A. de Bruin. Exploiting Graph Properties of Game Trees. In *AAAI National Conference*, pages 234–239, August 1996.

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

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

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

[31] A. Reinefeld and V. Schnecke. AIDA* — Asynchronous Parallel IDA*. In *Canadian Conference on Artificial Intelligence*, pages 295–302, Banff, Canada, 1994.

[32] J.W. Romein. *Multigame — An Environment for Distributed Game-Tree Search*. PhD thesis, Faculty of Sciences, Department of Mathematics and Computer Science, Vrije Universiteit, Amsterdam, the Netherlands, January 2001. `http://www.cs.vu.nl/~john/thesis/`.

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

[34] J.W. Romein, A. Plaat, H.E. Bal, and J. Schaeffer. Transposition Driven Work Scheduling in Distributed Search. In *AAAI National Conference*, pages 725–731, Orlando, FL, July 1999.

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

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

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

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

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