Universiteit Leiden
Opleiding Informatica

Transposition Driven Scheduling: Back to the Future?

A study on vectorization and shared memory CPU programming

Name: Florian Paolo Treurniet

Date: 07/04/2017

1st supervisor: Prof. Dr. Aske Plaat
2nd supervisor: Kristian F. D. Rietveld

MASTER'S THESIS

Leiden Institute of Advanced Computer Science (LIACS)
Leiden University
Niels Bohrweg 1
2333 CA Leiden
The Netherlands

Abstract

IDA* is an algorithm that finds minimal paths in graphs. It can be significantly sped up by adding a transposition table. Transposition Driven Scheduling (TDS) is an asynchronous parallel version of IDA*, developed to overcome the latencies of remote transposition table look-ups in a distributed memory environment. Modern general purpose processors have evolved into Non Uniform Memory Access (NUMA) machines. These architectures have high latencies for accesses to memory on a different NUMA node and for `lock`-prefixed instructions. This thesis shows that on NUMA machines TDS cannot hide the remote memory access latencies any better than other algorithms could potentially do, by using Hyper-Threading. Also no significant increase in speedup and scalability has been measured compared to a work-stealing implementation of IDA*. However, results show that it is plausible that TDS has the potential to hide the latency of locks by exchanging them for communication. Finally it is shown that TDS can be rewritten in a form, called Batched TDS, that processes a set of states at once instead of one state at a time, making it a candidate for vectorization using SIMD instructions.

Contents

Chapter 1

Introduction

Permutation puzzles, like the N-Puzzle (Section 1.1) can be solved by a parallel adaption of IDA* (Section 1.2). This thesis will focus on Transposition Driven Scheduling (TDS) [27] which is a way to run IDA* in parallel. Although TDS was designed for distributed systems, this thesis will investigate its use in shared memory systems. TDS will be further described in Section 1.4. In short the power of TDS lies in the fact that it schedules the work on the location of the data needed for the computation instead of sending work and the data it uses to an idle processor. This technique eliminates the need for costly synchronous remote look-ups. In fact, this method enables us to do the time-critical communication asynchronously, allowing the algorithm to scale super linear on distributed systems in some cases [26].

Since the original paper published in 1999 [27] the hardware has evolved far from the 200 MHz single core machines with 128 MB memory each. Machines these days run at the speed of several GHz, have multiple cores and and contain tens of gigabytes of memory which are accessible through a complex cache architecture. Sometimes they also include a computation accelerator unit like a GPU.

These changes came hand in hand with a lot of architectural changes. For instance, multi-core systems have their own on-die communication network. Systems with several sockets even have a different communication network to enable communication between processors in different sockets. Changes like these have transformed multi-core processors more and more into mini clusters. This change is represented by the fact that modern processors even have a Cluster-on-Die mode [22]. It is a transformation that will probably continue considering the increasing on-die core count.

One of the many impacts of this transformation on performance are the latencies of several operations (like memory access) relative to the amount of CPU clock cycles. The different communication networks, which are used to "emulate" a shared memory environment, have given rise to Non Uniform Memory Access (NUMA) machines, in which not all memory access times are equal. To get decent performance out of this shared memory environment, complicated coherency protocols are needed, which add a lot of latency to a data fetch from a different core, especially if it is from a different socket, much like in a distributed system where fetching data from a remote machine is a costly operation. It is therefore only reasonable to come with the following problem statement:

PS: Is there enough latency in modern multi-core systems to be hidden by TDS?

This problem statement will be answered by giving an answer to the following three research questions:

RQ1: Can parts of the TDS algorithm be vectorized, and is it worthwhile?

To get the full potential out of modern general purpose CPUs it is necessary to use SIMD instructions. The question is if TDS is a way to rewrite the IDA* algorithm to support these instructions.

RQ2: How does TDS perform when two CPUs communicate through sockets?

When CPUs are put in different sockets, communication between these sockets creates latency, for instance for memory access operations. It is a good environment for TDS to show how well it can mask latencies on NUMA architectures.

RQ3: How does TDS perform compared to work-stealing algorithms in a shared memory environment?

Work-stealing is a traditional approach to turn IDA* in a parallel algorithm in a shared memory environment. If there is enough latency on modern shared memory machines, TDS should be able to outperform this algorithm since work-stealing algorithms provide no measure against this latency.

The remainder of this chapter will go deeper into TDS, its background and the N-puzzle (Section 1.1), a benchmark problem used for our experiments. Section 1.2 discusses IDA*, a basic search algorithm to solve this puzzle. Section 1.3 discusses how the algorithm can be optimized in several ways, including how it can be run in a parallel environment. The discussed approaches have some drawbacks, and Section 1.4 discusses how TDS can solve these problems in a distributed environment, what the parallels are with the current shared memory environments, and finally it discusses a Shared Memory TDS algorithm that was used as a basis for most algorithms discussed in this thesis. This chapter ends with a discussion of the related work in Section 1.5.

The rest of the thesis is structured as follows. Chapter 2 will answer **RQ1**, showing that TDS can be used as a basis to restructure the IDA* algorithm in a way so it can support Single Instruction Multiple Data (SIMD) instructions. Chapter 3 will discuss **RQ2**. It gives an idea about the bottlenecks that appear when TDS is run on machines that have a shared memory environment based on multiple sockets. It will analyze how latencies that appear due to this environment can be hidden effectively. In Chapter 4 **RQ3** will be answered. It will show that TDS performs on-par with classical work-stealing algorithms. It will however be argued that TDS has more potential to get a better performance than the work-stealing algorithms since TDS can exchange the locks for communication. Finally in Chapter 5 it will concluded that we could not show that TDS is able to hide latency better than alternative algorithm, but that there is a potential gain in using TDS to avoid latency from locking, although this gain will be lower than the gain that can be achieved by using TDS in distributed memory systems.

1.1 N-puzzle

The problem that is used to analyze the performance of the TDS algorithm is the so called N-puzzle. The N-puzzle is a single-agent sliding puzzle consisting of N tiles and one empty position. A move consists of moving a tile to the neighboring empty position. In general, the goal is to find a path from the initial configuration to a given goal configuration. However, we are interested in the minimal path. Such a path only exists for half of the possible configurations [28].

	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

Figure 1.1: The goal position of the 15-puzzle

This research will only consider a version of the puzzle where the tiles and the empty space form a square. For instance the 15-puzzle consists of 15 tiles and one empty square, which are organized in a 4 by 4 square. An empty position is codified by a 0. In this thesis the goal configuration will always be the one shown in Figure 1.1. This goal configuration is encoded as $0, 1, \ldots, N$. Here the first $\sqrt{N+1}$ numbers lie on the first row from left to right, the second $\sqrt{N+1}$ numbers on the second row from left to right, and so forth. Only problem instances where a path exists from start to goal are used in the experiments.

1.2 IDA*

To solve the N-puzzle problem, an algorithm can be used to search the state-space for the shortest path between the two configurations. Common ways of searching graphs like these are depth-first search (DFS) or breadth-first search (BFS). For the ease of explanation, in the following discussion it is assumed that the graph is a tree with the root being the initial configuration and the edges represent moves made to transition between the different states. However, these algorithms can be easily generalized to search a general graph instead of a tree.

In DFS, when visiting a node, its children will be visited before the search backs up the tree. This forces the search to go as deep in the tree as allowed before backing up. Hence it is a depth-first search. In BFS, first the nodes of a tree on one level are searched, then the nodes on the next level, and so on. This way a shallow, but possibly wide, front is created which is about one layer of nodes deep. DFS only needs to remember the current path that is being searched. It is therefore more memory efficient than BFS which needs to remember the whole front of nodes and this front can be large. However, since it will go as deep in the tree as possible, if the solution is not deep inside the tree it might waste search time compared to BFS. Because of this property, if DFS is executed on a graph with cycles, it might not find an answer since it can get stuck in these cycles.

Both DFS and BFS are uninformed algorithms: they do not use extra information to guide the search, they traverse the state space in a predefined order. This possibly wastes time by searching parts of the state-space that can be proven to not contain the solution. One way of informing the search is by adding heuristics. A* [7] is an algorithm that incorporates heuristics.

The A* algorithm maintains a so-called open list of nodes that it has encountered, but which are not yet expanded. The nodes in the open list get a cost. This cost can be described as $f(x) = g(x) + h(x)$, where $g(x)$ is the known minimal cost of the path to that node and $h(x)$ is a heuristic value. The heuristic value gives an indication of the minimal length of the path that needs to be traversed from that node to the solution. Since the minimal path is sought, each time a new node needs to be processed, the node with the lowest $f(x)$ is processed first. It is added

to the closed list and its neighbors are evaluated (we say that the node is expanded). During the evaluation of the neighbors, each neighbor gets an $f(x)$ and is added to the open list if it is not already in the closed list. The search starts with putting the root node in the open list and continues until the solution is found on the path.

The A* algorithm is proven to find the optimal path when the heuristic is both admissible and monotone. A heuristic is admissible if it never overestimates the length of the minimal path from that node. A heuristic is monotonic if the cost never increases on a path, further down the path: $h(x) \leq h(y) + d(x, y)$ where $d(x, y)$ length of the edge between x and y.

A problem of A* is that the open list can contain a lot of nodes, sometimes more than can fit into memory. This problem can be solved using a technique called Iterative Deepening A* (IDA*) which uses less memory but takes more time [12]. The algorithm is described by Algorithm 1, in which the `search_with_bound()` method can be implemented either iterative (Algorithm 2 or recursive (Algorithm 3). In these algorithms the `expand_and_evaluate(state)` method expands the `state` and returns the children of this `state`, evaluated so they contain the cost, $f(x)$, of this child (`child.bound`). IDA* traverses the tree in a DFS fashion, but instead of going deeper until a leaf of the tree is reached, the algorithm backtracks when a node is found with a cost that exceeds a given threshold value (`bound`). When all the nodes within a given threshold value have been searched, and the solution is not found, the threshold is set to the lowest node cost found that exceeded the threshold value (`lowest_higher_bound`), and the search restarts from the root. In this fashion, in each iteration more nodes are included in the search. When the search finally finds a solution, it is only guaranteed to be optimal if the heuristic is admissible.

Algorithm 1 IDA* `ida_star(begin_state)`. Returns **true** if the solution is found, **false** if not

```
1: bound ← begin_state.evaluate()
2: while true and bound <= max_bound do
3:   found, lowest_higher_bound ← search_with_bound(begin_state, bound)
4:   if found then
5:     return true
6:   else
7:     bound ← lowest_higher_bound
8:   end if
9: end while
10: return false
```

Algorithm 2 IDA* `search_with_bound(state, bound)` iterative version. Returns `found` which indicates if the solution was found that iteration, and the lowest bound encountered during search (`lowest_higher_bound`) that was higher than `bound`.

```
1: workq.add(state)
2: lowest_higher_bound ← ∞
3: while true do
4:   if workq.empty() then
5:     return false, lowest_higher_bound
6:   end if
7:   state ← workq.get()
8:   if state.is_goal_state() then
9:     return true, lowest_higher_bound
10:  end if
11:  children ← expand_and_evaluate(state)
12:  for all child ∈ children do
13:    if child.bound < bound then
14:      workq.add(state)
15:    else
16:      if child.bound < lowest_higher_bound then
17:        lowest_higher_bound ← child.bound
18:      end if
19:    end if
20:  end for
21: end while
```

Algorithm 3 IDA* `search_with_bound(state, bound)` recursive version. Returns `found` which indicates if the solution was found that iteration, and the lowest bound encountered during search (`lowest_higher_bound`) that was higher than `bound`.

```
1: if state..is_goal_state() then
2:   return true, lowest_higher_bound
3: end if
4: children ← expand_and_evaluate(state)
5: for all child ∈ children do
6:   if child.bound < bound then
7:     found, child_lowest_bound ← search_with_bound(child, bound)
8:     if child_lowest_bound < lowest_higher_bound then
9:       lowest_higher_bound ← child_lowest_bound
10:    end if
11:    if found then
12:      return true, lowest_higher_bound
13:    end if
14:  else if child.bound < lowest_higher_bound then
15:    lowest_higher_bound ← child.bound
16:  end if
17: end for
18: return false, lowest_higher_bound
```

1.3 Optimizing IDA*

If the result of a search is optimal, one of the things that still can be improved upon is that a search is completed in as little time possible. This can be achieved by increasing the effectiveness of the heuristics. However, heuristics are usually very problem specific. Several other, more general modifications to the IDA* algorithm can be made. One of these solutions is called a *transposition table*. Section 1.2 describes IDA* when it is used on a tree. However, in reality the state space will be a graph with cycles, so there are multiple paths to the same state. States like these, which are identical except for the path towards the state are called *transpositions*.

During the search, when a transposition of an already encountered state is found, it is useless to continue the search if the path to the transposition is longer than the path to the already encountered state, for it is already known that the remaining path from that will be identical in both cases, resulting in a sub-optimal path. To eliminate this search overhead, the algorithm needs to remember which transpositions it has already encountered.

A transposition table is a hash table for states. It contains the hash of the state and some meta data like the length of the path to that state and the iteration in which the entry was created. When a node is encountered, it is matched against the transposition table. If a transposition is present in the table and the path of the current node is longer or equal than the path of the transposition in the table, the search does not continue from that node, effectively cutting off a part of the search space, reducing the total search time.

Since the amount of memory is limited, the transposition table usually has a limited size. As a result, a finite amount of transpositions can be stored in the table. If the searched graph is bigger than the amount of states that fit in the table, the pruning will not be perfect. Furthermore, it should be decided which states should be kept in the table. For this a replacement policy needs to be chosen.

The discussion so far was focused on running the algorithm on one processor. A further reduction in computing time can be achieved by performing the search on multiple processors. However, there are two decisions to be made when doing this: how to divide the work among the processors (Section 1.3.1) and how to keep the transposition table up to date across the processors (Section 1.3.2).

1.3.1 Work division

The first decision is which work can be processed in parallel. One option is to run different iterations of IDA* on different cores at the same time. However, the higher the iteration number, the larger the amount of nodes included in the search. This results in a load balance issue where some cores finish their part of the work earlier than others, making inefficient use of the available hardware. Besides, one does not know which iteration contains the solution. This effectively wastes the time of the processors running iterations higher than the iteration in which the solution is found, thus limiting the scalability of the algorithm. To illustrate this problem: a thousand processors take the same time as 4 processors solving the problem, if the solution is present in iteration 4.

Another option is to compute different parts of the search space in parallel. First the tree is expanded until there are as much sub-trees as there are processors. Then each processor gets its own sub-tree to work on. The problem of this solution is that the size of the sub-trees might differ a lot, creating a load imbalance like the first solution does.

Instead of looking at a different method to divide the work, it is also possible to think of a possible solution for this load balancing problem. One way is to apply work-stealing. When a processor runs out of work, it asks a random processor for work. If there is work available, it is

sent to the requesting processor. However, this solution is not ideal as the requesting processor needs to wait for the answer. During that time it has no useful work to do, reducing processor efficiency. This penalty is especially present in distributed memory systems where a messages takes an order of a 1000 CPU cycles to be transferred to the sender. As will be described in Section 1.4, this latency is one of the areas on which TDS improves.

1.3.2 Keeping the transposition table up to date

To make effective use of the transposition table during parallel search it needs to be as up-to-date as possible across all processors. To see why a local transposition table that only keeps track of the nodes encountered on that processor is not that effective, consider the following example. Processor P_a can process a state x, but when processor P_b comes across the transposition of x, called $T(x)$, it will still process $T(x)$ since P_b has no entry of x in its transposition table. As a result time is wasted on a duplicate search effort. This implies that the transposition table is most effective when it is organized in a way there is one global truth.

To get a better insight in the cost of this requirement, the following discussion will be based on the case where machines contain a single processor and therefore have a distributed memory when multiple processors work together. This way machines have to communicate over a network to share information, making the communication relatively costly. In Section 1.4 it will be argued why this is a similar situation to that of the modern multi-socket shared memory machines.

One straightforward solution is to duplicate the transposition table across all processors. When a processor updates its table, it will notify all other processors that it did, so they can update their transposition tables. This solution suffers from two problems. The first one being that each transposition table update creates $N - 1$ messages, N being the amount of processors performing the search. This amount of messages will in general clog up the network resources. The second problem is that there is a minor delay between the update on the local machine and the update on the other processors. During this delay the global truth differs from the local truth, and it might again not be known that x has already been processed, so $T(x)$ will still be processed.

To battle the inconsistency due to the delay between the local update and the global update, one could partition the transposition table across the processors. This means that every state can only be present in the transposition table of one processor: we say that this processor owns that state. When a state is encountered that is not owned by the local processor, a message is sent to the processor that does own it. The processor owning the state checks its table and sends a message back to the requesting processor if the state is present or not so it can make a decision on whether or not to continue processing the state. Although this solution has an additional benefit, namely that unlike the duplication scheme one can use the aggregated memory size of all the processors for the transposition table, still a large number of messages are needed to communicate the state of the table. Besides that, the processor can only continue processing the state when an answer has been received from the owning processor, possibly wasting a lot of computation time.

On a shared memory machine, it is possible to share one table in memory with all the processors connected to this shared memory environment. On one hand this would reduce the overhead of having to communicate transposition table changes with other processors. Unfortunately, since all processors can freely access all the memory, race conditions can appear when several processes access the same memory locations in the table. To eliminate this problem processors must have exclusive access to the parts of the program where they access these memory locations: the critical sections. This can be done using synchronization primitives like mutexes, or by using lock-free programming with atomic operations. Either way, memory addresses are effectively blocked off

from other processors, which now have to wait until the memory is free to use again, during which they run idle.

The costs of getting a lock are not low either since they are usually implemented by atomic operations. These operations require the processor to lock a specific memory address, which needs to be communicated with other cores. The cores need to confirm that they did this before the transaction can begin. This is especially costly in multi-socket systems, because sending a message between sockets is relatively slow. Using atomic operations therefore should be avoided.

1.4 TDS

Transposition Driven Scheduling (TDS) [27] is a way of running IDA* in a parallel way, designed for a distributed memory environment. It solves several problems, including those mentioned in Section 1.3.1 and Section 1.3.2. Instead of using the state space as a starting point to select which parts of the algorithm can run in parallel, it uses the transposition table. The main idea is that, like a partitioned transposition table, every state is owned by one processor. However, the owning processor will also always be responsible for expanding the states that are encountered during computation.

In TDS, each processor gets a work-queue. The work-queue is implemented using a stack to get as close to a DFS as possible, which is beneficial for the amount of nodes that are expanded during the search. During the search, a processor is popping states from its stack. Each state is matched against the transposition table, and if no transposition is encountered, it is expanded, generating its children. When the children are generated, a hash is calculated for each of them. Using this hash the owning processor is calculated. If the local processor is not the owner of the child, it is sent to the rightful owner. The receiving processor receives the state, puts it on its work stack, and continues until either the iteration ends or the solution is found. If the iteration ends, a termination algorithm makes sure no messages are in transport. If all sent messages are received, the bound of the search is increased, and a new iteration is started by putting the initial state on the stack.

TDS has several benefits. First of all, due to the way the transposition table is partitioned, a look-up is always local on the processor expanding the state. This saves time communicating with external processors, and maybe waiting for a reply, to obtain the status of the table. Besides that it keeps all the benefits from the partitioned transposition table scheme like being able to make a table with the size of the aggregated machine memory size.

Secondly, a benefit is that all the communication can be overlapped with computation when done asynchronously. This is especially true when using IDA* because no information flows up the tree, so a state only needs to be sent to another processor, and it never has to wait for a reply.

The question is this information compares to a shared memory environment. Since the original paper [27] relative timings for CPU operations have changed quite a bit. To give an indication, a memory access for a Pentium Pro would cost a minimum of 14 CPU cycles [18]. Compare that to the more than 300 cycles for a remote L3 miss on a Haswell multi-socket machine [22]. The latency that the original paper tried to cover up was the Myrinet network. With a latency of 32.5 microseconds [27] that a look-up would take, a Pentium Pro at 200MHz would be able to run about 6500 cycles. The question is if this kind of latencies are present on shared memory machines. Because if this is not the case, using TDS in a shared memory environment would not add a lot of performance compared to using a straightforward work-stealing algorithm, if it would add any performance at all, since it was designed to cover up these latencies.

Although remote L3 cache misses take an order of 10 cycles less than a Myrinet look-up,

note that more than one memory access must be done to do a look-up in a transposition table. It still would pose for a serious latency bottleneck if this latency would be encountered many times, especially if it could not be hidden by the reorder buffer of the processor. Another potential candidate for the latency in modern systems would appear in the form of `lock`-prefixed instructions, which need to block access to a piece of memory on all other processors connected to the memory. This requires quite a bit of communication between these processors, at least causing a remote L3 miss two times if two processors on different sockets require to access the same variable since the cache-line needs to transfer between them. Both observations lead to two conclusions: data should be accessed local, and locks should be avoided. Two things that can be achieved using TDS.

1.4.1 Shared Memory TDS

For the experiments done in this thesis, an implementation of TDS, further named Shared Memory TDS, has been made that should closely resemble the algorithm described in the original paper [27], although there are some differences to adapt it for a shared memory environment. The following section discusses the more relevant design choices that were made.

Several heuristics have been used: the Manhattan distance, the linear conflict heuristic, the corner heuristic and the last moves heuristic. The Manhattan distance is calculated by counting how many places every tile would have to move from the current position to the goal position if the board would have been empty. The distance the empty position should travel is excluded from the calculation. The Manhattan distance is a very rough admissible heuristic since it ignores the fact that there are other tiles on the board, so it will never overestimate the length of the remaining path. The other heuristics are extensively described in [13].

The hashes of the states are calculated using the Zobrist hash function [31]. It works by giving every tile-position combination a unique random number. For instance tile a in the first position will get random number $r_{a,1}$, the same tile on the second position will get random number $r_{a,2}$ and so on. By calculating an exclusive or of these numbers for all tile-position combinations, a unique random number is generated that describes the state. If the board consists of 3 spaces, where tile a is on position 1, tile b is on position 3 and position 2 is empty, the Zobrist hash is calculated like $r_{a,1} \oplus r_{b,3}$. Note how the associative property of the `xor` function and the fact that $X \oplus X = \vec{0}$ allow a tile slide to be calculated with only two operations: `xor` the moved tile on the old position out, and then `xor` the moved tile in on the new position, which is an optimization also used in our implementation.

Since each thread has its own transposition table, no locks are needed to ensure data consistency. The transposition table is implemented much like the Stadium Hash Table [8]. In this design the transposition table consists of two data structures: a ticket board and the actual table. The ticket board is used to store 7 bits of the hash of the state that is stored in the actual table and 1 bit to store if the entry is in use. For this research, a 4-way set associative table is implemented, meaning that 32 bits are used for one entry. This more compact representation easily fits on a cache line, while a row of the actual data in the table usually spans 2 cache lines. Furthermore the ticket board saves us the trouble of the extra look-up if the entry is not in use or if the 7 bits of the hash do not match. If all places to store states are used, a random one is evicted.

For the communication between threads, each thread has a receive buffer with a fixed number of places for states. If a thread A sends a state to thread B, it reserves a space in the receive buffer of thread A by executing an `atomicAdd()` operation on a reservation counter that is owned by A's buffer. To ensure that the data is only read by A after everything was written by B, a `done_writing` flag is set afterwards by B. In a similar way A keeps a counter to track at which

point in the buffer the next state resides that it will read. Unlike in the original TDS algorithm, no aggregation buffer was implemented.

At the end of an iteration, it is essential to know that it has actually ended. Otherwise it is not known if the search is actually continuing. It then might be the case that the result is found in a still running previous iteration or that a different bound is found at which the next iteration should start in some states that are still processed from the previous iteration. To know when the algorithm has actually ended is hard to know, since part of the work is traveling between processors. The only way to know that it has really ended is to check if there are still messages in transit. This can be detected by a termination detection algorithm. Our implementation uses the Vector Count method [20].

When using the Vector Count method, each processor P_i counts the amount of messages it sends to processor P_j, by increasing the number T_j in the termination vector $\vec{T}$. When processor P_i receives a message from P_j, it decreases the number T_j. If it is suspected that the algorithm is terminated, all vectors are added up, and if the result is $\vec{0}$, no messages are 'flying' since all sent messages have been received.

To achieve this effect without forcing a global synchronization once in a while, only one thread at a time owns the added-up vector and may send the vector to the next in the communication circle. The sending thread then resets its local vector to $\vec{0}$. The receiving process adds the received vector to its own vector. It can be proven that if the vector message has been forwarded once by all threads and the receiver gets $\vec{T} = \vec{0}$ after addition, there are no messages in transit anymore. In our implementation the message is then forwarded to each thread a second time to notify termination.

1.5 Related Work

The performance of TDS on a single core cluster for three single-agent games (15-puzzle, the double blank puzzle and the Rubiks cube) has been extensively analyzed by Romein et al. [26]. It underlines the original result that TDS outperforms classic work-stealing algorithms. More importantly it relates the super linear speedups that might occur to the fact that during sequential runs the transposition table is too small to store all the encountered transpositions. The paper also concludes that high latency is not a bottleneck for TDS. However, overhead of the communication layer degrades performance, albeit not as much as low bandwidth networks. An implementation of TDS that calculates a 3D state space of a virtual environment [14] for use in haptic interaction confirms it. They seem to get degraded performance due to overhead in the MPI library using another MPI implementation called MPICH. This raises questions how good the communication layer needs to be to achieve reasonable speedups.

Even in situations where low bandwidth is inevitable TDS can still function with some modifications. Multiple clusters of computers can be linked together by a wide-area network which has relatively low bandwidth compared to the network within a computing cluster. To overcome this problem one can use Wide-Area TDS (WA-TDS) [25]. It runs native TDS within a cluster. To minimize communication over the wide area network, each cluster works on different sub-trees. This creates a load-imbalance between clusters, which can be resolved by applying work-stealing between clusters over the wide area network. WA-TDS achieves good speedups compared to classic work-stealing algorithms.

TDS-like algorithms, that use the idea of using a hash function of the states to distribute the calculation, are also used in the area of Automated Planning in Artificial Intelligence. Vidal et al. present an algorithm based on a complete weighted A* algorithm [29]. Contrary to TDS and HDA* [9] the goal is to perform sub optimal planning and in implements a look-ahead

strategy. The algorithm is evaluated in different parallel configurations on both a normal cluster and on Intel its Single-chip Cloud Computer (SCC), an exotic architecture that embeds 48 cores on a single chip. This research gives several indications. Firstly there are indications that MPI gives significant overhead compared to hybrid (Open MP/MPI) communication schemes on clusters. Secondly TDS-like algorithms might benefit from the SCC architecture when the messages exchanged reach I/O capacities. Lastly, SCC might increase algorithm performance further than regular clusters since the memory contention seems to be less present on the SCC when comparing both.

Several successful attempts have been done bringing TDS to the area of 2-player search. Two-player search is harder than running IDA* in parallel since algorithms require information to go up the tree, besides that search results can result in a cut-off on nodes already distributed over the cores. TDSAB [10] is a parallel version of MTD(f) [23], which is a variation on $\alpha\beta$ search that performs the search using a minimal window. Results are reported regarding two games called Awari and Amazons that prove that TDSAB performs comparable or better compared to other algorithms at that time. The main problems in creating even better speedups seem to be the large synchronization overhead and that in problems with a large branching factor, the amount of parallelism can be too high, causing a lot of overhead when a cut-off is generated.

UCT-Treesplit [6] is another example of a 2-player algorithm run in parallel using a method closely resembling TDS. It is a parallel version of Upper Confidence Bound Applied to trees (UCT) [11] a Monte-Carlo Tree Search algorithm (MCTS) [2]. Since UCT is a best-first search, it is an interesting algorithm to run in parallel since simulations are dependent on each other. Therefore it can be that the search quality increases less when more simulations are run in parallel.

Interesting is the way they work with the multi-core architectures within clusters. One core per machine has the task to communicate with other machines and to distribute work to the other cores, the worker cores. The worker cores on a machine have a shared transposition table which is beneficial for synchronization steps like updates. It allows one worker to access more transposition table entries without extra synchronization allowing less of these messages to be sent over the inter-machine network. However, this algorithm needs locks on the transposition table. The research further shows that having a variable size work load can degrade performance, since the communication time might get too big for the work size, resulting in idle time for the processor.

A different parallel UCT version is called Depth First-UCT (DF-UCT) [30]. It remains closer to the idea of TDS than UCT-Treesplit since less modifications on it are needed. It introduces report messages next to the already existing search messages. Recognizing that a lot of the search time comes from the updates, df-UCT reformulates the UCT algorithm in such a way that the next node to expand can be selected near the location in the tree of the last play out. This way, report messages do not have to travel all the way up to the root in most cases. This saves time sending and waiting on report messages so simulations can be started quicker after one another. Good speedups are obtained, although the results might not hold for real-life problems, since results were reported on artificial game trees.

Algorithm research is now primarily focused on its performance given a fixed amount of virtually limited resources. A paper on Iterative Allocation (IA) [4] acknowledges that resources cost money, and over-allocation is not cost effective. It shows that TDS-like algorithms like HDA*, that scale well, can use IA to be run in a cost effective way on cloud computing services.

Chapter 2

Vectorizing TDS

2.1 Introduction

Vector processing is getting increasingly important to get the full potential out of a modern processor. Not only GPUs make use of vector instructions, but also general-purpose processors have extensions like "Intel SSE" [17]. These SIMD instructions make it possible to process several elements at once, which can increase the throughput of the processor.

This chapter will present a preliminary research on TDS that will investigate if it is possible to put the TDS algorithm in a form in which a vector, or block, of states will be processed instead of single elements like in the original algorithm. To this means an implementation of Batched TDS will be described in Section 2.1.1. Several experiments (Section 2.2) have been conducted which results will be presented in Section 2.3. Conclusions about its effectiveness have been made in Section 2.4.

2.1.1 Batched TDS

The main difference between Shared Memory TDS (Section 1.4.1) and Batched TDS is that instead of processing separate states, a batch of states will be processed. Aside from the local data structures each thread in Shared Memory TDS has, each thread has its own dedicated `processing_block` and `child_block`. The `processing_block` stores the states that need to be processed. The `child_block` stores the states resulting from the expansion of the states stored in the `processing_block`.

The algorithm, described by pseudocode in Algorithm 4, further works as follows. Within an iteration, each time the work-queue (`workq`) will be popped onto the `processing_block` until either the work-queue is empty, or the `processing_block` is full. During the popping of this block, the transposition table (`tt`) will be checked. If a better or equal state resides in the table, the state will be discarded instead of putting it on the `processing_block`. Next all the states of the `processing_block` will be expanded, and the expanded states will be evaluated and discarded if they are beyond the current search-bound (`expand_and_evaluate()`). The states that were not discarded are added to the `child_block`. Finally all the states in the `child_block` will be sent to their owner (`send_to_owner()`), just like in Shared Memory TDS. These steps are repeated until all threads run out of work for this iteration of IDA*, or if a solution is found on one thread.

Algorithm 4 Batched TDS `search_with_bound(bound)`

```
1: while not terminated() do
2:   receive states from the communicator onto the workq
3:   while not workq.empty() or not processing_block.full() do
4:     state ← workq.get()
5:     if tt.no_better_or_equal(state) then
6:       processing_block.add(state)
7:     end if
8:   end while
9:   for all state ∈ processing_block do
10:    expand_and_evaluate(state, child_block, bound)
11:  end for
12:  for all child ∈ processing_block do
13:    send_to_owner(child)
14:  end for
15:  processing_block.clear()
16:  child_block.clear()
17: end while
```

It must be noted that in this implementation of Batched TDS no SIMD instructions were used. It only uses the benefit of the less strict requirements on the order in which states are expanded, which is inherent to TDS, to rewrite the algorithm in a form that could be used to add SIMD instructions to TDS to speed up the algorithm.

2.2 Experiments

To evaluate the performance of Batched TDS, it will be compared with Shared Memory TDS on the areas of running time and cache performance. To measure the running time, the speedup of both algorithms will be measured for different logical core numbers. The baseline to which all reported speedups in this thesis ar e compared is the sequential iterative version of IDA* that uses a transposition table with 2^{27} hash targets which can each contain 4 transpositions. For the reported test-case it has an average running time, measured over 10 runs, of 1143 seconds. The sequential algorithm was used as a baseline since a lot of algorithms are compared in this thesis, and in the end the interesting measure is how much faster you can get on one machine than the sequential algorithm, given a maximum memory use (here represented by the transposition table size). The running time is the time measured between the start of the first iteration of IDA* and the ending of the last iteration. This means that this value excludes the setup and cleanup of the data structures inside the memory.

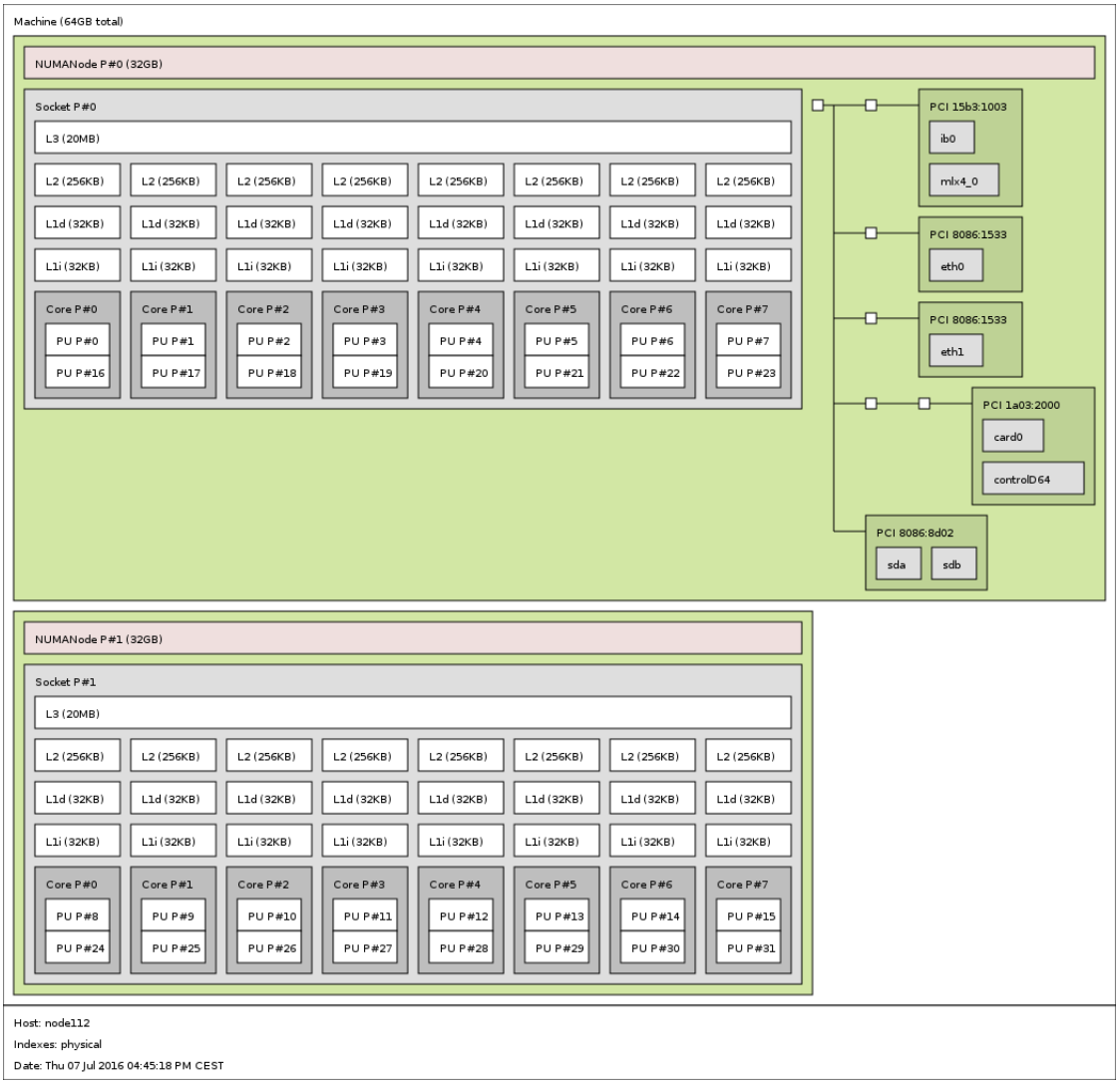

Figure 2.1: A hardware topology overview of a Xeon E5-2630 v3.

The testbed for all experiments in this paper is a dual socket system containing two Xeon E5-2630 v3 processors, each with 8 physical cores, or 16 cores by Hyper-Threading. The system is graphically depicted in Figure 2.1. One can see that it is a NUMA architecture, where each NUMA node shares half of the available memory, and the L3 cache. Logical cores of one physical processor share the L1 and L2 cache.

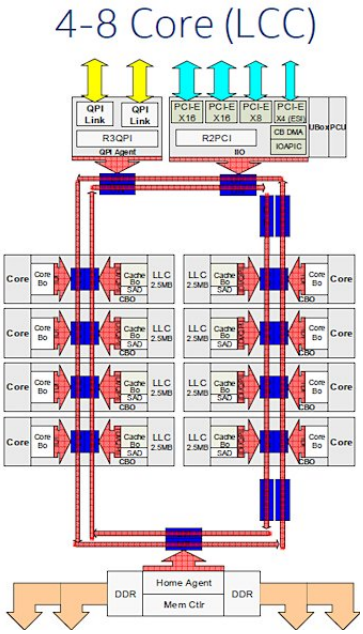

Figure 2.2: An overview of the so-called uncore topology of a Haswell-EP processor with 8 cores, just like the Xeon E5-2630 v3. Source: Intel.

Figure 2.2 shows that the cores of a processor are connected by a ring network. The L3 cache is splitted in as much slices as there are processors and is only accessible through the ring network. Memory is hashed into these slices, emulating one big shared L3 cache. Furthermore, this ring network is connected to a Quick Path Interconnect (QPI) transfer bus, which connects the cores to the inter-socket QPI network. All communication, including coherency protocol information travels over this network.

To get an idea of how important it is to consider the NUMA architecture of the machine while allocating memory and threads, two different sets of experiments will be executed to measure the running time. In the first set of experiments, both the threads and data structures will be allocated at NUMA nodes chosen by the operating system. In the second set of experiments, both the data and the processes will be allocated on the first NUMA node (NUMA node 1).

To measure the cache performance, the amount of L1 and Last Level Cache (LLC), in this case the L3 cache, misses will be measured for different logical core numbers. These units correspond to the default `perf` [3] utility counters for these values. In contrary to the running time, these values are measured over the complete running time of the applications, so including the allocation and deallocation of memory resources.

Both algorithms are configured to use the full set of heuristics (corner heuristic, last move heuristic, linear conflict heuristic and the Manhattan distance). As a benchmark, one of the 9 biggest 15-puzzle problems [5] will be used. It is encoded as "`15,14,8,12,10,11,9,13,2,6,5,1,3,7,4,0`". To make the results of these experiments more consistent, the last iteration of the search will not be performed. Since there is some randomness involved at which point in the final iteration the goal state will be found, cutting the last iteration out will result in more or less equal amount of nodes being expanded and processed. The experiments in this chapter have all been averaged over 10 runs

2.2.1 Implementation Details

In these runs, Batched TDS is configured with a `processing_block` that can hold 1024 states. The `child_block` has room for 4096 entries. Both algorithms have a transposition table that has 2^{23} hash targets that each contain room for 4 transpositions. So with each thread that is added, the effective transposition table grows with the same amount. All other implementation details are similar to those described in Section 1.4.1.

2.3 Results

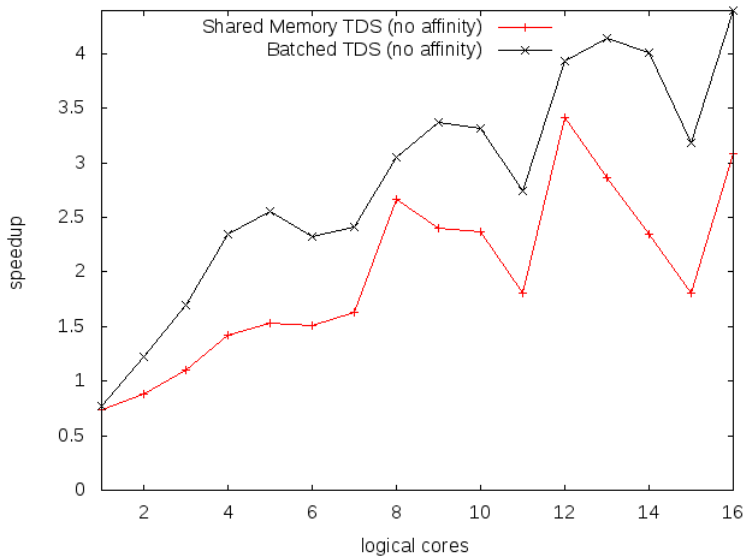

Figure 2.3: The speedup relative to sequential IDA* with a transposition table containing 2^{27} hash targets for both the Shared Memory TDS algorithm and the Batched TDS algorithm. Higher numbers are better.

Figure 2.3 shows that Batched TDS has a significant performance improvement over Shared Memory TDS. It shows a hilly landscape with valleys around the 6, 11 and 15 processor-points. Taking a look at Figure 2.4 and Figure 2.5, it seems that cache performance explains a lot about the performance of both algorithms. The hilly landscape of Figure 2.3 seems to be largely present, be it inverted: hills are peaks and the other way around.

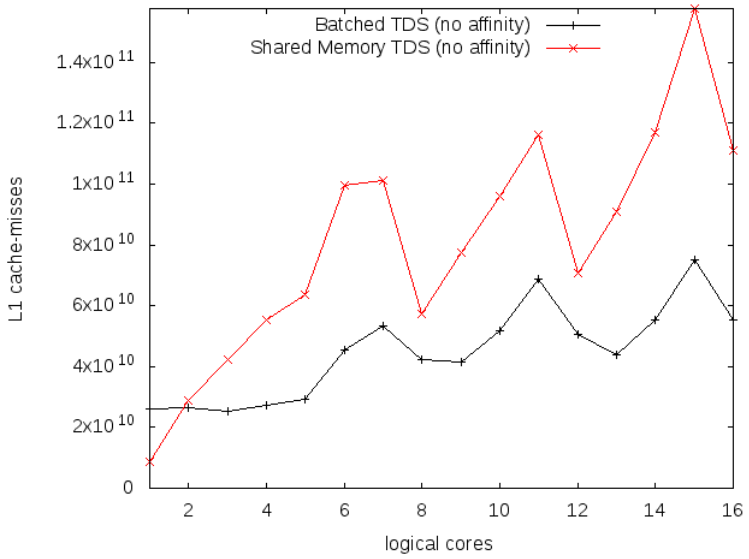

Figure 2.4: The L1 cache performance for different amounts of logical cores that are used for the Shared Memory TDS algorithm and the Batched TDS algorithm. Lower numbers are better.

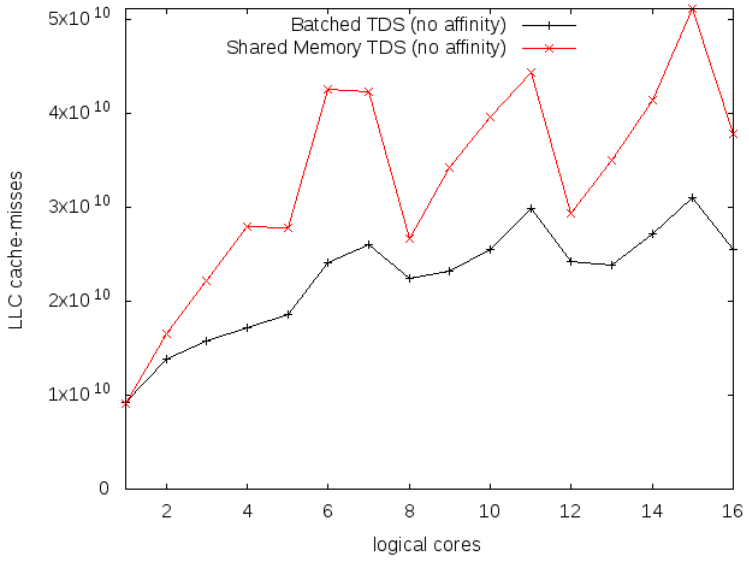

Figure 2.5: The LLC performance for different amounts of logical cores that are used for the Shared Memory TDS algorithm and the Batched TDS algorithm. Lower numbers are better.

The reason why Batched TDS has better cache performance is because Batched TDS makes more use of memory locality than Shared Memory TDS. One reason is that we re-iterate over the memory block, which contain the states every time which causes it to be inside the cache most of the time. The second reason is that the most important data structures are not used one after the other, but since we process an array of elements, the structures are used many

times after each other before using the next data structure. This access pattern keeps most of the data of these structures inside the caches, further increasing the data-locality. The increased data locality explains that the amount of misses are overall lower. If something is fetched from memory, a bigger amount of states use this piece of memory while it is low in the cache hierarchy, reducing cache-misses on higher cache levels level, and cache misses overall.

The peaks in the graphs are harder to explain. TDS is by nature a symmetric algorithm: all threads do more or less the same, be it that they work with different states. It even balances the work evenly among the threads due to the use of Zobrist hashing. It is therefore unlikely that the peaks are caused by something in semantics of the algorithm, so the cause must be in the hardware. In fact, this is not surprising since the hardware is indeed not symmetric since it is a NUMA system. Since both algorithms let the operating system decide where to put the threads and the data, both the data and the thread are not necessarily located on the same node. On top of that, getting data from a remote L3 cache is quite costly [22] and should be avoided.

One explanation for the peaks is that the QPI gets used a lot when nearly all data is present on the wrong node for some processes. This has an effect on how messages are passed over the QPI. It is suspected that latencies appear because the QPI protocol has problems with some core distributions over different chips, and works better with other. However, this could not be confirmed.

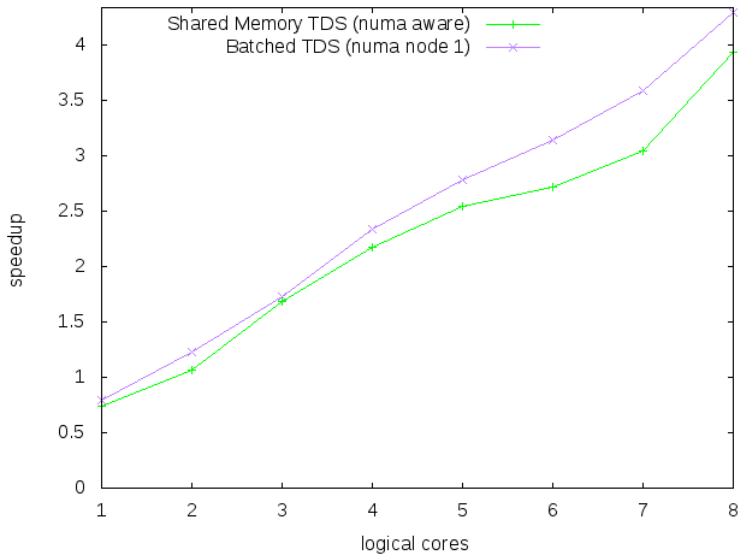

Figure 2.6: The speedup relative to sequential IDA* with a transposition table containing 2^{27} hash targets for both the Shared Memory TDS algorithm and the Batched TDS algorithm. Batched TDS allocates most of its memory on NUMA node 1, the Shared Memory TDS allocates all of its memory on NUMA node 1. Both algorithms only use processors on NUMA node 1. Higher numbers are better.

Figure 2.6 shows what happens when both the processors and the location of the memory are fixed to NUMA node 1. The first thing to notice is that the lines lie much closer to each other, but still Batched TDS outperforms Shared Memory TDS. The second thing to notice is that most of the hills have disappeared: the lines are much straighter. However, there is still a small performance drop when 6 or 7 cores are used.

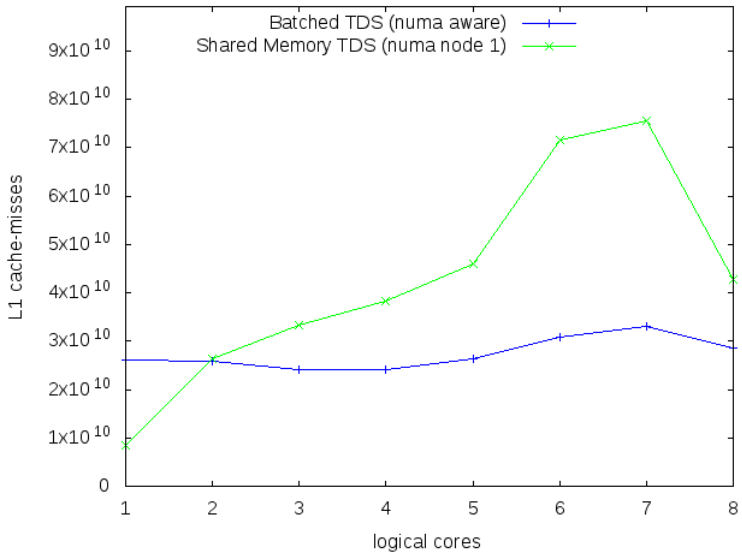

Figure 2.7: The L1 cache performance for different amounts of logical cores that are used for the Shared Memory TDS algorithm and the Batched TDS algorithm. Batched TDS allocates most of its memory on NUMA node 1, the Shared Memory TDS allocates all of its memory on NUMA node 1. Both algorithms only use processors on NUMA node 1. Lower numbers are better.

Looking at the L1 cache performance in Figure 2.4 a rise in cache misses around 6 and 7 cores is evident, explaining the dip in the run-time of both algorithms. Also the L1 performance of Batched TDS is still better than that of Shared Memory TDS, which is expected when the data locality is indeed better.

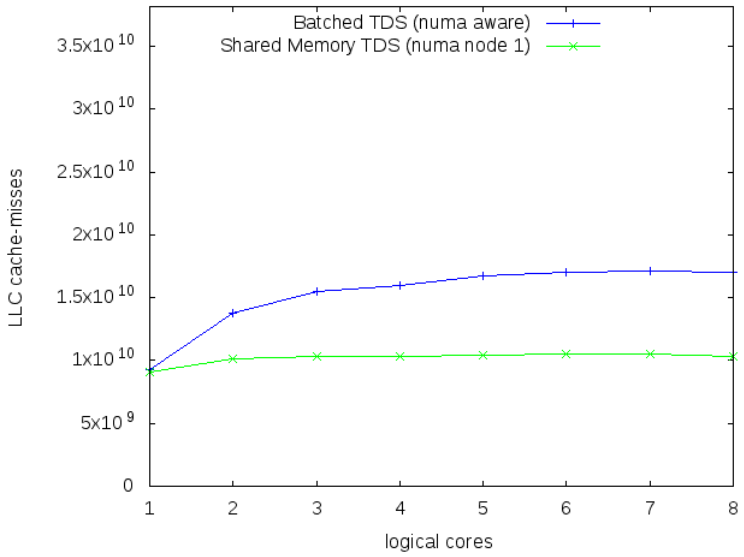

Figure 2.8: The LLC performance for different amounts of logical cores that are used for the Shared Memory TDS algorithm and the Batched TDS algorithm. Batched TDS allocates most of its memory on NUMA node 1, the Shared Memory TDS allocates all of its memory on NUMA node 1. Both algorithms only use processors on NUMA node 1. Lower numbers are better.

In the LLC, Figure 2.8, things are however different. The first thing that is apparent is that Shared Memory TDS performs better than Batched TDS on this point and it might account for the smaller gap between the two algorithms when comparing the speedup. The amount of LLC misses for the Batched TDS algorithm seems to converge to about 70% more misses than that of the Shared Memory TDS algorithm. The reason that Batched TDS performs worse is probably that because of the buffers the total memory use is bigger. When more threads, and with that cores, are added, the LLC cache needs to be shared among more processes, decreasing the per-thread LLC size, due to which the effectiveness per thread will be less, increasing the amount of misses. The data used by Shared Memory TDS is probably so small the amount of LLC misses remains about constant with increasing core counts.

The second thing to notice in this graph is that it is hard to spot a significant peak when using 6 cores. This might be the case because the L3 cache is shared between all the cores on this NUMA node. Like discussed in Section 2.2, it is distributed in slices across the die and only accessible through the ring network. The slices are also not local to a processor, and cache lines are distributed by a hash among the slices, making the average access time to it more or less uniform among cores. In addition to that, all memory is allocated from the same node, which therefore also has more or less uniform cost. Compare this to the case in Figure 2.5 where the L3 cache might sometimes need to fetch data from the remote NUMA node, making the access times irregular. Together with an irregular distribution of cores among the processors this can lead to peaks in memory access times and by that increase the amount of cache misses.

2.4 Conclusion

The results from Section 2.3 show that the overhead of putting the algorithm in a form where it processes the states using blocks of states instead of processing one state at a time is more than

compensated by the increased memory locality. The data locality seems to effectively reduce the amount of misses, and with that the amount of time the algorithm has to wait on data from a remote node. Therefore Batched TDS is a promising algorithm to vectorize using SIMD instructions, and to use in a NUMA architecture.

One should however note that the benefit of doing so seems to be limited. The amount of time spent on expansion and evaluation is only about 50%, so speeding this up can not even double the performance of the algorithm. A probable bottleneck is the transposition table. SIMD instructions for the Intel architecture need the data to be aligned and sequentially in memory. Since transposition tables are random access, a way is needed to access random memory locations. Although AVX2 instructions add a `gather` instruction which allows the programmer to load memory in a random access way, it does not provide a `scatter` instruction, which would provide a SIMD operation for stores [15]. Therefore storing data in the table probably still needs to be a sequential operation, at least on the processor used in these experiments, limiting the scalability available to this algorithm.

Chapter 3

TDS on a system with multiple sockets

3.1 Introduction

To increase the core count of single machines, servers can have multiple processors located on their motherboard which are located in different sockets. These sockets are connected with fast networks so the processors can communicate. Although these networks are fast, they will give some latency overhead, even compared to accessing the L3 cache.

Making the effect even more complicated, the memory in a machine is usually divided between the processors in the sockets. This gives rise to so-called Non Uniform Memory Access architectures (NUMA architectures), where parts of the memory are local to a processors, and other parts are remote: they can only be accessed through another processor, so the data needs to be sent over the inter-socket network, creating additional latency. The results from Chapter 2 give an indication about the huge effects of this NUMA architecture on the performance of the algorithm.

The correspondence to the situation of a distributed system is clear: sending data between processors takes time. The big difference is however that this time is explicit on distributed systems: one needs to explicitly send information to another machine. In a shared memory environment it is implicitly handled by the hardware when loads and stores are requested by the user of the system. The programmer can only influence where the data is put and where the work is done. It therefore is not suprising that modern Intel processors like the Xeon Haswell-EP CPUs are also called "distributed shared memory multiprocessors" to underline this correspondence to a distributed memory system. The question that this chapter investigates is how much impact this socketed environment has on TDS. The next question is if TDS can hide this latency in a way, much like it does in a distributed environment. To this means an implementation of TDS will be described in Section 3.1.1, called Split TDS, which tries to make communication asynchronous by adding another thread.

The remainder of this chapter is structured as follows. Section 3.2 will explain the experiments that were conducted in order to investigate this problem. The results of these experiments are stated in Section 3.3 and using those, some conclusions are given in Section 3.4.

3.1.1 Split TDS

Split TDS tries to further improve on Batched TDS in the way that it tries to take the latencies into account and tries to implement true asynchronous communication. To achieve this, the algorithm makes use of the fact that Intel processors can use the Hyper-Threading technology. Hyper-Threading [19] tries to make more effective use of the core by duplicating the instruction-issue logic of a core, creating an extra logical core on the processor, which shares the processor logic, the L1 and the L2 cache with another logical processor, which both reside on the same physical processor. The instruction of one Hyper-Thread is executed when the other Hyper-Thread is stalling, or just not using that part of the processor and vice-versa.

Using this technique as an inspiration, TDS is split into two parts which are scheduled on the two different logical cores of one physical core. The first part is called the "communication thread", the second part will do the processing of the states and is therefore called the "processing thread".

The communication thread will communicate with the other communication threads in the machine. Its second responsibility is to check the states against its local transposition table. The processing thread its sole responsibility is to expand and evaluate the states. To send a state, both threads communicate the `processing_block` and the `child_block` between the threads through a circular buffer. When the communicator has received all states, or the `processing_block` is full, the `processing_block` is passed to the processing thread. The processing thread expands and evaluates all the states in the `processing_block`, and the children are added to the `child_block` which is then sent to the communication thread.

The idea is that both threads will be idle for some time while they process memory requests, and they will rarely use all the processor resources. During that time the other thread can do some useful work and vice versa, effectively increasing the throughput of the processor. This allows the programmer to automatically overlap some of the latency caused by the fact that data needs to be fetched at a different processor. Besides that, it might be more effective to use both threads in this way than to just duplicate a second TDS thread on the logical core since in Split TDS both threads work on the same data: a second TDS thread would just effectively split the L1 and L2 cache in half for each TDS thread, possibly reducing cache performance.

3.2 Experiments

For the experiments, the speedup of Split TDS is compared with that of Batched TDS. Two versions of Batched TDS are compared, one with a communication network as described in described in Section 2.1.1, which is also used for Split TDS. The other implementation of Batched TDS is tested with a crossbar network. The crossbar network is emulated by giving each pair of processors a pair of circular buffers to communicate through in a lockless way. To speed up the look-up of which circular buffer has recently received something, each sending thread posts a message in a multiple producer, multiple consumer queue from the *moodycamel*[1] library. It was noticed that in contrary to the scaling of the space needed for this communication network, the time scaling was much better than the standard implementation mentioned in Section 2.1.1. The testing machine and test-cases are identical to those described in Section 2.2.

3.2.1 Implementation

Batched TDS is implemented just as described in Section 2.2.1. For Split TDS the same buffer sizes were used. The circular buffers between the communication thread and the processing

[1] `https://github.com/cameron314/concurrentqueue`

thread contain 3 blocks to pass between the threads, which should be more than enough to avoid threads waiting on each other to be processing the results. The crossbar network contains circular buffers which have 819200 entries each. It was noticed that using smaller buffers could cause buffer overflows in some cases.

Threads for Split TDS are allocated in pairs: the communication thread and processing thread are allocated on the same physical cores. First one NUMA node is filled, after which the second NUMA node is filled 1 physical processor at a time. For both implementations of Batched TDS first the physical cores of NUMA node 1 are allocated, then the physical cores of NUMA node 2, after which the logical cores created by Hyper-Threading are added in the same order.

Split TDS does not allocate its memory in a NUMA aware fashion. Most of the memory is probably allocated at NUMA node 1, creating some imbalance. The Batched TDS versions allocate their transposition table and work queues in a NUMA aware fashion by allocating the memory at the NUMA node of the thread that will use it.

3.3 Results

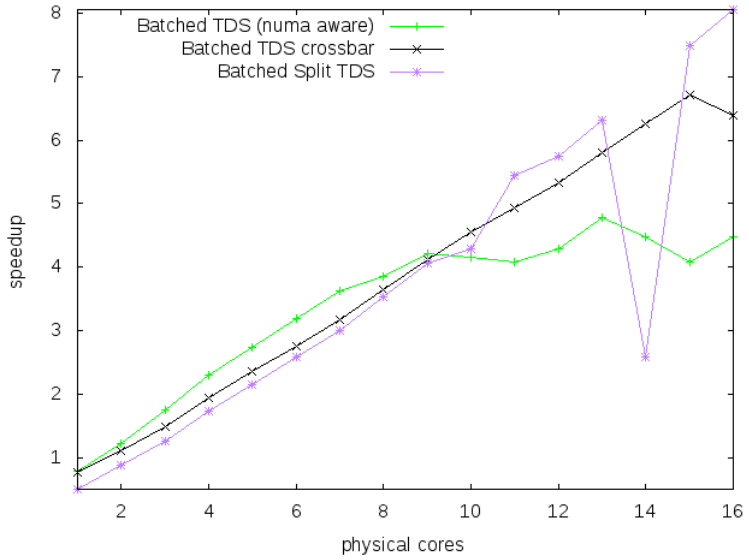

Figure 3.1: The speedup of both Batched TDS implementations and Split TDS. In this plot the amount of physical cores are plotted for easy reference, since Split TDS can only allocate threads in pairs. To get the amount of logical cores used, the amount of physical cores for Split TDS should be doubled. Higher numbers are better.

Looking at Figure 3.1 it can be seen that in contrary to the speedup of the normal Batched TDS implementation, the crossbar scales linearly, where the normal Batched TDS implementation flattens as it goes past the 7 or 8 cores point. This is an interesting result since it seems that the normal Batched TDS implementation suffers from adding a new CPU in a different socket, and Batched TDS with a crossbar does not.

When profiling it was discovered that the main reason for this decrease in performance is that past 7 cores, the algorithm spends an increasingly long time waiting for memory loads from the communication network. Upon further inspection, using `perf` to show the percentage of time

spent on each instruction, it was found that the memory load directly behind the `atomicAdd()` instruction becomes a lot more costly. The few memory loads after that point also suffer from higher cycle counts, but the effect is less severe. The cause of this effect is the `atomicAdd()` instruction, which is implemented by a `lock`-prefixed `xadd` operation. Before the atomic instruction is executed, the re-order buffer is cleared. Then the instruction is executed, and after that the next instructions are executed. Normally this instruction re-order buffer ensures that other issued instructions are executed while waiting on a memory operation. When this buffer is cleared, no operations can be executed while waiting on a load. So when this waiting time is increased because it is from remote NUMA memory, the observed latency of the memory load will also increase.

Another explanation of this reduction in performance can be found in the observation that Batched TDS with crossbar seems to be insensitive to the added latencies of processes running in a different socket. This reason could be that the ring buffers in the crossbar implementation are shared by two processes only, in contrary to the normal Batched TDS implementation which shares its communication memory spaces with all other processes. When a cache line is fetched, Intel processors automatically prefetch the next cache line. This is a good pattern for a ring buffer, especially when it is shared by two processors. For both processors the next position that they will access in the buffer will be adjacent to the one they are currently using: the sending processor will prefetch the next position in the buffer where the next state will be placed which needs to be sent, and the receiving processor will prefetch the next state it will receive. Compare this to the case when a buffer of states is shared between more than two processors. In that case the adjacent state in the buffer might not be the one the current processor will access, since access to it might be granted to another processor, reducing the effectiveness of the prefetching of this memory position. It, however, is hard to confirm this hypothesis from the given results.

Although the crossbar implementation of Batched TDS scales linearly, it does not achieve the perfect speedup like presented in the original TDS research [27]. The reason is not that Figure 3.1 shows the speedup relative to the sequential algorithm, where the original research compares it to the parallel version on one core. When comparing Batched TDS using a crossbar to a single processor run of itself, it achieves a speedup of 8.6 on 15 cores. When extrapolating the line to 16 cores, by ignoring the dip, the speedup when compared to itself can rise to 9.2. This is far from a perfect linear speedup.

Upon further inspection, the reason why Batched TDS with crossbar does not achieve the perfect linear speedup is because when using one core, about 75% of the time is spent in the expand and evaluate part of the program. When adding a core, this time is reduced to about 54%, and the algorithm spends about 22% of its time communicating. Assuming the total time spent in the expand and evaluate section remains constant, it is clear that the per-core running time of the algorithm is significantly slower when communication is added. This slowdown accounts for the less steep slope than it would have when the communication would not be a factor. Without the communication overhead the speedup would be nearly perfectly linear. Since the percentage of time spent in each part of the algorithm remains nearly constant for 2 or more cores, it still results in a straight line.

Another thing apparent is that Split TDS scales quite well. Beyond 9 physical cores it even achieves a higher speedup than Batched TDS using a crossbar. The reason might lie in the fact that when two sockets are used, the memory loads on average take slightly more time, allowing the other thread pair to run slightly longer on the Hyper-Thread, which might decrease overall waiting time. The dip at 14 cores is there because the transposition table and the workqueue for Split TDS are not allocated in a NUMA aware fashion, which might leads to an imbalance of processing speed between cores. It is interesting to see that still the speedup is quite similar to the one achieved by the crossbar implementation of Batched TDS, making it plausible that

Hyper-Threads can indeed cover up (part) of the extra communication time.

Looking at these results it is of course not totally fair to look at physical core numbers instead of logical core numbers. Both Batched TDS algorithms could potentially use those logical cores to achieve an extra speedup. Table 3.1 shows the speedups when we look at the logical cores instead of the physical ones. These are also probably the highest speedups achievable for these algorithms on the used machines since all logical cores are used, and for higher thread counts the threads need to be switched in and out more often by the operating system, causing some overhead. It is interesting to see that Split TDS and Batched TDS with a crossbar achieve about the same speedup, even when Split TDS has the overhead of sending an extra set of buffers between the communication and processing thread.

Algorithm	running time (s)	speedup
Split TDS	142	8
Batched TDS	121.5	9.4
Batched TDS with crossbar	143	8

Table 3.1: Speedup of Split TDS and the two implementations of Batched TDS when using 32 logical cores. Results are the average over 10 runs. The speedup is compared to IDA* with an equally sized transposition table.

Also interesting is that the normal Batched TDS implementation more than doubles its speedup over the one achieved using 16 physical cores. A theory on why that is possible is that, like discussed earlier, the main bottleneck at 16 cores for this algorithm seems to be the time spent just waiting on memory transfers. This amount is higher than the time the version with the crossbar is waiting on memory transactions. This creates a lot of idle time for the algorithm, which the Hyper-Threading system can use up quite well scheduling work from the other TDS thread.

The reason that it more than doubles might lie in the fact that when we look at Figure 3.1, the speedup at 16 cores is not the highest achievable speedup. In fact, the algorithm achieves better results at 13 cores. Although it is hard to prove from these numbers, a reason for this decrease in speedup from 13 to 16 cores might again be that the NUMA aware allocation is done just for the transposition table and the work queue. Since other data structures reside at NUMA node 1, while the threads itself are executed on node 2, the threads on node 2 progress slower than those on node 1 due to the added idle time for some memory accesses on node 2. This idle time can be used for useful computation by Hyper-Threading. Again this shows that Hyper Threads can at least cover some of the latency of the remote memory fetching.

It is interesting to think about the implications of some threads running slower than others. TDS distributes its work among the threads about equally using the Zobrist hash. If some cores effectively run slower than others, it causes a load imbalance since each core needs to process about the same amount of nodes. This results in an on-average low amount of per-core efficiency for the number of cores used. If Hyper-Threading is used, the imbalance gets smaller since useful work might be done in the other thread, effectively letting all cores on the machine run at more or less equal speeds. This has a positive effect on the overall running time of the algorithm since the per-core efficiency goes up.

The final ordering of the algorithms in Table 3.1 can be explained by the fact that Batched TDS without a crossbar is the fastest, because although its communication implementation generates high latencies, it is by far the most simple by design, using a minimum of instructions. The reason why Split TDS still performs quite well compared to Batched TDS might lie in the fact that both the communication and processing threads use the same dataset, where two

different Batched TDS threads would have to share the L1 with different data sets, possibly thrashing the cache. This however still needs to be confirmed.

3.4 Conclusion

The first result is that when not using Hyper-Threads, Batched TDS scales better when using a crossbar network than it does when using the normal communication network that uses an `atomicAdd()` instruction. When using a crossbar network, no performance degradation is present when adding threads on a CPU in a different socket. Although it was not possible to find the true cause of it, this performance degradation is present when using the normal implementation of Batched TDS. It therefore seems that TDS is able to hide the latency present in multi-socket systems when a crossbar network is used.

The second result is that Split TDS does not give a real performance gain over Batched TDS which uses a crossbar network. Apparently implementing asynchronous communication in an explicit way is not that beneficial.

The third result is that running other threads in a hardware Hyper-Threading is a way to cover up the latencies when the communication mechanism itself does not do that. This was confirmed by the normal implementation of Batched TDS which achieves quite some speedup using 32 cores, and Split TDS which increased its speedup when the second NUMA node was added: the increased latency gaps could be filled in by the other Hyper-Thread.

These conclusions lead to the final conclusion that TDS can overcome overcome (some) of the latency caused by NUMA architectures by using a crossbar network. The same effect can also be achieved when using Hyper-Threading instead of this crossbar network when using all logical cores of the system. It is however not necessary to use TDS if one wants to use Hyper-Threading since it is a hardware feature. So although this chapter has shown that Batched TDS can make good use of the Hyper-Threading hardware, every algorithm can use this feature to overlap CPU stall cycles, for instance caused by memory latency, with work from another thread. However, since these Hyper-Threads share the logic of a core, it is algorithm dependent how much performance can be gained using this method.

Chapter 4

TDS versus Work-Stealing Algorithms in Shared Memory

4.1 Introduction

TDS was originally designed for distributed systems. One of the problems that it solved in distributed systems is how to divide the work among the processors. Not only was it a solution for the load balancing problem, it also made it possible to do communication asynchronously in a lot of cases, reducing the idle time of the algorithm.

Of course, sending and receiving work takes more time in distributed systems than it does on a shared memory system. In a distributed environment the work needs to be copied over a network, which is several orders of magnitude slower than reading from RAM. However sharing data between cores is far from 'free' either. Modern processors have complex networks on the chip between the caches, processors and sockets, and communication of those takes time. In addition to that problem, a shared memory environment needs to be 'emulated' by using coherency protocols that need to send messages to different cores. It just might be the case that this complex environment creates latencies that can be hidden by TDS.

To investigate this, a parallel version of IDA* will be analyzed and compared to an implementation of Batched TDS (Section 2.1.1). This parallel version of IDA* will be created using the Cilk runtime system [1], which is a straightforward way to create a work-stealing algorithm from the original IDA* algorithm. Besides that, it makes use of the memory like a shared memory algorithm would do: it shares the transposition table with all other threads. Further details about the Cilk version of IDA*, and about the shared memory version of TDS will be discussed in Section 4.1.1.

The remainder of the chapter will be structured as follows. Section 4.2 will explain the experiments that have been done. The results from these experiments will be presented in Section 4.3 and the answer on the question if TDS has a benefit over this work-stealing algorithm will be presented in Section 4.4.

4.1.1 Cilk IDA*

The first algorithm that we compare is named Cilk IDA*. It is a parallel implementation of IDA* written using the Cilk multithreading system. The Cilk multithreading system allows the user to convert an application into a parallel one by adding Cilk keywords to the sequential code [16]. These keywords reveal parallelism to the compiler and the Cilk runtime. The Cilk

runtime uses a threadpool and a work-stealing scheduler to run work in parallel. For instance, putting the `cilk_spawn` keyword in front of a function will give the Cilk runtime the opportunity to run a function on a different thread. The `cilk_sync` keyword makes the thread wait until all its spawned children have returned. The `cilk_for` keyword will give the Cilk runtime the opportunity to run a for-loop in parallel.

Algorithm 5 Cilk IDA* `search_with_bound(state, bound)` recursive version

```
1: if state.is_goal_state() then
2:   return true, lowest_higher_bound
3: end if
4: children ← expand_and_evaluate(state)
5: for all child ∈ children do
6:   if child.bound < bound then
7:     cilk_spawn found, child_lowest_bound ← search_with_bound(child, bound)
8:     if child_lowest_bound < lowest_higher_bound then
9:       lowest_higher_bound ← child_lowest_bound
10:    end if
11:    if found then
12:      return true, lowest_higher_bound
13:    end if
14:  else if child.bound < lowest_higher_bound then
15:    lowest_higher_bound ← child.bound
16:  end if
17: end for
18: cilk_sync
19: return false, lowest_higher_bound
```

To transform IDA* into a parallel algorithm by using Cilk, several modifications to IDA* have been made. The first difference is that a recursive version of the IDA* algorithm has been used, like stated in Algorithm 3. The recursive definition of the IDA* algorithm makes it easy to make it parallel since each recursive step can be done in parallel. Algorithm 5 shows the result of this modification, adding a `cilk_spawn` call before the recursive step. Since the `cilk_spawn` keyword resides in a **for** loop, instead the `cilk_for` keyword was used in the real implementation, which should be nearly semantically equivalent.

Apart from this relatively easy change, the transposition table is shared among threads, and should be made concurrent. To make it concurrent, a `concurrent_hash_map` from the Thread Building Blocks (*TBB*) [24] library was used which can hold up to 4 states per hash target, creating a 4-way set associative hash table. This implementation uses fine-grained locking of the values inside the table. In our implementation the critical section consists of the whole time between finding the correct hash-target and finally releasing the lock after all changes were made to the table, if they were necessary. *TBB* is a popular library for parallel programming, so it can be expected that this implementation of a hash table would give agreeable performance.

For the last noteworthy change, consider that each recursion step needs to keep track of the lowest number that was out-of-bound for the IDA* search. In this implementation, for the ease of programming, this value is tracked by using a Cilk-reducer that keeps track of this value in a lock-free way.

4.2 Experiments

To get a better idea how both algorithms scale, the time one of the biggest 15-game puzzle cases take for both Batched TDS and Cilk IDA* will be measured against the number of cores that are used for the search. Their running times will be compared. This will give an indication of the bottlenecks of both algorithms. On these bottleneck points time breakdowns are made to get an indication of the true cause of the the bottlenecks.

The breakdown graphs were made by analyzing the output of the annotated code generated by the `perf record` utility, which was run once. Since this command was run on optimized code, and a lot of code was inlined, they reflect an estimate, since a lot of instructions have very low percentages of running time, and some were aggregated to make the count easier. However, the estimations should be accurate enough to give an accurate view of the distribution of the time spent in each part of the code. The measurements in the speedup plots however represent the averages over 10 experiments, and should represent an accurate value. The test-case and testing machine used for these experiments is identical to that described in Section 2.2.

4.2.1 Implementation Details

For these experiments both algorithms use all the heuristics as discussed in Section 1.4.1. Although Cilk IDA* will choose its own cores to execute its threads on, Batched TDS is configured to use processor affinity. First all the physical cores of NUMA node 1 will be used, then the physical cores of NUMA node 2 will be used. After that the logical cores of NUMA node 1 will be added, and finally the logical cores of NUMA node 2 will be added, adding up to 32 logical cores on a machine.

Batched TDS will have its biggest data structures (the transposition table and the work-queues) allocated at the NUMA node where the owning thread is run on. However, unfortunately a problem of the implementation is that not all of the other memory is guaranteed to be allocated on the same NUMA node that the thread is working on. Since "local allocation" is the system default, it is expected that most of the other memory will be located at NUMA node 1, since it was allocated by a thread that runs on that node.

A small difference in the implementation of the communication of Batched TDS is that, unlike mentioned in Section 2.1.1, it will emulate a crossbar network by giving each pair of processors a pair of circular buffers to communicate through in a lockless way. To speed up the look-up of which circular buffer has recently received something, each sending thread posts a message in a multiple producer, multiple consumer queue from the *moodycamel* library. It was noticed that in contrary to the scaling of the space needed for this communication network, the time scaling was much better than the standard implementation mentioned in Section 2.1.1.

For the transposition table in Batched TDS, a standard size of 2^{23} hash targets, of which each can have 4 states were used for each additional thread. So the aggregated transposition table grows linearly with the amount of threads. The same effect was not easy to achieve for Cilk IDA*. Therefore the size was manually set for different amounts of threads to match the transposition table size of Batched TDS when an amount of threads is used that is a power of 2. Unfortunately the implementation of the transposition table does not support table sizes that are not a power of 2. Table 4.1 shows these sizes for different amount of threads.

AMOUNT OF THREADS	TABLE SIZE (AMOUNT OF STATES THAT CAN BE STORED)
2	$2^{24} * 4$
3,4	$2^{25} * 4$
5-8	$2^{26} * 4$
9-16	$2^{27} * 4$

Table 4.1: Amount of states that can be stored in the transposition table of the Cilk IDA* algorithm for different thread numbers.

An unfortunate difference between the two implementations is that the TBB `concurrent_hash_map` does not allow to pre-allocate all entries. This means that some memory allocation needs to be done during the run of the algorithm. The transposition table of Batched TDS is allocated before the algorithm runs.

4.3 Results

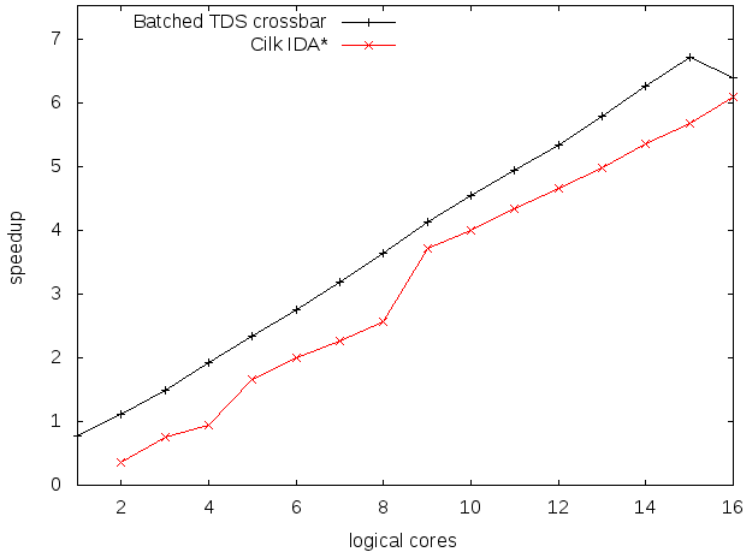

Figure 4.1: The speedup relative to sequential IDA* with a big transposition table for both the Cilk IDA* algorithm and the Batched TDS algorithm with crossbar. Higher numbers are better.

Figure 4.1 shows the speedup of both the Cilk IDA* algorithm and the Batched TDS algorithm. Batched TDS outperforms Cilk IDA* by a few percentages. However, when looking closer, the scaling behavior is nearly identical. The bumps in the line of Cilk IDA* are caused by the suddenly increasing transposition table sizes, as reported in Table 4.1. When a straight line is drawn between the 2 thread point and the 16 thread point on the Cilk IDA* line, it seems to run about parallel to the Batched TDS line.

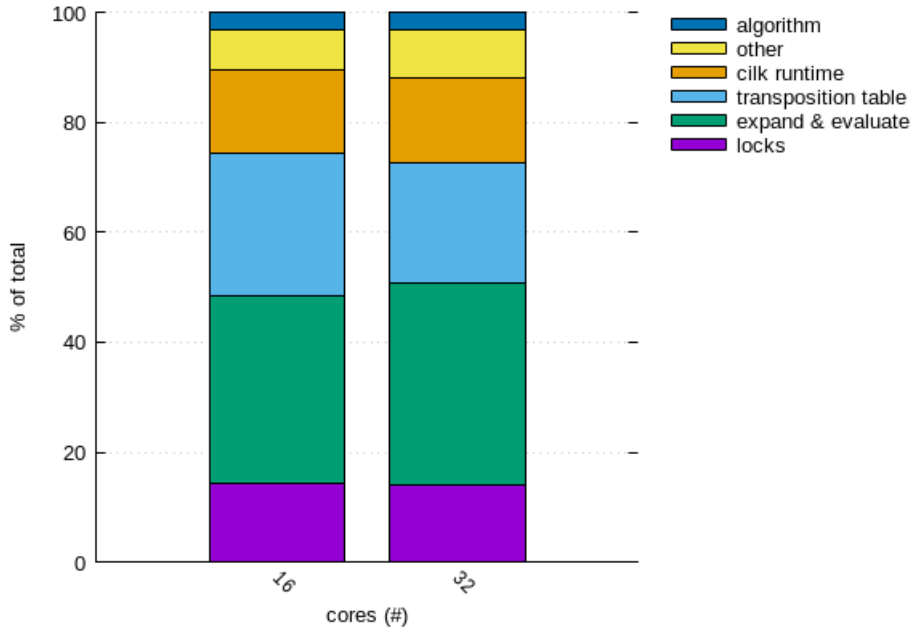

Figure 4.2: The percentages of time spent in the different parts of the Cilk IDA* algorithm.

The reason that the two algorithms run on different heights but parallel can be explained looking at the breakdown graphs in Figure 4.2 which shows the breakdown for Cilk IDA*, and Figure 4.3 which shows the breakdown graph for the Batched TDS algorithm. When looking at the 16 core case, it is interesting to see that Batched TDS spends a lot more time in the expand and evaluate phase than Cilk IDA*, although the algorithms for this part are identical. This is largely explained by looking at the time spent in the transposition table.

Cilk IDA* spends more time in the transposition table logic than Batched TDS, especially when it is taken into account that all "locks" time is spent in the transposition table, and therefore should be added. It is very reasonable that this difference comes thanks to the fact that some allocations for the table need to be done at run-time. Also Cilk IDA* its transposition table uses a more complicated replacement strategy than the table used by Batched TDS. It is not unthinkable that another implementation of the transposition table would be, except for the locking, as effective as the one used by Batched TDS.

Taking the transposition table explanation into account, still the expand and evaluate phase of Cilk IDA* takes less time than that of Batched TDS. Upon inspection of the L1 cache misses, the amount of L1 cache misses in the expand and evaluate phase of Cilk IDA* is half of amount of L1 cache misses encountered in the Batched TDS algorithm, even when the total L1 misses over all parts of the algorithm are about equal. A more efficient cache use would explain why this phase takes less time in Cilk IDA*. The reason why Cilk IDA* performs this well is because it uses the recursive definition of IDA*, where all memory is allocated on the call stack, which has a very good data locality.

Another difference is that the Cilk runtime seems to give more overhead than the TDS runtime, which also causes some performance degradation. Nevertheless, it seems plausible that an implementation of Cilk IDA* can be made which performs on par or even better than Batched TDS.

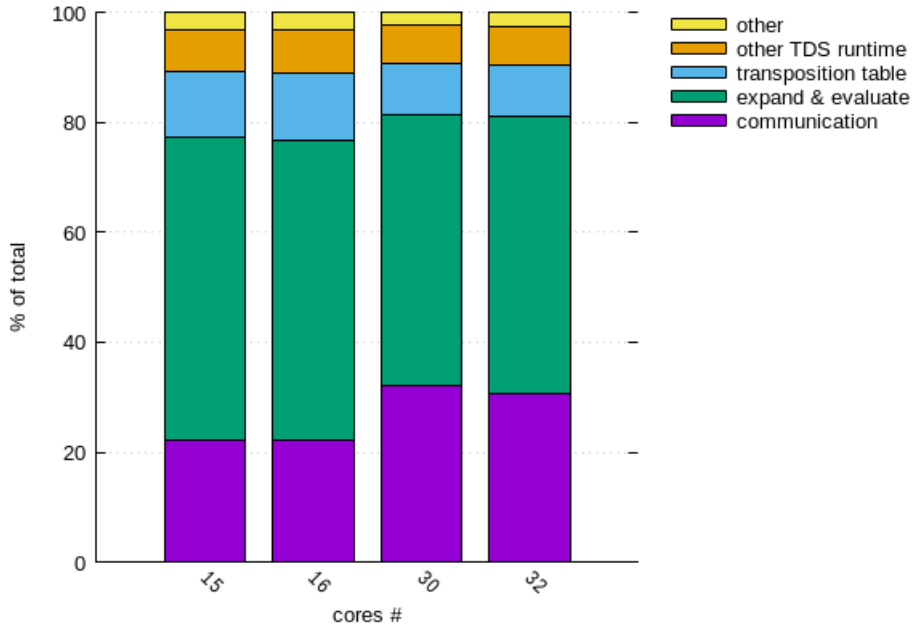

Figure 4.3: The percentages of time spent in the different parts of Batched TDS that uses a crossbar network.

The remaining question seems to be the scaling behavior of both algorithms. Looking at Figure 4.1 one can see that Cilk IDA* scales linearly, ignoring the sudden increases of the transposition table. However, Batched TDS shows a bump down from the transition of 15 cores to 16 cores. Looking at Figure 4.3 at both 15 and 16 cores does not show any significant change in the time spent in both cases. Although this can be because of the fact that these percentages come from single runs. A further inspection indeed revealed that some of the runs when using 16 cores had increased expand counts, resulting in a higher run-time, lowering the average run-time.

The most probable reason that these fluctuations appeared is because of the fact that not all data structures are allocated to the local NUMA node. For instance the `processing_blocks` all are located at NUMA node 1. This means that there is a probability that the caches accidentally evict one of these blocks, or that they are not prefetched efficiently enough since a prefetch from a remote node takes longer increasing the chances that the algorithm needs it before the memory is actually in the cache. When this happens, some processors on the second NUMA node run slightly slower compared to their brothers on NUMA node 1. Since the work is partitioned, these slower nodes still have to process a specific amount of work before the graph is fully searched, which takes them more time. During this time the faster nodes will process some additional states, resulting in a higher expansion count.

It is interesting to see that these NUMA problems do not happen on the Cilk IDA* implementation. This is because the algorithm only needs the transposition table as a data structure. All other structures are not necessary because they are implicitly created in the call-stack of the algorithm. Cilk allocates the call-stack on the thread that is run, so it is always allocated in local memory.

Figure 4.3 can give insights on why Batched TDS scales like it does. Like discussed in chapter 3, Batched TDS with a crossbar scales linearly, but does not achieve a perfect linear speedup because of the communication cost. To understand why this limits the steepness of the slope of

the line in Figure 4.1, consider that in the sequential version, about 80% of the time is spent in the expand and evaluate phase and 10% is spent in the transposition table. The amount of time in the expand and evaluate phase decreases when using more cores because of the communication time, which accounts for about 22% of the time spent in Batched TDS with crossbar.

Most of the communication time in the crossbar network is spent in the multiple consumer, multiple producer queue which was discussed in Section 3.2. The problem of this queue is that, although it is lockfree, it uses `lock`-prefixed instructions, and locking is expensive (Section 1.4): it takes nearly 15% of the communication time. It is still cheaper than using mutexes like Cilk IDA* does, because only one `lock`-prefixed instruction is needed instead of two, and there is no critical path in the software. Most of the other communication time is spent in other parts of this queue, although it is hard to pinpoint the exact cause of the queue being so expensive to use because it is hard to understand how it exactly works.

Algorithm	running time (s)	speedup
Cilk IDA*	153	7.5
Batched TDS	143	8

Table 4.2: Speedup of Cilk IDA* and Batched TDS at 32 cores.

Table 4.2 shows the speedups when 32 cores are used. It is interesting to see that the gap between Batched TDS and Cilk IDA* gets smaller. Looking at Figure 4.3 the reason probably is the communication network which scales bad. Upon further inspection the cause is again the multiple producer and multiple consumer queue which scales bad when Hyper-Threading cores are added. However, the reason why it behaves like this is hard to pinpoint because the logical core added due to Hyper-Threading shares its resources with another logical core, and scaling behavior therefore can be very algorithm and implementation dependent. Since there is little difference between the 30 core and the 32 core breakdown the same behavior might not appear when only physical cores are used. Further research is required.

Since the locks of Cilk IDA* seem to scale better than the communication of Batched TDS, at least when Hyper-Threads are used, results do not show that TDS is able to hide the latencies caused by the locking. Instead of locking, TDS introduces another problem, namely communication. However, it is plausible that by adding aggregation buffers to the algorithm, like described in the original research [26], the communication overhead can be reduced quite a a bit. It is less likely that the locking overhead can be reduced in a similar way in Cilk IDA* since the transposition table is random access. This makes it hard to aggregate states in a way only one locking action is needed for them, without blocking the table for all other threads.

This chapter, in a way, recreates some of the experiments done in the original paper [27]. A comparison with this paper gives us insights in whether TDS is needed on distributed shared memory multiprocessors. The original paper discusses two work-stealing implementations Work-Stealing Partitioned (WPS) where a transposition table is partitioned just like TDS, and Work-Stealing Replicated, where the transposition table is of a fixed size and updates are broadcasted over the network.

WSR did not scale because the amount of messages that were sent did not scale well with the amount of processes, clogging up the network. Furthermore the transposition table in WSR is not up-to-date until the broadcasted messages have been processed, generating a bigger workload.

WSP did scale linearly because the amount of sent messages scales well, the transposition table could grow when more processors were added and the transposition table is always up-to-date. The slope of WSP was however less steep than that of TDS because of a costly remote look-up that was needed when the state was not local to the processor. During this look-up over

Myrinet the processor would stay idle for about 6500 cycles. It is possible to process one or two states in that time, resulting in quite an efficiency drop of a process compared to the sequential algorithm, and to TDS where this idle time was removed by using asynchronous communication.

Cilk IDA* is quite similar to WSP, with a difference that only one process actually has the transposition table (there is no transposition table on NUMA node 2). Since the remote look-up "messages" are handled by the coherency protocol, no broadcast is needed for a transposition table update, so in that respect the algorithm scales well much like WSP. Since WSP scaled linearly, it is only logical that Cilk IDA* scales linearly as well. Cilk IDA* however has a performance much closer to Batched TDS than WSP had to TDS since the penalty of a remote look-up has been reduced from 6500 cycles to about 300 cycles.

In addition to this already smaller amount of cycles for a remote look-up, it is hard to overlap the remaining time and achieve real asynchronous communication on the Xeon 2630 v3, since these processors have a strong memory model, which ensures coherency among all cores. In order to achieve that, when writing or reading, the cache line should be present on the core which is doing this write or read. The time spent waiting on this cache line transfer can be reduced. One way is to overlap the computation by the instruction re-order buffer, but its effectiveness depends on the instruction level parallelism. The other way is to make use of memory locality by ensuring the cache line is in a fast part of the caching hierarchy. Still, in normal applications these two do not give guarantees: if the load or store creates a LLC cache miss, there will be some cycles of stalling time, reducing the effectiveness of this implicit asynchronous communication and by that the possibility of TDS to hide the remote look-up times.

4.4 Conclusion

This research could not find the big difference in scaling and performance between work-stealing and TDS on a distributed shared memory multiprocessor that the original paper [27] has found on distributed memory systems. Although Batched TDS was a bit faster than Cilk IDA*, it was argued that the difference was not significant. Regarding scalability and speedup, both algorithms operate on par. Especially when considering that the Cilk IDA* algorithm is easier to implement than Batched TDS. The performance gain of overlapping communication with computation is smaller on distributed shared memory multiprocessors since the memory model makes it hard to do real asynchronous communication, and the remote-look-up latencies that can be hidden are significantly smaller than those present on distributed memory machines.

The latency of distributed shared memory multiprocessors also shows itself through the increased cost of locking in these systems. Although it could not be confirmed by this research, it could be argued that it is plausible that TDS can hide the overhead of locks indirectly since it uses less `lock`-prefixed instructions than Cilk IDA* and does not have a critical section in the software. TDS can achieve this since it replaces (some of the) locks by adding communication, a method that has shown its benefits in other research [21].

Reducing the communication overhead of the current implementation could let it scale better than Cilk IDA*. This could be done by decreasing the overhead of the communication network to a few percent by aggregating states in an aggregation buffer so the penalty of the bookkeeping of the crossbar network (the multiple producer, multiple consumer queue) or the normal network (`atomicAdd()`) is only encountered once for each buffer of states. Using a similar technique to reduce the locking overhead for Cilk IDA* is less plausible because of the random-access nature of the transposition table.

Chapter 5

Conclusions and future work

5.1 Conclusions

This thesis has given an answer to three research questions.

RQ1: Can parts of the TDS algorithm be vectorized, and is it worthwhile?

The results from Section 2.3 show that the overhead of putting the algorithm in a form where it processes the states using blocks of states instead of processing one state at a time is more than compensated by the increased memory locality. The data locality seems to effectively reduce the amount of misses, and with that the amount of time the algorithm has to wait on data from a remote node. Therefore Batched TDS is a promising algorithm to vectorize using SIMD instructions, and to use in a NUMA architecture.

One should however note that the benefit of doing so seems to be limited. The amount of time spent on expansion and evaluation is only about 50%, so speeding this up can not even double the performance of the algorithm. A probable bottleneck is the Transposition Table. SIMD instructions for the Intel architecture need the data to be aligned and sequentially in memory. Since transposition tables are random access, a way is needed to access random memory locations. Although AVX2 instructions add a `gather` instruction which allows the programmer to load memory in a random access way, it does not provide a `scatter` instruction, which would provide a SIMD operation for stores [15]. Therefore storing data in the table probably still needs to be a sequential operation, limiting the scalability available to this algorithm.

RQ2: How does TDS perform when two CPUs communicate through sockets?

The first result is that Batched TDS scales better when using a crossbar network than it does when using the normal communication network that uses an `atomicAdd()` instruction. When using a crossbar network, no performance degradation is present when adding threads on a CPU in a different socket. Although it was not possible to find the true cause of it, this performance degradation is present when using the normal implementation of Batched TDS. It therefore seems that TDS is able to hide the latency present in multi-socket systems when a crossbar network is used.

The second result is that Split TDS does not give a real performance gain over Batched TDS which uses a crossbar network. Apparently implementing asynchronous communication in an explicit way is not that beneficial.

The third result is that running other threads in a hardware Hyper-Threading is a way to cover up the latencies when the communication mechanism is worse than that of the crossbar network. This was confirmed by the normal implementation of Batched TDS which achieves quite some speedup using 32 cores, and Split TDS which increased its speedup when the second NUMA node was added: the increased latency gaps could be filled in by the other Hyper-Thread.

These conclusions lead to the final conclusion that TDS can overcome overcome (some) of the latency caused by NUMA architectures by using a crossbar network. The same effect can also be achieved when using Hyper-Threading instead of this crossbar network when using all logical cores of the system. It is however not necessary to use TDS if one wants to use Hyper-Threading since it is a hardware feature. So although this chapter has shown that Batched TDS can make good use of the Hyper-Threading hardware, every algorithm can use this feature to overlap CPU stall cycles, for instance caused by memory latency, with work from another thread. However, since these Hyper-Threads share the logic of a core, it is algorithm dependent how much performance can be gained using this method.

RQ3: How does TDS perform compared to work-stealing algorithms in a shared memory environment?

This research could not find the big difference in scaling and performance between work-stealing and TDS on a distributed shared memory multiprocessor that the original paper [27] has found on distributed memory systems. Although Batched TDS was a bit faster than Cilk IDA*, it was argued that the difference was not significant. Regarding scalability and speedup, both algorithms operate on par. Especially when considering that the Cilk IDA* algorithm is easier to implement than Batched TDS. The performance gain of overlapping communication with computation is smaller on distributed shared memory multiprocessors since the memory model makes it hard to do real asynchronous communication, and the remote-look-up latencies that can be hidden are significantly smaller than those present on distributed memory machines.

The latency of distributed shared memory multiprocessors also shows itself through the increased cost of locking in these systems. Although it could not be confirmed by this research, it could be argued that it is plausible that TDS can hide the overhead of locks indirectly since it uses less `lock`-prefixed instructions than Cilk IDA* and does not have a critical section in the software. TDS can achieve this since it replaces (some of the) locks by adding communication, a method that has shown its benefits in other research [21].

Reducing the communication overhead of the current implementation could let it scale better than Cilk IDA*. This could be done by decreasing the overhead of the communication network to a few percent by aggregating states in an aggregation buffer so the penalty of the bookkeeping of the crossbar network (the multiple producer, multiple consumer queue) or the normal network (`atomicAdd()`) is only encountered once for each buffer of states. Using a similar technique to reduce the locking overhead for Cilk IDA* is less plausible because of the random-access nature of the transposition table.

By answering these three research questions, the problem statement can be answered.

PS: Is there enough latency in modern systems to be hidden by TDS?

It was shown that the latency in distributed shared memory multiprocessors shows itself in two ways when running IDA* in parallel. The first way is the latency caused by a look-up in a transposition table that resides on a different NUMA node. It was argued that this latency is significantly lower than in the original paper [27].

The answer on **RQ3** shows that it is hard to hide the latency that remains because it is hard to do real asynchronous communication in these systems due to the strong memory model. The

answer on **RQ2** further shows that it has no use trying to create asynchronous communication explicitly by using a variation of TDS called Split TDS. Hyper-Threading seems to overlap (part of) this latency at least equally effective. Although Batched TDS seems to perform quite well when using Hyper-Threads and the benefits of Hyper-Threading are implementation dependent, it is a hardware feature that is available to every algorithm that can be run on more threads. Therefore in theory any threaded algorithm could cover this latency up, at least partially, and there is no performance benefit in using TDS to cover up this first type of latency.

The second way latency in distributed shared memory multiprocessors shows itself is through the cost of locking. When more processors are added to a machine, it takes more time to lock a memory location on all processors. Unfortunately this needs to happen before a certain operation is executed on these memory locations to ensure the atomic properties of these operations. It was shown that TDS can trade the locks necessary in a work-stealing implementation of IDA*, Cilk IDA*, for communication. Although the results of the experiments did not find a significant performance improvement of TDS over Cilk IDA*, it was argued that the potential performance of TDS could be (slightly) better than that of Cilk IDA* since it is hard to perform lock batching on a transposition table because of its random access properties. On the contrary, for TDS, it is plausible that by using an aggregation buffer, like in the original TDS research, it would be possible to reduce the communication time significantly. Therefore it is concluded that TDS at least has more potential to hide the latency of locking, by using communication, than Cilk IDA*.

Taking this all into account, this research could not confirm that TDS is able to hide the latency in modern systems. It also can be concluded that at this point in time even the potential benefits of using TDS over a work-stealing implementation on a distributed shared memory multiprocessor are a lot smaller than the benefits that were once encountered in distributed memory machines. Although this is the case, one needs to take into account that due to increasing core and socket numbers, communication times between cores will keep increasing in the coming years. This results in less performance of locked instructions, and increases the maximum penalty of a remote transposition table look-up. If the communication overhead could be reduced, by using aggregation buffers like discussed before, or maybe architectures where asynchronous sends and receives between cores are easy to implement, then it is still plausible that TDS can be used to outperform work-stealing implementations.

5.2 Future Work

A first thing that can be added to future work is to strengthen the result from Chapter 4. Adding an aggregation buffer to the algorithm would definitely confirm or falsify the theory that TDS can hide lock latency. Also taking more care during the allocation of the data structures with respect to the NUMA architecture would clean a lot of noise from the results, strengthening the whole research.

To strengthen the results from Chapter 3, it would be interesting to measure prefetcher performance counters that show why the crossbar network behaves better than the communication network that uses `atomicAdd()` instructions. It might give insights in how to implement inter-thread communication even more efficient.

Since only 16 physical cores and 32 logical cores were considered during this research, only a limited insight has been gained into the scalability of TDS. Increasing the core count and the socket count could give light to new bottlenecks for all the considered algorithms.

It would be interesting to see how TDS would behave on a GPU or on a Xeon Phi, since Chapter 2 shows that it is feasible to create a blocked implementation of TDS, which generate opportunities to use SIMD instructions. CUDA enabled GPU's even deliver some performance on

random memory accesses, so TDS might even scale beyond the slight speedup that is promised when only the expansion and evaluation can be sped up. The Xeon Phi also provides the `scatter` SIMD instruction in addition to the `gather` instruction that is present in the AVX2 set, possibly allowing a similar speedup to that of a GPU. Another thing that is promising about CUDA GPU's is that they have a weaker memory model the Intel processor used in this research, allowing the user to post asynchronous stores to memory, making it easier to implement true asynchronous communication.

There is also a potential use for TDS in hybrid environments using accelerators and CPUs to do the computation. Both the bandwidth and latency are worse for PCI busses. Also both the Xeon Phi and CUDA GPUs support asynchronous memory transfers between the CPU and the accelerator, making it an even more interesting environment for TDS to work in.

Bibliography

[1] Robert D Blumofe, Christopher F Joerg, Bradley C Kuszmaul, Charles E Leiserson, Keith H Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. *Journal of parallel and distributed computing*, 37(1):55–69, 1996.

[2] Bernd Brügmann. Monte carlo go. Technical report, Citeseer, 1993.

[3] Arnaldo Carvalho de Melo. The new linux'perf'tools. In *Slides from Linux Kongress*, volume 18, 2010.

[4] Alex Fukunaga, Akihiro Kishimoto, and Adi Botea. Iterative resource allocation for memory intensive parallel search algorithms on clouds, grids, and shared clusters. In *AAAI*, 2012.

[5] Ralph Udo Gasser. *Harnessing computational resources for efficient exhaustive search*. PhD thesis, Citeseer, 1995.

[6] Tobias Graf, Ulf Lorenz, Marco Platzner, and Lars Schaefers. Parallel monte-carlo tree search for hpc systems. In *Euro-Par 2011 Parallel Processing*, pages 365–376. Springer, 2011.

[7] Peter E Hart, Nils J Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968.

[8] Farzad Khorasani, Mehmet E Belviranli, Rajiv Gupta, and Laxmi N Bhuyan. Stadium hashing: Scalable and flexible hashing on gpus. In *2015 International Conference on Parallel Architecture and Compilation (PACT)*, pages 63–74. IEEE, 2015.

[9] Akihiro Kishimoto, Alex S Fukunaga, and Adi Botea. Scalable, parallel best-first search for optimal sequential planning. In *ICAPS*, 2009.

[10] Akihiro Kishimoto and Jonathan Schaeffer. Distributed game-tree search using transposition table driven work scheduling. In *Parallel Processing, 2002. Proceedings. International Conference on*, pages 323–330. IEEE, 2002.

[11] Levente Kocsis and Csaba Szepesvári. Bandit based monte-carlo planning. In *European conference on machine learning*, pages 282–293. Springer, 2006.

[12] Richard E Korf. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial intelligence*, 27(1):97–109, 1985.

[13] Richard E Korf and Larry A Taylor. Finding optimal solutions to the twenty-four puzzle. In *Proceedings of the national conference on artificial intelligence*, pages 1202–1207. Citeseer, 1996.

[14] Aleš Křenek, Igor Peterlík, and Luděk Matyska. Building 3d state spaces of virtual environments with a tds-based algorithm. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 529–536. Springer, 2003.

[15] Olaf Krzikalla, Kim Feldhoff, Ralph Müller-Pfefferkorn, and Wolfgang E Nagel. Auto-vectorization techniques for modern simd architectures. In *Proc. of the 16th Workshop on Compilers for Parallel Computing, Padova, Italy (January 2012)*, 2012.

[16] Charles Leiserson and Aske Plaat. Programming parallel applications in cilk. *SINEWS: SIAM News*, 31(4):6–7, 1998.

[17] Chris Lomont. Introduction to intel advanced vector extensions. *Intel White Paper*, 2011.

[18] Optimization Manual. Intel architecture optimization manual. 1996.

[19] Deborah T Marr, Frank Binns, David L Hill, Glenn Hinton, David A Koufaty, J Alan Miller, and Michael Upton. Hyper-threading technology architecture and microarchitecture. *Intel Technology Journal*, 6(1), 2002.

[20] Friedemann Mattern. Algorithms for distributed termination detection. *Distributed computing*, 2(3):161–175, 1987.

[21] Zviad Metreveli, Nickolai Zeldovich, and M Frans Kaashoek. Cphash: A cache-partitioned hash table. In *ACM SIGPLAN Notices*, volume 47, pages 319–320. ACM, 2012.

[22] Daniel Molka, Daniel Hackenberg, Robert Schöne, and Wolfgang E Nagel. Cache coherence protocol and memory performance of the intel haswell-ep architecture. In *Parallel Processing (ICPP), 2015 44th International Conference on*, pages 739–748. IEEE, 2015.

[23] Aske Plaat, Jonathan Schaeffer, Wim Pijls, and Arie de Bruin. Best-first fixed-depth minimax algorithms. *Artificial Intelligence*, 87(1):255–293, 1996.

[24] James Reinders. *Intel threading building blocks: outfitting C++ for multi-core processor parallelism.* " O'Reilly Media, Inc.", 2007.

[25] John W Romein and Henri E Bal. Wide-area transposition-driven scheduling. In *High Performance Distributed Computing, 2001. Proceedings. 10th IEEE International Symposium on*, pages 347–355. IEEE, 2001.

[26] John W Romein, Henri E Bal, Jonathan Schaeffer, and Aske Plaat. A performance analysis of transposition-table-driven work scheduling in distributed search. *Parallel and Distributed Systems, IEEE Transactions on*, 13(5):447–459, 2002.

[27] John W Romein, Aske Plaat, Henri E Bal, and Jonathan Schaeffer. Transposition table driven work scheduling in distributed search. In *AAAI/IAAI*, pages 725–731, 1999.

[28] William E Story. Notes on the" 15" puzzle. *American Journal of Mathematics*, 2(4):397–404, 1879.

[29] Vincent Vidal, Simon Vernhes, and Guillaume Infantes. Parallel ai planning on the scc. In *4th Many-core Applications Research Community (MARC) Symposium*, page 15, 2012.

[30] Kazuki Yoshizoe, Akihiro Kishimoto, Tomoyuki Kaneko, Haruhiro Yoshimoto, and Yutaka Ishikawa. Scalable distributed monte-carlo tree search. In *Fourth Annual Symposium on Combinatorial Search*, 2011.

[31] Albert L Zobrist. A new hashing method with application for game playing. *ICCA journal*, 13(2):69–73, 1970.