Early Experiences on Accelerating Dijkstra's Algorithm Using Transactional Memory

Nikos Anastopoulos, Konstantinos Nikas, Georgios Goumas and Nectarios Koziris

National Technical University of Athens
School of Electrical and Computer Engineering
Computing Systems Laboratory
{anastop,knikas,goumas,nkoziris}@cslab.ece.ntua.gr

Abstract

In this paper we use Dijkstra's algorithm as a challenging, hard to parallelize paradigm to test the efficacy of several parallelization techniques in a multicore architecture. We consider the application of Transactional Memory (TM) as a means of concurrent access to shared data and compare its performance with straightforward parallel versions of the algorithm based on traditional synchronization primitives. To increase the granularity of parallelism and avoid excessive synchronization, we combine TM with Helper Threading (HT). Our simulation results demonstrate that the straightforward parallelization of Dijkstra's algorithm with traditional locks and barriers has, as expected, disappointing performance. On the other hand, TM by itself is able to provide some performance improvement in several cases, while the version based on TM and HT exhibits a significant performance improvement that can reach up to a speedup of 1.46.

1 Introduction

Dijkstra's algorithm [8] is a fundamental graph algorithm used to compute single source shortest paths (SSSP) for graphs with non-negative edges. SSSP is a classic combinatorial optimization problem used in a variety of applications such as network routing or VLSI design. The algorithm maintains a set S of visited nodes, whose shortest path has already been calculated. In each iteration, the unvisited node with the shortest distance from the set S is selected and inserted into S and the distances of its neighbors are updated. The set of unvisited nodes is implemented as a priority queue. This serializes a large part of the algorithm's operations, thus making Dijkstra a hard to parallelize graph algorithm [5, 16].

Delving into implementation details, the algorithm involves a two-level nested loop: the outer loop iterates over all the nodes of the graph, selecting in each step the one closest to set S, while the inner loop updates the distances from S of all the neighbors of the extracted node. To implement parallel versions, researchers follow two general strategies. The first strategy attempts to relax the sequential nature of Dijkstra by creating more parallelism in the outer loop. This leads to alternative algorithms like Δ-stepping [13, 16] that enable concurrent extraction of multiple nodes from the unvisited set. The second strategy works on pure Dijkstra and seeks parallelism in the inner loop, by enabling concurrent acceses to the priority queue. However, practical implementations of concurrent binary heaps as priority queues [12] are based on unavoidable fine-grain locking of the binary heap, which is expected to kill the performance of such a scheme.

In this paper we face the challenges of parallelizing Dijkstra's algorithm for a multicore architecture. To decrease the synchronization cost we employ *Transactional Memory (TM)* [2, 11] as a means of efficient concurrent thread accesses to shared data. TM is a novel programming model for multicore architectures that allows concurrency control over multiple threads. The programmer is able to envelop parts of the code within a transaction, indicating that within this section exist accesses to memory locations that may be performed by other threads as well. The TM system monitors the transactions of the threads and, if two or more of them perform conflicting memory accesses, it resolves this conflict. TM seems a promising approach for dynamic data structures and applications with independent threads. It remains though to be investigated how TM can speedup a single application.

The parallelization of the inner loop does not exploit a significant amount of parallelism. Therefore, we choose to coarsen the granularity of parallelism by employing the idea of *Helper Threads* (HT) [7, 20]. Dijkstra's algorithm spends a large part of its execution in the relaxations of the nodes of the priority queue. Parallel threads can update (relax) the distances of several nodes' neighbors without changing the semantics of the algorithm. Thus, while the main thread extracts and updates the neighbors of the head of the priority queue, k helper threads update the neighbors of the next k nodes in the priority queue. This approach exploits parallelism in the outer loop, without changing the algorithm.

We have implemented several versions of the multithreaded Dijkstra algorithm using traditional synchronization primitives (locks and barriers), TM and HT and evaluated them using Simics [14] and GEMS [1, 15], which allow the simulation of multicore systems and provide support for TM. Our results demonstrate that the combination of TM and HT achieves significant speedup on a hard to accelerate application, while requiring only a few extensions to the original source code.

The rest of the paper is organized as follows. Section 2 presents the basics of Dijkstra's algorithms and the details of the various multithreaded implementations. Section 3 demonstrates simulation results comparing the performance of the versions under consideration. Related work is presented in Section 4, while Section 5 summarizes the paper and discusses directions for future work.

2 Parallelizing Dijkstra's algorithm

2.1 Dijkstra's algorithm

Dijkstra's algorithm solves the SSSP problem for a directed graph with non-negative edge weights. Specifically, let $G = (V, E)$ be a directed graph with $n = |V|$ vertices, $m = |E|$ edges, and $w : E \rightarrow \mathbf{R}^+$ a weight function assigning non-negative real-valued weights to the edges of G. For each vertex v, the SSSP problem computes $\delta(v)$, the weight of the shortest path from a source vertex s to v. For each vertex v, Dijkstra's algorithm maintains a *shortest-path estimate* (or *tentative distance*) $d(v)$, which is an upper bound for the actual weight of the shortest path from s to v, $\delta(v)$. Initially, $d(v)$ is set to ∞ and through successive edge relaxations it is gradually decreased, converging to $\delta(v)$. The relaxation of an edge (v, w) sets $d(w)$ to $min\{d(w), d(v) + w(v, w)\}$, which means that the algorithm tests whether it can decrease the weight of the shortest path from s to w by going through v.

The algorithm maintains a partition of V into *settled* (visited), *queued* and *unreached* vertices (the latter two representing unvisited nodes). Settled vertices have $d(v) = \delta(v)$; queued have $d(v) > \delta(v)$ and $d(v) \neq \infty$; unreached have $d(v) = \infty$. Initially, only s is queued, $d(s) = 0$ and all other vertices are unreached. In each iteration of the algorithm, the vertex with the smallest shortest-path estimate is selected, its state is permanently changed to settled and all its outgoing edges are relaxed, causing any of its neighbors that were unreached by the source vertex until this point to become queued. The algorithm is presented in more detail in Figure 1a.

The basic data structure lying at the heart of Dijkstra's algorithm is a min-priority queue of vertices, keyed by their $d(\cdot)$ values. The queue is used to maintain all but the settled vertices of the graph. At each iteration, the vertex with the smallest key is removed from the queue (`ExtractMin` operation) and its outgoing edges are relaxed, which could result to reductions of the keys of the corresponding neighbors (`DecreaseKey` operation). To amortize the cost of the multiple `ExtractMin` and `DecreaseKey` operations, especially for realistic, sparse graphs, the min-priority queue is implemented as a binary heap.

2.2 Lock-based parallel implementation

An intuitive choice for parallelizing Dijkstra's algorithm is to exploit parallelism at the inner loop by relaxing all outgoing edges of vertex u in parallel. This is a fine-grain parallelization scheme. In each step, one thread extracts u from the heap and then its outgoing edges are assigned (e.g. via cyclic assignment) to parallel threads for relaxation. This idea is depicted in Figure 2 while a generic implementation is shown in Figure 1b.

A number of observations can be made concerning this parallelization scheme. First, the speedup is bounded by the average out-degree of the vertices, i.e. the density of the graph. Clearly, if vertices have on average a small number of neighbors, then the parallel segment of the algorithm (lines 6–14) will consume a small fraction of the total execution time, making the sequential part (lines 3–4, `ExtractMin`) dominant.

Algorithm 1: Dijkstra's algorithm.

```
   Input   : Directed graph $G = (V, E)$, weight function $w : E \rightarrow \mathbf{R}^+$,
             source vertex $s$, min-priority queue $Q$
   Output  : shortest distance array $d$, predecessor array $\pi$
   /* Initialization phase */
 1 foreach $v \in V$ do
 2     $d[v] \leftarrow$ INF;
 3     $\pi[v] \leftarrow$ NIL;
 4     Insert($Q, v$);
 5 end
 6 $d[s] \leftarrow 0$;
   /* Main body of the algorithm */
 7 while $Q \neq \emptyset$ do
 8     $u \leftarrow$ ExtractMin($Q$);
 9     foreach v adjacent to u do
10         $sum \leftarrow d[u] + w(u, v)$;
11         if $d[v] > sum$ then
12             DecreaseKey($Q, v, sum$);
13             $d[v] \leftarrow sum$;
14             $\pi[v] \leftarrow u$;
15     end
16 end
```

(a)

Algorithm 2: Fine-grain parallel implementation of Dijkstra's algorithm.

```
   Input   : Directed graph $G = (V, E)$, weight function $w : E \rightarrow \mathbf{R}^+$,
             source vertex $s$, min-priority queue $Q$
   Output  : shortest distance array $d$, predecessor array $\pi$
   /* Initialization phase same to the serial code */
   /* Main body of the algorithm */
 1 while $Q \neq \emptyset$ do
 2     Barrier
 3     if $tid = 0$ then
 4         $u \leftarrow$ ExtractMin($Q$);
 5     Barrier
 6     foreach v adjacent to u do in parallel
 7         $sum \leftarrow d[u] + w(u, v)$;
 8         if $d[v] > sum$ then
 9             Begin-Atomic
10             DecreaseKey($Q, v, sum$);
11             End-Atomic
12             $d[v] \leftarrow sum$;
13             $\pi[v] \leftarrow u$;
14     end
15 end
```

(b)

Figure 1: Serial and multithreaded implementations of Dijkstra's algorithm.

The second observation concerns the concurrent accesses to the binary heap by the parallel `DecreaseKey` operations. The binary heap is implemented as a linear array and can be considered as a nearly complete binary tree. The smallest element in the heap is stored at the root and the subtree rooted at a node contains values no smaller than the value of the node. During a `DecreaseKey` operation, a vertex obtains a smaller value as its new, updated shortest path estimate. If this new value is smaller than that of its parent, the vertex has to move upwards the tree until it is placed to a location that satisfies the min-heap property. During this traversal, the node is repeatedly compared to its parent and if its value is smaller, the nodes are swapped.

The first, and rather naive, approach to enable parallelization of the relaxation phase, is to use a global mutex to lock the entire heap during each `DecreaseKey` operation. This constitutes a conservative, coarse-grain synchronization scheme that permits only one `DecreaseKey` operation at a time and obviously limits concurrency. We refer to this scheme as *cgs-lock*. The alternative, more optimistic approach is to allow multiple sequences of node swaps to execute in parallel as long as they access different parts of the heap. More specifically, instead of using one lock for the entire heap, one can utilize separate locks for each parent-child pair of nodes. Whenever a thread executes a `DecreaseKey` operation and a node swap is required, it must first acquire the appropriate lock that guards this specific pair of nodes (a scheme similar to [12]). In this way atomicity is guaranteed and the algorithm can be executed safely in parallel. We refer to this scheme as *fgs-lock*.

To obtain a a first picture of the efficiency of these schemes, we simulated their execution on Dijkstra's algorithm for a graph with 10K vertices and 100K edges, which were randomly added between pairs of vertices. A detailed description of the simulation framework can be found in Section 3.1. Figure 3 demonstrates the speedup of the two schemes for 2 to 16 threads. The speedup is calculated as the ratio of the execution time (in terms of cycles) of the serial to the parallel scheme in each case. The performance of the *cgs-lock* scheme is disappointing. Although the limited parallelism of the scheme explains the lack of speedup, a more detailed execution profiling revealed that the vast performance drop is attributed to the overhead of barriers that surround the `ExtractMin` operation and decouple the serial phases from the parallel ones. More specifically, for 2 threads the time spent in barriers accounts for 71% of the total execution time. This percentage rises up to 88% when using 8 threads, explaining why the performance degrades when more threads are used. We used the barriers provided by the Pthreads library, yet we argue that this should not be a problem of the specific barrier implementation, since alternative software-based implementations are expected to provide similar results.

In an attempt to isolate the effect of the barriers, we implemented a version of idealized, zero-latency barriers that rely solely on hardware in our simulated environment. This scheme is named *perfbar+cgs-lock*. It is clear from Figure 3 that the replacement of barriers with "perfect" ones deals with the poor

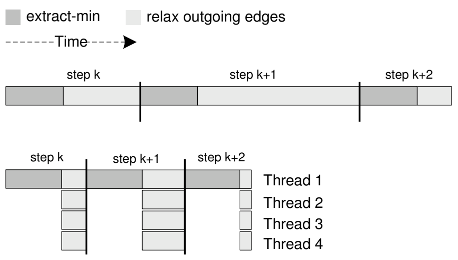

Figure 2: Execution patterns of serial and multithreaded Dijkstra's algorithm.

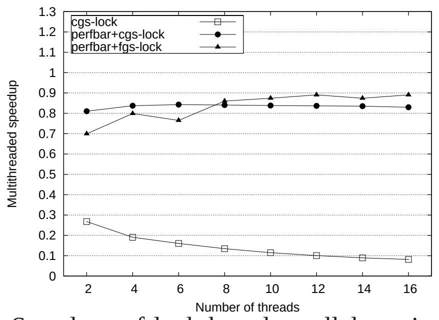

Figure 3: Speedups of lock-based parallel versions with real and perfect barriers.

scalability problem of the *cgs-lock* scheme. Nevertheless, the scheme still performs worse than the serial execution of the algorithm, revealing that this coarse-grain synchronization is too conservative and cannot expose enough parallelism.

Finally, the *fgs-lock* scheme combined with "perfect" barriers fails to outperform the serial execution, despite being more optimistic than the *cgs-lock* scheme. As the number of threads increases, its performance improves slightly indicating that there does exist an amount of parallelism. However, the *fgs-lock* scheme fails to exploit it efficiently and there are two possible reasons for this failure. The first reason is that in order to allow concurrent accesses to the heap, a pair of spin-locks is used for each pair of nodes, causing the total overhead to be high and lowering the performance gains from the exploited parallelism. The second reason is that the *fgs-lock* scheme allows concurrent accesses to the binary heap only when the threads access different parts of the heap. The probability of threads touching the same nodes of the binary heap depends on the structure of the graph as well as on the order by which the neighbors of a vertex are examined during the `DecreaseKey` operations. Whenever this occurs, the threads are serialized, thus limiting the total available parallelism.

2.2.1 TM-based parallel implementation

The *fgs-lock* scheme described in Section 2.2 allows concurrent accesses to the binary heap used in the implementation of Dijkstra's algorithm. Unfortunately, it has a high overhead due to the numerous locks needed limiting its efficiency severely. Looking for alternatives, we test the efficacy of TM, as a means of concurrent accesses to the shared binary heap. The first approach is to enclose each `DecreaseKey` operation within a transaction, and rely on the underlying system to ensure atomicity. When concurrent transactions access the same elements in the heap and at least one of these accesses is a write operation, a conflict arises and the system needs to resolve it deciding which transaction succeeds. The `DecreaseKey` operation includes a series of swaps, as a node traverses the heap until it is placed in the final correct position. When two or more vertices are relaxed in parallel, the paths of these heap traversals might share one or more common nodes. The TM system will detect the conflict, only one of the transactions will be allowed to commit and only one vertex will be relaxed. The other conflicting threads will have to pause or repeat their work, depending on the implementation of the TM system and its conflict detection and resolution policy. This scheme is not as fine-grain as the *fgs-lock* scheme, where atomicity is enforced at the level of a single swap and not for a series of swaps. We will refer to this scheme as *cgs-tm*. It is implemented as shown in Figure 1b by replacing the `Begin-Atomic` and `End-Atomic` operations with the appropriate `Begin-Transaction` and `End-Transaction` primitives.

A second alternative is to implement a scheme as fine-grain as the *fgs-lock* scheme using TM. To accomplish this, each swap executed by the `DecreaseKey` operation is enclosed into a transaction. This means that the transactions will be shorter than those of the *cgs-tm* scheme resulting, hopefully, into fewer conflicts and thus more parallelism. We will refer to this scheme as *fgs-tm*. For the implementation of the *fgs-tm* scheme the `DecreaseKey` presented in Figure 4 is used. Similarly to the lock-based schemes, both these TM-based schemes require the incorporation of barriers to decouple the serial from the parallel phases. Having observed the disastrous effect of the barriers on the performance of the lock-based schemes, we employ the “perfect”, zero-latency barriers in the evaluation of our TM-based implementations as well.

Algorithm 3: DecreaseKey

```
Input : min-priority queue $Q$, vertex $u$, new key $value$ for vertex $u$
1 $Q[u] \leftarrow value$;
2 $i \leftarrow u$;
3 while $(parent(i).key \geq value)$ do
4     Begin-Transaction
5     $swap(u, i)$;
6     End-Transaction
7     $i \leftarrow parent(i)$;
8 end
```

Figure 4: `DecreaseKey` implementation for *fgs-tm* scheme.

To obtain a first insight on the efficiency of the TM-based schemes we used the same graph as in Section 2.2 and present speedups in Figure 5. In contrast to the lock-based schemes, the TM-based ones outperform the serial implementation for more than 4 threads. For 2 threads the overhead of the TM scheme seems to be too high, cancelling out any performance gains from the exploitation of parallelism. As more threads are used though, the performance is improved providing a speedup of up to almost 1.1. More detailed results are presented in Section 3.3. Thus, it seems that TM is a promising mechanism to exploit the available parallelism of the `DecreaseKey` operations on the binary heap. However, to obtain these performance improvements ideal barriers are employed.

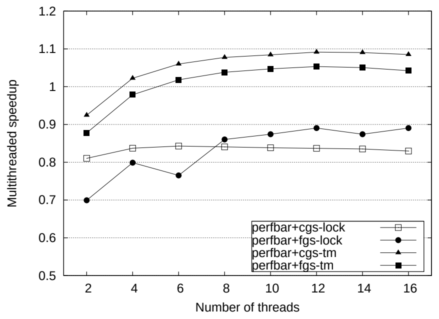

Figure 5: Speedups of lock-based and TM-based versions.

2.3 A multithreaded version based on HT

In this section we present an alternative multithreaded version of Dijkstra’s algorithm based on helper threading. The motivation arises from the poor performance of all aforementioned versions, which is due to their limited parallelism and excessive synchronization. Our goal is to coarsen the granularity of parallelism, as in [10, 13, 16], without, changing the algorithm itself. Thus, instead of partitioning the inner loop and assigning only a few neighbors to each thread, we seek to assign the relaxation of a complete set of neighbors to each thread. To accomplish this, we take advantage of a basic property of Dijkstra’s algorithm: the relaxations (lines 11,13–14 in Figure 1a) lead to monotonically decreasing values for the distances of unvisited nodes until each distance reaches its final minimum value. As long as a graph node is inserted in the queued set (i.e. the node’s distance from S is not infinite) its neighbors could also be relaxed to newer updated values. This property is not utilized by the original serial algorithm, since all the updates occur for the neighbors of the extracted node. Practically, the algorithm avoids calculating intermediate distances that will eventually be overwritten. Our key idea is that parallel threads can serve as *helper threads* and perform relaxations for neighbors of nodes belonging in the queued set. Optimistically, some of these relaxations will be utilized and offloaded by the *main thread*.

In our implementation the main thread operates like in the sequential version, *extracting* in each iteration the minimum vertex from the priority queue and relaxing its outgoing edges. At the same time, the k-th helper thread *reads* the tentative distance of the k-th vertex in the queue (let us call it x_k for short) and relaxes all its outgoing edges based on this value. When the main thread accomplishes all relaxations, it notifies the helper threads to stop their relaxations, and they all proceed to the next iteration. This scheme is demonstrated in Figure 6. The rationale behind it is that vertices occupying the top k positions in the queue might already be settled with some probability, so that when the helper threads read their distances and relax their outgoing edges, they will make their corresponding neighbors settled, as well. As a result, when the main thread checks these vertices later, it will not have to perform any re-

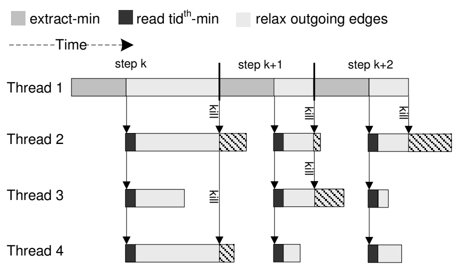

Figure 6: Execution pattern of the helper threads version.

laxations. On the other hand, if the k-th thread reads x_k, it is possible that x_k might not have been settled yet and thus have a suboptimal tentative distance. The thread would then update the neighbors according to a new tentative value, which will eventually be set to the appropriate minimum value, when x_k will be examined by the main thread later on. At that moment, all its outgoing edges will be re-relaxed using the correct final distance. A significant aspect of this multithreaded scheme is that the main thread stops all helper threads after finishing each iteration of the outer loop. At this time, the helper threads stop their computations and proceed with the main thread to the next iteration. It is possible that at this time a helper thread might have updated only some of the neighbors of its vertex x_k, leaving the other ones with their old, possibly suboptimal, distances. As explained above, however, this is not a problem since all neighbors of x_k with suboptimal distances will be correctly updated when x_k reaches the top of the priority queue.

The code executed by the main and helper threads is shown in Figure 7a and Figure 7b, respectively. In the beginning of each iteration, the main thread extracts the top vertex from the queue. At the same time, the helper threads spin-wait until the main thread has finished the extraction, and then each one reads –without extracting– one of the top k vertices in the queue (this is what `ReadMin` function does). Next, all threads relax all the outgoing edges of the vertices they have undertaken in parallel. Note that, compared to the original algorithm, a performance improvement is expected, since, due to the work done by the helper threads, the main thread will evaluate the expression of line 7 as true fewer times and thus, will not need to execute the operations of lines 8–9.

The proposed helper threading scheme is largely based on TM. Updates to the heap via the `DecreaseKey` function, as well as updates to the tentative distances and predecessor arrays are enclosed within a single transaction for both the main and helper threads. This ensures atomicity of these updates, i.e. that they will be performed in an “all-or-none” manner. Furthermore, it guarantees that in case of conflict only one thread will be allowed to commit its transaction and perform the neighbor update. A conflict can arise when two or more threads update simultaneously the same neighbor, or when they update different neighbors but change the same part of the heap. The interruption of helper threads is implemented using transactions as well. Specifically, when the main thread has completed all the relaxations for its vertex, it sets the notification variable `done` to 1 within a separate transaction. This value denotes a state where the main thread proceeds to the next iteration and requires all helper threads to stop and follow, terminating any operations that they were performing on the heap. All helper threads executing transactions at this point will abort, since `done` is in their read sets as well. The helper threads will immediately retry their transactions, but there is a good chance that they will find `done` set to 1, stop examining the remaining neighbors in the inner loop and continue with the next iteration of the outer loop. In the opposite case that the main thread performs the `ExtractMin` operation too quickly, `done` will be set back to 0 and the helper threads will miss the last notification, continuing from the point where they had stopped. This might yield suboptimal updates to the distances of the neighbors, but as explained above, these will be overwritten once the vertices examined by the helper threads reach the top of the queue. So, correctness is guaranteed.

Summarizing, the main concept of our implementation is to decouple as much as possible the main thread from the ex-

Algorithm 4: Main thread’s code.

```
Input  : Directed graph $G = (V, E)$, weight function $w : E \to \mathbf{R}^+$,
         source vertex $s$, min-priority queue $Q$
Output : shortest distance array $d$, predecessor array $\pi$
/* Initialization phase same to the serial code */
1  while $Q \neq \emptyset$ do
2      $u \leftarrow$ ExtractMin($Q$);
3      $done \leftarrow 0$;
4      foreach $v$ adjacent to $u$ do
5          $sum \leftarrow d[u] + w(u, v)$;
6          Begin-Transaction
7          if $d[v] > sum$ then
8              DecreaseKey($Q, v, sum$);
9              $d[v] \leftarrow sum$;
10             $\pi[v] \leftarrow u$;
11         End-Transaction
12     end
13     Begin-Transaction
14     $done \leftarrow 1$;
15     End-Transaction
16 end
```

(a)

Algorithm 5: Helper threads’ code.

```
1  while $Q \neq \emptyset$ do
2      while $done = 1$ do ;
3      $x \leftarrow$ ReadMin($Q, tid$);
4      $stop \leftarrow 0$;
5      foreach $y$ adjacent to $x$ and while $stop = 0$ do
6          Begin-Transaction
7          if $done = 0$ then
8              $sum \leftarrow d[x] + w(x, y)$;
9              if $d[y] > sum$ then
10                 DecreaseKey($Q, y, sum$);
11                 $d[y] \leftarrow sum$;
12                 $\pi[y] \leftarrow x$;
13         else
14             $stop \leftarrow 1$;
15         End-Transaction
16     end
17 end
```

(b)

Figure 7: Multithreaded implementation of Dijkstra’s algorithm based on Helper Threading.

Simics	Processor	configurations up to 32 cores UltraSPARC III Cu (III+)
Ruby	L1 caches	Private, 64KB, 4-way set-associative, 64B line size, 4 cycle hit latency
	L2 cache	Unified and shared, 8 banks, 2MB, 4-way set-associative, 64B line size, 10 cycle hit latecny
	Memory	160 cycle access latency
	TM System	HYBRID resol. policy, 2Kb HW signatures

Table 1: Simulation framework.

ecution of the helper threads, minimizing the time that it has to spend on synchronization events or transaction aborts. The helper threads are allowed to execute in an aggressive manner, being at the same time as less intrusive to the main thread as possible, even if they perform a notable amount of useless work. The semantics of the algorithm guarantee that any intermediate relaxations made by the helper threads are not irreversible. Finally, by using a TM with a conflict resolution policy that favors the main thread, transaction abort overheads are mainly suffered by the helper threads.

3 Experimental Evaluation

3.1 Experimental setup

We evaluated the performance of the various implementations of Dijkstra's algorithm through full-system simulation, using the Wisconsin GEMS toolset v.2.1 [1, 15] in conjunction with the Simics v.3.0.31 [14] simulator. Simics provides functional simulation of a SPARC chip multiprocessor system (CMP) that boots unmodified Solaris 10. The GEMS Ruby module provides detailed memory system simulation and for non-memory instructions behaves as an in-order single-issue processor, executing one instruction per simulated cycle.

Hardware TM is supported in GEMS through the LogTM-SE subsystem [19]. It is built upon a single-chip CMP system with private per-processor L1 caches and a shared L2 cache. It features *eager version management*, where transactions write the new memory values "in-place", after saving the old values in a log. It also supports *eager conflict detection*, as conflicts, i.e. overlaps between the write set of one transaction and the write or read set of other concurrent transactions, are detected at the very moment they happen. On a conflict, the offending transaction stalls and either retries its request hoping that the other transaction has finished, or aborts if LogTM detects a potential deadlock. The aborting processor uses its log to undo the changes it has made and then retries the transaction. In our experiments we used the `HYBRID` conflict resolution policy, which tends to favor older transactions against younger ones. Table 1 shows the configuration of the simulation framework.

For our programs we used the Pthreads library for thread creation and conventional synchronization operations such as spin-locking and barriers. For our "perfect" barriers, we encoded a global barrier as a single assembly instruction, exploiting the functionality offered by Simics' magic instructions.

Graph	*Edges*	*Parameters*	*Sequential part (%)*	*Parallel. part (%)*	*Ideal speedup*
rand1	10K		97.8	2.2	1.02
rand2	100K		38.7	61.3	2.58
rand3	200K		26.2	73.8	3.81
rmat1	10K	$a = 0.45$	78.3	21.7	1.27
rmat2	100K	$b, c = 0.15$	39.3	60.7	2.54
rmat3	200K	$d = 0.25$	26.8	73.2	3.73
ssca1	28K	$(P, C) = (0.25, 5)$	94.1	5.9	1.06
ssca2	118K	$(P, C) = (0.5, 20)$	35.2	64.8	2.84
ssca3	177K	$(P, C) = (0.5, 30)$	28.9	71.1	3.46

Table 2: Graphs used for experiments.

Thus the synchronization of threads is handled by the simulator and not the operating system, providing instant suspension/resumption of the arriving/departing threads. To avoid resource conflicts between our programs and the operating system's processes as much as possible, we used CMP configurations with more processor cores than the number of threads we required. So, the experiments for 2 and 4 threads were performed on an 8 core CMP, while the 8 threads experiments were done on an 16 core CMP. To schedule a thread on a particular processor and avoid migrations, we used the `pset_bind` system call of Solaris. Finally, all codes were compiled with the C compiler of Sun Studio 12 using the O3 optimization level.

3.2 Reference graphs

To evaluate the different schemes we strived to work on graphs which vary in terms of density and structure. In that attempt, we used the GTgraph graph generator [4] to construct graphs with $10K$ vertices from the following families:

Random: Their m edges are constructed choosing a random pair among n vertices.

R-MAT: Constructed using the Recursive Matrix (R-MAT) graph model [6].

SSCA#2: Used in the DARPA HPCS SSCA#2 graph analysis benchmark [3].

Table 2 summarizes the characteristics of the graphs used. To obtain an estimate of possible speedups, we profiled the serial execution of Dijkstra's algorithm on each graph in order to calculate the distribution of the sequential, i.e `ExtractMin` operations, and the parallelizable parts, i.e `DecreaseKey` operations. In the ideal case where parallel execution would manage to zero out the time spent for edge relaxations, the speedup would be $\frac{100\%}{\%Serialpart}$. This is presented in the sixth column of Table 2 and constitutes a theoretical upper bound for any performance improvement.

3.3 Results

Figure 8 shows the speedups of all evaluated schemes. Consistently to the discussion in Section 2, the concurrent thread accesses to shared data implemented with the aid of TM (*cgs-tm* and *fgs-tm*) clearly outperforms the ones using traditional

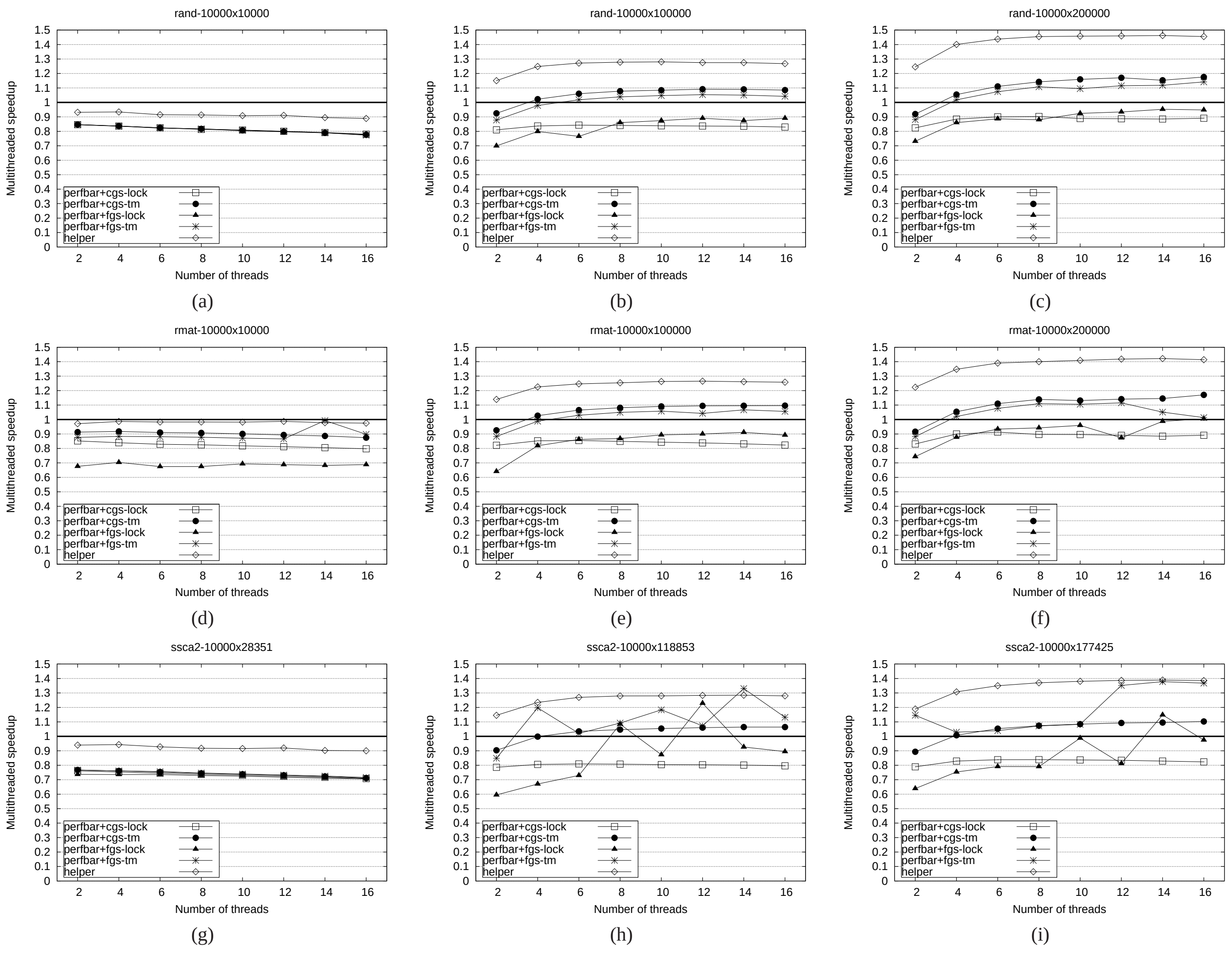

Figure 8: Multithreaded speedups for the graphs tested.

synchronization primitives (*cgs-lock* and *fgs-lock*). This fact reveals the existence of fine-grain parallelism in the updates of the priority queue of the algorithm, in the sense that, statistically, it is highly probable that the paths of various concurrent updates do not overlap. Thus, optimistic parallelism seems a good approach for Dijkstra's algorithm.

Nevertheless, only by employing "perfect" barriers are the TM-based schemes able to outperform the serial case. Therefore, the fine-grain parallelism exposed by the inner loop of the algorithm, is not sufficient to achieve significant performance improvement. On the contrary, the helper threading scheme (*helper*), which exploits parallelism at a coarser granularity, is able to achieve significant speedups in the majority of the cases (6 out of 9 experiments). The maximum speedup achieved is 1.46 as shown in Figure 8c.

For more dense graphs, the performance improvements are greater since more parallelism can be exposed in the inner loop of the algorithm. These are the cases where *helper* achieves the best speedups and scalability (Figures 8c, 8f and 8i). Conversely, sparse graphs leave limited space for parallelism leading to low performance. Therefore they can serve as test cases for the overhead of the co-existence of numerous threads. The results for these cases are shown in Figures 8a, 8d and 8g. It is obvious that the *helper* scheme is the more robust one, as it exhibits the smallest slowdown. In the worst case, the performance of the main thread is degraded by around 10%.

A closer look at the results reveals that the main thread suffers a really low number of aborts (less than 1% of the total aborts). This means that even when the helper threads are not contributing any useful work, they still do not obstruct the main thread's progress. Therefore, the main thread is allowed to run almost at the speed of the serial execution, thus explaining the robustness of the scheme. The low overhead of the *helper* scheme is also illustrated by the fact that the addition of more threads does not lead to performance drops in any case.

4 Related Work

A significant part of Dijkstra's execution is spent in updates in the priority queue. Therefore, enabling concurrent accesses to this structure seems a good approach to increase performance. Brodal et al. [5] utilize a number of processors to accelerate the `DecreaseKey` operation and discuss the applicability of their approach to Dijkstra's algorithm. However, this work is evaluated on a theoretical Parallel Random Access Machine (PRAM) execution model. Hunt et al. [12] implement a concurrent priority queue which is based on binary heaps and supports parallel Insertions and Deletions using fine-grain locking on the nodes of the binary heap. Since these operations do not traverse the entire data structure, local locking leads to performance gains. However, in the case of `DecreaseKey` which performs wide traversals of the data structure it degrades performance greatly, unless special hardware synchronization is supported by the underlying platform.

To expose more parallelism, it would be beneficial to concurrently extract a large number of nodes from the priority queue. This can be achieved if several nodes have equal distances from the set S of visited nodes. Thus, if the priority queue is organized into buckets of nodes with equal distances, then the extraction and neighbor updates can be done in parallel per bucket (Dial's algorithm [9]). A generalization of Dial's algorithm called Δ-stepping is proposed by Meyer and Sanders [16]. Madduri et al. [13] use Δ-stepping as the base algorithm on Cray MTA-2, an architecture that exploits fine-grain parallelism using hardware synchronization primitives, and achieve significant speedups. In the Parallel Boost Graph Library [10] Dijkstra's algorithm is parallelized for a distributed memory machine. The priority queue is distributed in the local memories of the system nodes and the algorithm is divided in supersteps, in which each processor extracts a node from its local priority queue. The aforementioned approaches are based on significant modifications to Dijkstra's algorithm to enable coarse-grain parallelism and lead to promising parallel implementations. In this paper we adhere to the pure Dijkstra's algorithm to face the challenges of its parallelization and test the applicability of TM and HT.

TM has attracted extensive scientific research during the last few years, focusing mainly on its design and implementation details in software and hardware. Nevertheless, its efficacy on a wide set of real, non-trivial applications is only now starting to be explored. Scott et al. [17] use TM to parallelize Delaunay triangulation and Watson et al. [18] exploit it to parallelize Lee's routing algorithm. Moreover, a set of TM applications is offered in STAMP [21].

5 Conclusions – Future work

This work applies several parallelization techniques to Dijkstra's algorithm, which is known to be hard to parallelize. The schemes that parallelize each serial step by incorporating traditional synchronization primitives (locks and barriers) fail to outperform the serial algorithm. In fact, they exhibit low performance even if the necessary barriers are replaced by ideal ones. To deal with this we employ Transactional Memory (TM), which reduces synchronization overheads, but still fails to provide meaningful overall performance improvement, as speedups can be achieved in some test cases only with using ideal barriers. To improve the performance further, we propose an implementation based on TM and Helper Threading, that is able to provide significant speedups (reaching up to 1.46) in the majority of the simulated cases.

As future work, we will investigate the application of these techniques on other algorithms solving the SSSP problem, such as Δ-stepping [16] and Bellman-Ford [8]. We also aim to explore the impact of various TM characteristics on the behavior of the presented schemes, such as the resolution policy, version management and conflict detection. Finally, preliminary results demonstrated interesting variations in the available parallelism between different execution phases, motivating us to explore more adaptive schemes in terms of the number of parallel threads and the tasks assigned to them.

References

[1] Wisconsin multifacet gems simulator. `http://www.cs.wisc.edu/gems/`.

[2] A.-R. Adl-Tabatabai, C. Kozyrakis, and B.E. Saha. Unlocking concurrency: Multicore programming with transactional memory. *ACM Queue*, 4(10):24–33, 2006.

[3] D.A. Bader and K. Madduri. Design and implementation of the hpcs graph analysis benchmark on symmetric multiprocessors. In *HiPC*, 2005.

[4] D.A. Bader and K. Madduri. Gtgraph: A suite of synthetic graph generators. 2006. `http://www.cc.gatech.edu/~kamesh/GTgraph/`.

[5] G.S. Brodal, J.L. Traff, C.D. Zaroliagis, and I. Stadtwald. A parallel priority queue with constant time operations. *Journal of Parallel and Distributed Computing*, 49:4–21, 1998.

[6] D. Chakrabarti, Y. Zhan, and C. Faloutsos. R-mat: A recursive model for graph mining. In *ICDM*, 2004.

[7] J. D. Collins, H. Wang, D. M. Tullsen, C. Hughes, Y. Lee, D. Lavery, and J. P. Shen. Speculative precomputation: Long-range prefetching of delinquent loads. In *ISCA*, 2001.

[8] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, 2001.

[9] R. Dial. Algorithm 360: Shortest path forest with topological ordering. *Communications of the ACM*, 12:632–633, 1969.

[10] N. Edmonds, A. Breuer, D. Gregor, and A. Lumsdaine. Single-source shortest paths with the parallel boost graph library. In *9th DIMACS Implementation Challenge*, 2006.

[11] M. Herlihy and E. Moss. Transactional memory: Architectural support for lock-free data structures. In *ISCA*, 1993.

[12] G.C. Hunt, M.M. Michael, S. Parthasarathy, and M.L. Scott. An efficient algorithm for concurrent priority queue heaps. *Inf. Proc. Letters*, 60:151–157, 1996.

[13] K. Madduri, D.A. Bader, J.W. Berry, and J.R. Crobak. Parallel shortest path algorithms for solving large-scale instances. In *9th DIMACS Implementation Challenge*, 2006.

[14] P.S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A full system simulation platform. *Computer*, 35(2):50–58, 2002.

[15] M. Martin, D. Sorin, B. Beckmann, M. Marty, M. Xu, A. Alameldeen, K. Moore, M. Hill, and D. Wood. Multifacet's general execution-driven multiprocessor simulator (gems) toolset. *SIGARCH Comput. Archit. News*, 33(4):92–99, 2005.

[16] U. Meyer and P. Sanders. Delta-stepping: A parallel single source shortest path algorithm. In *ESA*, 1998.

[17] M. L. Scott, M. F. Spear, L. Daless, and V. J. Marathe. Delaunay triangulation with transactions and barriers. In *IISWC*, 2007.

[18] I. Watson, C. Kirkham, and M. Lujan. A study of a transactional parallel routing algorithm. In *PACT*, 2007.

[19] L. Yen, J. Bobba, M.R. Marty, K.E. Moore, H. Volos, M.D. Hill, M.M. Swift, and D.A. Wood. Logtm-se: Decoupling hardware transactional memory from caches. In *HPCA*, 2007.

[20] W. Zhang, B. Calder, and D.M. Tullsen. An event-driven multithreaded dynamic optimization framework. In *PACT*, 2005.

[21] C. Cao Minh, J. Chung, C. Kozyrakis, and K. Olukotun STAMP: Stanford Transactional Applications for Multi-Processing. In *IISWC* 2008.