Employing Transactional Memory and Helper Threads to Speedup Dijkstra's Algorithm

Konstantinos Nikas, Nikos Anastopoulos, Georgios Goumas and Nectarios Koziris
National Technical University of Athens
School of Electrical and Computer Engineering
Computing Systems Laboratory
Members of HiPEAC
{knikas,anastop,goumas,nkoziris}@cslab.ece.ntua.gr

***Abstract*—In this paper we work on the parallelization of the inherently serial Dijkstra's algorithm on modern multicore platforms. Dijkstra's algorithm is a greedy algorithm that computes Single Source Shortest Paths for graphs with non-negative edges and is based on the iterative extraction of nodes from a priority queue. This property limits the explicit parallelism of the algorithm and any attempt to utilize the remaining parallelism results in significant slowdowns due to synchronization overheads. To deal with these problems, we employ the concept of Helper Threads (HT) to extract parallelism on a non-traditional fashion and Transactional Memory (TM) to efficiently orchestrate the concurrent threads' accesses to shared data structures. Results demonstrate that the proposed implementation is able to achieve performance speedups (reaching up to 1.84 for 14 threads), indicating that the two paradigms could be efficiently combined.**

I. Introduction

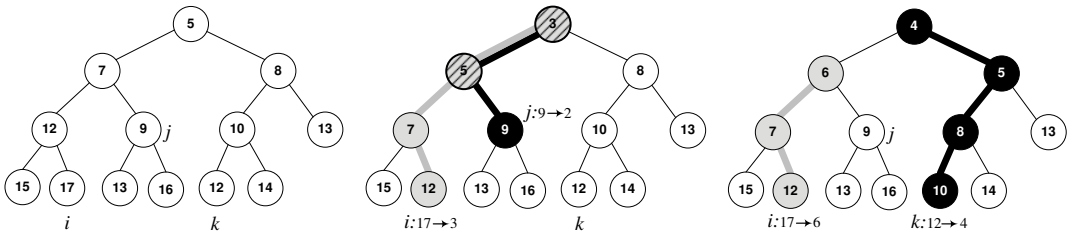
Parallel programming is a very intricate, yet increasingly important, task as we have entered the multicore era and more cores are made available to the programmer. Although separate applications or independent tasks within a single application can be easily mapped on multicore platforms, the same is not true for applications that do not expose parallelism in a straightforward way. Dijkstra's algorithm [1] is a challenging example of such an application that is difficult to accelerate when executed in a multithreaded fashion. It is a fundamental algorithm applied to compute single source shortest paths (SSSP) for graphs with non-negative edges and is used in a variety of applications, like network routing or VLSI design.

Dijkstra's algorithm iteratively extracts one node from a min-priority queue and performs relaxations to this node's neighbors. To preserve the semantics of the algorithm the extractions must be performed sequentially, a fact that greatly prohibits efficient parallelization [2], [3]. Straightforward parallelism can be sought in the relaxation of the neighbors, but this approach leads to significant performance slowdowns, since the threads need to synchronize their concurrent access to shared data very frequently [4]. Its fundamentally serial nature has led researchers to seek performance through significant modifications of the algorithm [3], [5], [6], [7]. However, in this work we adhere to the original version and attempt to improve its performance by utilizing the capabilities provided by modern multicore processors. To this direction, we need to face the two major issues inherent to the algorithm: *limited explicit parallelism* and *excessive synchronization*.

Since Dijkstra's algorithm does not favor the utilization of multiple symmetric threads in any standard parallelization scheme (e.g. data-parallel, task-parallel, pipeline), we elaborate on the concept of *Helper Threads (HT)* [8], [9] and test whether the incorporation of helper threads is a good strategy to provide performance speedups. The key idea is to employ a number of threads that will offload operations from the main thread in a transparent way.

To amortize the cost of excessive synchronization, we employ *Transactional Memory (TM)* [10], [11]. TM is a novel programming model for multicore architectures that allows concurrency control over multiple threads and is getting adopted by the industry, as it is demonstrated by Sun's coming processor Rock [12] or Intel's STM [13]. The programmer is offered the capability to envelop parts of the code within a transaction, indicating that some of the memory accesses in this code segment may be performed by other threads as well. The TM system monitors the transactions of the threads and if two or more perform conflicting memory accesses, it decides how to handle this conflict. The common case is to allow one thread to commit its transaction and restart the transaction(s) of the other conflicting thread(s). In the case of non-conflicting transactions, TM systems perform the appropriate accesses with (almost) no overhead. TM seems a promising approach which increases programmability while being capable of providing performance gains through the concept of optimistic parallelism. Therefore, if for a given problem the threads access the same memory location too rarely, then locking seems a pessimistic exaggeration, making TM a more appropriate approach. Lately, TM's usage in the parallelization of specific algorithms has attracted scientific attention [14], [15], [16], as its potential on the speedup of real-world applications is still under investigation.

The evaluation of our scheme demonstrates that the combination of the aforementioned approaches can provide speedups, while requiring only a few extensions to the original source code. The rest of the paper is organized as follows: Section II discusses the basics of Dijkstra's algorithm. Section III presents our scheme while Section IV presents its evaluation. Related work is presented in Section V and Section VI



(a) Min-priority queue as a binary heap. (b) Conflicting concurrent updates. (c) Non-conflicting concurrent updates.

Fig. 1: Min-priority queue and `DecreaseKey` operations.

summarizes the paper and discusses future work directions.

II. The basics of 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. The weight of the path is the sum of the weights of its edges. If v is not reachable from s, then $\delta(v) = \infty$. 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*, *queued* and *unreached* vertices. 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 Alg. 1.

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 maintains 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 time complexity of these operations, the min-priority queue is implemented as a binary heap. Thus, a `DecreaseKey` operation on a relaxed node involves an upward traversal of the heap with consecutive parent-child swaps, until the node reaches its correct position which satisfies the min binary heap's property, i.e. all children have a key value larger or equal to that of their parent. An example is shown in Fig. 1a.

The algorithm involves a two-level nested loop. The outer loop iterates over all the nodes and each time extracts the one closest to the settled set. It clearly prioritizes the nodes and thus, is inherently serial. The inner loop relaxes the neighbors of the extracted node. The order of the relaxations is irrelevant and thus, this loop is conceptually parallel.However, its operations include `DecreaseKey`, which means that the threads may need to modify the binary heap concurrently. Fig. 1b depicts how the parallel relaxations of two nodes can lead to conflicting `DecreaseKey` operations. In this example, the relaxation of node i causes its traversal to the root of the heap. If j is relaxed in parallel, a conflict arises as it tries to travel through the parts of the heap that i traverses.

Algorithm 1: Dijkstra's algorithm.

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

To preserve the semantics of the algorithm, we need to synchronize the threads' accesses to the heap. In [4] we evaluated two multithreaded versions of the algorithm, one based on a coarse-grain synchronization scheme which locks the entire binary heap and one based on a fine-grain synchronization scheme where the threads lock pairs of nodes. Note that, apart from the synchronized accesses to the priority queue, the threads need to synchronize further (e.g. with a barrier) at the end of their parallel relaxation phase, in order for the execution to proceed correctly to the next iteration of the outer loop. Due to this excessive synchronization, both versions exhibited poor performance, motivating us to look for alternatives.

III. Speeding up Dijkstra's algorithm

This section presents our scheme for parallelizing Dijkstra's algorithm. It tries to deal with the two major problems, the lack

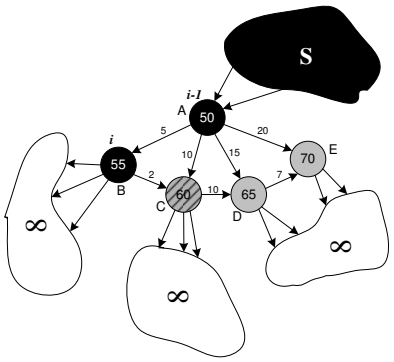


Fig. 2: Example of HT scheme's functionality.

of sufficient explicit parallelism and the synchronization costs.

A. Extracting more parallelism

As discussed in Section II, explicit parallelism exists only in the inner loop of Dijkstra's algorithm. Our goal is to coarsen the granularity of parallelism as in [3], [6], [7], without though changing the algorithm itself. Thus, instead of partitioning the inner loop and assigning only a few neighbors to each thread, we parallelize the outer loop by assigning the relaxation of a complete set of neighbors to each thread.

We specifically exploit the following basic property of Dijkstra's algorithm: the relaxations lead to monotonically decreasing values for the distances of unsettled nodes until each distance reaches its final minimum value. As long as a node is inserted in the queued set (i.e. its distance from S is not infinite) its neighbors can also be relaxed to newer updated values. This property is not utilized by the original algorithm, which avoids calculating intermediate distances that will eventually be overwritten by updating only the neighbors of the extracted node. Our key idea is that parallel threads can serve as *Helper Threads* and relax neighbors of nodes belonging to the queued set. Optimistically, the load corresponding to some of these relaxations will be taken off the *main thread*.

The rationale behind our scheme is that vertices occupying the top k positions in the queue might be, with some probability, already settled. When the helper threads read their distances and relax their outgoing edges, there is a high probability they will set their neighbors to settled as well. Thus, when the main thread checks these vertices later, it will avoid any further relaxations. On the contrary, if a helper thread reads a node that has not been settled yet, it will update its neighbors to suboptimal tentative values. When, though, the node is extracted by the main thread later on, all its outgoing edges will be re-relaxed using the correct final distance.

This is illustrated in Fig. 2, where the i-th iteration of the outer loop is depicted. In the previous step, node A was extracted and its neighbors were relaxed to the values shown. In the current step, the main thread extracts node B, while the helper threads are assigned the next three nodes in the priority queue, namely C, D and E. Thus, C's neighbors will be relaxed using value 60. However, at the end of this step, C's distance will be updated to 57 by the main thread. In step $i+1$ the main thread will extract C and relax again its neighbors

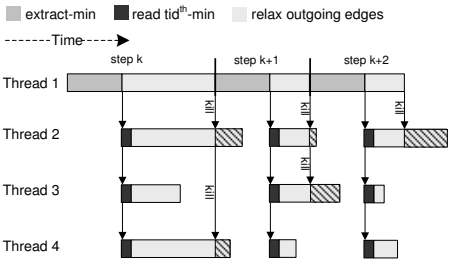


Fig. 3: Execution pattern of the HT scheme.

using now the correct distance. In this case, the helper thread's work has been wasted. On the contrary, the distances for nodes D and E will not change, as they obtained their minimum value in the $i-1$ step. Thus, in step i, the helper threads relax their neighbors correctly and when the main thread extracts them it will not have to perform any relaxations.

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 all 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 attempts to relax its outgoing edges based on this value. When the main thread accomplishes all its relaxations, it notifies the helper threads to stop their relaxations, and they all proceed to the next iteration. This execution pattern is illustrated in Fig. 3.

This orchestration by the main thread has a potential drawback. It is possible, that at this point a helper thread might have updated only some of the neighbors of its vertex x_k, leaving the rest 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.

B. Efficient Concurrency Control

In our scheme the threads need to access the binary heap as well as the data structures that implement the graph (lines 10–14 in Alg. 1) in parallel. For efficient concurrency control, we propose the use of Transactional Memory.

A TM system allows non-conflicting updates, like those shown in Fig. 1c, to occur in parallel with no overhead. At the same time, it guarantees atomicity, which means that if a conflict arises, it will allow one of the threads to update the heap (e.g. perform the traversal of node i in Fig. 1b) while the rest will have to repeat their work (e.g. relax node j in Fig. 1b). To implement this, we enclose each `DecreaseKey` operation within a transaction using the appropriate `Begin-Transaction` and `End-Transaction` primitives, as shown in Alg. 2 and Alg. 3 for the main and helper threads 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 one has finished the extraction, and then each one reads –without extracting– one of the top k vertices in the queue (implemented by the `ReadMin`

function). Next, all threads relax in parallel the outgoing edges of the vertices they have undertaken. Compared to the original algorithm, a performance improvement is expected, since, due to the helper threads, the main thread will evaluate the expression of line 7 in Alg. 2 as true fewer times and thus, will not need to execute the operations of lines 8–10.

Algorithm 2: Main thread's code.

```
1  while Q ≠ ∅ do
2      u ← ExtractMin(Q);
3      done ← 0;
4      foreach v adjacent to u do
5          sum ← d[u] + w(u, v);
6          Begin-Transaction
7          if d[v] > sum then
8              DecreaseKey(Q, v, sum);
9              d[v] ← sum;
10             π[v] ← u;
11         End-Transaction
12     end
13     Begin-Transaction
14     done ← 1;
15     End-Transaction
16 end
```

Algorithm 3: Helper threads' code.

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

Our scheme employs TM not only for the concurrent accesses to the various data structures, but for the orchestration of the helper threads as well. Specifically, when the main thread completes 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 included in their read sets. Then they will 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. If the main thread happens to perform 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.

Employing TM instead of traditional locking primitives, i.e. locks and barriers, offers two significant advantages: First, it is too difficult and error-prone to develop a fine-grain locking scheme for these threads. The programmer would probably have to use a series of locks in a composable fashion to guard all the data structures that must be accessed atomically (lines 7–10 in Alg.2). This is a quite intricate task, since correctness requires avoiding potential deadlocks or livelocks, while efficiency requires avoiding serialization of accesses as much as possible. On the other hand, this functionality is achieved easily with TM, just by enclosing the critical section in *one* transaction, as shown in Alg. 2 and Alg. 3.

Even if such a complex locking scheme was implemented, it would incur a very high overhead on non-conflicting parallel accesses. This would be acceptable if the majority of concurrent accesses led to conflicts. However, in this work we show that the opposite is true. Therefore, the optimistic nature of TM, where non-conflicting accesses are allowed to execute with no overhead, makes it a better solution.

IV. Experimental Evaluation

A. Experimental setup

The performance of the proposed scheme was evaluated through full-system simulation, using the Wisconsin GEMS toolset v.2.1 [17], [18] in conjunction with the Simics v.3.0.31 simulator [19]. 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 [20]. 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 occur. 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 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 I shows the configuration of the simulation framework.

TABLE I: Simulation framework.

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 latency
	Memory	160 cycle access latency
	TM System	HYBRID resol. policy, 2Kb HW signatures

TABLE II: Graphs used for experiments

random			*rmat*			*ssca*		
E (K)	Ser. (%)	Id. Sp.	E (K)	Ser. (%)	Id. Sp.	E (K)	Ser. (%)	Id. Sp.
10	52.9	1.89	10	68.4	1.46	28	45.0	2.22
50	62.2	1.61	50	58.8	1.70	60	55.2	1.81
100	50.9	1.96	100	48.3	2.07	118	46.6	2.15
200	40.1	2.49	200	38.0	2.63	177	41.5	2.41
500	28.4	3.52	500	27.3	3.66	590	27.4	3.65
1000	22.6	4.42	1000	22.2	4.50	1157	22.4	4.64

To avoid resource conflicts between our programs and the operating system's processes, we used CMP configurations with more processor cores than the number of threads we required. At the same time, each thread is bound to a specific processor to avoid migrations. Finally, all codes were compiled with Sun's Studio 12 C compiler (O3 level).

B. Reference graphs

In our evaluation we strived to work on graphs which vary in terms of density and structure. In that attempt, we used the GTgraph graph generator [21] to construct graphs with $10K$ vertices from the *Random*, *R-MAT* and *SSCA#2* families.

To obtain an estimate of possible speedups, we profiled the stand-alone execution of the main thread of our scheme on each graph to calculate the extent of the sequential part. As sequential we define the non-transactional part of the code, which includes mainly the `ExtractMin` operations. In the ideal case where the helper threads would manage to offload all the relaxations of the main thread, the speedup would be $\frac{100\%}{\%SerialPart}$. Note that this is optimistic, since even in this case the main thread would still have to check if any relaxations are required. In general, it constitutes a theoretical upper bound for any performance improvement and is presented in Table II for each graph family.

C. Performance results

Fig. 4 presents the speedups achieved by our HT+TM based implementation of Dijkstra's algorithm for our graph suite. The speedup obtained for p threads is the ratio of the execution time of the serial algorithm to the execution time with p threads, $p-1$ of them being helper threads. The maximum speedup is 1.84, achieved for 14 threads in Fig. 4f. Considering the serial nature of the algorithm and the inherent difficulties in its parallelization, this is a significant performance gain. Note also that the performance is strongly related to the density of the graph. In the serial case the execution time can be estimated as follows:

$$T_{serial} = n \times O(\lg n) + d \times n \times O(\lg n) \qquad (1)$$

where n denotes the number of vertices in the graph and d the average out-degree of the nodes. The first part of (1) estimates the time spent on `ExtractMin` operations, while the second part approximates the time spent on `DecreaseKey` operations. Similarly, the execution time of our scheme can be estimated as follows:

$$T_{HT} = n \times O(\lg n) + a \times d \times n \times O(\lg n), a < 1 \qquad (2)$$

where a the ratio of the main thread's `DecreaseKey` operations to those executed in the serial case. This is a simple estimate and does not take into account the time spent in thread orchestration or delays due to conflicting transactions. The speedup s could be approximated by $s = \frac{1+d}{1+a\times d}$ which implies that s should increase with the average out-degree and thus, the density of the graph, explaining the results of Fig. 4. This figure also reveals that the speedup increases as more threads are utilized. This tendency reaches a maximum point, after which employing more threads leads to a slight performance degradation. The number of threads needed to achieve this maximum, is again related to the graph's density.

D. Interpretation of the HT scheme's behavior

In this section, we attempt to gain a better insight into the behavior of our scheme. We focus our study on one family of graphs, the rmat, as the other families exhibit similar behavior and we select only three representative graphs with different density degrees; low (10K), medium (200K) and high (1000K).

Fig. 5 shows the distribution of `DecreaseKey` operations between the main and helper threads and compares them to those performed in the serial case. As more threads are used, the main thread's `DecreaseKey` operations are reduced, justifying the performance improvement. However, not all the helper threads' operations are useful, as illustrated in Figs. 5b and 5c, where the total number of `DecreaseKeys` is greater than that of the serial case, explaining why the performance does not keep improving. Interestingly, similar reductions in the main thread's operations are also achieved for the sparse graph, as it is shown in Fig. 5a. However, Fig. 4a shows that in this case the performance is actually degraded. This can be attributed to the transactions' abort rate, which is defined as the ratio of aborts to commits and is depicted in Fig. 6. It is obvious, that for the sparse graph, the abort rate is too high, causing any performance improvements due to the exploitation of parallelism to be canceled out. However, for the more dense graphs, the abort rate is significantly reduced and thus, speedups are achieved. An important observation though, is that in any case the abort rate of the main thread is significantly low, which means that it is not obstructed by the helper threads. This explains the robustness of our scheme, as in the worst case the slowdown is around 0.95.

The same conclusion can be derived from Fig. 7, where the execution cycles of the main thread are depicted. The non-transactional cycles remain stable for each graph, as they represent the time spent on `ExtractMin` operations, which are not affected by our scheme. The addition of helper threads reduces the time spent in transactions, i.e. the parallel part of our scheme, since the main thread executes less `DecreaseKeys`, as shown before. The overhead cycles represent the time spent in aborts or stalls caused by transaction conflicts. This overhead is relatively small, illustrating once again that the main thread is not hindered by the helper threads.

To gain a better understanding of the wasted work due to transaction aborts, Fig. 8 plots the percentage of the total

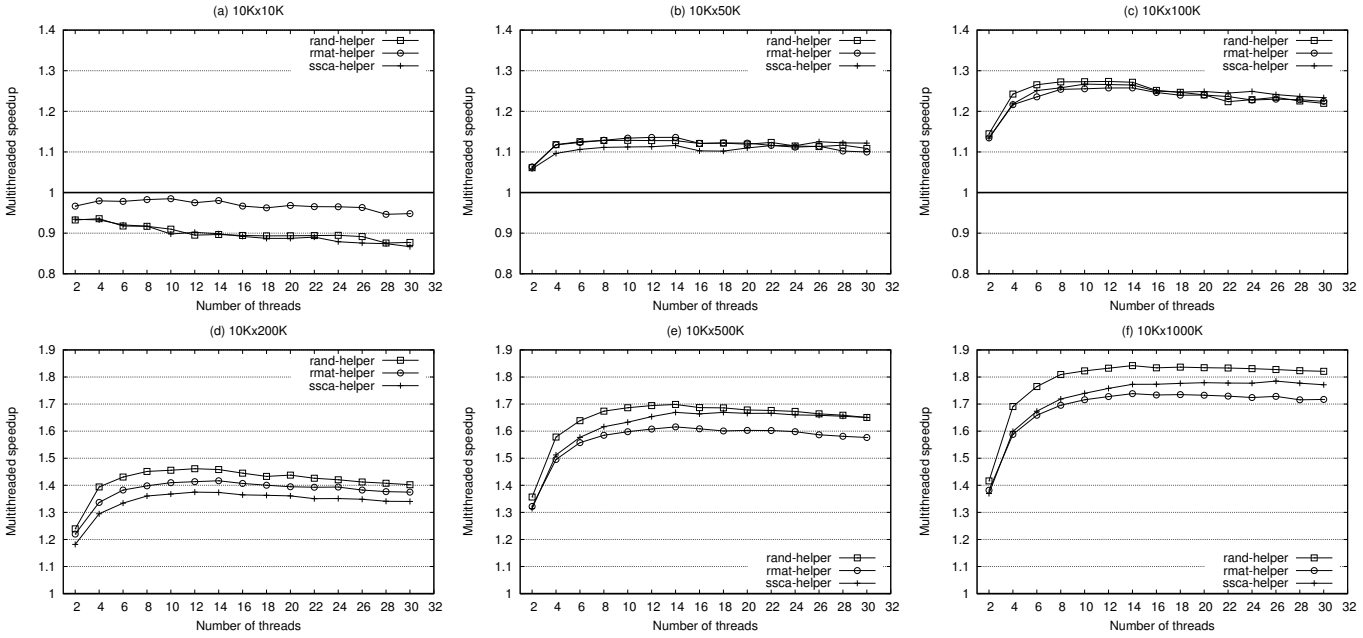


Fig. 4: Multithreaded speedups for graphs of different density.

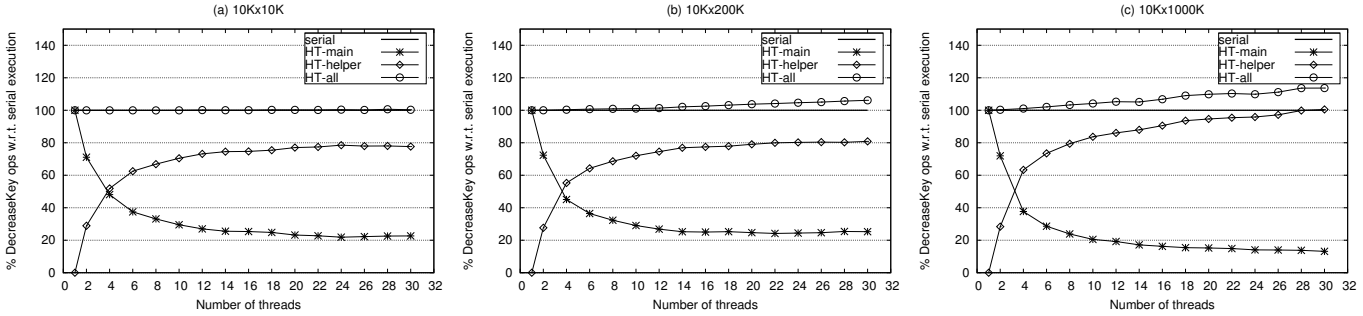


Fig. 5: Distribution of `DecreaseKey` operations between the main and helper threads.

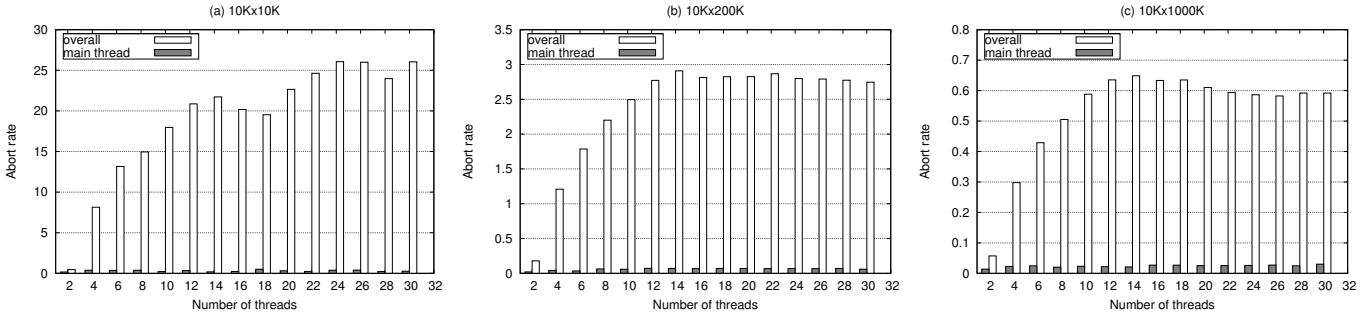


Fig. 6: Overall and main thread transaction abort rates.

cycles spent by all threads in aborted transactions with respect to the total number of cycles spent in successfully committed transactions. Again, for graphs of medium or high density the amount of wasted work is relatively small, justifying the observed speedups. On the contrary, a lot of work is wasted for the sparse graph, explaining the absence of performance improvements in this case.

In general, the small amount of wasted work shows that most of the concurrent accesses to the shared data structures are non-conflicting. The number of aborts depends also on the size of the transactions' write sets. The larger the write sets, the higher the probability of a conflict. Table III presents the range of the average write set size of all transactions, together with that of the transactions that envelop the `DecreaseKey` operations. Note that the average sizes are quite small, leading to a low probability for conflicts. These findings confirm that, due to its optimism, TM is a better approach than locks for the implementation of our scheme, as explained in Section III-B.

Finally, Fig. 9 compares the cycles the main thread needs for every 100 iterations of the algorithm's outer loop for graph

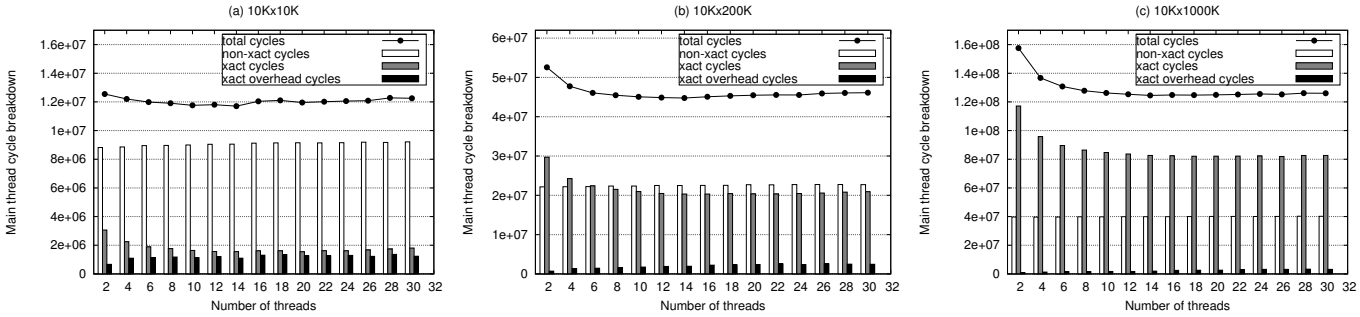


Fig. 7: Breakdown of main thread's total cycles: non-transactional (non-xact), transactional (xact) and overhead (xact overhead).

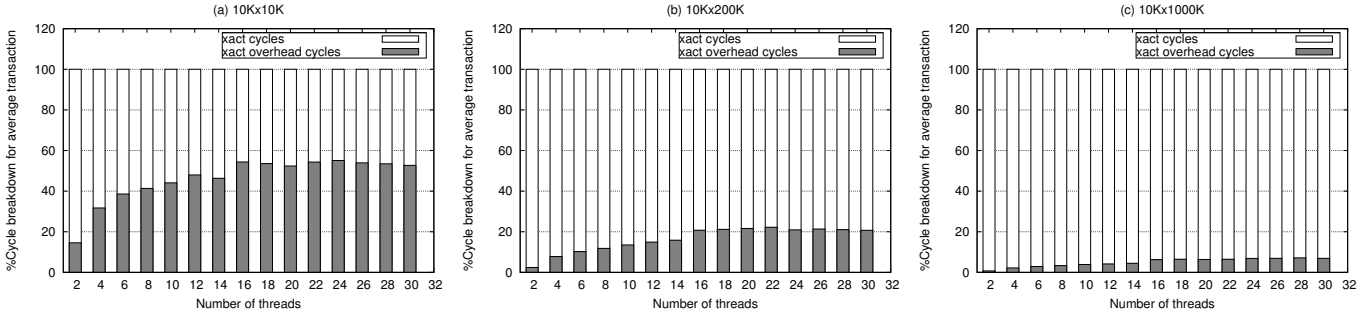


Fig. 8: Percentage of useful (xact) and wasted (xact overhead) transactional cycles.

Density	*Avg. write-set size*	*Avg. write-set size for DecreaseKey operations*	*Max write-set size*
10K	1.31 - 3.14	12.44 - 20.02	28 - 31
50K	1.16 - 2.07	8.26 - 12.08	29 - 31
100K	1.08 - 1.71	7.84 - 10.79	28 - 30
200K	1.04 - 1.52	7.66 - 9.83	28 - 31
500K	1.02 - 1.20	7.54 - 8.81	27 - 31
1000K	1.01 - 1.12	7.67 - 8.68	29 - 36

TABLE III: Write-set size.

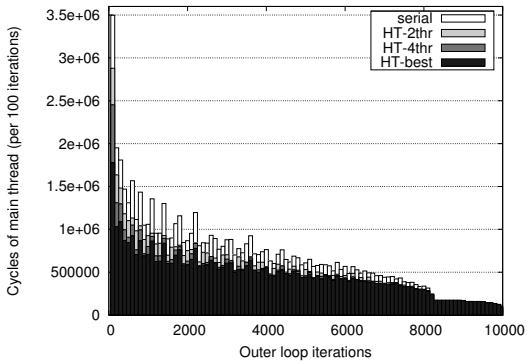


Fig. 9: The timeline of execution.

rmat-10Kx200K, when running in parallel with 0, 1, 3 and 13 helper threads. The first observation is that the majority of the execution time is spent on the first 30% of the iterations. The second observation is that as the algorithm proceeds, the available parallelism is reduced and the gains from the use of more helper threads are negligible. In fact, for the last 20% of the iterations, the main thread spends the same amount of time both in the serial case and with 13 helper threads. This motivates us to explore adaptive schemes, where the number of helper threads will be dynamically adjusted.

V. 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. [2] 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. [22] 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 [5]). A generalization of Dial's algorithm called Δ-stepping is proposed by Meyer and Sanders [3]. Madduri et al. [7] use Δ-stepping as the base algorithm on Cray MTA-2. In the Parallel Boost Graph Library [6] Dijkstra's algorithm is parallelized for a distributed

memory machine where the priority queue is distributed in the local memories of the system nodes. 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. Nevertheless, its efficacy on a wide set of real, non-trivial applications is only now starting to be explored. Scott et al. [15] use TM to parallelize Delaunay triangulation, Watson et al. [14] exploit it to parallelize Lee's routing algorithm and Kang and Bader [16] employ it for computing minimum spanning forests of sparse graphs.

VI. Conclusions - Future Work

In this work, we attempt to parallelize Dijkstra's algorithm, which is known to be inherently serial. Our scheme utilizes the notion of "Helper Threads" (HT) to offload the main thread by speculatively executing a notable portion of its `DecreaseKey` operations. For the implementation, we choose to employ Transactional Memory (TM), not only for its ease of programmability, but also for its nature, which allows to explore any optimistic parallelism inherent in our scheme. The evaluation revealed that the proposed scheme is able to provide significant speedups (reaching up to 1.84) in the majority of the simulated cases. The results further confirmed the existence of optimistic parallelism, justifying the selection of TM.

An important outcome of this work is the indication that the TM mechanism could be efficiently leveraged for the implementation of speculative multithreading, as it is also discussed in [23]. We feel that studying the combination of these two models is extremely important, especially as new systems are coming that will provide support for TM[12].

As future work, we will investigate the application of this technique on other algorithms solving the SSSP problem, such as Δ-stepping and Bellman-Ford. We also aim to explore the impact of various TM characteristics, such as the resolution policy, version management and conflict detection, on the performance of our scheme. Moreover, 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. Finally, we aim to further explore the integration of the two programming models, namely Transactional Memory and Speculative Multithreading.

Acknowledgements

The experiments were executed on hardware platforms generously provided by Intel Hellas S.A. This work was supported by the Greek Secreteriat of Research and Technology (GSRT) and the European Commission under the program 05AKMWN95.

References

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

[2] G. Brodal, J. Traff, C. Zaroliagis, and I. Stadtwald, "A parallel priority queue with constant time operations," *J. Parallel and Distributed Computing*, vol. 49, pp. 4–21, 1998.

[3] U. Meyer and P. Sanders, "Delta-stepping: A parallel single source shortest path algorithm," in *Proc. 6th Ann. European Symp. on Algorithms (ESA'98)*, 1998.

[4] N. Anastopoulos, K. Nikas, G. Goumas, and N. Koziris, "Early experiences on accelerating dijkstra's algorithm using transactional memory," in *Proc. 3rd Workshop on Multithreaded Architectures and Applications (MTAAP'09)*, 2009.

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

[6] N. Edmonds, A. Breuer, D. Gregor, and A. Lumsdaine, "Single-source shortest paths with the parallel boost graph library," in *9th DIMACS Implementation Challenge – The Shortest Path Problem*, 2006.

[7] K. Madduri, D. Bader, J. Berry, and J. Crobak, "Parallel shortest path algorithms for solving large-scale instances," in *9th DIMACS Implementation Challenge – The Shortest Path Problem*, 2006.

[8] 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 *Proc. 28th Ann. Int'l Symp. on Computer Architecture (ISCA'01)*, 2001.

[9] W. Zhang, B. Calder, and D. Tullsen, "An event-driven multithreaded dynamic optimization framework," in *Proc. 14th Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT'05)*, 2005.

[10] M. Herlihy and E. Moss, "Transactional memory: Architectural support for lock-free data structures," in *Proc. 20th Ann. Int'l Symp. on Computer Architecture (ISCA'93)*.

[11] A. Adl-Tabatabai, C. Kozyrakis, and B. Saha, "Unlocking concurrency: Multicore programming with transactional memory," *ACM Queue*, vol. 4, no. 10, pp. 24–33, 2006.

[12] M. Tremblay and S. Chaudhry, "A third-generation 65nm 16-core 32-thread plus 32-scout-thread CMT SPARC processor," in *Proc. Int'l Solid State Circuits Conf. (ISSCC '08)*, 2008.

[13] B. Saha, A.-R. Adl-Tabatabai, R. L. Hudson, C. C. Minh, and B. Hertzberg, "Mcrt-stm: a high performance software transactional memory system for a multi-core runtime," in *Proc. 11th Symp. on Principles and Practice of Parallel Programming (PPoPP'06)*, 2006.

[14] I. Watson, C. Kirkham, and M. Lujan, "A study of a transactional parallel routing algorithm," in *Proc. 16th Int'l Conf. on Parallel Architecture and Compilation Techniques (PACT'07)*, 2007.

[15] M. L. Scott, M. F. Spear, L. Daless, and V. J. Marathe, "Delaunay triangulation with transactions and barriers," in *IEEE Intl. Symp. on Workload Characterization (IISW'07)*, 2007.

[16] S. Kang and D. A. Bader, "An efficient transactional memory algorithm for computing minimum spanning forest of sparse graphs," in *Proc. 14th Symp. on Principles and Practice of Parallel Programming (PPoPP'09)*, 2009.

[17] 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," *Comput. Archit. News*, vol. 33, no. 4, pp. 92–99, 2005.

[18] "Wisconsin multifacet gems simulator," http://www.cs.wisc.edu/gems/.

[19] P. 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*, vol. 35, no. 2, pp. 50–58, 2002.

[20] L. Yen, J. Bobba, M. Marty, K. Moore, H. Volos, M. Hill, M. Swift, and D. Wood, "Logtm-se: Decoupling hardware transactional memory from caches," *Proc. 13th Int'l Symp. on High Performance Computer Architecture (HPCA'07)*, 2007.

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

[22] G. Hunt, M. Michael, S. Parthasarathy, and M. Scott, "An efficient algorithm for concurrent priority queue heaps," *Inf. Proc. Letters*, vol. 60, pp. 151–157, 1996.

[23] L. Porter, B. Choi, and D. Tullsen, "Mapping out a path from hardware transactional memory to speculative multithreading," in *Proc. 18th Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT'09)*, 2009 – in press.