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

Nikos Anastopoulos, Konstantinos Nikas, Georgios Goumas and Nectarios Koziris[1]

Computing Systems Laboratory, School of Electrical and Computer Engineering, National Technical University of Athens, Zografou Campus, Zografou 15780, Greece

ABSTRACT

In this work we deal with 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. Simulation results demonstrate that the proposed HT+TM based implementation is able to achieve performance speedups up to 1.84.

1 Motivation

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 [CLRS01] 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. 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 [ANGK09]. Its fundamentally serial

[1]E-mail: {anastop,knikas,goumas,nkoziris}@cslab.ece.ntua.gr

nature has led researchers to seek performance through significant modifications of the algorithm. 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)* 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.

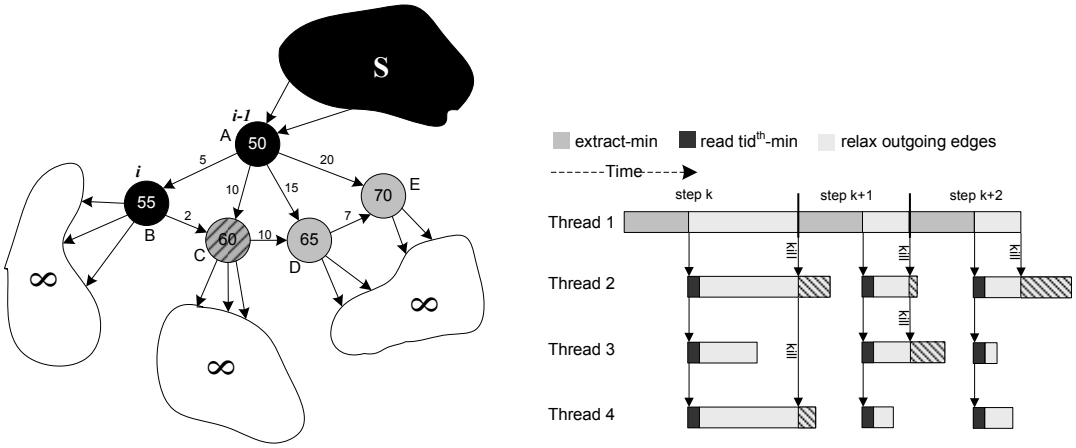
To amortize the cost of excessive synchronization, we employ *Transactional Memory (TM)* [HM93]. TM is a novel programming model for multicore architectures that allows concurrency control over multiple threads and is getting adopted by the industry. It seems a promising approach which increases programmability, while being capable of providing performance gains through the concept of optimistic parallelism. Lately, TM's usage in the parallelization of specific algorithms has attracted scientific attention [WKL07, SSDM07, KB09], as its potential on speeding up real-world applications is still under investigation.

2 Helper Threading Scheme

The rationale of our proposed scheme is largely based on the greedy nature of the algorithm, i.e. its attempt to make in each step locally optimal choices. In the serial algorithm the relaxations are performed only from the extracted vertex, which *is known* to have at this point an optimal distance. In our scheme, we relax this limitation by allowing relaxations to happen from queued vertices, *hoping* that some of them *might* already have their optimal, final distances. Therefore, we employ parallel threads to serve as *Helper Threads* and perform relaxations for 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 helper threads are assigned the vertices occupying the top k positions in the queue, which are the next best choices with respect to the vertex being extracted and the most likely ones to have their distances optimal. Thus, when the helper threads read their distances and relax their outgoing edges, there is a high probability they will make correct relaxations. As a result, 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. 1a, where the i-th iteration of the algorithm 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. So, in step $i+1$ the main thread will extract C and relax again its neighbors 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 have already obtained their minimum value in the $i-1$ step. Therefore, in step i, the helper threads relax their neighbors correctly and when the main thread extracts them it will not have to perform any



(a) Example of HT scheme's functionality.

(b) Execution pattern of the HT scheme.

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. 1b. 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 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.

In our scheme the parallel threads need to access the priority queue as well as the data structures that implement the graph in parallel. For efficient concurrency control, we propose the use of Transactional Memory. In each thread (main or helper), we enclose the check for relaxation, the update to the priority queue and the update to the distance and predecessor arrays within a single transaction. This guarantees that these dependent operations will be performed in an "all-or-none" fashion, and that on a conflict, only one thread will be allowed to commit. Transactional Memory is the option that makes feasible the composition of these sub-operations into a larger atomic operation, without limiting concurrency, without suffering the overhead of any lock-based scheme and with minimal additions to the original code.

3 Performance Evaluation

The performance of the proposed scheme was evaluated through full-system simulation, using the Wisconsin GEMS toolset [gem] in conjunction with Simics simulator. Hardware TM is supported in GEMS through the LogTM-SE subsystem. 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* and *eager conflict detection*. In our experiments we used the `HYBRID` conflict resolution policy, which tends to favor older transactions against younger ones.

In our evaluation, we worked on graphs which varied in terms of density and structure. Our scheme yielded speedups in 15 out of 18 graphs, reaching up to 1.84 [NAGKar]. The evaluation revealed the existence of optimistic parallelism in the concurrent updates of the priority queue, justifying the selection of TM. It also showed the robustness of our scheme, since in any case the abort rate of the main thread was significantly low, which means that it was not obstructed by the helper threads (less than 0.95 slowdown in the worst case).

4 Conclusions and 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 executing a notable portion of edge relaxations. 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 in the majority of the simulated cases.

As future work, we will investigate the applicability of our proposed scheme on other algorithms that have similar nature ("greedy"). We also aim to explore the impact of various TM characteristics on the scheme's performance, such as the resolution policy, version management and conflict detection. Finally, 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.

References

[ANGK09] N. Anastopoulos, K. Nikas, G. Goumas, and N. Koziris. Early experiences on accelerating dijkstra's algorithm using transactional memory. In *MTAAP'09*.

[CLRS01] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, 2001.

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

[HM93] M. Herlihy and E. Moss. Transactional memory: Architectural support for lock-free data structures. In *ISCA'93*.

[KB09] S. Kang and D. A. Bader. An efficient transactional memory algorithm for computing minimum spanning forest of sparse graphs. In *PPoPP'09*.

[NAGKar] K. Nikas, N. Anastopoulos, G. Goumas, and N. Koziris. Employing transactional memory and helper threads to speedup dijkstra's algorithm. In *ICPP'09* – to appear.

[SSDM07] M. L. Scott, M. F. Spear, L. Daless, and V. J. Marathe. Delaunay triangulation with transactions and barriers. In *IISWC'07*.

[WKL07] I. Watson, C. Kirkham, and M. Lujan. A study of a transactional parallel routing algorithm. In *PACT'07*