

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

Nikos Anastopoulos

Computing Systems Laboratory
School of Electrical and Computer Engineering
National Technical University of Athens
anastop@cslab.ece.ntua.gr
<http://www.cslab.ece.ntua.gr>

June 4, 2009



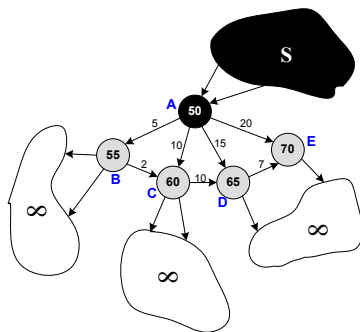
Motivation

- TM community needs real-world applications
- Graph algorithms are described as good candidates for TM, due to irregular accesses of data structures
- Dijkstra's algorithm
 - ▶ fundamental SSSP algorithm
 - ▶ widely used
 - ▶ inherently serial, thus challenging to parallelize
 - ▶ previous attempts resulted in major changes in algorithm's semantics (e.g. Δ -stepping, Boost implementation)
- Early results published in MTAAP'09, extended version will appear in ICPP'09

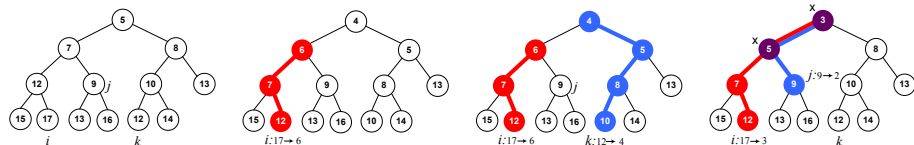
The Basics of Dijkstra's Algorithm

Serial algorithm

```
1  Input      :  $G = (V, E)$ ,  $w : E \rightarrow \mathbb{R}^+$ ,  
                source vertex  $s$ , min  $Q$   
2  Output   : shortest distance array  $d$ ,  
                predecessor array  $\pi$   
3  foreach  $v \in V$  do  
4       $d[v] \leftarrow \text{INF}$ ;  
5       $\pi[v] \leftarrow \text{NIL}$ ;  
6      Insert( $Q, v$ );  
7  end  
8   $d[s] \leftarrow 0$ ;  
9  while  $Q \neq \emptyset$  do  
10      $u \leftarrow \text{ExtractMin}(Q)$ ;  
11     foreach  $v$  adjacent to  $u$  do  
12          $sum \leftarrow d[u] + w(u, v)$ ;  
13         if  $d[v] > sum$  then  
14             DecreaseKey( $Q, v, sum$ );  
15              $d[v] \leftarrow sum$ ;  
16              $\pi[v] \leftarrow u$ ;  
17     end  
18 end
```



The Basics of Dijkstra's Algorithm



Min-priority queue implemented as binary min-heap

- maintains all but the settled (“optimal”) vertices
- min-heap property: $\forall i : d(\text{parent}(i)) \leq d(i)$
- amortizes the cost of multiple ExtractMin's and DecreaseKey's
 - ▶ $O((|E| + |V|)\log|V|)$ time complexity

Straightforward Parallelization

Fine-grain parallelization at the inner loop level

Fine-Grain Multi-Threaded

```
1  /* Initialization phase same to the serial
2     code */
3  while Q ≠ ∅ do
4     Barrier
5     if tid = 0 then
6         u ← ExtractMin(Q);
7         Barrier
8         for v adjacent to u in parallel do
9             sum ← d[u] + w(u, v);
10            if d[v] > sum then
11                Begin-Atomic
12                DecreaseKey(Q, v, sum);
13                End-Atomic
14                d[v] ← sum;
15                π[v] ← u;
16            end
17        end
18    end
```

Issues

- speedup bounded by average out-degree
- concurrent heap updates due to DecreaseKey's
- barrier synchronization overhead

Evaluation

- conventional synch. mechanisms yield major slowdowns
- TM
 - ▶ better performance
 - ▶ highlights optimistic parallelism
 - ▶ suffers from barriers overhead

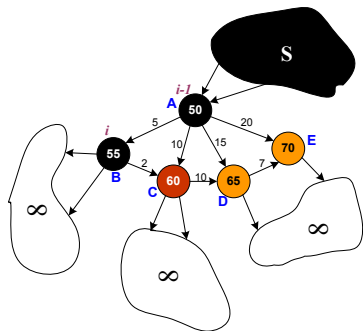
Helper-Threading Scheme

Motivation

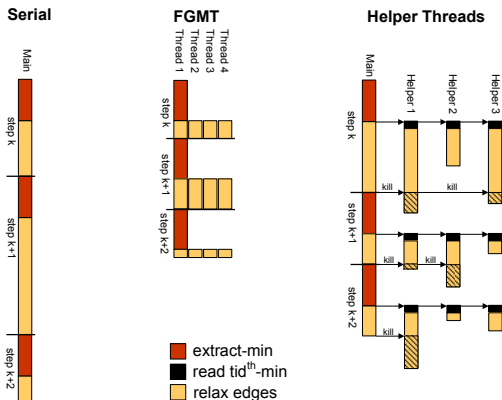
- expose more parallelism to each thread
- eliminate costly barrier synchronization

Rationale

- in serial, updates are performed only from *definitely* optimal vertices
- allow updates from *possibly* optimal vertices
 - ▶ main thread operates as in the serial case
 - ▶ helper threads are assigned the next minimum vertices (x_k) and perform updates from them
- speculation on the status of x_k
 - ▶ if **already optimal**, main thread will be offloaded
 - ▶ if **not optimal**, any suboptimal relaxations will be corrected eventually by main thread



Execution Pattern



Decoupling of sequential/parallel parts is achieved through TM

- the main thread stops all helpers at the end of each iteration
- unfinished work will be corrected, as with mis-speculated distances

Helper-Threading Scheme

Main thread

```
1 while  $Q \neq \emptyset$  do
2    $u \leftarrow \text{ExtractMin}(Q)$ ;
3    $done \leftarrow 0$ ;
4   foreach  $v$  adjacent to  $u$  do
5      $sum \leftarrow d[u] + w(u, v)$ ;
6     Begin-Xact
7     if  $d[v] > sum$  then
8        $\text{DecreaseKey}(Q, v, sum)$ ;
9        $d[v] \leftarrow sum$ ;
10       $\pi[v] \leftarrow u$ ;
11     End-Xact
12   end
13   Begin-Xact
14    $done \leftarrow 1$ ;
15   End-Xact
16 end
```

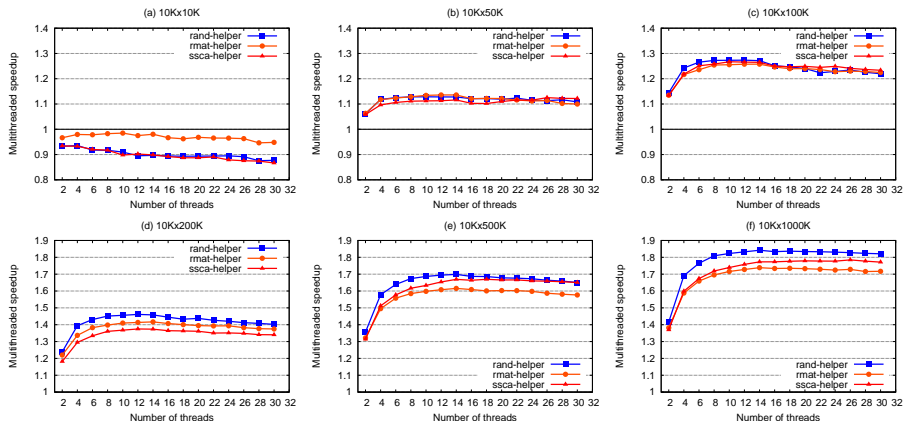
Helper thread

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

Why with TM?

- composable
 - ▶ all dependent atomic sub-operations composed into a large atomic operation, without limiting concurrency
- optimistic
- easily programmable

Speedups



Helper-Threading

- speedups in 15 out of 18 cases, up to 1.84
- performance improves with increasing density
- main thread not obstructed by helpers (<1% abort rate in all cases)

Conclusions

FGMT

- conventional synchronization mechanisms incur unacceptable overhead
- TM reduces overheads and highlights the existence of parallelism, but still requires very efficient barriers to offer some speedup

HT+TM

- exposes more parallelism and eliminates barrier synchronization
- noteworthy speedups with minimal code extensions

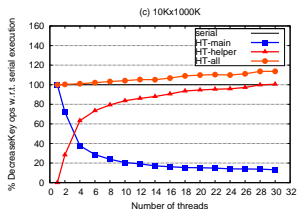
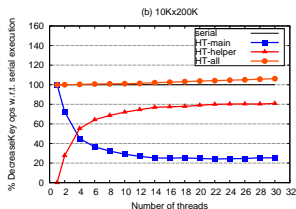
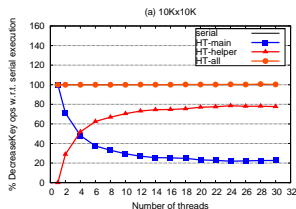
Future work

- HT+TM as a programming model for other graph problems (MSTs, maximum flow, SSSP) and other similar (“greedy”) applications
- dynamic adaptation of helper threads to algorithm’s execution phases
- explore impact of TM characteristics

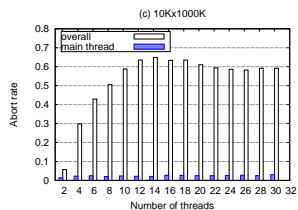
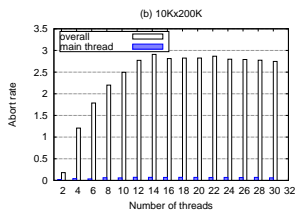
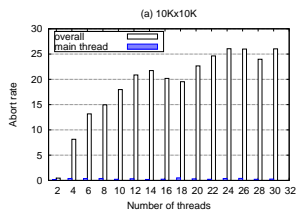
Thank you!

Questions?

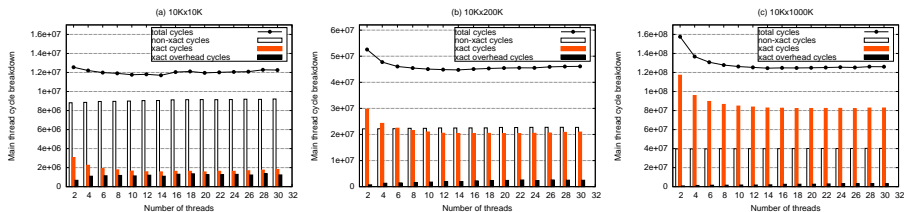
Distribution of DecreaseKey's between main and helper threads



Overall and main thread transaction abort rates

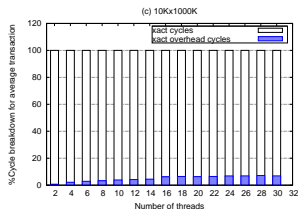
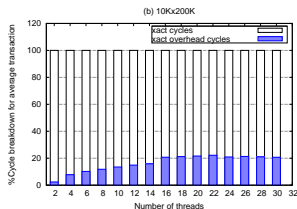
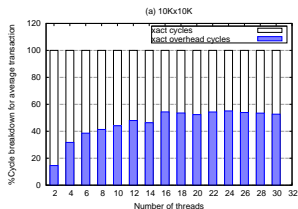


Breakdown of main thread's total cycles



Non-transactional (non-xact), transactional (xact) and overhead (xact-overhead)

Percentage of useful (xact) and wasted (xact-overhead) transactional cycles



Write-set sizes

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

The timeline of execution (rmat-10Kx200K)

