Early Experiences on Accelerating Dijkstra's Algorithm Using Transactional Memory

Nikos Anastopoulos, Konstantinos Nikas, Georgios Goumas and Nectarios Koziris

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

May 31, 2009

Outline

1. Dijkstra's Basics
2. Straightforward Parallelization Scheme
3. Helper-Threading Scheme
4. Experimental Evaluation
5. Conclusions

The Basics of Dijkstra's Algorithm

SSSP Problem

- Directed graph $G = (V, E)$, weight function $w : E \rightarrow \mathbf{R}^+$, source vertex s
- $\forall v \in V$: compute $\delta(v) = min\{w(p) : s \overset{p}{\leadsto} v\}$

Shortest path estimate $d(v)$

- gradually converges to $\delta(v)$ through *relaxations*
- relax (v,w): $d(w) = min\{d(w), d(v) + w(v, w)\}$
  - can we find a better path $s \overset{p}{\leadsto} w$ by going through v?

Three partitions of vertices

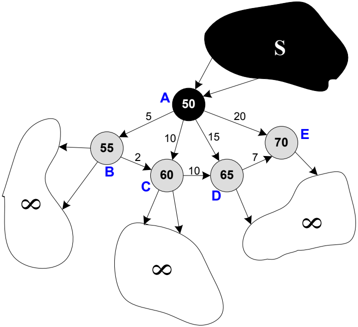
- Settled: $d(v) = \delta(v)$
- Queued: $d(v) > \delta(v)$ and $d(v) \neq \infty$
- Unreached: $d(v) = \infty$

The Basics of Dijkstra's Algorithm

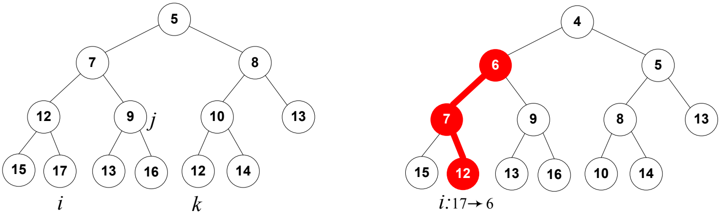
Serial algorithm

Input : $G = (V, E)$, $w : E \rightarrow \mathbf{R}^+$,
source vertex s, min Q
Output : shortest distance array d,
predecessor array π

foreach $v \in V$ **do**
  $d[v] \leftarrow$ INF;
  $\pi[v] \leftarrow$ NIL;
  Insert(Q, v);
end
$d[s] \leftarrow 0$;
while $Q \neq \emptyset$ **do**
  $u \leftarrow$ ExtractMin(Q);
  foreach v *adjacent to* u **do**
    $sum \leftarrow d[u] + w(u, v)$;
    if $d[v] > sum$ **then**
      DecreaseKey(Q, v, sum);
      $d[v] \leftarrow sum$;
      $\pi[v] \leftarrow u$;
  end
end



The Basics of Dijkstra's Algorithm



Min-priority queue implemented as binary min-heap

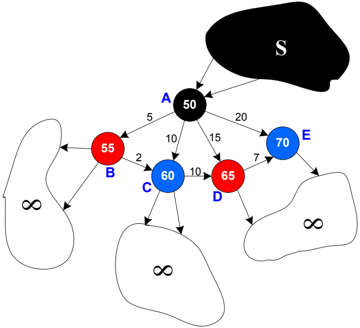
- maintains all but the settled vertices
- min-heap property: $\forall i : d(parent(i)) \leq d(i)$
- amortizes the cost of multiple `ExtractMin`'s and `DecreaseKey`'s
  - $O((|E| + |V|)log|V|)$ time complexity

Straightforward Parallelization

Fine-grain parallelization at the inner loop level

Fine-Grain Multi-Threaded

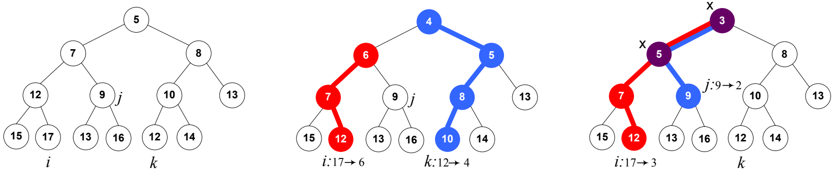
```
/* Initialization phase same to the serial
   code */
while Q ≠ ∅ do
    Barrier
    if tid = 0 then
        u ← ExtractMin(Q);
    Barrier
    for v adjacent to u in parallel do
        sum ← d[u] + w(u, v);
        if d[v] > sum then
            Begin-Atomic
            DecreaseKey(Q, v, sum);
            End-Atomic
            d[v] ← sum;
            π[v] ← u;
    end
end
```



Issues:

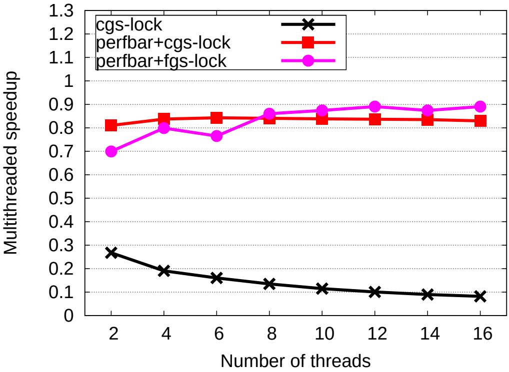
- speedup bounded by average out-degree
- concurrent heap updates due to `DecreaseKey`'s
- barrier synchronization overhead

Concurrent Heap Updates with Locks



- Coarse-grain synchronization (*cgs-lock*)
  - enforces atomicity at the level of a `DecreaseKey` operation
  - one lock for the entire heap
  - serializes `DecreaseKey`'s
- Fine-grain synchronization (*fgs-lock*)
  - enforces atomicity at the level of a single swap
  - allows multiple swap sequences to execute in parallel as long as they are *temporally* non-overlapping
  - separate locks for each parent-child pair

Performance of FGMT with Locks



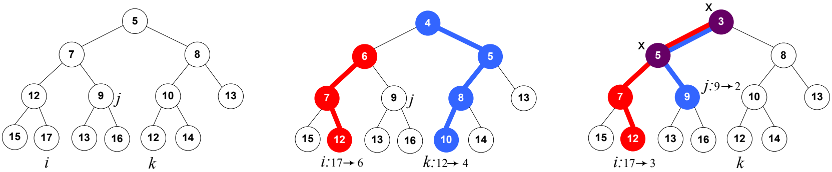
Software barriers dominate total execution time

- 72% with 2 threads, 88% with 8
- replace with idealized (simulated) zero-latency barriers

Fgs-lock scheme more scalable, but still fails to outperform serial

- locking overhead (2 locks + 2 unlocks per swap)

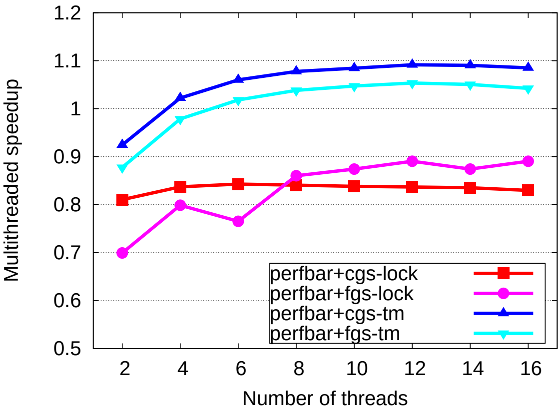
Concurrent Heap Updates with TM



TM-based

- Coarse-grain synchronization (*cgs-tm*)
  - enclose `DecreaseKey` within a transaction
  - allows multiple swap sequences to execute in parallel as long as they are *spatially* (and temporally) non-overlapping
  - conflicting transaction stalls and retries or aborts
- Fine-grain synchronization (*fgs-tm*)
  - enclose each swap operation within a transaction
  - atomicity as in *fgs-lock*
  - shorter but more transactions

Performance of FGMT with TM



TM-based schemes offer speedup up to ~ 1.1

- less overhead for *cgs-tm*, yet equally able to exploit available concurrency

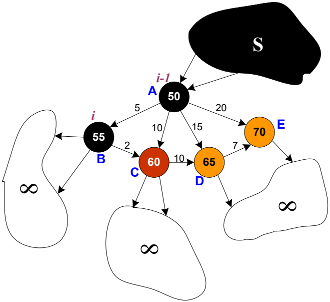
Helper-Threading Scheme

Motivation

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

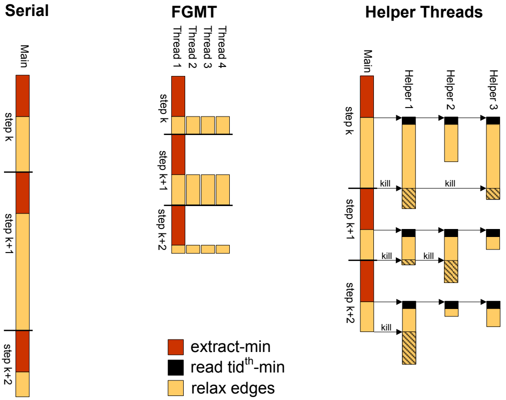
Rationale

- in serial, relaxations are performed only from the extracted (settled) vertex
- allow relaxations for out-edges of queued vertices, hoping that some of them might already be settled
  - main thread operates as in the serial algorithm
  - assign the next t vertices in the queue $(x_2 \ldots x_{t+1})$ to t helper threads
  - helper thread k relaxes *all* out-edges of vertex x_k



- speculation on the status of $d(x_k)$
  - if already optimal, main thread will be offloaded
  - if not optimal, any suboptimal relaxations will be corrected eventually by main thread

Execution Pattern



- 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

while $Q \neq \emptyset$ **do**
    $u \leftarrow$ `ExtractMin`(Q);
    $done \leftarrow 0$;
    foreach *v adjacent to u* **do**
        $sum \leftarrow d[u] + w(u, v)$;
        Begin-Xact
        if $d[v] > sum$ **then**
            `DecreaseKey`(Q, v, sum);
            $d[v] \leftarrow sum$;
            $\pi[v] \leftarrow u$;
        End-Xact
    end
    Begin-Xact
    $done \leftarrow 1$;
    End-Xact
end

Helper thread

while $Q \neq \emptyset$ **do**
    while $done = 1$ **do** ;
    $x \leftarrow$ `ReadMin`(Q, tid)
    $stop \leftarrow 0$
    foreach *y adjacent to x* **and while** $stop = 0$ **do**
        Begin-Xact
        if $done = 0$ then
            $sum \leftarrow d[x] + w(x, y)$
            if $d[y] > sum$ **then**
                `DecreaseKey`(Q, y, sum)
                $d[y] \leftarrow sum$
                $\pi[y] \leftarrow x$
        else
            $stop \leftarrow 1$
        End-Xact
    end
end

- for a single neighbour, the check for relaxation, updates to the heap, and updates to d,π arrays, are enclosed within a transaction
  - performed "all-or-none"
  - on a conflict, only one thread commits
- interruption of helper threads implemented through TM, as well

Helper-Threading Scheme

Main thread

```
while Q ≠ ∅ do
    u ← ExtractMin(Q);
    done ← 0;
    foreach v adjacent to u do
        sum ← d[u] + w(u, v);
        Begin-Xact
        if d[v] > sum then
            DecreaseKey(Q, v, sum);
            d[v] ← sum;
            π[v] ← u;
        End-Xact
    end
    Begin-Xact
    done ← 1;
    End-Xact
end
```

Helper thread

```
while Q ≠ ∅ do
    while done = 1 do ;
    x ← ReadMin(Q, tid)
    stop ← 0
    foreach y adjacent to x and while stop = 0 do
        Begin-Xact
        if done = 0 then
            sum ← d[x] + w(x, y)
            if d[y] > sum then
                DecreaseKey(Q, y, sum)
                d[y] ← sum
                π[y] ← x
        else
            stop ← 1
        End-Xact
    end
end
```

Why with TM?

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

Experimental Setup

Full-system simulation

- Simics 3.0.31 in conjunction with GEMS toolset 2.1
- boots unmodified Solaris 10 (UltraSPARC III Cu)

LogTM ("Signature Edition")

- eager version management
- eager conflict detection
  - on a conflict, a transaction stalls and either retries or aborts
- HYBRID conflict resolution policy
  - favors older transactions

Hardware platform

- single CMP system (configurations up to 32 cores)
- private L1 caches (64KB), shared L2 cache (2MB)

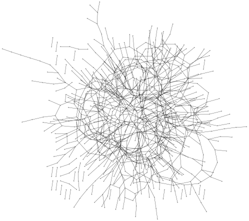
Software

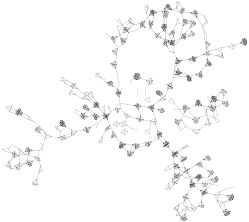
- Pthreads for threading and synchronization
- Simics "magic" instructions to simulate idealized barriers
- Sun Studio 12 C compiler (-xO3)

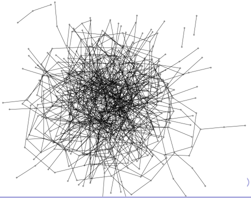
Graphs

Three graph families

- **Random**: $G(n, m)$ model
- **SSCA#2**: cliques with varying size $(1 - C)$ connected with probability P
- **R-MAT**: power-law degree distributions





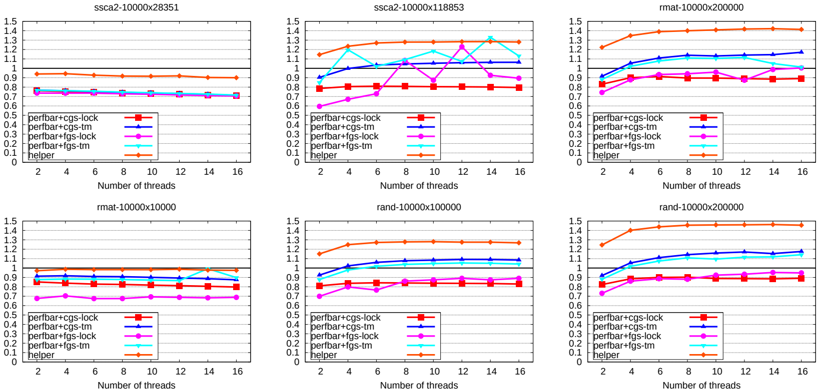


GTgraph graph generator

Fixed #nodes (10K), varying density

- sparse ($\sim$10K edges)
- medium ($\sim$100K edges)
- dense ($\sim$200K edges)

Speedups



Helper-Threading

- speedups in 6 out of 9 cases (not all shown), up to 1.46
- performance improves with increasing density
- main thread not obstructed by helpers (<1% abort rate in all cases)

FGMT with TM

- speedups only with perfect barriers
- optimistic parallelism does exist in concurrent queue updates

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 with TM

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

Future work

- more aggressive parallelization schemes
- dynamic adaptation of helper threads to algorithm's execution phases
- explore impact of TM characteristics
- applicability of HT on other SSSP algorithms (Δ-stepping, Bellman-Ford) and other similar ("greedy") applications

Thank you!

Questions?