Motivation
TM community needs real-world applications
- Graph algorithms are described as good candidates for TM, due to irregular accesses of underlying data structures

Dijkstra’s algorithm
- fundamental and widely used algorithm for the single-source shortest paths (SSSP) problem
- inherently serial, thus challenging to parallelize
- previous attempts resulted in major changes in algorithm’s semantics or structure (e.g. Δ-stepping, Boost library implementation)

Straightforward Parallelization

**Fine-Grain Multi-Threaded**

```
/* Initialization: phase same to serial code */
while Q is empty do
  Barrier
  if Q[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;
      n[v] += u;
    End-Atomic
  end
end
```

Issues:
- speedup bounded by average out-degree
- concurrent heap updates to due DecreaseKey’s
- barrier synchronization overhead (more than 70% of total execution time)

Evaluation
- conventional synchronization mechanisms yield major slowdowns
- TM gives better performance, highlights optimistic parallelism, but still suffers from barriers overhead

Implementation of HT-Scheme

**Main thread**

```
while Q is empty do
  u = ExtractMin(Q);
  done = 0;
  foreach v adjacent to u do
    sum += d[u] + w(u,v);
    if d[v] > sum then
      Begin-Atomic
        DecreaseKey(Q,v,sum);
      End-Atomic
      d[v] = sum;
      n[v] += u;
      End-Atomic
    if done = 1 then
      done = 0;
      End-Atomic
    End-Atomic
  end
end
```

**Helper thread**

```
while Q is empty do
  while done = 1 do
    done = 0;
    foreach v adjacent to u do
      sum += d[u] + w(u,v);
      if d[v] > sum then
        Begin-Atomic
          DecreaseKey(Q,v,sum);
        End-Atomic
        d[v] = sum;
        n[v] += u;
        End-Atomic
      end
      if done = 1 then
        done = 0;
        End-Atomic
      else
        done = 1;
        End-Atomic
    End-Atomic
end
```

Why with TM?
- composable: all dependent atomic sub-operations (check for relaxation, heap updates, d,u updates) composed into a large atomic operation without limiting concurrency
- optimistic
- easily programmable
- offers a way for helper threads interruption

The Basics of Dijkstra’s Algorithm

<table>
<thead>
<tr>
<th>Serial algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input: G=(V,E), s ∈ V, min Q</td>
</tr>
<tr>
<td>Output: shortest distance array d.</td>
</tr>
<tr>
<td>foreach v ∈ V do</td>
</tr>
<tr>
<td>d[v] = d(s,v);</td>
</tr>
<tr>
<td>while Q is empty do</td>
</tr>
<tr>
<td>u = ExtractMin(Q);</td>
</tr>
<tr>
<td>foreach v adjacent to u do</td>
</tr>
<tr>
<td>sum += d(u) + w(u,v);</td>
</tr>
<tr>
<td>if d[v] &gt; sum then</td>
</tr>
<tr>
<td>Begin-Atomic</td>
</tr>
<tr>
<td>DecreaseKey(Q,v,sum);</td>
</tr>
<tr>
<td>End-Atomic</td>
</tr>
<tr>
<td>d[v] = sum;</td>
</tr>
<tr>
<td>n[v] += u;</td>
</tr>
<tr>
<td>End-Atomic</td>
</tr>
<tr>
<td>end</td>
</tr>
</tbody>
</table>

**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 serial**
- helper threads are assigned the next minimum vertices (xₖ) and perform updates from them

Speculation on the status of xₖ:
- if already optimal, main thread will be offloaded
- if not optimal, any suboptimal relaxations will be corrected eventually by main thread

Experimental Evaluation

**Simics 3.0.31, GEMS 2.1**

- LogTM-SE (EE, HYBRID resolution policy)
- 3 different graph families, 6 graph densities

**Results**
- speedups in 15 out of 18 cases (not all shown), up to 1.84
- main thread not obstructed by helpers (<1% abort rate)

Future work

**TM for Optimistic parallelization**

**HT+TM as a programming model for other graph problems (MSTs, maximum flow, SSSP) and other similar (“greedy”) applications**

**Adjustments of existing TM systems for explicitly supporting speculative parallelization**
(or, what if we didn’t rely on Dijkstra’s algorithm semantics to correct things?)

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

Nikos Anastopoulos, Konstantinos Nikas, Georgios Goumas and Nectarios Koziris

{anastop,knikas,goumas,nkoziris}@cslab.ece.ntua.gr

Computing Systems Laboratory, School of Electrical and Computer Engineering, National Technical University of Athens

**Execution Patterns**

**Example of HT scheme**
- In step 1, node A was extracted, and its neighbors were relaxed in the value array. In step 2, the main thread extracts B while the helpers are assigned the next nodes in the queue (C,D,E). For both A and B, the helper performs correct relaxations, since in step 1 they obtained their final distances. As a result, the main thread will be offloaded from these relaxations. On the contrary, the helper that took over C will perform suboptimal relaxations, since at the end of step 1, C’s distance will be updated to 37 by the main thread. However, in step 1 the main thread will correct C and relax again its out-edges, using now the correct distance.

**Number of threads**

- 2 to 32 threads
- 4 categories of threads
- Barriers, helper threads, rand-helper threads
- Unreached: d(v) = α(v)
- Queued: d(v) = β(v) and d(v) ≠ γ(v)
- Settled: d(v) = δ(v)

**Barriers**

- Consecutively, barriers do not violate TM semantics to correct things?