Understanding Priority-Based Scheduling of Graph Algorithms on a Shared-Memory Platform

Serif Yesil*, Azin Heidershenas*, Adam Morrison+, and Josep Torrellas*

*University of Illinois at Urbana-Champaign and +Tel Aviv University

November 20, 2019
Task Based Graph Processing

- Many graph processing frameworks use a task based model
- Computation is broken down into dynamically created tasks
- Easy to program
Unordered Task Execution

- Tasks can be processed in any order
- Correct result regardless of execution order
- Easy to parallelize
- Problem: may cause unnecessary work

1. Vertex 4 is scheduled
2. Vertex ∞ becomes 7
3. Vertex 1 is scheduled
4. Vertex 7 becomes 2

- **Update in step 2 is unnecessary**
Adding Task Priorities and Using a Priority Scheduler

- Each task becomes associated with a priority
- Tasks are sorted by priorities in the scheduler
  - For SSSP priority=distance, ordered in ascending order
  1. Vertex 1 is scheduled
  2. Vertex ∞ becomes 2
Fundamental Trade-off in Concurrent Priority Schedulers: Synchronization vs. Useless Work

- Scheduling in strict priority order causes synchronization overhead
- Solution: Use **Relaxed Concurrent Priority Scheduler (CPS)**
- Highly relaxed priority processing $\rightarrow$ **more useless work**
Contributions

- Provide an extensive empirical analysis of existing CPS designs
- Identify shortcomings of high performance CPS designs
- Propose a CPS approach (PMOD) to provide robust performance automatically
Types of Concurrent Priority Schedulers

- **Combined-Priority Queue CPSs**: Each queue has multiple priorities
  - SprayLists (SL) – Global Queue [Alistarh et.al.]
  - Multiqueues (MQ) – Distributed Queues [Rihani et.al.]
  - Remote Enqueue Local Dequeue (RELD) – Distributed Queues [Jeffrey et.al.]
- **Per-Priority Queue CPSs**: Each priority in a different queue
  - Ordered-By Integer Metric (obim) [Nguyen et.al.]
SprayList

- A global queue design
- Tasks are stored in a lock-free skip list, sorted in priority order
- To minimize synchronization, dequeue retrieve a random high priority task
  - Done with a short random walk
Distributed Queues

MultiQueues
- Create \( k \) queues per core (Total: \( k \times p \))
- Enqueue to a random queue
- Dequeues retrieve the highest priority task from two random queues

Remote Enqueue Local Dequeue
- One queue per core (Total: \( p \))
- Enqueue to a random queue
- Dequeues go to the local queue
OBIM Chunking

- Creates a queue for each priority
- Uses a *global map* for ordering
  - Access bottleneck
- A *local map* caches a snapshot of the global map in each core
- When needed, local maps are refreshed from the global map
- To reduce contention, a core enqueues and dequeues a chunk of tasks at a time
Minimizing Useless Work vs. Minimizing Communication

Minimize Useless Work

- Combined-priority queues
  - SL and MQ relaxation guarantees
- Retrieve tasks close to priority order
- Invest in synchronization and communication
- Higher enqueue and dequeue cost

Minimize Communication

- Per-priority queues: obim
- Emphasis on reducing enqueue dequeue cost
- Coarse-grain enqueue/dequeue operations
- Prone to useless work
CPS Performance Comparison

How well do relaxed CPSs work in practice?

- Useless work behavior
- Performance of CPS operations
Types of Execution Cycles in an Application

- Work Cycles
  - **Good Work (GWork)**: The work ends up being useful.
  - **Useless Work (UWork)**: The work later proves useless.

- CPS Operation Cycles
  - **Enqueue (Enq)**
  - **Dequeue (Deq)**
  - **Failed Dequeue (FDeq)**: Attempting and failing to dequeue

- Other
  - **Other**: Executing framework code.
Analysis Setup

- Galois framework
- 5 applications
  - Single source shortest path (SSSP)
  - Breadth first search (BFS)
  - PageRank (PR)
  - Minimum spanning tree (MST)
  - A* (A*)

5 inputs graphs

<table>
<thead>
<tr>
<th>Graph</th>
<th># Verts</th>
<th># Edges</th>
</tr>
</thead>
<tbody>
<tr>
<td>USA roads (rUSA)</td>
<td>24 M</td>
<td>58 M</td>
</tr>
<tr>
<td>West USA roads (rW)</td>
<td>6 M</td>
<td>15 M</td>
</tr>
<tr>
<td>Twitter40 (tw)</td>
<td>42 M</td>
<td>1469 M</td>
</tr>
<tr>
<td>Web-Google (wg)</td>
<td>875 K</td>
<td>5 M</td>
</tr>
<tr>
<td>Soc-LiveJournal1 (lj)</td>
<td>5 M</td>
<td>69 M</td>
</tr>
</tbody>
</table>
Speedups: Combined-Priority Queues vs. Per-Priority Queues

Per-priority queue performs better than combined priority queues
Cycles: Combined-Priority Queues vs. Per-Priority Queue CPSs

Small amount of useless work
Combined-priority queues spend upto 90% of cycles on CPS operations
Per-Priority Queues: Effect of Chunking

- Enqueueing and dequeueing a chunk of tasks is very beneficial.
- Chunking decreases the \( \text{Deq} \) time.
  - No chunks: Higher \( \text{Deq} \)
  - Large chunk sizes: Higher \( \text{FDeq} \)

BFS with \( rUSA \)
- \( c0 \)⇒ no chunks
- \( c16 \)⇒ chunk size 16
- \( c64 \)⇒ chunk size 64
- \( c256 \)⇒ chunk size 256
Per-Priority Queues: Effect of Priority Distribution

- Different algorithms create different priority distributions
  - BFS: Close by priorities
  - SSSP: Dispersed priorities
- Dispersed priorities lowers the performance of per-priority queue CPSs
  - Create many per-priority queues with few tasks
Ad-Hoc Solution: Manually Merging Priorities

- Different priorities are grouped together in the same per-priority queue
- Merging Factor (\(\Delta\)):  
  - Range of priorities that map to the same queue  
  - Best \(\Delta\) parameter is input dependent
- Used in SSSP application:  
  - Optimal \(\Delta\) for SSSP with \(rUSA\) is \(\Delta = 14 \Rightarrow \text{Range}=[0 - 2^{14}]\)  
  - Optimal \(\Delta\) for SSSP with \(tw\) is \(\Delta = 0 \Rightarrow \text{No priority merging}\)
Effect of Merging Factor for SSSP

- Priority merging has significant impact
- Too-small $\Delta$: Dequeue has high overhead
- Too-large $\Delta$: High useless work
- Optimal $\Delta$: 20-35% of cycles spent for useless work

SSSP with $rUSA$

- $\Delta_{10}$ $\Rightarrow$ Merging Factor=10
- $\Delta_{14}$ $\Rightarrow$ Merging Factor=14
- $\Delta_{18}$ $\Rightarrow$ Merging Factor=18
Proposed CPS: Priority Merging on Demand (PMOD)

- Too many per-priority queues $\rightarrow$ higher enqueue and dequeue times
- Too few per-priority queues $\rightarrow$ more useless work
- Idea: Dynamically detect inefficient execution and tune Merging Factor
  - Too many per-priority queues $\rightarrow$ Merge priorities (increase $\Delta$)
  - Too few per-priority queues $\rightarrow$ Unmerge priorities (decrease $\Delta$)
Detect and Adjust Merging Factor

- Too many per-priority queues $\rightarrow$ Merge priorities (increase $\Delta$)
  1. Trigger: Frequent global/local map synchronizations
  2. Check: Density of per-priority queues
  3. Increase merging factor

- Too few per-priority queues $\rightarrow$ Unmerge priorities (decrease $\Delta$)
  1. Trigger: Too many tasks dequeued from a per-priority queue
  2. Check: Number of per-priority queues and priority range
  3. Decrease merging factor

- PMOD challenge: Priority ordering with different merging factors
  - Details in the paper...
PMOD Performance

SSSP with rUSA

SSSP with tw

Same performance as obim without manual/application specific tuning
Merging factor converges to 13 for SSSP rUSA (Δ = 14)
PMOD Performance

Same performance as obim without manual/application specific tuning
Merging factor doesn’t increase when not needed: BFS
Conclusions

- Fundamental trade-off between useless work and synchronization overhead
- Current CPSs often incur large enqueue/dequeue overheads
- Per-priority CPSs that minimize CPS operation cost give better performance
  - But performance is application & input dependent
- PMOD provides high performance automatically
  - Same performance as per-application manually optimized application with obim
- In the paper
  - Performance sensitivity to chunk sizes & priority merging
  - obim performance sensitivity to priority ranges
  - More applications and insights