Lock-Free Priority Queue Based on Multi-Dimensional Linked Lists

Jun Tao Luo   Yumin Chen

Background

- Priority queues are central to many CS problems
- Investigate concurrent implementations
- Explore lock-free implementation using Multi-Dimensional Linked Lists (MDList)[1]

Single Source Shortest Path

- Classic CS problem, compute the shortest paths from source node to all other nodes in a graph
- Sequentially solved using Dijkstra's algorithm
- Parallelized Dijkstra's algorithm [2]
  - Uses fine-grain locks and a concurrent priority queue

```
Graph (E,V,w)
done[1..TNum] = [false,..,false]
D[1..|V|] = [∞,..,∞]
Element[1..|V|] Offer =
                    [null,..,null]
Lock [1.. |V|]  DLock
Lock [1.. |V|]  OfferLock
```

```
relax(v,vd)
 lock(OfferLock[v])
   if (vd < D[v])
     vo = Offer[v]
     if (vo = null)
       Offer[v] = insert(v,vd)
     else
       if (vd < vo.key)
         Offer[v] = insert(v,vd)
 unlock(OfferLock[v])
```

```
parallelDijkstra()
 while (!done[tid])
   o = extractMin()
   if (o ≠ null)
     u = o.data
     d = o.key
     lock(DLock[u])
     if(dist < D[u])
       D[u] = d
       explore = true
     else
       explore = false
     unlock(DLock[u])
     if (explore)
       foreach ((u,v) ∈ E)
         vd = d + w(u,v)
         relax(v,vd)
   else
     done[tid] = true
     i = 0
     while (done[i] and i<TNum)
       i =  i + 1
     if(i == TNUM)
        return
     done[tid] = false
```

Priority Queue using Multi-Dimension Lists

Key Conversion

- key -> $(\text{key})_{\lceil \sqrt[D]{N} \rceil} \rightarrow [k_0, k_1, ..., k_{D-1}]$

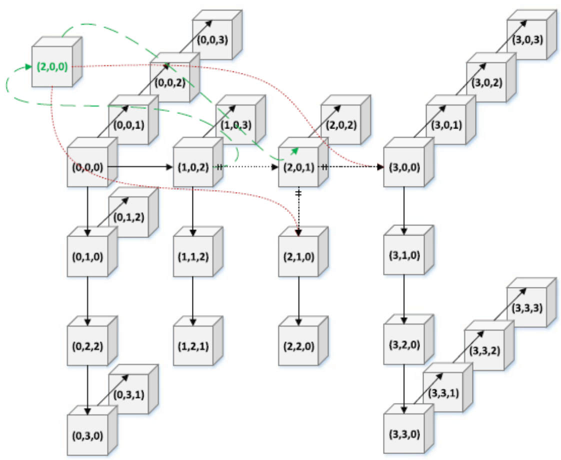
Basic Rule

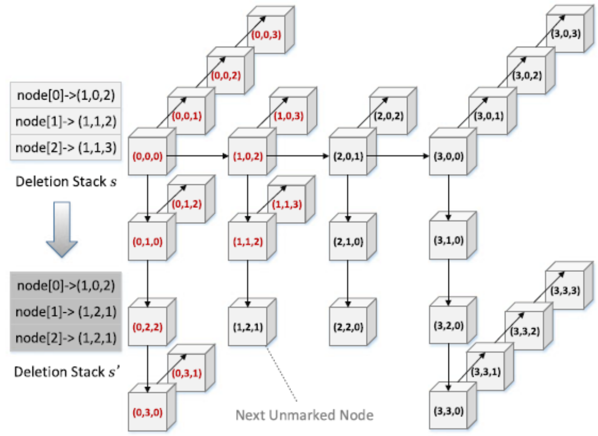
- Dimension range: [0, D); each node has up to D children
- The coordinates of a child is greater than or equal to the coordinate of the parent for all dimensions

Data Structures

- Adesc (Adoption Description)
  - parent node of adoption
  - child slot to be adopted (dp, dc)
- Node
  - key, value
  - key coordinates
  - child[D]
  - adesc
- Stack (deletion stack to trace the last known deleted node)
- Priority Queue
  - D, N
  - head (dummy head)
  - stack

Insert

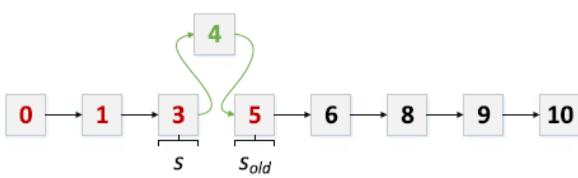


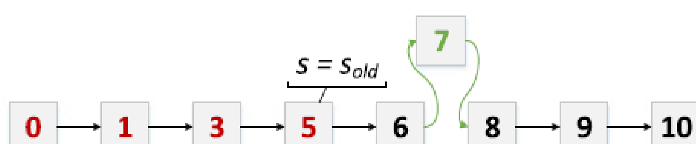


- Build the new node
- LocatePred → predNode, childNode, dp, dc
- FinishInserting → child adoption for predNode, childNode
- Fill in the new node(CAS)
- Rewind the deletion stack
- Retry if 1) another thread inserted a child; 2) the child slot has been marked as invalid

Rewind the stack

- Synchronize info between insert and deleteMin
- The old stack points to a position beyond the new node





DeleteMin

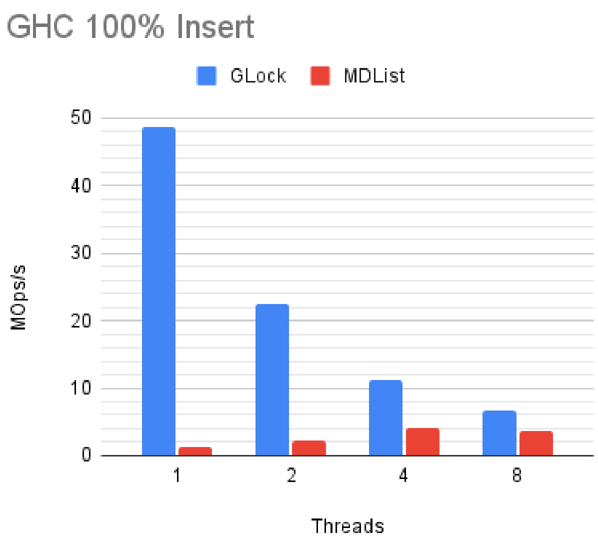
- Backtrack the deletion stack to find the next minimum stack
- Retry if the found node has been marked as deleted

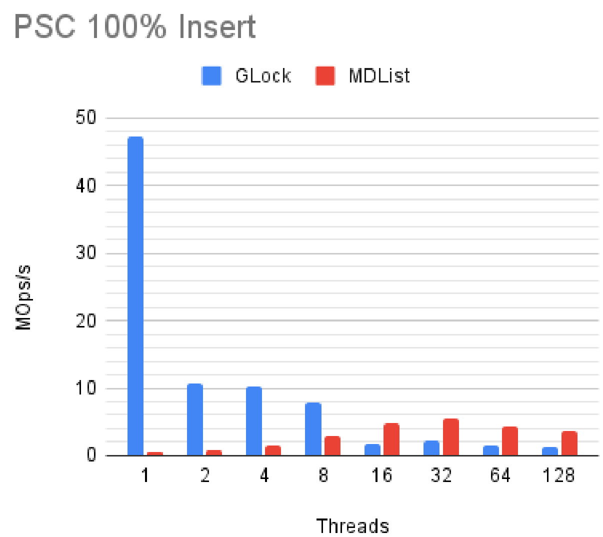
Implementation Extensions

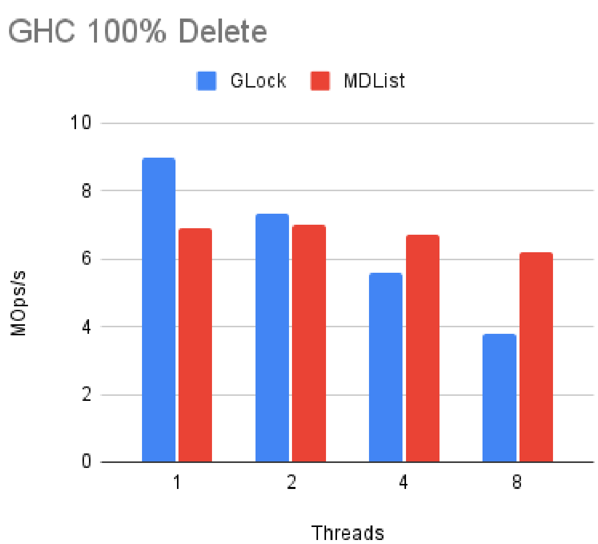
- Duplicated priorities
    - Reserve 13/32 bits in key space for unique identifiers
    - Identifiers are assigned using an atomic counter
- DeleteMin return values
    - SSSP requires the priority and value of the min entry
    - SSSP requires to know when priority queue is empty
- Support non-reference values
    - Reference implementation assumes stored values are referenced as pointers
- Exclusion of physical deletion
    - The reference paper indicated they use only logical deletion for benchmarks
- Pseudocode bugs
    - There were several instances of wrong or missing pseudocode from both the MDList and Parallel Dijkstra algorithms

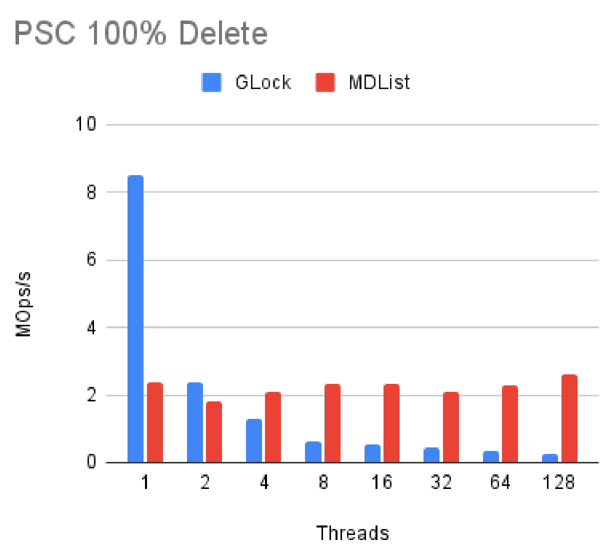
Results

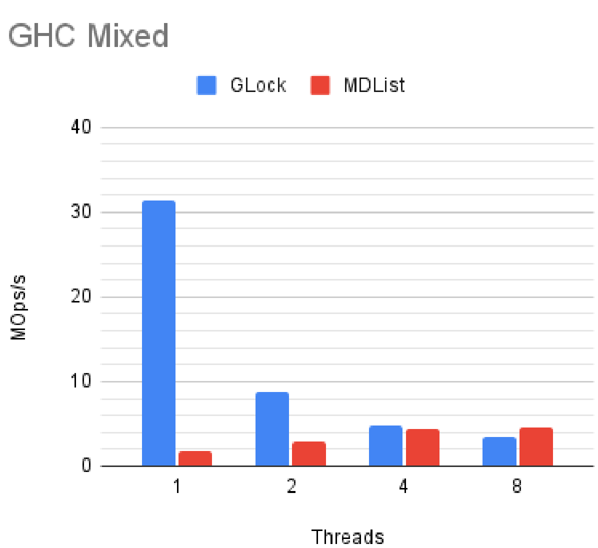
Microbenchmarks

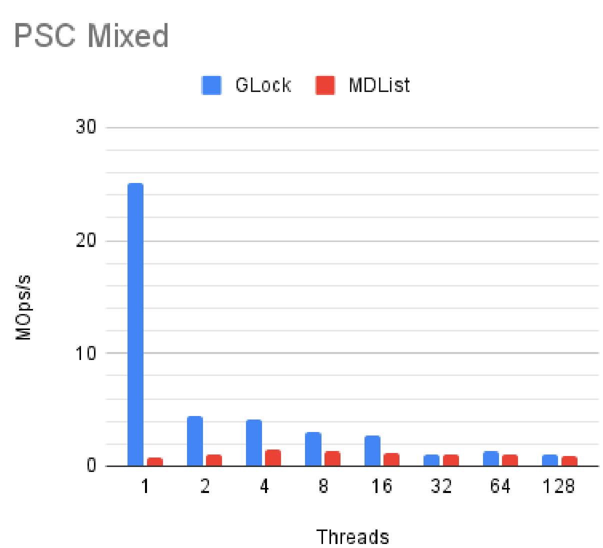




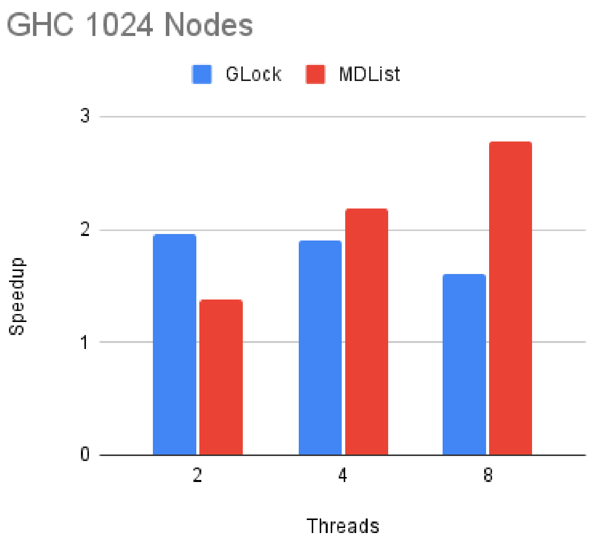


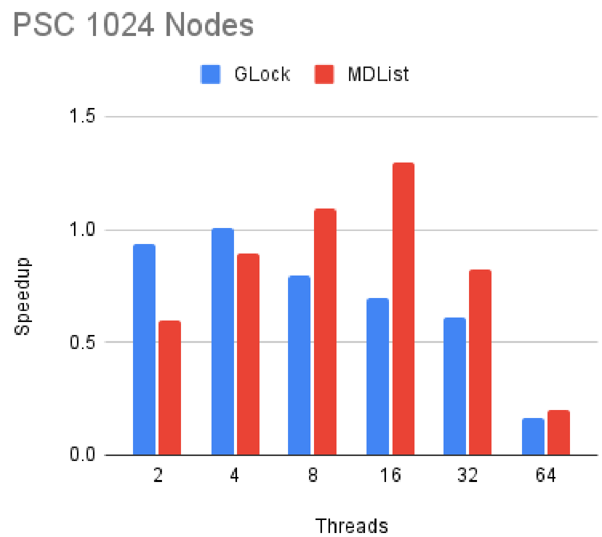


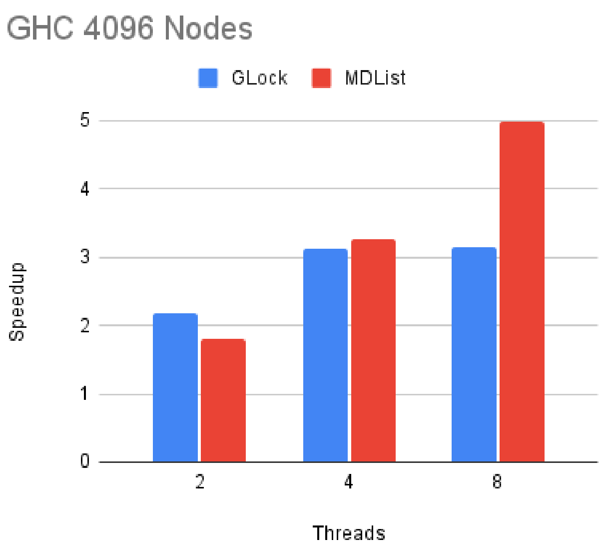


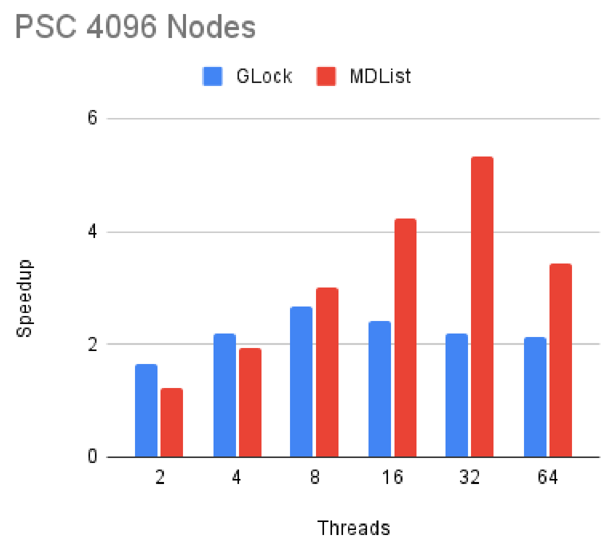


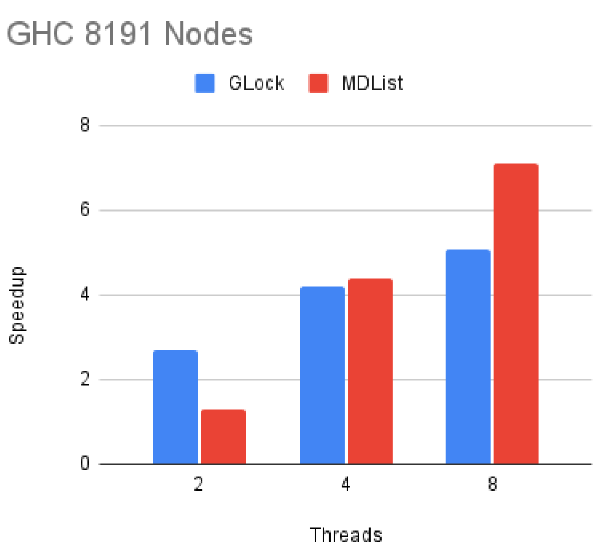
SSSP Benchmark

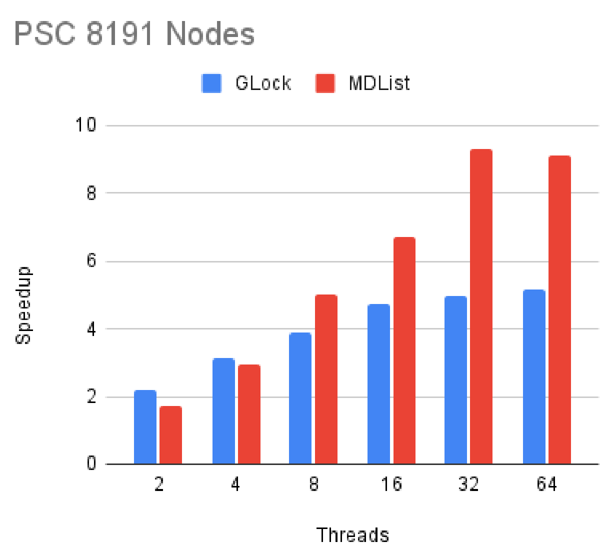












Discussion and Analysis

Microbenchmarks

- Coarse grain lock performs poorly with more threads
    - Due to more lock contention
- MDList outperforms coarse grain for >= 4 threads
    - Insert performs poorly < 8 threads due to overhead
    - Insert performs well relatively for 8 - 32 threads
    - Insert performs poorly > 32 due to context switching
    - Insert bottlenecked by allocation (via VTune results)
    - Delete performance constant
        - Due to inherent sequential nature of DeleteMin
    - Mixed performance inconclusive since coarse grain implements physical deletion while MDList does not

SSSP Benchmark

- Coarse grain lock performs poorly with more threads
    - Due to more lock contention
    - Speedup mostly attributed to parallelized SSSP
- MDList outperforms coarse grain for >= 4 threads on GHC and >= 8 threads on PSC
    - Performs poorly for low thread counts due to overhead
    - Scales well to up to 32 threads on PSC
    - DeleteMin bottlenecked by CAS retry (via diagnostics)
- Superlinear speedup on GHC due to lower branch miss rate and cache miss rate based on prof result

Further Work

- Concurrent memory allocation
  - Experiment with other allocators instead of glibc [3]
- Comparisons with alternative priority queue implementations
  - Intel's TBBPQ [4]
  - Fine Grained Locking [5]
- Physical Deletion
  - Implement Purge
- Further Optimizations of MDList implementation

Conclusion

- Extended MDList to solve Parallelized SSSP
- Observed performance improvement over coarse grained locking on SSSP benchmarks and microbenchmarks
- Limitations of concurrent priority queues limits its application in highly parallelized algorithms [6]

References

[1] Zhang, D. and Dechev, D. A lock-free priority queue design based on multi-dimensional linked lists. IEEE Transactions on Parallel and Distributed Systems , 27(3):613–626,2016. doi: 10.1109/TPDS.2015.2419651.

[2] Tamir, O., Morrison, A., and Rinetzky, N. A heap-based concurrent priority queue with mutable priorities for faster parallel algorithms. In OPODIS, 2015.

[3] Evans, J. A scalable concurrent malloc (3) implementation for freebsd. In Proc. of the bsdcan conference, Ottawa, Canada, 2006.

[4] Shavit, N. and Lotan, I. Skiplist-based concurrent priority queues. In Proceedings 14th International Parallel and Distributed Processing Symposium. IPDPS 2000, pp. 263–268. IEEE, 2000.

[5] Hunt, G. C., Michael, M. M., Parthasarathy, S., and Scott, M. L. An efficient algorithm for concurrent priority queue heaps. Information Processing Letters, 60(3):151–157, 1996.

[6] Srinivasan, S., Riazi, S., Norris, B., Das, S. K., and Bhowmick, S. A shared-memory parallel algorithm for updating single-source shortest paths in large dynamic networks. In 2018 IEEE 25th International Conference on High Performance Computing (HiPC), pp. 245–254, 2018. doi: 10.1109/HiPC.2018.00035.