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

Juntao Luo Yumin Chen

URL: https://juntaoluo.github.io/618-Project/Index

Schedule

Here is our updated schedule for this project. Note that past tasks were more general since they were completed before this report. Future tasks are more specific.

Week	Task	Assignee
Nov. 7 - Nov. 13	Finish project proposal and study related research paper thoroughly	Completed, Jun Tao Luo and Yumin Chen
Nov. 14 - Nov. 20	Build Data Structures, including Node, Stack, AdoptDesc and PriorityQueue	Completed, Yumin Chen
Nov. 14 - Nov. 20	Implement the coarse-grained concurrent priority queue	Completed, Jun Tao Luo
Nov. 21 - Nov. 27	Implement concurrent DELETEMIN, including Logical deletion and batch physical deletion	Completed, Yumin Chen
Nov. 21 - Nov. 27	Implement benchmark scripts and reference algorithm (parallel Dijkstra's) using concurrent priority queues.	Completed, Jun Tao Luo
Nov. 21 - Nov. 27	Start to work on project milestone report	Completed, Jun Tao Luo and Yumin Chen
Nov 30	Finish the project milestone report	Completed, Jun Tao Luo and Yumin Chen
Dec 2	Implement concurrent INSERT	In Progress, Jun Tao Luo
Dec 2	Test and debug correctness of the MDList priority queue	In Progress, Yumin Chen
Dec. 6	Profile the MDList priority queue, and analyze performance	Pending, Yumin Chen

Dec. 6	Create graphs showing performance statistics and make comparisons with other mature lock-free concurrent priority-queue if applicable	Pending, Jun Tao Luo
Dec. 9	Final project report draft	Pending, Jun Tao Luo
Dec. 9	Final project poster draft	Pending, Yumin Chen
Dec. 9	Evaluate feasibility of HOPE TO ACHIEVE tasks	Pending, Jun Tao Luo and Yumin Chen
Dec.13	Complete final project report	Pending, Jun Tao Luo
Dec. 13	Complete HOPE TO ACHIEVE tasks	Pending, Jun Tao Luo and Yumin Chen
Dec. 17	Complete the final poster	Pending, Yumin Chen

Summary

We changed the order of some of the tasks outlined in our original proposal, but the planned items are mostly unchanged.

In addition to using only microbenchmarks as a method to evaluate our data structure in our proposal, we decided to add a reference algorithm as a more realistic benchmark. For this purpose we have implemented a parallelized Dijkstra's algorithm that uses a concurrent priority queue [1]. This has a further benefit of allowing us to empirically confirm the correctness of the concurrent priority queue by comparing the results of the parallelized algorithm with the sequential algorithm.

We have completed our testing harness which includes scripts to generate input graphs for parallelized Dijkstra's algorithm, test the correctness of the output and compute the performance of the parallelized algorithm using different implementations of concurrent priority queues against the sequential algorithm and each other. We have put off writing specific microbenchmarks since we believe it will be simple but potentially dependent on the exact APIs of each of our priority queue implementations so we'll work on this once we complete our priority queue implementations.

For the implementation of MDList based lock free concurrent priority queue, we have implemented the data structures it uses, the DeleteMin algorithm and are currently working on Insert. We are also in the process of debugging our current implementation and have not yet tested the functionality end to end using our parallelized Dijkstra's algorithm.

Goals and Deliverables

Our objectives are similar to the ones stated in the proposal with a minor change with respect to coarse-grained and fine-grained concurrent priority queues. We found that the fine-grained concurrent priority queue to be similar in complexity to the MDList based lock free concurrent priority queue so we decided to use a coarse-grained concurrent priority queue instead as our baseline in order to maximize the time we have for implementing our lock free data structure. Instead we'll implement a fine-grained concurrent priority queue for further comparison if we have additional time.

Here's our updated goals:

- PLAN TO ACHIEVE
    - Implement a coarse-grained concurrent priority queue
    - Implement a concurrent lock-free priority queue based on multi-dimensional linked lists in C++, which theoretically can achieve an average of 50% speedup over other versions of priority queue[1].
    - Implement a benchmarking script to evaluate the performance of the concurrent lock-free priority queue and come up with an analysis report regarding the influence of both the dimensions and number of threads on the throughput
- HOPE TO ACHIEVE
    - Compare the performance of different versions of concurrent priority queue(e.g. TBBPQ and LJPQ) with the MDList priority queue
    - Improve the concurrent lock-free priority queue based on profiling statistics, such as memory access and cache management using concepts learned in the course
    - Implement a fine-grained concurrent priority queue

Poster Session

Our plan for our poster session remains mostly the same as the proposal with the exception of using a coarse-grained concurrent queue instead of fine-grained concurrent queue:

- Graphs displaying the comparison of the performance of coarse-grained concurrent priority_queue and the MDList lock-free concurrent priority_queue across different threads, and different INSERT / DELETEMIN ratios.
- Graphs displaying the lock-free concurrent priority queue's performance across different workload ratios (e.g. 50% of INSERT, 75% of INSERT), different number of threads and different INSERT / DELETEMIN ratios.
- The comparison of existing skip list lock-free concurrent priority queue and MDList lock-free concurrent priority queue across different number of threads and different INSERT / DELETEMIN ratios, if time we complete these objectives.

Preliminary Results

We measured the speed up for coarse-grained concurrent priority queue compared to the sequential implementation of priority queue of std::priority_queue for the Dijkstra's benchmark:

```
 -- Performance Table ---
 Scene Name      | 4               | 8
-----------------------------------------------------
 bench-64        | 0.000247        | 0.000989
 bench-256       | 0.0003          | 0.000503
 bench-1024      | 0.002944        | 0.00348
 bench-4096      | 0.032007        | 0.032112
 bench-8192      | 0.12489         | 0.11085

-- Speedup Table ---
 Scene Name      | 4               | 8
-----------------------------------------------------
 bench-64        | 0.068826        | 0.017189
 bench-256       | 0.946667        | 0.564612
 bench-1024      | 1.756454        | 1.485920
 bench-4096      | 3.089168        | 3.079067
 bench-8192      | 3.310625        | 3.729941
```

Concerns and Unknowns

Our current progress is mostly on track and we will likely achieve our planned goals on time. However, we did encounter some surprises that made us adjust our plans slightly from the initial proposal. As mentioned previously we changed our baseline concurrent priority queue benchmark from fine-grained to coarse-grained.

We also found that there are few state of the art parallel algorithms that use concurrent priority queues. For example, while we used a parallelized Dijkstra's algorithm that's described in [1], we found that current best performing single source shortest paths algorithms avoid using concurrent priority queues [2]. Although this does not invalidate our usage of parallelized Dijkstra's as a more realistic benchmark to evaluate performance over synthetic microbenchmarks, it is unfortunate it is not representative of current state of the art algorithms.

References

[1] Tamir, O., Morrison, A., & Rinetzky, N. (2016). A heap-based concurrent priority queue with mutable priorities for faster parallel algorithms. In 19th International Conference on Principles of Distributed Systems (OPODIS 2015).
[2] S. Srinivasan, S. Riazi, B. Norris, S. K. Das and S. Bhowmick, "A Shared-Memory Parallel Algorithm for Updating Single-Source Shortest Paths in Large Dynamic Networks," 2018 IEEE 25th International Conference on High Performance Computing (HiPC), 2018, pp. 245-254, doi: 10.1109/HiPC.2018.00035.