15-418/618, Parallel Computer Architecture and Programming Final Project: Parallel Minimum Spanning Tree Algorithms

Xuren Zhou (xurenz), Wenting Ye(wye2)

November 19, 2019

1 URL

GitHub Page: https://allenchou.github.io/CMU-15618-Final-Project/.

2 Schedule

The up-to-date schedule in provided in Table 1.

Week	Due	Task	Assigned to	Status
1	11.3	Project Proposal	Both	Completed
1	11.3	Implement graph data structure and graph data generator	Both	Completed
2	11.10	Implement disjoint set data structure	Wenting Ye	Completed
2	11.10	Implement sequential Kruskal's algorithm	Wenting Ye	Completed
2	11.17	Implement parallel Kruskal's algorithm with parallel sorting	Xuren Zhou	Completed
3	11.17	Implement sequential Kruskal's algorithm with Filter-Kruskal	Xuren Zhou	Completed
3	11.17	Project checkpoint report	Xuren Zhou	Completed
4	11.21	Implement parallel Kruskal's algorithm with Filter-Kruskal	Xuren Zhou	In progress
4	11.24	Initial benchmark and profile parallel Kruskals' algorithms	Xuren Zhou	In progress
4	11.24	Implement correctness checker using Boost	Wenting Ye	
4	11.28	Implement sequential Borůvka's algorithm	Wenting Ye	
5	11.28	Implement sequential Borůvka's algorithm with edge contraction	Xuren Zhou	
5	12.1	Implement parallel Borůvka's algorithm	Wenting Ye	
6	12.1	Initial benchmark and profile parallel Borůvka's algorithm	Wenting Ye	
6	12.8	Large benchmark for both algorithms on GHC/CloudLab	Xuren Zhou	
6	12.8	Poster and final report	Both	

Table 1: Project schedule.

3 Summary of Completed Work

We have implemented graph data structure, random graph generator, sequential Kruskal's algorithm, Kruskal's algorithm with parallel sorting and sequential Filter-Kruskal's algorithm. Some initial benchmark experiments are in progress and we will show some preliminary results in the later section.

4 Goals and Deliverables

4.1 Goals

4.1.1 Plan to achieve

- Implement two sequential algorithms, Kruskal's algorithm and Borůvka's algorithms, as the baseline of our performance benchmark.

- Parallelize these two sequential algorithms in shared-memory model using OpenMP:
  - Krushal's algorithm with parallel sorting and Filter-Kruskal [1],
  - Borůvka's algorithm with edge contraction [2].
- Benchmark the speedup performance of our parallel implementations under different types and sizes of input graphs. We plan to consider two kinds of graphs: dense graphs and sparse graphs.
- Explore the speedup performance of different parallel components, such as the parallel sorting in Krushal's algorithm and the parallel finding adjacent edge with the smallest weight of each vertex in Borůvka's algorithm.

4.1.2 Hope to achieve

- Explore the speedup performance on input graphs with power-law distribution.
- Implement a faster shared-memory MST algorithm proposed by Bader *et al.* [3].
- Explore the possibility to implement parallel Krushal's and Borůvka's algorithms in message-passing model using MPI.

4.2 Progress

4.2.1 Original goals

For the algorithm part, We have implemented most of Kruskal's algorithm: including the sequential baseline, Kruskal with parallel sorting and sequential Filter-Kruskal. For sequential baseline, we use `C++` STL sorting algorithm while in our parallel sorting, we implement a recursive quicksort and use OpenMP task to parallelize it. Although our qicksort implementation is slower than `C++` STL sorting, the parallel version is faster than `C++` STL sorting.

The parallel version of Filter-Kurskal is still under exploration. The basic idea of Filter-Kruskal is to construct minimal spanning forest for small edges first, and then filter out large edges whose endpoints within the same tree and then run Filter-Kruskal on the remaining large edges. It is a recursive algorithm and there is a strong dependency between each recursive layer. We try to use OpenMP task dependency to generate tasks but the dependency makes the running time worse. Instead of parallelizing the recursive tasks, we decided to focus on the tasks within each recursive layer: namely the partition and filter. After the checkpoint, we are going to implement our partition and filter via prefix-sum. So far, we just use `C++` STL, which is a single thread algorithm.

For the graph generator, we implement a random graph generator: For each pair of vertices u and v, there is p probability such that $e = (u, v) \in G$. Once $e \in G$, the weight of e is uniformly distributed on a given range. Because the probability is a fixed value, our graph generator outputs a dense graph. During our initial benchmark, we notice that Kruskal's algorithm spends most of its running time on sorting, therefore it is expected to get good speedup once we parallelize the sorting. However, when we run the same data on sequential Filter-Kruskal, we get better running time, even better than Kruskal with parallel sorting. This inspires us to explore the performance of our implementation on different graph type. To accelerate the saving and loading data, we use binary file to store the data.

4.2.2 Extra goals and new goals

For extra goals, we consider to add graphs with power-law distribution as our testing data to our final deliverables. We want to focus on our parallel implementation on shared-memory model so we decided not to consider to explore these two algorithms using MPI. We will spend more time on the benchmark experiment design. If we have time, we will consider to implement Bader's algorithm.

There are things we want to add to our new goals: correctness and high-level parallel modules. We consider to add a correctness checker implemented via Boost, which contains a MST implementation and can be used as a new baseline to test our speedup performance. Meanwhile, we notice that there are some native parallel module in some specific `C++` compiler, such as `libstdc++` parallel mode in GCC and TBB. `libstdc++` parallel mode uses shared-memory model to parallel some STL algorithms such as sorting, partition, etc. So we think it is also a good point to explore.

To summarize, we add two new basic goals, graphs with power-law distribution and Boost checker, and one extra goal, exploring `libstdc++` parallel mode.

5 Poster Deliverables

We will deliver the speedup graphs of our parallel implementations under different numbers of threads and different types and sizes of input graphs. We will also deliver speedup graphs, or tables, of different parallel components. We think a reasonable speedup with respect to the number of processors will demonstrate that demonstrate we did a good job. If we cannot get a good speedup, we will try to profile our implementations to show that the parallel overhead is inevitable under our testing platform and provides potential specs of the testing platform to improve the speedup of our implementations.

6 Preliminary results

We conduct an initial benchmark with graph of node size 5000, 10000 and 20000 and edge probability 0.05, 0.1 and 0.2. We use `random-N-p` to represent the case of a random graph of node size N and edge probability p. We use STL sorting implementation of sequential Kruskal's algorithm as our baseline. The running platform is on my personal MacBook Pro with 2.6 GHz 6-Core Intel Core i7 CPU. The result is shown in Table 2, in which the sequential sort represents our OpenMP implementation without OpenMP `#pragma`.

case	baseline	sequential sort	parallel sort	sequential filter
random-5000-0.05	1	0.6732376416	1.657943461	2.657509953
random-10000-0.05	1	0.6603400285	1.690410944	2.682852532
random-20000-0.05	1	0.5990178821	1.592128318	4.339100392
random-5000-0.1	1	0.693282639	1.535450117	3.063387064
random-10000-0.1	1	0.5830809685	1.37825495	4.677607176
random-20000-0.1	1	0.5829203993	1.85125072	4.067878587
random-5000-0.2	1	0.6493750125	1.847814554	3.441612889
random-10000-0.2	1	0.5766389356	1.425928101	3.410266252
random-20000-0.2	1	0.5862568891	1.904486512	4.300331156

Table 2: Preliminary results of speedup performance.

This result is only used to show that our OpenMP indeed has some speedup. Meanwhile, it reflects some potential issues in our implementation: we expect the speedup can be about $\times 6$ because we have 6 cores in our CPU. However the result shows that it is about $\times 1.5$ speedup. Even if we consider the speedup with respect to sequential sort, the speedup is still only about $\times 3$. Therefore, further analysis and optimization is required to get better performance.

We breakdown the running time of baseline and sequential sort: Sorting takes about 84% running time in baseline and 91% running time in our sequential sort.

One interesting observation is that sequential filter is quite efficient, which indicates that there is a lot of useless sorting work in original Kruskal's algorithm.

7 Issues

Issues that concern us the most have been discussed in previous sections. Here we summarize them into a list:

- How to optimize the parallel performance to fully utilize the CPU parallel resource. So far, our initial implementation can only achieve about $\times 3$ w.r.t. our sequential sort implementation. Further optimization is required after we finished basic requirement.
- Parallel partition: it is not trivial to parallelize partition algorithm. The main reason is that the position of the element is not independent with others. One possible way is to use prefix-sum. It is natural to implement prefix-sum using SIMD model, such as CUDA or SSE CPU instruction. The performance of OpenMP implementation is unknown to us, so we need to spend some time to explore it.
- Different algorithms have different performance on different graph type. It is not enough to only consider random graph with uniform distribution. How to design the benchmark is also a potential issue in our project

deliverables. So far, we consider to add sparse graph (constant vertex degree upper bound) and random graph with power-law distribution.

- There is no correctness checker in our current implementation. There is a MST implementation in Boost, which can be used as checker in our project. However, we are not sure whether it is easy for us to use it.
- We have noticed that `C++` STL algorithm library is quite efficient in a single thread. We are worried about that the bad speedup performance is due to some unknown optimization in STL. One possible solution is to compute the speedup without STL. Another way is to take advantage of `libstdc++` parallel mode. We have no experience using the later one so we put it in our extra goals.

References

[1] V. Osipov, P. Sanders, and J. Singler, "The filter-kruskal minimum spanning tree algorithm," in *Proceedings of the Meeting on Algorithm Engineering & Expermiments*, pp. 52–61, Society for Industrial and Applied Mathematics, 2009.

[2] S. Chung and A. Condon, "Parallel implementation of bouvka's minimum spanning tree algorithm," in *Proceedings of International Conference on Parallel Processing*, pp. 302–308, IEEE, 1996.

[3] D. A. Bader and G. Cong, "Fast shared-memory algorithms for computing the minimum spanning forest of sparse graphs," *Journal of Parallel and Distributed Computing*, vol. 66, no. 11, pp. 1366–1378, 2006.