Milestone Report: Parallelizing MST Using Borůvka's Algorithm

Shubham Bhargava & Jake Zaia

Project Webpage Link: https://jzaia18.github.io/15618-Final/

Project Status

Work Thus Far

Over the last few weeks we have implemented Boruvka's algorithm 3 separate times: sequentially in C++, using parlaylib, and using CUDA. The sequential version is lightly optimized for sequential execution and will be used to benchmark our parallel implementations against. For our CPU implementation, we have first translated our implementation to Python (in a functional style). Then, we converted it to parlaylib code ensuring that our code still matches the theoretical guarantees of the algorithm while maximizing usage of parlaylib primitives as they've been optimized for memory usage and scheduling. This step involved designing a simple concurrent union-find where we relied on C++ atomics. Compare and exchange was also useful in updating shared memory when finding the lightest edge per vertex (the first part of the algorithm). For the GPU parallel implementation, the CUDA code runs and is correct, but is currently not doing any useful parallelization or obtaining any speedup. Optimizing the CUDA implementation will be the focus of the remainder of our effort on this project.

We have also built up infrastructure for testing our implementations, including a graph generation python script, several datasets, and a python implementation of Kruskal's algorithm to verify correctness of outputs. While these scripts are not the primary portion of our project, they serve an important role in terms of gathering data to benchmark and analyze our code.

Goals & Deliverables Going Forward

Of our 3 goals, we have already achieved 2. We have a lightly optimized sequential version of Boruvka's that performs decently well and a CPU-parallel version written in parlaylib that achieves a noticeable speedup. Our final goal was to implement a parallel version of MST in CUDA that scales near-linearly. There are more challenges with implementing the CUDA version than initially anticipated (further details in the challenges section), so while this is still a goal, we also propose an alternate goal. The alternate goal would be to benchmark Boruvka's algorithm running with different implementations of a concurrent disjoint-set. The disjoint-set data structure is crucial to the algorithm's operation, and different design decisions would likely have significant

performance ramifications. If we are not able to achieve good speedups, we would like to instead compare and contrast these effects. Our stretch goals remain the same.

Thus, the current list of our goals is:

Primary Goals (Aim to complete 3)

- ~~Implement an optimized sequential version of MST using Baruvka's algorithm for benchmarking the parallel version against.~~ (COMPLETED)
- ~~Implement a CPU parallel version of Baruvka's in parlaylib that can be used for further benchmarking. In the event that we encounter significant difficulty and are unable to implement a CUDA version that achieves good speedup, we will instead use the benchmark data of our parlaylib for analysis.~~ (COMPLETED)
- Implement a parallel version of MST in CUDA that scales near-linearly as more GPU threads are used.
- Implement at least 2 meaningfully different versions of a concurrent disjoint-set. Gather significant benchmark data for a parallel implementation of Boruvka's running using these different implementations.

Additional/Stretch Goals (Aim to complete ≥0)

- Implement a parallel version of MST that performs close to, or better than, the existing implementations in Open MP and MPI.
- Implement an MST-approximation algorithm that performs better than existing MST implementations while getting close to optimal results.

Plans for Poster Session

We plan to present a series of graphs and figures for the poster session. Given that our project processes data from arbitrary graphs, it would not be particularly interesting for a live demonstration. Moreover, given that we will have multiple implementations benchmarked against one another, the best way to display this data will be through graphs and charts. If we do not pivot, we will be demonstrating the relative speedup of the 2 parallel implementations compared to the sequential implementation. If we need to pivot, we will instead benchmark the parlaylib implementation using the 2 different disjoint-set implementations against the sequential implementation. If we are forced to pivot, we will also provide data on the execution of the CUDA implementation that shows that the SIMD execution caused issues parallelizing over the disjoint-set.

Preliminary Results

Time taken to find MST on dense highly connected graph with 1000 vertices:

Sequential	Parlaylib (4 cores)
0.0535	0.0113
0.0561	0.0087
0.0946	0.0085

Time taken to find MST on dense highly connected graph with 1000 vertices:

Sequential	Parlaylib (4 cores)
1.933	0.360
2.054	0.282
2.233	0.224

In the future, we intend to use RMAT graphs, Erdos-Renyi graphs, and other types of random graphs to better represent realistic graphs (in addition to actual graph datasets). Our randomly generated graphs were highly connected which may not be representative of all real datasets.

Challenges

Now that we have working implementations for Boruvka's algorithm, there are some potential hurdles with parallelizing the algorithm which may be difficult to overcome. Namely, there are several conditioned loops within the algorithm that terminate early or skip iterations. These will not perform well for SIMD execution, and we may not be able to circumvent this issue. Thus, for the CUDA implementation, it may be the case that a speedup will be miniscule due to warp stall. We will have to expend considerable effort to mitigate this issue, and may need to pivot our project to instead focus on benchmarking different implementations for parallel disjoint-set and how they affect the runtime of our implementations. This will still produce interesting results since, when trying to parallelize using parlaylib, we realized that there were a lot of different ways to implement the algorithm which made the space of potential optimizations large.

Another challenge has been sourcing data. We expected that it would be easy to get datasets of weighted undirected graphs in similar formats and with various sizes. It turns out that accumulating datasets has been a challenge. We have created a Python script for generating graphs to use as test data in addition to attempting to use

open-source data. Obtaining good, diverse, and large datasets remains an ongoing challenge.