15-418 Project Milestone Report

Parallelizing Boruvka's Minimum Spanning Tree Algorithm and Union Find Data Structure

Noor Mostafa and Rohan Shenoy

---

https://noor-5.github.io/parallelgraphs.github.io/

Summary
We had initially proposed performing parallel spectral clustering on graphs as our project. This would have entailed representing a graph as a matrix, performing linear algebra computations on the matrix, then running K-Means clustering. Based on our proposal feedback, we were made aware that K-means would not be enough for the project. As we investigated the steps needed for the projection, we soon recognized that much of it was linear algebra, which has been parallelized quite extensively already.

As a result, we pivoted to a new project. We have decided to write parallel implementations of Boruvka's algorithm for determining the minimum spanning tree of a graph. To quote Wikipedia, a minimum spanning tree (MST) is a "subset of the edges of a connected, edge-weighted undirected graph that connects all the vertices together, without any cycles and with the minimum possible total edge weight."

Boruvla's algorithm works by finding the minimum edge out of each vertex, grouping these together in each step, and iterating with the contracted graph. As we researched the algorithm further, we came across two main implementations of this contraction step: one using a hashmap of each vertex to its new vertex in the contracted graph, and another using the Union-Find (Disjoint Set) data structure.

We have decided we will work on writing parallel versions of both implementations in OpenMP, as well as experiment with different types of locking on the Union-Find data structure, namely: coarse-grained locking, fine-grained locking, and lock-free. We will be using C++ and running on the GHC machines.

Deliverables

1. Fast sequential versions of Borukva's algorithm, using a star contraction and Union-Find approach.
2. OpenMP parallel versions of the sequential versions.
3. Coarse grained locking on the Union-Find Data structure.
4. Fine grained locking on the Union-Find Data structure.
5. Lock-Free locking on the Union-Find Data structure.

Nice-to-Have

1. Experiment with Graph sharding to overcome memory constraints
2. Implement Boruvka's on a GPU using a data parallel approach.

Schedule

Date	**Task**	**Status**
March 31 - April 6	Research Boruvka's algorithm and opportunities for parallelism - Noor and Rohan	Done
April 7 - April 10	Write a sequential version of Boruvka's algorithm using star contraction - Noor and Rohan	Done
April 10 - April 13	Write graph generation script and MST validation script - Rohan	Done
April 14 - April 16	Write a sequential version of Boruvka's algorithm using a sequential Union Find - Noor	Done
April 17 - April 20	Use OpenMP to parallelize both sequential versions - Noor and Shenoy take on one each	In Progress
April 21 - April 23	Implement coarse grained locking for Union Find - Noor and Shenoy	To Do
April 24 - April 27	Implement fine grained locking for	To Do

	Union Find - Noor and Shenoy	
April 28 - May 1	Implement lock-free Union Find - Noor and Shenoy	To Do
May 2 - May 5	Implement a nice-to-have feature, Create poster, write - Noor and Shenoy	To Do

Progress

We have implemented both sequential versions, one using star contraction, and another using Union-Find. This required research to understand the algorithm and the various implementations. We have also set up the testing infrastructure that we will use for the remainder of the project to ensure correctness. This involved writing a Python script to create connected graphs with the requested number of nodes and edges. We also have written a python script to test whether our output is truly an MST by using Python's Networkx graph library.

We have also begun assessing our sequential code for parallelism. We have considered parallelizing over for loops for vertices and edges, and the tradeoffs between them. We have decided to implement OpenMP dynamic scheduler for assigning vertices to processors since there is variable time when it comes to finding minimum edge, depending on how many outgoing edges a vertex has. We will also use a static approach as a reference comparison. One challenge is maintaining shared vectors between threads and preventing race conditions. The race conditions can be problematic especially in the contracted implementations because we are directly changing the graph hence we don't want threads to simultaneously be contracting the same vertex. Furthermore, for the union find, we implemented a shared data structure to keep track of the groups of vertices that have been grouped together. Hence we need to ensure atomicity when accessing regions of this data structure which is where our different locking/ lock free implementations will come into play.

Poster Session

We plan on showcasing graphs displaying performance in terms of speedup of the parallel versions of each implementation as compared to the sequential version on different sizes of graph inputs. This will allow us to directly compare tradeoffs between each implementation as well as how well the implementations scale as we increase processor count. We also plan on demoing visuals of smaller graphs and their computed MST using online graph illustration software to display the structure behind a MST. Furthermore, we will be displaying stall times/ overhead along with memory/cache misses with the different types of locking to demonstrate their tradeoffs when it comes to performance.

Issues and Concerns

For the most part, it is about coding up the parallel versions and implementing the locking on each implementation. Since we were able to guarantee a correct sequential version correctly and have full understanding of the underlying algorithm , we are pretty confident that we can implement the OpenMP version parallel version of the implementations. The main concern would be with the lock free implementation on the Union-Find (Disjoint Set) data structure as we haven't exactly planned out how we are going to go about implementing it and verifying correctness. However, we understand where the parallelism and data dependencies lie in our program so we are hopeful we can resolve this issue.