Parallelizing Minimum Spanning Tree Using Borůvka's Algorithm in Parlaylib and CUDA

Shubham Bhargava & Jake Zaia

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

Summary

We are going to implement an algorithm for finding the Minimum Spanning Tree of a graph in parallel. We will be implementing this algorithm in CUDA and comparing its performance to an optimized sequential version of the same algorithm benchmarked on road and social network graph datasets. We will additionally be benchmarking against a CPU parallel version implemented using parlaylib graph primitives.

Background

Finding the Minimum Spanning Tree (MST) is a common graph problem with many applications in approximate algorithms, network design, image segmentation, and taxonomy. Many sequential algorithms have been designed for it such as Prim, Kruskal, and Boruvka that run in O(m log n) time. Boruvka's algorithm is commonly used for parallel applications as it is easy to parallelize in polylogarithmic time.

Boruvka's algorithm works by using the lightest edge property where the lightest neighboring edge of a vertex is placed in the MST. By adding the minimum edge for each vertex to the list of MST edges, we get a bunch of connected components. We can treat these connected components as vertices (removing any self-edges). Then we repeat the previous steps until all vertices are connected. This is inherently parallel as the minimum edge for each vertex can be calculated in parallel.

The Challenge

Implementing Boruvka's algorithm is tricky as it requires maintaining a shared disjoint set data structure. We have to ensure fast memory access to the disjoint-set while ensuring correctness. Optimizing the disjoint-set might involve figuring out ways to break it down into smaller sets that don't interact with each other as much. What makes this truly challenging is that warps in a GPU are SIMD and traversing the disjoint set data structure could result in divergent execution. Additionally, the disjoint set keeps track of the connected components that act as vertices in each iteration in the algorithm. This means that we need to contract newly connected components into vertices. It will

be a challenge to choose the most efficient contraction method for this step removing any self-edges.

We are hoping to become better CUDA programmers through this project. Additionally, we are hoping to apply our knowledge of parallel algorithms beyond just the asymptotics taught in class. We are hoping to implement efficient shared data structures suited to the specific problem at hand.

Resources

Since we will be implementing this algorithm in CUDA, we will need access to Nvidia GPUs. We are planning on using the GHC machines with the same setup as assignment 2. Parlaylib is a header-only library, and thus requires no extra resources. We will not be using any starter code, however as Boruvka's is a well-studied algorithm, plenty of pseudocode is available for reference (such as in the 15-210 and 15-852 notes). It would be interesting to test this project on the PSC machines as well, if possible.

There are a couple of papers, (such as Paper 1, Paper 2), that attempt to optimize parallel Boruvka's that we can use as a reference and possibly compare against. We can also use the same datasets as the papers used for testing in addition to randomly generated test sets.

Goals and Deliverables

Plan to Achieve

- Implement an optimized sequential version of MST using Baruvka's algorithm for benchmarking the parallel version against.
- 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.
- Implement a parallel version of MST in CUDA that scales near-linearly as more GPU threads are used. Since this algorithm is highly parallelizable, we should be able to realize a speedup which scales well.

Hope to Achieve

- 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.

Platform Choice

We will be coding in C++ so we can use Parlaylib for our CPU parallel implementation. We will be testing this implementation on the GHC and, if possible, PSC machines to get CPU baselines to compare with our CUDA implementation. The CUDA implementation will be tested on the lab machines. The baseline sequential implementation will be done in C++ for fairness.

Schedule

Week	**Goal**
Apr 1	Implement sequential version of Baruvka's algorithm, begin programming CPU-parallel version of Baruvka's using parlaylib
Apr 8	Finish implementation of Baruvka's using parlaylib
Apr 15*	Intermediate milestone deadline (Apr 16) Implement Baruvka's in CUDA: divide parallelizable code into kernels and ensure correctness (without speedup)
Apr 22	Begin optimizing CUDA implementation
Apr 29	Fine tune CUDA implementation and start aggregating metrics into a presentation
May 6*	Final poster presentation (May 6)