# Parallel MST Algorithms

*Mengrou Shou (mshou)*
*Jacky He (junchenh)*

## Summary

We will create optimized implementations of Minimum Spanning Tree (MST) algorithms on both GPU and multi-core CPU platforms, and perform a detailed analysis of both systems' performance characteristics. Specifically, we'll be working with enumerable graphs and exploring using existing MST algorithms such as Prim's Kruskal's, Boruvka's, and their variants.

## URL

https://meng87.github.io/parallel-mst/

## Background

Formally, the spanning tree T = (V', E') of a graph G = (V, E) is a tree whose edges are subsets of the set of all edges (E) in the graph, and V' = V (the tree spans all vertices in the graph).

A minimum spanning tree (MST) is a spanning tree of minimum weight, where the weight of a tree is defined as the sum of the weight of all edges in that tree.

MST algorithms rely on the Light-Edge Property, which states that if G is a connected, undirected, and weighted graph with distinct edge weights, then for any cut of G, the minimum weight edge that crosses the cut is contained in the MST of G.

To find an MST, we just need to find all edges that are MST edges as determined by the Light-Edge Property.

A typical parallel algorithm for finding the MST is the Boruvka's algorithm, whose pseudocode is shown below:

Boruvka's: V vertices, E edges

- Start out with V disconnected components
- while (size of edge set > 0):
    - For each vertex, mark the minimum outgoing edge from that vertex
    - Flip a coin to split vertices into heads and tails
    - For each edge, check if it's a tail head edge. If yes, that edge is marked for contraction.
        - Map each tail vertex to it's star center (head vertex)

  - For each edge (u, v), check if u and v have the same star center. If yes, remove that edge.

This algorithm might benefit from parallelism because at many stages of the algorithm, we're simply operating on a collection of vertices or edges independently of each other before moving to the next stage. Therefore, if the graph is large enough, each of those stages might benefit from parallel computation.

## The Challenge

**Challenge:**
The primary challenge of the problem comes from the fact that we need to contract the graph to a smaller graph at each step in the algorithm. Rebuilding the graph is expensive, and we need to find an efficient way to represent the contracted graph (possibly trying out different contraction approaches). Additionally, contracting the graph is an inherently sequential process. It's a point of synchronization that limits the amount of parallelism in this problem. If we can find a way to reduce such synchronization (by cutting down the number of rounds of contraction, or not contracting at all), perhaps there are some ways to increase parallelism.

**Workload:**
There are dependencies from each step of the algorithm to the next. With Boruvka's algorithm, the graph needs to be contracted some number of times. The process of contraction is inherently sequential.

The memory access patterns are based on the vertex numbers of each vertex. There are lots of accesses where we're just looking at the neighborhood of each vertex. For a typical graph, those accesses might be random, which would pose a challenge for locality. However, this also presents opportunities for optimization if we can artificially create locality by restructuring the graph so that neighboring vertices are also stored closer in memory.

The algorithm is very memory access driven in that it frequently performs a bulk operation on some collection of edges and vertices (those collections are usually stored in the form of an array). This means that there's a high communication-to-computation ratio, where the communication here refers to memory accesses. We wouldn't have many opportunities to do a lot of arithmetic operations after each load or store before we have to do another load or store, which makes taking advantage of locality crucial.

**Constraints:**
Because our workload is very memory-access driven, this means that the problem is at least constrained by the memory transfer time of the hardware and by how big the cache sizes are. For example, we might choose to map the problem to GPUs or CPUs in very different ways because of how memory is organized when computing with either of those hardware components.

## Resources

### Algorithmic Resources

Given that most MST algorithms are not very long, we'll start by first writing a fast sequential MST algorithm (most likely using either Kruskal's or Prim's), and then we'll code up a single-threaded version of Boruvka's from scratch (most likely using Disjoint-Set Union).

When optimizing our algorithm, we plan to consult several papers in achieving good performance on different platforms. Two examples of papers that we plan to reference are: A Generic and Highly Efficient Parallel Variant of Boruvka's Algorithm and A High-Performance MST Implementation for GPUs.

### Machines

We plan to use the GHC machines for coding with GPUs. For CPUs, we plan to use the GHC machines or the PSC machines benchmarking performance with a large number of cores.

## Goals and Deliverables

Plan to Achieve

1. Data Parallelism
    a. Different variations of the MST algorithm using CUDA, and seeing what's the best variant for this specific platform by performance benchmarking
2. Shared Address Space
    a. Try out different approaches to speedup MST on a shared address space model
    b. Goal: Near linear-speedup from single-threaded variant of this problem. The theoretical work and span for Boruvka is $O(m \log n)$ and $O(\log^2(n))$, which means that parallelism is approximately $P = m/\log(n)$. This is the theoretical maximum number of processors that the algorithm can effectively utilize. Therefore, if the graph is large enough, it should be able to effectively utilize all processors of a machine.
3. Message Passing
    a. Different variations of the MST algorithm using MPI, and seeing what's the best variant for this platform by performance benchmarking
    b. Goal: Just like in the shared address space model, we'd like to achieve near linear-speedup from single-processor variant of this problem. However, this might pose more challenges because of potential extra communication overhead with an increasing number of processors.

Hope to Achieve

1. Explore using Graph DSL to speedup MST
    a. CUGraph

    b. GraphLab
2. Explore heterogenous approaches and explore the different thresholds with which different parallelization paradigms are desirable. Ideally, this would involve figuring out the best way to combine message passing and shared address space models or message passing and CUDA and experimentation with problem sizes.

## Platform Choice

We'll use CUDA for data parallelism since this is the mainstream method for coding on a GPU. For shared address space, we'll most likely use OpenMP and ISPC. If we find the need for additional threads control, then we will use a combination of pthreads and OpenCilk.  We chose OpenMP and ISPC because we're familiar with those tools by having used them in projects in the past, and they both work under the assumption of a shared address space model. For a similar reason, we'll use MPI for message passing.

Our programming languages will be in C++ to take advantage of its high performance. However, if we decide to explore Graph DSLs such as CUGraph, then we might also need to code in Python.

## Schedule

| Date | Deliverable |
|---|---|
| April 6 | Initial Fast Sequential MST Implementation + CUDA Approach Complete + Benchmarked |
| April 13 | Shared Address Space Implementation Complete + Benchmarked |
| April 16 | Intermediate Milestone<br>1. Complete CUDA and OpenMPI approach |
| April 20 | Message Passing Implementation Complete + Benchmarked |
| April 27 | Explore CuGraph + GraphLab |
| May 4 | Heterogenous Parallelism Approach Tuning Complete |