Parallelizing MST Algorithms: Project Proposal

Alex Knox (akknox), Elizabeth Knox (emknox)

March 27, 2025

1 Summary

In this project, we will explore the parallelization of three algorithms for finding the minimum spanning tree of a graph: Prim's algorithm, Kruskal's algorithm, and Boruvka's Algorithm. We will implement them using different parallelization strategies, comparing the speedup obtained in a shared address space model or a message passing model.

2 Webpage

https://akknox.github.io/mstparallelization.github.io/

3 Background

Minimum spanning trees are useful in many applications, such as network design, clustering, and efficiently approximating the traveling salesman problem. Many of these real-world applications involve handling large datasets. Thus, we are interested in scaling the performance of minimum spanning tree algorithms through parallelization.

Minimum Spanning Tree (MST). The formal definition of a minimum spanning tree is as follows. Consider a connected, undirected graph $G = (V, E)$ where each edge e_i is assigned a positive weight w_i. A minimum spanning tree is a connected subgraph $G' = (V', E')$ of G with $V' = V$ and $E' \subseteq E$ such that $\sum_{e_i \in E'} w_i$ is minimized over all such subgraphs. In other words, it is a tree subgraph of G with minimum total edge weight.

There are several algorithms for finding an MST, which each rely on different properties of an MST.

Prim's Algorithm. The basis for Prim's algorithm is a property of MSTs called the Light Edge Property: if we take any subset of vertices X, the edge with least weight spanning X and $V \setminus X$ must be in the MST. The algorithm iteratively constructs a visited set by selecting the lightest edge spanning the current visited set and the rest of the graph for inclusion in the MST, and adding the unvisited endpoint of this edge to the visited set. Once every vertex is in the visited set, the MST is complete. The pseudocode for this algorithm is replicated below.

Algorithm 1 Prim's Algorithm

```
X = { s } // s is arbitrary
T = { }
while X != V do:
    Find minimum weight edge (a, b) with a ∈ X and b ∈ V \ X
    X = X ∪ {b}
    T = T ∪ {(a, b)}
end while
return T
```

There are opportunities for parallelism in computing the minimum weight of all edges under consideration at each iteration (line 4), since these reads are independent. This can be helpful, especially in dense graphs, where each vertex has many neighbors. However, the outer loop is inherently sequential, since the choice of lightest edge on one iteration can change the lightest edge candidates for the next iteration.

Kruskal's Algorithm. This algorithm uses a greedy approach to select edges belonging to the MST. On every iteration, the algorithm chooses the lightest edge that has not already been selected and does not create a cycle with edges already chosen. This is based on the cycle property of an MST: for every cycle in the graph, the heaviest edge is excluded from the MST. By selecting edges in weight order, the algorithm ensures that all other edges in a cycle are always considered for inclusion before the heaviest edge in the cycle. Instead of building up a single connected component until it forms an MST, this algorithm builds up multiple connected components. Eventually, when the algorithm terminates, there will only be one connected component left, and it will be an MST.

Algorithm 2 Kruskal's Algorithm

```
1: T = { }
2: for (a,b) from lightest to heaviest do:
3:     if a and b are not in the same connected component then T = T ∪ {(a, b)}
4:     end if
5: end for
6: return T
```

Like Prim's, the outer loop of Kruskal's algorithm is difficult to parallelize because the addition of an edge during some iteration will affect whether later edges will be included in the same connected component or not, and because edges must be checked in weight priority order for the algorithm to produce correct results. However, there are ways to parallelize some of the auxiliary steps like the sorting of edges, and modifications to the algorithm which allow for more parallelism.

Boruvka's Algorithm. This algorithm uses the same light edge property as the previous two algorithms, with the additional observation that multiple cuts can be examined at the same time. For example, if we take every vertex V_i as an individual cut, the lightest edge connected to that vertex (called a vertex bridge) must be in the MST because it is the lightest edge spanning V_i and $V \setminus V_i$. Then, the components connected during that process can be contracted, so that each connected component is treated as a single vertex, and the process repeats. The contraction step can be implemented a variety of ways: most notably, tree contraction and star contraction.

Algorithm 3 Boruvka's Algorithm

```
1: T = {}
2: G = V
3: while G has more than one vertex do
4:     for each vertex v in G do
5:         find the lightest edge (v, b)
6:         T = T ∪ {(v, b)}
7:         Contract G on the edge (v, b)
8:     end for
9: end while
```

Kruskal's algorithm has more room for parallelism, since every vertex can find its lightest incidental edge at the same time. It is also possible to combine some of the ideas used in Boruvka's algorithm to adapt Prim's algorithm or Kruskal's algorithm, and make them better suited for parallelism.

4 The Challenge

Prim's algorithm and Kruskal's algorithm exhibit some inherently sequential behavior because updates to the MST T are made on every iteration of the outer loop and later updates depend on what vertices are already connected. Parallelism in the outer loops of these algorithms is not obvious and would require algorithmic modifications based on understanding underlying data dependencies. The parallelism opportunities in Boruvka's algorithm are more obvious, but as described below the computation cost is slightly higher. We aim to analyze tradeoffs between the computational differences in MST-finding algorithms and opportunities for parallelism and synchronization costs in the data structures we use to implement them. We would also aim to learn how to use properties of the input data (properties like edge density and degree of connectedness) to identify parallelism opportunities and adapt or combine approaches accordingly.

Dependencies in the MST workload for these algorithms come from updates to the MST T, because across all of the MST-finding algorithms, the computation involves computing a minimum and then checking if adding a particular edge is *necessary* to connect a new vertex to the overall tree. If other updates were made by other threads, the update may connect two vertices already in the tree. In a general graph, this forces sequential updates. Thus, updates would require synchronization to ensure they are done correctly. The cost of synchronization (and similarly, the cost of communication between threads) depends on implementation details of how we track updates. Prim's algorithm is often implemented with a priority queue and Kruskal's algorithm is often implemented with a union-find data structure; the former is more costly to synchronize.

There is additional opportunity for parallelism over two sides of a *critical edge* if we can identify an edge whose removal would disconnect the graph. This involves algorithmic changes to identify sets of vertices that are connected to the rest of the graph only by edges like these.

Overall, the memory accesses involved in these two algorithms primarily surround checking whether endpoints a and b of particular edges are connected (either with respect to T in Kruskal's or with respect to a cut X in Prim's). There is some locality, because we often want to access all neighbors of a particular vertex and an adjacency matrix (or sorted adjacency list) lends itself to this access pattern. However, there is also random access based on what is in T and X or not.

Compared to the communication cost of Prim's and Kruskal's algorithm, Boruvka's algorithm seems to have less overhead in parallelization. Boruvka's algorithm updates T from the perspective of each connected component (starting from each vertex being disconnected), rather than from the perspective of the graph as a whole, which means there will be less need for communication between two threads trying to make updates. The challenge with Boruvka's algorithm, as compared to Kruskal's is that more computation is repeated. Kruskal's algorithm sorts edges by weight first and iterates through them, so finding a minimum weight edge satisfying certain properties requires less additional computation. Boruvka's algorithm does not do this, so although communication requirements decrease, computation per thread increases.

5 Resources

We will be using GHC machines to run both shared address space and message passing implementations. We will also benefit from PSC machine access for testing the scalability of our approaches.

For testing purposes, we will want to a variety of benchmarks: some which are particularly dense, since that increases the computational cost of finding an MST, and some that are similar to real world applications, which we will need to find.

We will also look at some research into parallelizing these algorithms, including attempts to parallelize Prim's and Kruskal's. For example, this paper[1] describes a method of parallelizing Kruskal's algorithm directly, while this paper[2] provides a variant of Kruskal's better suited for parallelism. In addition, this paper[3] provides a method of augmenting Prim's algorithm with ideas from Boruvka's algorithm to produce a more parallelizable solution.

[1] https://ieeexplore.ieee.org/document/6270833
[2] https://dl.acm.org/doi/abs/10.5555/2791220.2791225
[3] https://wiki.eecs.yorku.ca/course_archive/2010-11/W/6490A/_media/public:xiwen.pdf

6 Goals and Deliverables

The goals we plan to achieve are:

- Implement a sequential version of Prim's algorithm or Kruskal's algorithm to compare speedup (because sequential versions of these algorithms will have less data-structure overhead than a sequential version of Boruvka's algorithm)
- Parallelize Boruvka's algorithm in the shared address space model: because there is not much communication, we would aim to achieve near-linear speedup. We would test this on benchmarks that cover a variety of different features of graphs, increasing the number of vertices in the graph and changing the relative density of edges.
- Parallelize Boruvka's algorithm in the message passing model: again, because of the limited communication requirements we would hope to achieve near-linear speedup. Then we would compare the performance of the two different parallel models.
- Parallelize Kruskal's algorithm and Prim's algorithm as best we can given their sequential nature. For Kruskal's this would involve creating helper threads to check for cycle-creating edges while the main thread does the sequential computation, and for Prim's algorithm, this would involve parallelizing the inner loop that finds the minimum weight edge. We would aim to compare the speedup of all three algorithms as we change different properties of our input graphs.

If this goes well, we hope to implement some more complicated algorithmic changes which would improve the opportunities for parallelism. The exact algorithmic changes we do at this point would depend on what types of graphs do not scale up as well, i.e. whether our current implementations scale worse for increasing number of vertices or increasing number of edges. Some potential algorithmic change, inspired by the papers we reference in section 5, include:

- One example is a modification of Kruskal's algorithm which has more potential for parallelism. It filters out certain edges that are not in the MST in order to decrease the computation cost, and could help improve speedup for not-too-sparse graphs.
- Another algorithm, derived from Prim's, creates multiple threads each responsible for growing an MST for a subset of edges or vertices, and then uses parts of Boruvka's algorithm to combine them. This would use the shared address space model.

7 Platform Choice

For the most part, we plan to use OpenMP to implement shared address space parallelism. For some algorithms, we also want to try out message passing models using MPI, particularly for Boruvka's, which would have relatively low communication cost. The details of which platform we would use for each algorithm are addressed in more detail in section 6.

8 Proposed Schedule

- Week 1: Implement sequential version of all three algorithms, and write benchmark test cases for varying sizes of graphs and their density. For varying numbers n of vertices, we want a full test suite for graphs with $O(n)$ vertices and graphs with $O(n^2)$ vertices.
- Week 2: Get working parallel implementations of Prim's, Kruskal's, and Boruvka's algorithm using the shared address space model, and begin benchmark testing on multiple processors. The implementation goals are described in more detail in the previous section. This is approximately where the milestone report falls.

- Week 3: Work on improving the implementations of the above to get better speedup. Also, experiment with message passing implementations as described in the goals section. Also, if time allows, we might expand the benchmark tests to include "real-world" datasets, of graphs representing some application.
- Week 4: Improve the message passing implementation and compare the best achieved results between this and the shared address space models. If time allows, we will work on algorithmic modifications here in an attempt to meet some of the "hope" goals.