15-418 Project Milestone Report

Alex Knox (akknox), Elizabeth Knox (emknox)

April 15, 2025

1 Revised Schedule

The first few weeks of the project have been proceeding according to schedule. The goals stated in the proposal are repeated in the revised schedule, along with more specific goals for the coming weeks. Details on the work completed so far can be found in the following sections.

April 28	Complete poster	**Both**
April 26	Complete writeup	**Both**
April 23	Implement helper thread parallelization strategy for Kruskal's algorithm (on top of the parallel implementation that we have developed up to this point). Message passing implementation of Boruvka's algorithm. We believe we could accomplish similar parallelization here, by having each thread compute a preliminary result and then use messages to take an overall minimum.	Alex Elizabeth
April 19	Create more benchmarks to better compare varying levels of graph size and graph density, with other parameters kept consistent. Performance debugging for Boruvka's Algorithm. Some ideas (discussed further below) involve measuring the contention on a parallel array and decreasing shared memory accesses.	Alex Elizabeth
April 16	Performance debugging for Kruskal's Algorithm. At the least, understand why there is a drop off in performance at high thread counts. Ideally, if there is something that might address the drop off, adapt the implementation to achieve better performance.	Alex
April 15	Checkpoint	**Completed**
Week 2: April 6 - April 12	Get working implementations of Prim's, Kruskal's, and Boruvka's using the shared address space model, and begin benchmark tests on multiple processors.	**Completed**
Week 1: March 30 - April 5	Implement sequential versions of Prim's, Kruskal's, and Boruvka's. Develop testing infrastructure, and write benchmark test cases for varying sizes and densities of graphs.	**Completed**

2 Completed Work + Initial Results

For each of the three algorithms, we have implemented a sequential version in C++. Then, we profiled the sequential implementation to assess the percentage of time spent in parallelizable sections of the algorithm versus inherently sequential parts of the algorithm. Since the algorithms differ in their capacity for parallelism, this has given us a good model to interpret the results we have obtained so far. For all three algorithms, we have an initial attempt at parallelization–in the future, we will do further performance debugging work, and we will explore different opportunities and abstractions for parallelism.

Kruskal's Algorithm. The opportunities for parallelism in Kruskal's algorithm are primarily in sorting the edges by weight. After this sort, edges must be processed sequentially in order by weight, since cheaper edges determine whether heavier edges should be included or not. However, on benchmark tests, between 60% to 65% of the runtime is spent on the sort, so there is still opportunity for overall speedup due to parallelism. We have implemented merge sort using OpenMP to perform the sort: we observe some speedup, particularly for small numbers of processors and when compared against the maximum possible speedup due to Amdahl's Law, but there is a drop off in performance at higher processor counts.

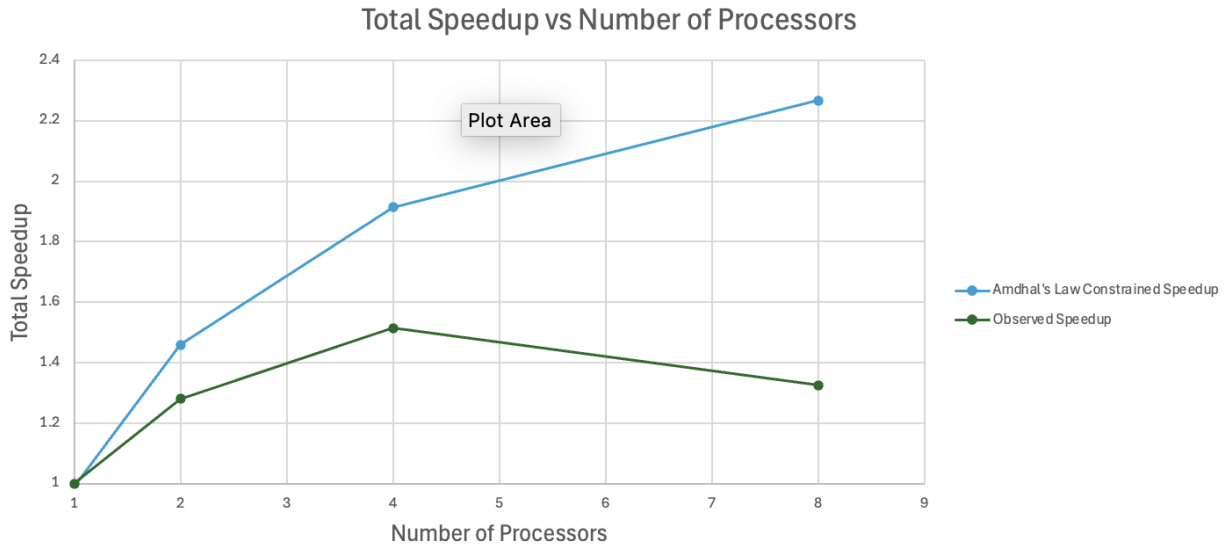

Figure 1: The effect of different numbers of processors on the speedup of Kruskal's Algorithm on a benchmark test. Speedup is limited by the fraction of parallelizable work, shown by the blue line, which is the maximum possible speedup under Amdhal's Law. Results were generated from GHC71.

Prim's Algorithm The opportunities for parallelism in Prim's Algorithm are in finding the minimum weight edge among all edges in the "frontier".

In a naive implementation, it is possible to find the minimum by iterating through the frontier. This is easy to parallelize with multiple processors and a shared address space model with a minimum reduction, and it scales well with more processors.

One of the reasons that this method scales so well is that finding the minimum by iterating through the frontier causes the algorithm to have a higher than optimal asymptotic cost bound, in which most of the work (>95% on benchmark tests) is spent finding a minimum.

Alternatively, an implementation of Prim's algorithm could perform this step using a priority queue. Instead of parallelizing the step of finding the minimum, we considered parallelizing the insertion into the priority queue. However, analysis of the algorithm shows that only about 8-10% of the runtime is spent inserting elements into the priority queue, so the potential for speedup with multiple processors using a priority queue is quite limited.

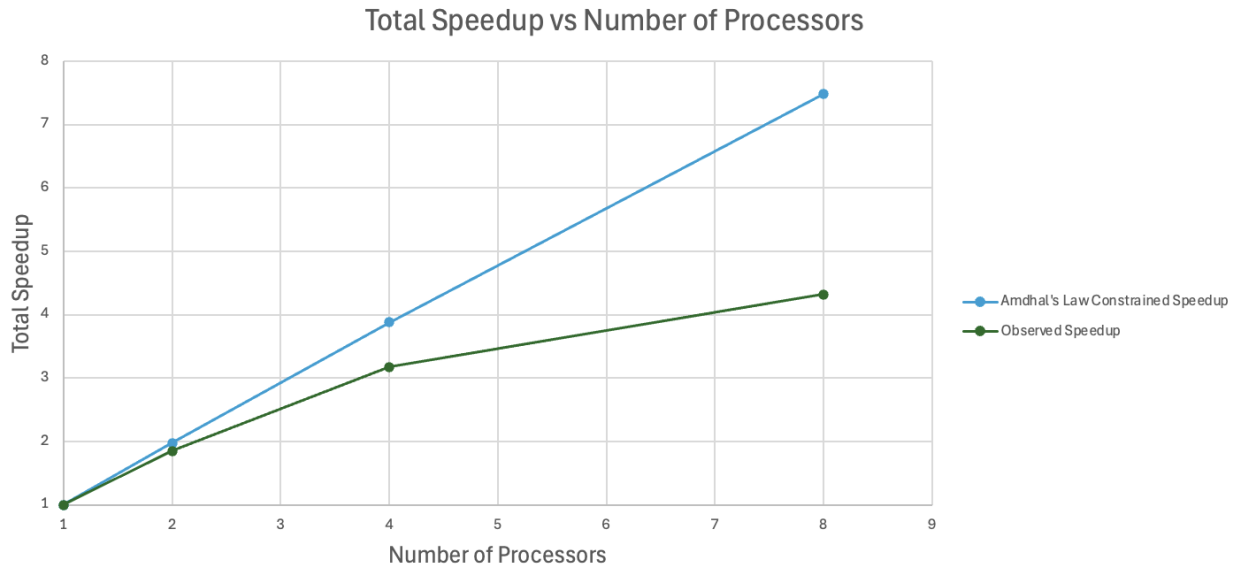

Figure 2: The effect of different numbers of processors on the speedup of Prim's Algorithm on a benchmark test. Results were generated from GHC71.

Comparing these two implementations, while the iterative minimum finding has opportunities for parallelism, the different asymptotic bounds means that it has performed empirically worse than the sequential, priority queue implementation. Increasing the number of processors may help to close the gap, and we intend to conduct experiments on the PSC machines to analyze this further.

Boruvka's Algorithm The majority of the parallelization so far has occurred in computing a minimum outgoing edge for each current connected component (unlike the other two algorithms, we are not computing just one minimum at a time). Connected components are represented using a union-find data structure, so as long as we do not do any contractions (unions of connected components), obtaining a vertex's connected component by reading from the data structure during this phase is safe, even in a multiprocessor setting.

In order to parallelize this phase, we stored the minimum outgoing edge for each connected component in an array that supports parallel updates — this was accomplished by performing updates with `atomic_compare_exchange`. We use the weak version of this function call, and wrap it in a loop that repeatedly tries to update the minimum while the value we have is still less than the current minimum. Assuming edges are distributed somewhat randomly among processors, contention will be lower when there are more connected components and higher towards the end of the algorithm.

As we continue performance debugging, we will try to quantify the contention on the array. A preliminary observation is that the compare/exchange will not fail arbitrarily many times: either we succeed in updating the minimum, or we fail, which means the minimum has been decreased by another processor. The minimum cannot decrease arbitrarily far: after repeated failures it would be low enough that we no longer want to do our initial update.

We do see some speedup with our initial parallelization across edges. However, we would like to improve the performance significantly. One concern is in the implementation of contraction: as the number of connected components decreases through the algorithm, the processors all updating the same few elements of the parallel `minimums` array, and are (perhaps unnecessarily) repeatedly searching for the same elements in the union-find data structure. In future performance debugging, we may try to restructure the algorithm so that processors do less frequent reads + writes on shared memory.

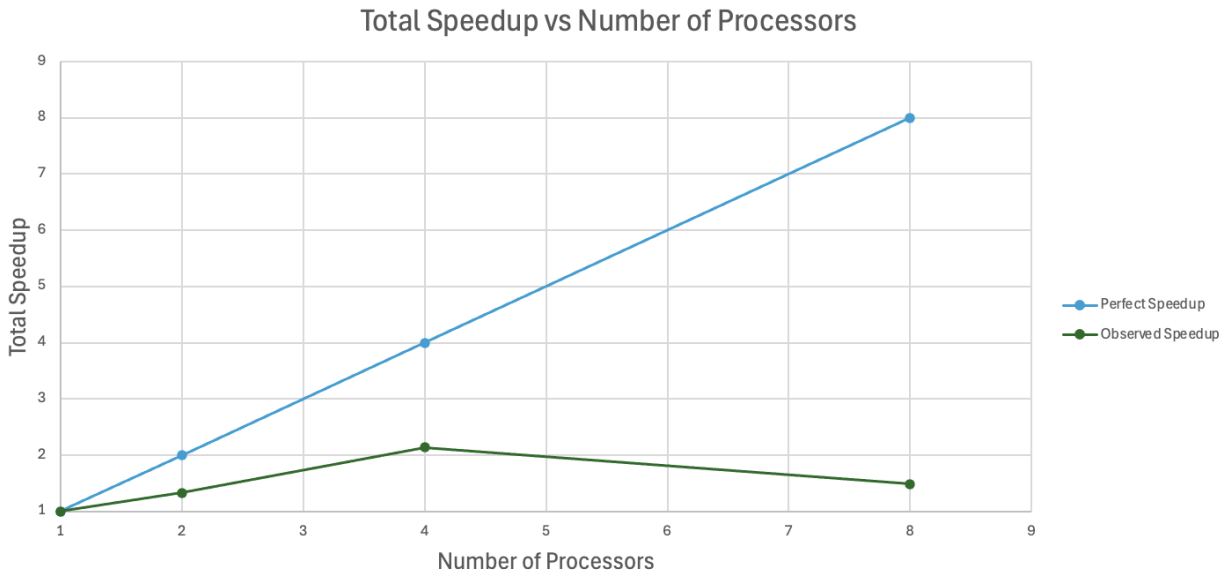

Figure 3: The effect of different numbers of processors on the speedup of Boruvka's Algorithm on a benchmark test. Results were generated from GHC56.

3 Reflection + Goals

We are on track to complete our deliverables. Our current parallel implementations require some performance debugging, but we expect it to be a normal amount for this point in the process. In particular, we stated in the proposal that we expected a parallel version of Boruvka's to achieve near linear speedup due to a lack of required communication. As it stands, our parallel implementation of Boruvka's is substantially sublinear. With further performance debugging and analysis, we will be able to more concretely evaluate why this is the case. Including this, there are a few open questions raised by our work up to this point which we would like to address:

- What is the limiting factor for speedup in a parallel implementation of Boruvka's algorithm?
- How does the parallelization of Prim's algorithm scale at processor counts beyond 8? Is it possible for the more expensive, parallel algorithm to achieve runtimes comparable to the cheaper, sequential algorithm with enough processors?
- What is the cause of the drop off in performance of the parallel implementation of Kruskal's algorithm at higher processor counts?

In addition, we have some remaining delieverables that we will accomplish before the poster session.

- Implement a parallel implementation of Boruvka's algorithm using a message passing abstraction. Since there is not very much communication in the algorithm, we would expect this to perform quite well, and we would like to compare it to the current implementation in a shared address model.
- Implement an idea for parallelizing a part of Kruskal's algorithm, which is running sequentially in our current parallel implementation. More specifically, while one thread performs the normal algorithm, other helper threads can be identifying edges which are definitely not in the MST.
- Investigate the way that varying different qualities of a test case, such as the size or density of the input graph, affect the speedups of our different implementations.

At the moment, we don't have any major issues. While we don't yet understand the performance of our parallel Boruvka implementation, we expect that this can be resolved by further performance debugging.