Parallelizing Kruskal's Algorithm and Boruvka's Algorithm for Finding Minimum Spanning Trees

By Ishika Saxena and Sarayu Namineni

Video Link to Presentation: https://youtu.be/4fs-OUm6-Sw

SUMMARY

We parallelized minimum spanning tree algorithms by, first, parallelizing Kruskal's algorithm and, then, parallelizing Boruvka's algorithm. In implementing Kruskal's algorithm, we parallelized the sorting step (a divide and conquer merge sort algorithm). For Boruvka's algorithm, we first wrote a sequential version and then optimized its performance via parallelization using edge contraction. We used C++ and OpenMP to conduct parallelization across different numbers of threads in an attempt to achieve speedup, and we analyzed performance (via computation time taken using various numbers of threads) on the GHC machines. For deliverables, we have:

- Plots of computation time vs. number of threads used for Kruskal's algorithm on 3 different graphs (small and large) over 2, 4, 8, 16, and 32 threads. Calculated via comparing computation times on these threads with the time taken for the parallel version to run on 1 thread, along with time taken for the completely sequential (no threads) algorithm.
    - 200 vertices, 15000 edges [Graph 1]
    - 500,000 vertices, 2 million edges [Graph 2]
    - 1 million vertices, 9 million edges [Graph 3]
    - Also, a table of speedups for 1 thread vs. 2 threads
- Analyzing Amdahl's Law effect of only parallelizing sorting for Kruskal's
    - Bar graph that breaks down time taken for inherently sequential parts of the code and parallel time
    - Computing the max theoretical speedup possible via Amdahl's Law and comparing this to the speedup achieved on the 3 graph inputs
- Computation time vs. number of threads used for Boruvka's algorithm on 3 different graphs (small and large) over 2, 4, 8, 16, and 32 threads. (Baseline is again 1-thread runtime of parallel version and the general sequential algorithm.)
    - Graphs of computation times for each input (number of threads on x-axis, computation times on y-axis) with fine-grained locking implementation
    - Table of associated speedups for Graphs 2 and 3
- Breakdown of computation times in order to analyze the effect of parallelism over substeps of our algorithm
    - Bar graph that breakdowns computation times over parallelizable and sequential steps and compares times across sequential and fine-grained locking implementations

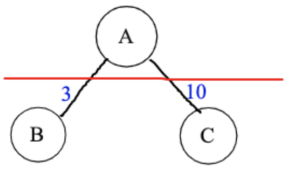
    - Bar graph that breakdowns parallelizable work into substeps and compares times across critical section and fine-grained locking implementations

BACKGROUND

We parallelized 2 different minimum spanning tree (MST) algorithms: Kruskal's algorithm and Boruvka's algorithm.

Key Terms

- A **tree** is an undirected graph that has no cycles such that given any pair of vertices, exactly 1 path connects them.
- Given a connected, undirected graph G, we can find a subset of its edges that, together, include all of its vertices while still being acyclic. This set of edges and vertices defines the **spanning tree** of graph G.
    - A spanning tree must have |V|-1 edges, where |V| is the number of vertices in G. (This can be shown by induction and using the fact that the tree is acyclic.)
- A graph G can have many spanning trees, so we are sometimes interested in finding the **minimum spanning tree (MST)** of G, which is the spanning tree with the smallest total weight.
    - The **weight** of a tree is defined by the sum of the weights of its edges.
- Most MST algorithms utilize the **Light-Edge Property**, which states that if G is a connected, undirected, weighted graph (where each edge weight is distinct), for any cut of G (partitioning G's vertices into two nonempty subsets), the minimum weight edge that crosses the cut is contained in G's MST (Diderot, 2021).



  In this case, we see that there are 2 edges that cross the cut (as one of their vertices is in one subset and the other vertex is in the other subset). Edge (A,B) mus be in the MST by this property.
- In **Kruskal's algorithm**, the MST of G is built by adding the least-weight edge that doesn't create a cycle with the edges already chosen. This utilizes the Light Edge Property, since among all the edges that don't create a cycle with the connected graph being created so far, the lightest one must be in the MST, since the edge that is being added is going across a cut (since one vertex in the already connected subgraph is being connected to one vertex from the edges that haven't been chosen yet). It also uses the fact that a tree has |V|-1 vertices (where |V| = number of vertices in G) to know when to stop adding edges.

Algorithm's Inputs and Outputs

Both Kruskal's and Boruvka's take in connected, undirected graphs as input and output a set of edges that directly represent the MST.

Key Data Structures & Key Operations on These Data Structures

Kruskal's

- One key data structure is the **array of edges**, where an edge is a struct containing a starting point, an ending point, and a weight. The array is populated from the input text file. In Kruskal's, it's an undirected graph but is stored as a directed graph that has 2 edges for every undirected edge (i.e. stores both directions).
    - A key operation on the array of edges is **sorting** the edges by weight. Since Kruskal's selects the lightest weight edge to add, the edges of the graph must be processed from least to greatest weight.
- At each iteration, Kruskal's algorithm must also check whether the edge being added creates a cycle or not (with the edges that have already been added to the resultant MST being accumulated). Thus, one key data structure is the **Union Find** data structure, which consists of an array storing the canonical representative of each vertex (i.e. which larger component each vertex is a part of) and an array storing the rank at each vertex.
    - The key operations on these data structures are **find** and **union**
        - find(v): finds the specific subset/component that a particular vertex v is a part of (note that each component has a representative vertex). When called on two separate vertices, find can be used to figure out whether two vertices are in the same component, upon which adding an edge between them would create a cycle.
        - union(v1, v2): joins the two components/subsets that these two vertices are individually a part of into one subset of the vertices. This is needed once the algorithm has decided to add an edge into the resultant MST to ensure that the meaning of being in the same component is maintained

Boruvka's

- One key data structure is the **array of edges**, where an edge is a struct containing a starting point, an ending point, and a weight. The array is populated from the input text file. In Boruvka's, it is an undirected graph which stores each edge once. No sorting is necessary.
- Another key data structure is the **array of cheapest edge indices**. This array is used to keep track of the edge with minimum weight that has exactly one endpoint in the given vertex component. It is indexed by vertex so that we can iterate through all the vertices, adding their cheapest edges to the MST, if possible.

- **Union Find** data structure is used just like above to store the canonical representative of each vertex and the size of each of the components. In Boruvka's, the union find structure is used to determine which edges to contract.
    - We perform find operations to determine if we can contract an edge. If both the endpoints of an edge have the same canonical representative, then they belong to the same vertex component, which means that they have already been contracted. On the other hand, if the endpoints of an edge have different canonical representatives, then they belong to different vertex components and can be contracted.
    - We perform the union operation on the endpoints of the vertices we want to contract. We update the representatives of the vertices so that they both share the same canonical representative after the operation.

In Both Algorithms

- We utilized "rank[s]" to ensure that a smaller subset/tree is added "under the root" of the larger subset/tree (to maintain a balanced sort of structure) and "path compression" to "traverse" upwards towards the root to find the parent vertex rather than traverse all vertices (Chaudhary, 2020).

<u>Parts that are Computationally Expensive & Could Benefit from Parallelization</u>

Kruskal's

- The only part of Kruskal's that is computationally expensive AND could benefit from parallelization is the sorting of the graph's edges by weight. This step is important, as everything else in the algorithm depends on the edges being sorted. We performed this sorting in parallel through parallelizing the divide and conquer merge sort algorithm and, particularly, used OpenMP tasks to run the two recursive calls that sort the two halves of the array at every step in parallel.
- A major dependency in the sorting algorithm is that the two halves can only be merged once they're sorted, meaning we needed to insert a barrier before the merge step, which prevented some parallelism.
- Overall, parallelizing the sorting is just a small part of the larger program. The actual insertion of edges into the result must be done sequentially, as we must process the edges from least to greatest weight and stop once there are n-1. Thus, Kruskal's is severely limited in speedup by Amdahl's Law.

Boruvka's

- There are two parts to Boruvka's algorithm: finding the cheapest edges out of every vertex component and contracting those edges. On a baseline sequential implementation, finding the cheapest edges out of every vertex component takes approximately 97.5% of our computation time. Thus, we identified this part of the algorithm as the one that could

benefit the most from parallelization. We can parallelize the process of finding the cheapest edge of each vertex component and we discuss various approaches, including the one we ultimately chose, in further detail in the Approach section.

- One dependency in the program is that before we begin the step of contracting edges we must wait until we have finished finding the cheapest edges out of every vertex component. This is because we must wait until we have iterated through all the edges in the graph in order for the array of cheapest edges to be correct.
- We were unable to parallelize the process of contracting the cheapest edges adjacent to each vertex component. This is because it is possible that two cheapest edges share a vertex component as an endpoint, which would result in multiple threads trying to update the representative of the vertex component that was a shared endpoint for multiple cheapest edges.

APPROACH

Languages/Machines Used

We used C++ to implement the 2 MST algorithms, and OpenMP to parallelize our implementations.

To test our implementations (times taken, speedups, etc), we used the GHC machines, which have 8 cores with 1 thread per core.

Graph Input Generator

An important component of our implementation is testing it on different graph inputs (with a varied number of vertices and edges). Initially, we implemented this in C by generating random edge weights and endpoints (based on an inputted number of edges, vertices, and max weight possible) and outputting these onto a text file. However, due to needing to support millions of nodes and edges, we switched to using C++, as the in-built vectors, uniform distribution random number generator (for random edge weights), and other data structures aided us in writing more efficient code.

Specifically, we use an adjacency list representation of a graph to store the edges being added, along with an edge list that allows for ease of printing out each edge separately. We've taken many steps to ensure correctness i.e. that the graph we are outputting is connected and doesn't have self-loops or multi-edges (so that it can satisfy all preconditions of the MST algorithms). For instance, if the number of edges exceeds n choose 2 (maximum number of edges possible) or is less than n-1 (minimum number of edges needed to be connected), we output errors. We also ensure that there are no isolated vertices by ensuring at least one edge is adjacent to every vertex.

Kruskal's Approach

Kruskal's Algorithm Implemented with Sequential Merge Sort

The MST of the input graph is built by adding the least-weight edge that doesn't create a cycle with the edges already added (initially starting with no edges added). This can be done by sorting the edge list by weight and processing them in order of least to greatest. At each iteration, the algorithm checks if the edge being added creates a cycle with the edges already added using the above-described union find operations. If it doesn't create a cycle, we add the edge to our resultant MST and continue iterating until we've collected |V|-1 edges, because by definition, a spanning tree of a graph always has |V|-1 edges.

In the sequential version of the algorithm, we utilized merge sort but ran the two recursive calls (that sort either half of the input array) sequentially with no OpenMP pragma statements to parallelize.

Kruskal's Algorithm Implemented with Parallelized Merge Sort

The only parallelizable component of Kruskal's algorithm is the sorting step. As such, an opportunity for parallelism arises due to the divide-and-conquer nature of merge sort, as the two recursive calls executed to sort the halves of the input array can be done in parallel, since they are sorting completely independent halves of the array.

Since our goal was to measure performance on various threads using OpenMP parallel constructs, we decided to utilize #pragma omp parallel to section off the portion of the code that wanted to parallelize, which was the 2 recursive mergeSort calls. Using this directive, we actually chose to spawn the group of threads across the entirety of the 2 mergeSort calls. This is because the merge call can't be parallelized with the 2 recursive calls, so we knew we could utilize the implicit barrier at the end of the parallel section to prevent threads from executing the merge call until both recursive calls had completed.

We chose to use the task pragma, as it made most sense that separate threads (specifically, two different ones) would compute one of the recursive calls and other threads would compute the other recursive call. Thus, by using a #pragma omp task directive explicitly around each of the recursive calls, we solidified this as a specific "unit[...] of work" that we wanted a specific thread to perform in parallel with everything outside this task directive (Monsalve, 2019, p. 26). By surrounding each recursive call with a different #pragma omp task directive, we ensured that the 2 calls would be executed in parallel with each other. This is because if these 2 recursive calls were in the same #pragma omp task block, the 2 calls would not be run in parallel with anything else, as there are only these 2 function calls within the #pragma omp parallel block.
Again, we made sure to conduct the merge call after the barrier that's implied by the end of the #pragma omp parallel block, as the merge call explicitly depends on the 2 halves being sorted (i.e. the 2 recursive calls being completed). This implied barrier waits for all the tasks within the #pragma omp parallel block to be completed before moving on. Without this, we couldn't ensure correctness, as merge's steps require the 2 inputs to be sorted fully.

After this initial iteration of implementation, we decided to optimize further, as through research, we realized that when tasks are declared one after the other within a #pragma omp parallel block, every thread in the spawned group of threads ends up adding these two tasks "into the work queue" ((Monsalve, 2019, p. 25). This would cause replicated work and explained why we were seeing very minimal speedup, even when executing using 1 thread vs. 2 threads. Thus, to ensure

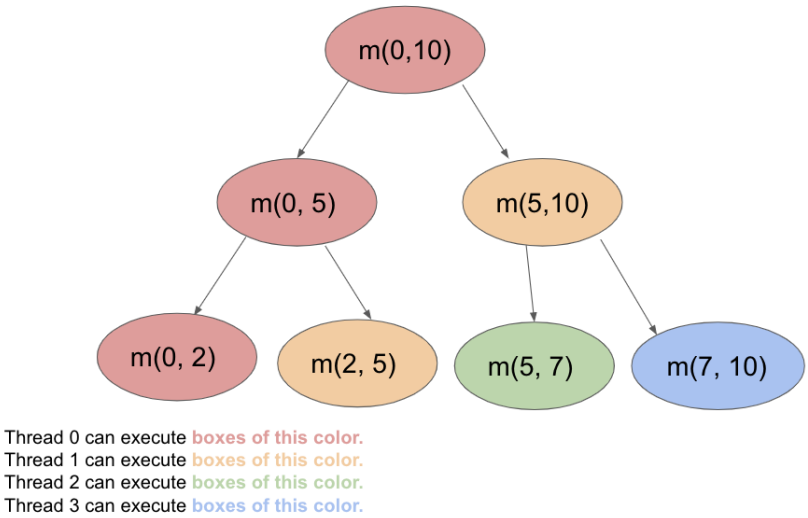
that just 1 thread entered the task declaration region (i.e. the 2 #pragma omp task statements) and added the 2 tasks to the queue just twice (rather than many more times), we utilized the #pragma omp single. This ensured that the block was "executed by only one of the threads in the team" ("OpenMP 3.1 API", 2021), meaning only 1 thread would add the 2 tasks to the task queue, after which each thread in the spawned group of threads could try to dequeue a task and execute it.

Note for our parallelizing approach here, we didn't necessarily change the initial serial algorithm but rather added OpenMP directives that enabled units of work to be matched to concurrently executing threads on a parallel machine:

```
// For parallelizing mergeSort using tasks, we adapted structure of
// code from slide 7 of this Oregon State University lecture
// https://web.engr.oregonstate.edu/~mjb/cs575/Handouts/tasks.1pp.pdf
// (Namely just using tasks to do 2 things at once and single to ensure
// only 1 thread enqueues the tasks.)
void mergeSort(edge *edgeList, int start, int end) {
    using namespace std::chrono;
    typedef std::chrono::high_resolution_clock Clock;
    typedef std::chrono::duration<double> dsec;
    // end is exclusive
    if(start >= end-1) {
        return;
    }
    int mid = ((end-2)+start)/2;

    omp_set_num_threads(1);
    #pragma omp parallel
    {
        #pragma omp single
        {
            // start 2 parallel tasks to sort 2 halves
            #pragma omp task
            {
                mergeSort(edgeList, start, mid+1);
            }
            #pragma omp task
            {
                mergeSort(edgeList, mid+1, end);
            }
        }
    }
    auto compute_start = Clock::now();
    merge(edgeList, start, mid, end); // merge start to mid WITH mid to end
    globalTime += duration_cast<dsec>(Clock::now() - compute_start).count();
}
```

As an example, using this approach, thread 0 could enter mergeSort(edgeList, 0, 10) and create tasks for mergeSort(edgeList, 0, 5) and mergeSort(edgeList, 5, 10) by adding them onto the task queue. Thread 0 and thread 1 could now execute these tasks. In doing so, thread 0 and 1 create 4 new tasks: thread 0 creates mergeSort(edgeList, 0, 2) and mergeSort(edgeList, 2, 5) and thread 1 creates mergeSort(edgeList, 5, 7) and mergeSort(edgeList, 7, 10) ((Terboven & Schmidl, 2018, p. 6)).
This diagram outlines this (diagram structure somewhat adapted from slide 6 of RWTH Aachen University linked here and cited here and below (Terboven & Schmidl, 2018, p. 6)):



In this diagram, arrows represent the parent node enqueuing both of the child tasks. So for instance, the m(0,10) call that thread 0 executes ends up enqueuing m(0,5) and m(5,10) onto the task queue. Now, after thread 0 is freed up (after enqueuing these tasks for m(0,10)), thread 0 can dequeue m(0,5) and some other thread (in the diagram, it's thread 1) can dequeue m(5,10). And the process can continue.
Note also that at the bottom of the diagram, threads 0, 1, 2, and 3 are now able to execute these 4 tasks (at the bottom of the diagram) in parallel. It's possible that without the #pragma omp single directive, every thread would enqueue 2 tasks, meaning there would be a lot more duplicated/replicated work being done, which would increase the time taken but not change the computation itself.

With this optimized approach, we saw about 2x speedup when comparing 2 threads to 1 thread. This makes sense because we're just running 2 separate recursive calls in parallel. (As discussed in below Results section, the speedup stays the same as the 2-threads speedup after increasing threads beyond just 2 threads, as predicted by Amdahl's Law)

As another optimization, we felt that the overhead of setting up tasks and adding the queue may not be worth the small speedup gain on smaller inputs, so we tried adding an if conditional at the top of the merge sort that would just run the sequential version of merge sort if the length of the inputted edge list was small (about 200 edges or less). We saw a very slight decrease in time taken for Graph 1 (0.028016 seconds vs. 0.027987 seconds) but for Graph 2, the time actually slightly increased (3.921218 seconds vs. 3.933790). This may mean that the threshold of number of edges upon which we should call sequential mergeSort should be dependent on the input size, rather than just a flat 200 edges amount. For instance, since Graph 2 has 2 million edges and Graph 1 has 15000, perhaps the threshold should be increased for Graph 2 (something like 15000/200 = 75 times less than 2 million (so around 27,000 edges) to even see a slight speedup with this approach.

If you started with an existing piece of code, please mention it (and where it came from) here.
For parallelizing mergeSort using tasks (in optimized Kruskal's algorithm implementation), we adapted structure of basic starter code (regarding using tasks) on slide 7 of this Oregon State University Computer Graphics lecture slides PDF on OpenMP Tasks by Mike Bailey: https://web.engr.oregonstate.edu/~mjb/cs575/Handouts/tasks.1pp.pdf
Namely, we just used the structure of using tasks to do 2 things at once and a #pragma omp single construct to ensure only 1 thread enqueues the tasks rather than all of them.

(Here's a full citation (also included in References section):
Bailey, Mike. (2022, March 16). *OpenMP Tasks.* [Lecture Slides/Handout from Oregon State University CS575 as per course number in link]. Computer Graphics, Oregon State University. https://web.engr.oregonstate.edu/~mjb/cs575/Handouts/tasks.1pp.pdf)

Boruvka's Approach

Sequential Boruvka's Implementation

For each iteration of Boruvka's algorithm, we identify the cheapest edges adjacent to each vertex and contract those edges, if possible. We continue this process of identifying cheapest edges and contracting them until the graph is reduced to a single vertex component.

The key insight of Boruvka's algorithm is that the cheapest (or "lightest") edge that crosses any cut must be in the minimum spanning tree. At every iteration of the algorithm, we iterate through all the edges in the array in order to determine the cheapest edges out of every vertex component. For each edge, we find the vertex components of the endpoints by consulting the union find data structure. If the endpoints do not belong to the same vertex component, we proceed because we can contract this edge. We compare each edge to the cheapest edges we have seen thus far for either of the vertex component endpoints of the edge and update our array of cheapest edges accordingly.

After finding the cheapest edges adjacent to each of the vertex components, we proceed to contract these edges, if possible. We iterate through all of the vertex components and find the vertex components of the endpoints of its cheapest edge. It is crucial to perform this step before proceeding because it is possible that the two vertex endpoints now belong to the same vertex component as a result of previous edge contractions. If the two endpoints do not belong to the same vertex component, we contract this edge by performing the union operation and add this edge to our MST.

Parallelized Boruvka's Implementation

There are a couple approaches to parallelizing the work of finding the cheapest edge adjacent to each vertex component. One method is to parallelize the search over the vertices, and another method is to parallelize the search over edges.

A naive approach to parallelizing over the vertices would require O(nm) work, since each vertex iterates through the entire edge array in order to determine the cheapest edge adjacent to it.
If we store the edges in an adjacency list, each vertex only iterates through the edges that are adjacent to it. This results in a total of O(m) work, which is optimal. However, assigning each thread to a vertex could result in load imbalance, since the amount of work performed by each thread is proportional to the degree of the vertex it was assigned to. Since we are randomly generating our input graphs, we can make no guarantees about the maximum degree of any vertex.

We choose to parallelize the search across all the edges, since this approach results in a total of O(m) work, which is optimal, and we can guarantee a balanced workload across all threads using a static, interleaved mapping of edges to threads. This is because there is a constant amount of

work associated with checking each edge. We compare the current edge to the cheapest edge seen thus far in the vertex components of the endpoints of the current edge and update the cheapest edge array accordingly.

This implementation went through three versions: sequential implementation, parallel implementation with critical sections, and parallel implementation with fine-grained locking. When changing the original serial algorithm in order to parallelize the search for cheapest edges, our first approach was to lock the array of cheapest edges so that only one thread would perform updates at any given time, thereby ensuring correctness.

```
while(nsets > 1){
    std::vector<unsigned int> cheapest(n, UINT_MAX);

    // Iterates through all the edges and updates the cheapest edges for the
    // associated endpoints
    #pragma omp parallel num_threads (num_threads)
    {
        unsigned int threadId = omp_get_thread_num();
        for(unsigned int i = threadId % m; i < m; i += num_threads){
            unsigned int v1 = edges[i].v1;
            unsigned int v2 = edges[i].v2;
            int w = edges[i].w;

            unsigned int pv1 = findParent(v1);
            unsigned int pv2 = findParent(v2);
            if(parent[pv1].first != parent[pv2].first){
                #pragma omp critical
                {
                    if(cheapest[pv1] == UINT_MAX || edges[cheapest[pv1]].w > w){
                        cheapest[pv1] = i;
                    }
                }


                #pragma omp critical
                {
                    if(cheapest[pv2] == UINT_MAX || edges[cheapest[pv2]].w > w){
                        cheapest[pv2] = i;
                    }
                }
            }
        }
    }
```

The above code snippet demonstrates important aspects of our first parallel approach. First, we can observe the static, interleaved assignment of edges to threads. Next, we can observe the critical sections that effectively lock access to the cheapest edges array such that only one thread makes updates to it at a time.

A key realization that we made when trying to remove these critical sections was that any edge that can be contracted must have endpoints that are part of different vertex components. We realized that we only needed one critical section, instead of two, since we could perform our updates to the cheapest edge array in parallel. This is because the indices that we are accessing are the vertex components of the endpoints, which must be distinct if we are going to contract this edge.

After further consideration, we realized that this idea generalized to all threads updating the cheapest edges on distinct vertex components. Instead of locking the entirety of the cheapest

edge array, we implemented fine-grained locking for each vertex component in our cheapest edge array.

```
omp_lock_t lock[n];
for(unsigned int i=0; i < n; i++){
    omp_init_lock(&(lock[i]));
}
```

```
// Iterates through all the edges and updates the cheapest edges for the
// associated endpoints
#pragma omp parallel num_threads (num_threads)
{
    unsigned int threadId = omp_get_thread_num();
    for(unsigned int i = threadId % m; i < m; i += num_threads){
        unsigned int v1 = edges[i].v1;
        unsigned int v2 = edges[i].v2;
        int w = edges[i].w;

        unsigned int pv1 = findParent(v1);
        unsigned int pv2 = findParent(v2);
        if(parent[pv1].first != parent[pv2].first){
            omp_set_lock(&lock[pv1]);
            if(cheapest[pv1] == UINT_MAX || edges[cheapest[pv1]].w > w){
                cheapest[pv1] = i;
            }
            omp_unset_lock(&lock[pv1]);


            omp_set_lock(&lock[pv2]);
            if(cheapest[pv2] == UINT_MAX || edges[cheapest[pv2]].w > w){
                cheapest[pv2] = i;
            }
            omp_unset_lock(&lock[pv2]);
        }
    }
}
```

```
for(unsigned int i=0; i < n; i++){
    omp_destroy_lock(&(lock[i]));
}
```

The above code snippets demonstrate important aspects of our final fine-grained locking approach. First, we initialize an array of locks, one for each of the vertex components in our graph, in order to implement fine-grained locking. Next, we are able to remove the critical sections from our code by setting and unsetting the lock associated with the vertex component whose cheapest edge we want to update. Finally, we destroy all the locks that we created.

RESULTS OF KRUSKAL'S ALGORITHM IMPLEMENTATION

We tested Kruskal's algorithm implementation with no parallel sorting (i.e. completely sequential sorting), along with Kruskal's algorithm with our final, optimized (above-described) parallel implementation (involving parallel merge sort using tasks). Since we knew we would see parallelism results the most on larger graphs (as there would be more work to do, specifically more sorting, as there would be more edges), we tested on 1 smaller graph (200 vertices, 15000 edges [Graph 1]) and 2 larger graphs (500,000 vertices, 2 million edges [Graph 2]) and (1 million vertices, 9 million edges [Graph 3]). We used the C++ chrono library for timing code.

Plotting Computation Time vs. Number of Threads

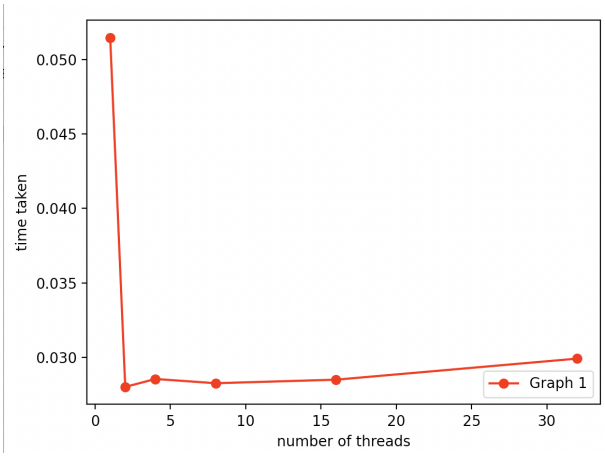
We first tested the computation time that our algorithm takes (i.e. the time taken for the algorithm to find an MST given the input graph). We tested on the no parallel sorting implementation (i.e. completely sequential) and on the parallel sorting implementation.

With sequential sorting (no use of OpenMP constructs), our Kruskal's algorithm implementation takes:

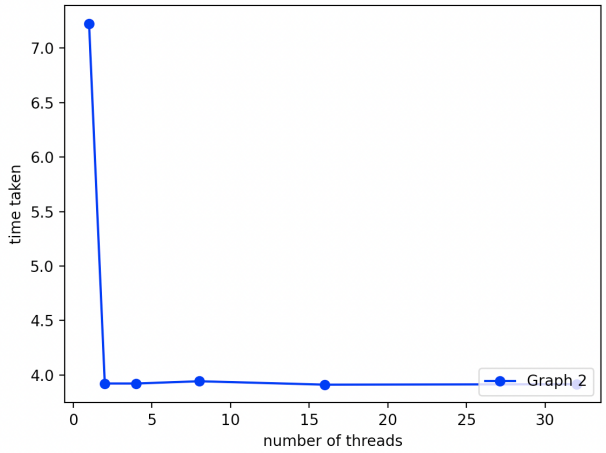
- 0.005316 s on Graph 1
- 1.043964 s on Graph 2
- 4.793265 s on Graph 3

It makes sense that with more vertices and edges, the sequential implementation takes more time, as most of the algorithm has dependencies preventing it from conducting anything in parallel. Further, the sorting conducts its recursive steps sequentially, causing more time to be taken.

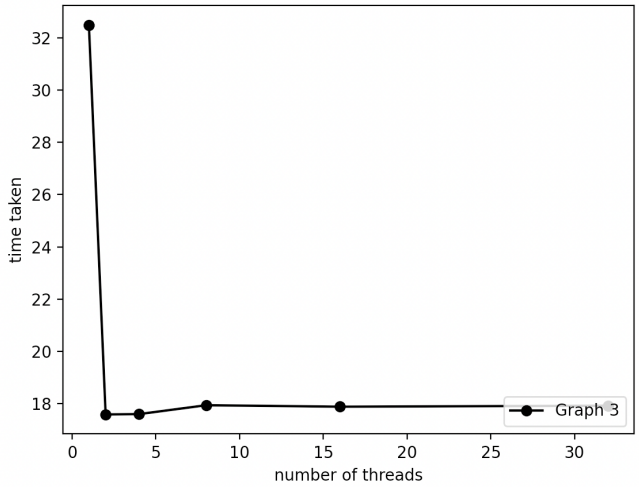
With the parallel sorting implementation using tasks, our Kruskal's algorithm implementation follows these trends:



Kruskal's Algorithm Implementation with Parallelized MergeSort on Graph 1 (200 vertices, 15000 edges)

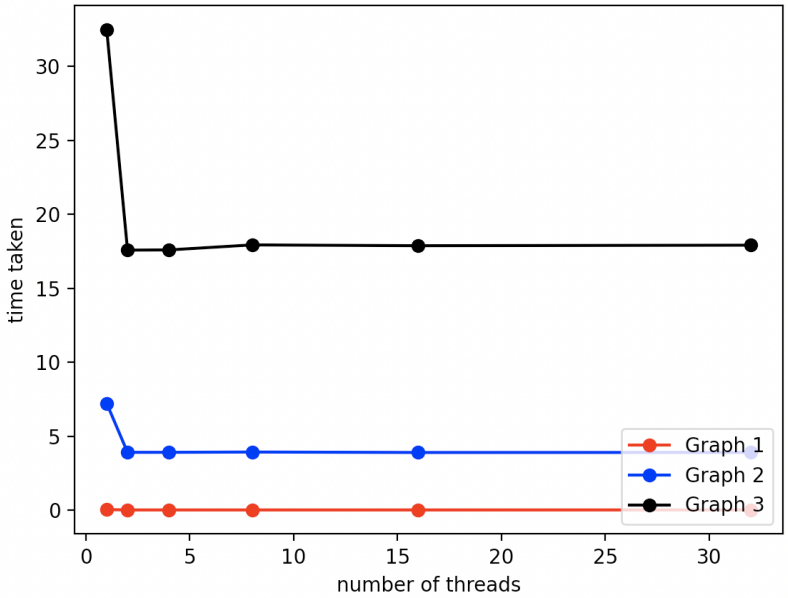


Kruskal's Algorithm Implementation with Parallelized MergeSort on Graph 2 (500,000 vertices, 2 million)



Kruskal's Algorithm Implementation with Parallelized MergeSort on Graph 3 (1 million vertices, 9 million edges)

And all 3 lines plotted on the same graph give this trend:



Kruskal's Algorithm Implementation with Parallelized MergeSort on All 3 Graphs

The baseline is just single-threaded CPU code (i.e. time taken for parallel version to run using 1 thread).

Analysis of Results

Note that in all graphs, regardless of their size, the time that Kruskal's algorithm takes to complete in the completely sequential version is much smaller than the time taken for the parallel version to complete on 1 thread. This shows how the overhead of initializing tasks in the pragma construct is truly not worth it if running the implementation on 1 thread. In terms of specific numbers, we see that in Graph 1, there's a 9.68x slowdown (0.005316 seconds vs 0.051474 seconds). In Graph 2, there's a 6.92x slowdown. In Graph 3, there's also a 6.92x slowdown. The slowdown might be decreasing as the graph size increases, since with more work to do, we see that the benefits of parallelism increase, as less threads are idle and thus there's more potential for speedup.

Also, upon comparing the time taken on 1 thread with the time taken on 2 threads, we see:

	Time taken for Graph 1	Time Taken for Graph 2	Time Taken for Graph 3
1 thread	0.051474	7.225179	32.491217
2 threads	0.028016	3.921218	17.588175
Speedup	1.837x	1.843x	1.847x

Note that for all graphs, there's almost a 2x speedup for when the implementation is run with 2 threads. Note that running with 2 threads would mean that once the initial 2 tasks are added to the queue (i.e. the 1st 2 recursive calls to mergeSort), these 2 threads can begin executing them concurrently. Thus, in an ideal case, due to 2 threads being used, we would expect a 2x speedup.

One reason we may not be observing exactly 2x speedup is because of unequal work partition, as it's possible that one half of the array may be completely sorted (meaning calling mergeSort on any subsequent halves of this half will require less work), meaning more often than not, this thread could be idle. (This is due to there being an implicit barrier at the end of a pragma omp parallel block.) Further, due to halving the array, the array inputted into one of the recursive calls could always have an extra element, which could again cause certain threads to be idle.

A major reason for non-ideal speedup is also the use of the implicit #pragma omp parallel barrier that occurs at the end of the pragma omp parallel block. This barrier prevents threads from merging until all threads have completed the sorting of the array (i.e. wait until the 2 halves are now sorted). This prevents more work from progressing, causing threads to be idle, especially in the aforementioned case of one half of the array being more sorted than the other.
Since Graph 3 is larger than Graph 2, which is larger than Graph 1 (both in terms of number of vertices and number of edges), it makes sense that we observe slightly more speedup as the graph size increases. This is because relative to the number of vertices, there's actually more work that can be parallelized in the sorting step in the larger graphs.

Note also that Graph 1 may be relatively denser than the other 2 graphs (since it's closer to having n choose 2 edges), which hints that perhaps if Graph 1 wasn't as dense, due to the size sheer graph size increase from Graph 1 to Graph 2, we may have observed significantly less speedup on Graph 1. This is also because upon comparing Graph 2 to Graph 3, we see that speedup only increases by 0.004, which is close to the amount that increases between Graphs 1 and 2 (0.006). However, the number of vertices in Graph 2 is 2500 times the number of vertices in Graph 1 (as compared to the number of vertices in Graph 3 being 2 times the number of vertices in Graph 2). All this goes to say is that a denser graph (maintaining the same number of

vertices but just changing number of edges to be closer to "n choose 2") may relate to having more maximized speedup, as there may be more work, as there are more edges relative to number of vertices.

Amdahl's Law Analysis

Another major reason of non-ideal speedup is due to the fact that a major part of Kruskal's algorithm is sequential: keep in mind that we only parallelized the sorting step of Kruskal's, which means the rest is completely sequential (such as finding whether 2 edges create a cycle or adding an edge to the resultant MST). Even the merging step of merging the 2 sorted halves must be done sequentially after the parallel recursive calls. And everything else (such as actually adding edges to the MST) must run sequentially.

This relates to Amdahl's Law and can help us explain why after 2 threads (so with 4, 8, 16, and 32 threads), we achieve the same speedup as we did with 2 threads. (We can see this in our plots above where there's a relatively horizontal line after 2 threads for every graph.) In other words, after 2 threads, there's no point of adding more resources (in the form of more threads/processors), because the observed speedup on our parallel merge sort implementation of Kruskal's algorithm will stay the same.

According to Amdahl's Law, if we let *s* be the "fraction of execution that is inherently sequential", then the "maximum speedup in latency due to parallel execution is $S <= 1/s$" ("Exercise D Solutions. Problem 1", 2022, p. 2).
In essence, no matter how many threads are used in the parallel execution context, if *s* is the fraction of work that is sequential and thus must run in a specific amount of time (as it can't run faster since it must be performed sequentially), then the speedup is limited to $1/s$.
According to the speedup numbers above, it seems like our speedup is limited to around 1.85x to 1.9x.

To find more concrete numbers, we timed the entire Kruskal's algorithm implementation (involving parallelized merge sort) on 1 thread. Then, we timed the part of the code that's completely inherently sequential (this would be the merging part of mergeSort, the creation of the edge list from the input graph, the initialization of union find data structures, and the actual while loop that processes edges in order of increasing weight and decides whether to insert an edge into the MST).

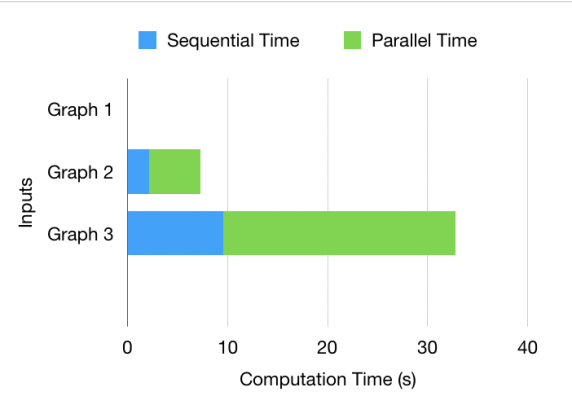
	Sequential Work Time Taken	Total Computation Time	Sequential Work Percentage
Graph 1	0.013231 s	0.051934 s	25.48%
Graph 2	2.120656 s	7.284130 s	29.11%
Graph 3	9.578349 s	32.775596 s	29.22%

Through applying Amdahl's Law, we see that:

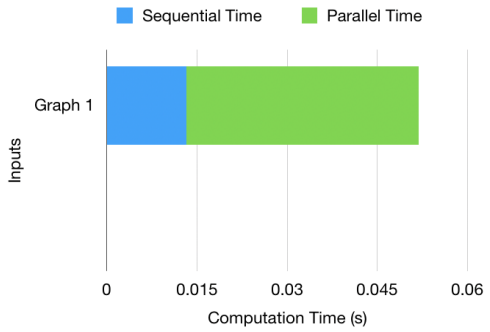
- On Graph 1, the max speedup that we should be able to achieve is *1/.2548 = 3.92*
- On Graph 2, the max speedup that we should be able to achieve is *1/.2911 = 3.44*
- On Graph 3, the max speedup that we should be able to achieve is *1/.2922 = 3.42*

Note that the max speedup decreases as graph size increases. This may be happening as more of the computation time is spent inherently on sequential work due to the edgeList being longer in a larger graph.

This bar graph showcases how much of the time was spent on sequential tasks in blue and how much of the time component was spent on parallelizable work in green. The parallelizable work in particular is the 2 recursive merge calls in particular. The sequential work includes the merging which is dependent on the 2 halves being sorted.



Note that Graph 1's small size causes its time amounts to be quite small relative to the other graphs, so here is a plot showing it enlarged:



As we can see, generally, for all the inputs, the amount of parallelizable time was around 70 to 75% of total computation time. As such, the max speedup possible is around 3.42 to 3.92x, as per Amdahl's Law.

This bound is slightly more than we had speculated it to be, given that our observed speedups across the 3 graphs were 1.837x to 1.847x. Nevertheless, since our observed speedups are less than the ideal Amdahl's Law speedup, the Law helps explain why regardless of how many threads we tried beyond 2, our speedup was staying the same. Our speedups were staying the same, because our speedup in this problem is bounded by the amount of work that can be parallelizable. Amdahl's Law being idealistic may be a reason why our observed speedup isn't closer to the max bound. Further, intuitively, it makes sense that our speedup was being maxed out at 2x, since our mergeSort parallelizes 2 recursive calls.

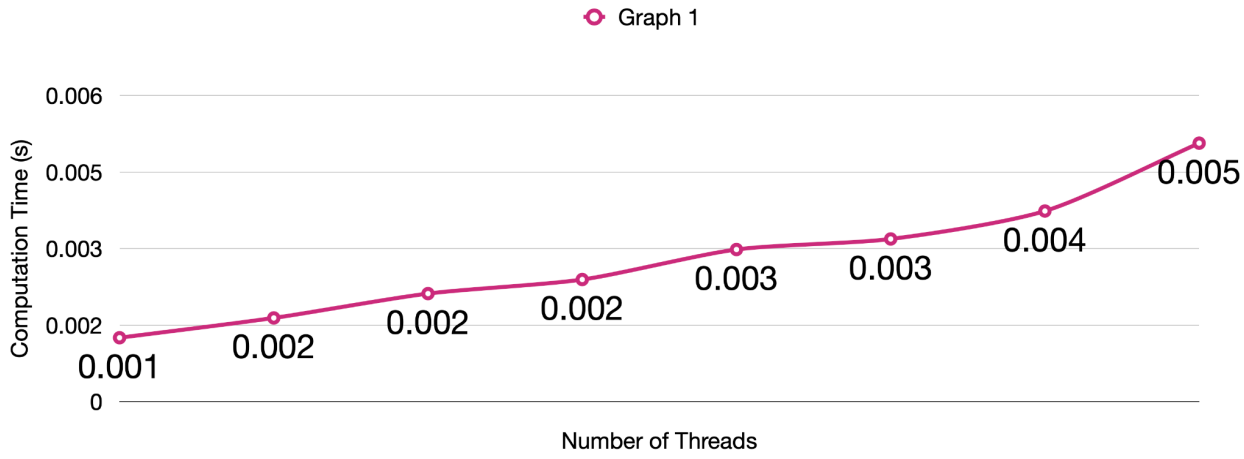
This shows that there may be room to make the parallelizable portion of the code even faster: even though we exhausted the ways to make the 2 recursive mergeSort calls run in parallel and optimized the number of times a task is enqueued, perhaps future studies can look into trying to make the merge function itself more amenable to parallelism, even though merge depends fully on the 2 halves being sorted and is, thus, in some ways inherently sequential.

Regardless, as Amdahl's Laws shows us, even with such a change, our speedup would still not be able to exceed 3.4x to 3.9x speedup, regardless of how many threads are used on these 3 input graphs for Kruskal's Algorithm. With Boruvka's Algorithm, we aimed to eliminate dependencies that inherently prevent speed up from going past 2x and get closer to max idealistic speedup ranges for finding MSTs.

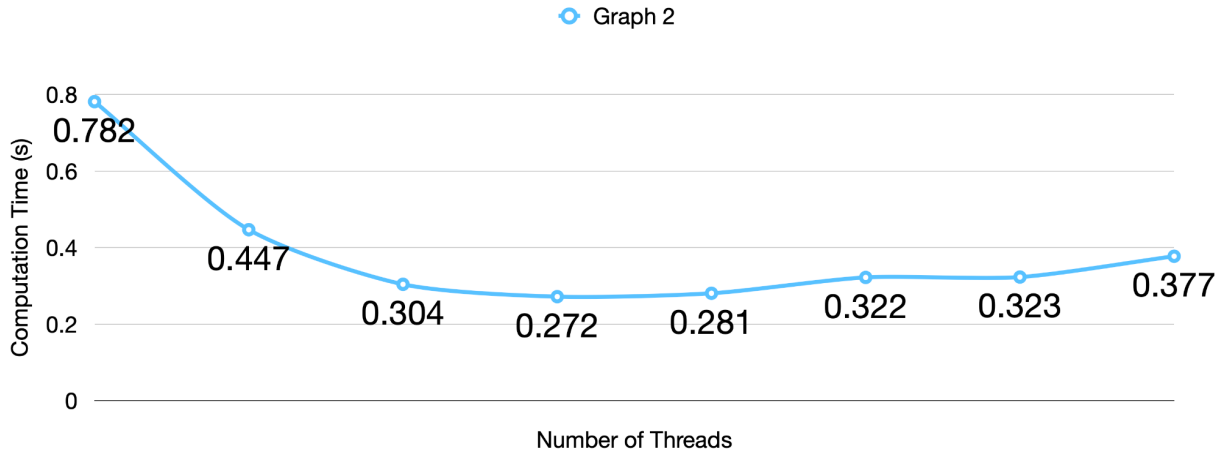
RESULTS OF BORUVKA'S ALGORITHM IMPLEMENTATION

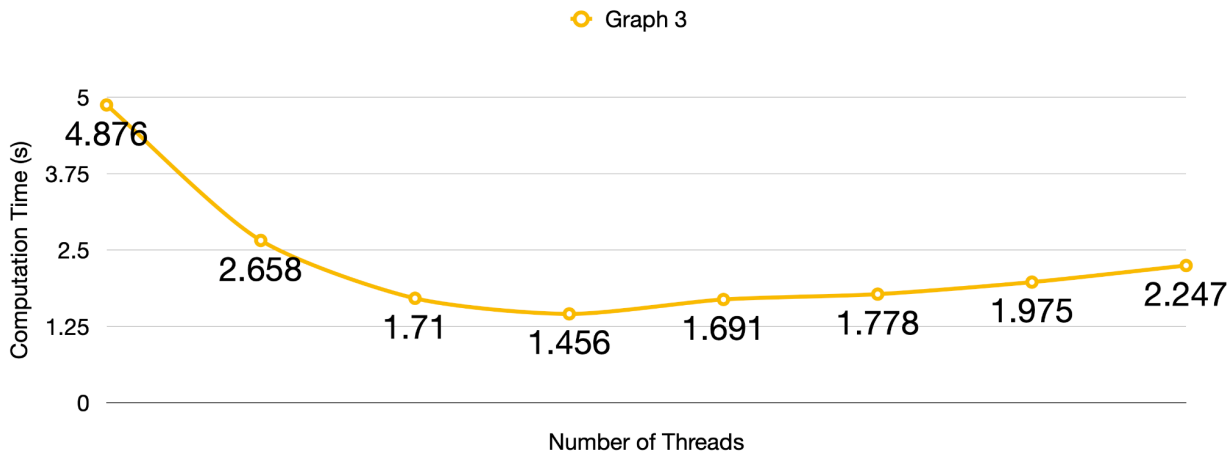
Computation Time v. Number of Threads

We tested our fine-grained locking implementation of Boruvka's on the same three types of graphs that we tested Kruskal's algorithm on. Specifically, we measured the computation time, as defined above, on thread counts of 1, 2, 4, 8, 16, 32, 64, and 128. The following graphs chart the computation time on a specific thread count for each input.



From the above trendline, we can determine that the size of the first graph (200 nodes and 15000 edges) is too small to achieve any speedup from parallelism. Our sequential implementation runs in 0.000637 secs, which is faster than any of the results we got for our fine-grained locking implementation. The trend we notice here is that on small graphs, the overhead of parallelism outweighs any potential benefits from parallelism. As we spawn more threads, our performance worsens since the computation time increases by some factor of new threads we must spawn.





For both our second and third input graphs, we notice speedups between 1 and 8 threads. On the second input, we achieve a maximum speedup of 2.88 at 8 threads. On the third input, we achieve a maximum speedup of 3.35 at 8 threads.

After this point, we begin to see a drop-off in performance due to a number of potential reasons. One potential reason is that the overhead of thread creation begins to outweigh the amount of parallelizable work. As the algorithm progresses, we have fewer edges that we can contract, which means that a thread could be assigned to many edges that already belong to the same vertex component. At later iterations of the algorithm, having a higher thread count will create more overhead since there is less parallelizable work.

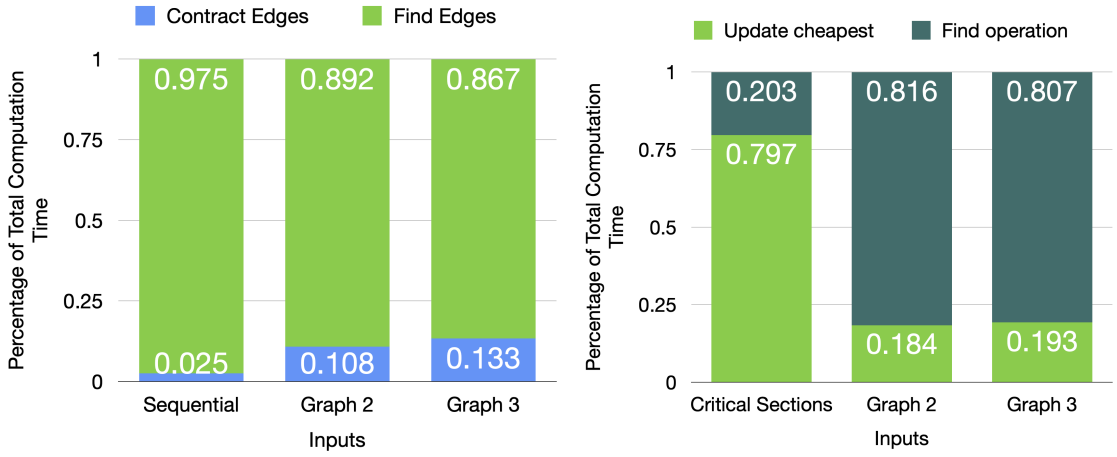
However, as seen in the table below, the speedups observed between 16 and 128 threads for Graph 3 exceed those of Graph 2 since there is a greater amount of parallelizable work in Graph 3 than in Graph 2.

Number of Threads	Graph 2	Graph 3
2	1.75	1.83
4	2.57	2.85
8	2.88	3.35
16	2.78	2.88
32	2.43	2.74
64	2.42	2.47
128	2.07	2.17

Takeaways

Overall, we were able to notice a better trend in speedup in comparison to Kruskal's algorithm since we had a more parallelizable workload in Boruvka's algorithm. As we identified earlier in this report, out of the two steps in our implementation of Boruvka's algorithm, namely finding and contracting edges, the step for finding edges made up roughly 97.5% of the work of our sequential implementation of Boruvka's algorithm. With our parallel fine-grained locking implementation, the amount of time spent finding edges for Graph 2 and Graph 3 reduced to 89.2% and 86.7% of our computation time.

Although we were unable to parallelize the contraction step of Boruvka's algorithm, we do not think that seriously limited our speedup, as it made up 2.5% of the total workload in our sequential implementation. However, there is still a sequential dependency between the two steps of our implementation, or an implicit synchronization barrier that prevents us from moving onto the contraction step before we have searched through our entire array of edges, which is one potential reason for limitations with speedups.



The above bar graphs demonstrate our success in speeding up what we initially identified as the most computationally expensive parts of our implementation. The bar graph on the left shows the breakdown of how much time is spent on the two steps of finding and contracting edges. In Graphs 2 and 3, we see that the proportion of time that we spend on finding edges has been reduced in comparison to our sequential implementation due to our parallelization efforts. Note that we see the greatest reduction in Graph 3 since it has the greatest amount of parallelizable work.

The bar graph on the right breaks down the step of finding edges into two substeps. The first substep is performing the find operation on the union find data structure in order to determine which vertex components the endpoints of the current edge belong to. The second substep is

performing the update to the cheapest edge array. In our parallel implementation with critical sections, we identified the step of updating the edge in the cheapest array to be more computationally expensive than the find operation on our union find data structure. By reducing contention over the cheapest edge array through fine-grained locking, we can see that the proportion of time spent updating the cheapest edge array decreased.

Ultimately, with our implementation of Boruvka's algorithm, we were able to achieve speedups over Kruskal's sequential algorithm on large graph inputs. Note that on our smallest input, Graph 1, Kruskal's algorithm performed the best out of all implementations. On larger inputs, we were able to achieve a speedup of 3.292 and 3.838 over Kruskal's by running Boruvka's with 8 threads.

REFERENCES

Bailey, Mike. (2022, March 16). *OpenMP Tasks.* [Lecture Slides/Handout from Oregon State University CS575 as per course number in link]. Computer Graphics, Oregon State University. https://web.engr.oregonstate.edu/~mjb/cs575/Handouts/tasks.1pp.pdf

Chaudhary, Dhanesh. (2020, Jan 26). *Union find by rank and path compression.* Medium. https://medium.com/@dhaneshchaudhary99/union-find-by-rank-and-path-compression-5f461a5b9839

Diderot. (2021). Textbook (III): Part 4: Minimum Spanning Trees: Section 14: Sequential MST Algorithms. Diderot: 15210. https://www.diderot.one/courses/89/books/352/chapter/4581

Diderot. (2021). Textbook (III): Part 4: Minimum Spanning Trees: Section 15: Parallel MST Algorithms. Diderot: 15210. https://www.diderot.one/courses/89/books/352/chapter/4589

Monsalve, Jose. (2019, April 26). *OpenMP Workshop: Quick Overview of OpenMP*. Argonne National Laboratory, U.S. Department of Energy (managed by UChicago, Argonne, LLC). https://www.alcf.anl.gov/sites/default/files/2020-01/OpenMP_Jose.pdf

OpenMP 3.1 API C/C++ Syntax Quick Reference Card. (2021). OpenMP ARB, Miller & Matson (http://www.millermattson.com/). https://www.openmp.org/wp-content/uploads/OpenMP3.1-CCard.pdf

Terboven, C. Schmidl, D. RWTH Aachen University. (1 October 2015 – 31 March 2018). *Using OpenMP Tasking*. EU H2020 Centre of Excellence. https://pop-coe.eu/sites/default/files/pop_files/pop-webinar-openmptasking.pdf

15-418/618 Spring 2022 Exercise D Solutions. Problem 1: Resource Oriented Scaling (2022). [Referenced Problem 1 and Given Solution Posted on Canvas] https://canvas.cmu.edu/courses/26708/files/7762924?wrap=1

LIST OF WORK BY EACH STUDENT, AND DISTRIBUTION OF TOTAL CREDIT

Both of us contributed equally to the project (i.e. 50%-50%).

We collaborated to determine strategies for representing graphs and parallelizing our two algorithms, and we split the work of coding and running tests so that one partner worked on each algorithm. Throughout the project, we helped debug each other's code and analyze performance results. We also both contributed equally to the write ups for the project proposal, checkpoint submission, and final submission.