Comparing Parallel Implementations of Minimum Spanning Tree Algorithms

Alex Knox (akknox), Elizabeth Knox (emknox)

April 28, 2025

1 Summary

We designed parallel implementations for three MST algorithms: Prim's algorithm, Kruskal's algorithm, and Boruvka's algorithm. We evaluated the performance of each of our implementations on benchmarks tests with properties designed to test the impact of graph size and density. We will compare the differences in the sources of parallelism between the algorithms, and the effect that has on benchmark performance.

2 Introduction

Consider a connected, undirected graph $G = (V, E)$ where each edge e_i is assigned a positive weight w_i. A **minimum spanning tree** (MST) 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.

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.

There are three well-known algorithms for finding a minimum spanning tree: Prim's Algorithm, Kruskal's Algorithm, and Boruvka's Algorithm, all with asymptotic complexity $O(m \log n)$. However, they vary significantly in their capacity for parallelism. We will describe the algorithms in more detail in their respective sections, but at a high level, both Prim's and Kruskal's are inherently very sequential, while Boruvka's is more parallelizable. However, all three algorithms have some capacity to be parallelized. In this project, we explore opportunities for parallelism in all three algorithms, comparing their speedup and empirical performance on benchmark tests.

3 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.

Most of the work comes from finding the minimum weight edge among the current

3.1 Available Parallelism in Prim's Algorithm

To achieve $O(m \log n)$ asymptotic complexity, Prim's is generally implemented with a priority queue. This makes extracting the minimum edge and inserting elements into the priority queue $O(\log n)$ operations. However, it leaves little room for parallelism. The main loop is inherently sequential, since the choice of lightest edge on one iteration can change the lightest edge candidates for the next iteration.

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
```

An alternative implementation for Prim's iterates through a list of possible lightest edges. This takes $O(n)$ to find the minimum but $O(1)$ to add new elements to the list, leading to $O(mn)$ asymptotic complexity. However, here there is opportunity to parallelize the minimum finding across multiple processors. With perfect speedup for finding the minimum and enough processors, it is theoretically possible for this implementation to become competitive with the priority queue implementation

3.2 Parallel Minimum Finding

First, we attempted to parallelize the minimum finding in a version of Prim's algorithm which computes the minimum by iteration. Here, empirical timing measurements show that 99% of the runtime is spent on this step, and so parallelism has good capacity for speedup in this case.

We implement a parallel minimum reduction over the current edge list using a custom reduction function with OpenMP to obtain the following speedup results shown in Figure 1

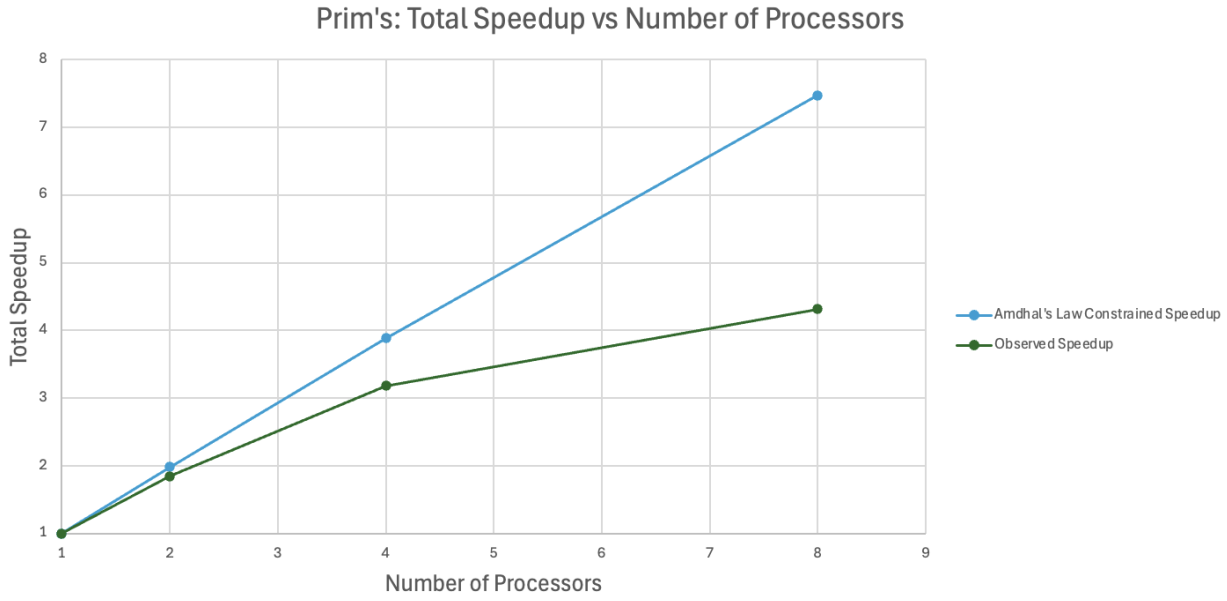

Figure 1: Parallelizing the minimum computation of the lightest edge achieves good speedup on an implementation of Prim's algorithm which finds the minimum through iteration, because this step takes up a large percentage of the total runtime. Results for the speedup graph were generated on GHC71 using a graph with $N = 2000$ and $M = 20000$

The speedup is limited by the overhead of finding the minimum across multiple threads: even though we can find the minimum of components of the edge list separately, there must be some communication at the end in order to compute the overall minimum. We begin to see some of the effects of higher communication costs at high thread counts here.

In addition, while speedup is achieved and scales decently, the runtime even at 8 threads perform substantially worse than a sequential implementation of Prim's algorithm using a priority queue. This is somewhat expected: even ignoring communication costs, the $O(n)$ cost of finding the minimum in the iterative implementation would need to be split among $O(\frac{n}{\log n})$ processors in order to achieve $O(\log n)$ effective minimum finding. For large numbers of vertices, this is infeasible, and so we do not continue this line of investigation in the remainder of our project.

3.3 Parallel PQ Insertion

There is another opportunity for parallelism in Prim's algorithm, in the addition of neighbors to the priority queue. In dense graphs, a given vertex may have many neighbors. While we can only insert one edge into the priority queue at at time, we only need to insert edges into the priority queue if they reach vertices that have not already been visited. In the best case, if we have already visited all of the neighbors, the checks will become entirely independent. Otherwise, parallelism may be limited by contention to the priority queue.

There is a `for` loop over all neighbors of the most recently visited vertex which determines whether a neighbor needs to be added to the priority queue. We start by parallelizing this loop, splitting up the list of neighbors among the available threads, and requiring priority queue insertion to be done by only one thread at a time. To amortize the overhead of creating new threads, we use the parallel approach only when at least half of the vertices have been visited before, as a heuristic that a significant percentage of the neighbors we consider have already been visited, and thus will not need to be added to the priority queue.

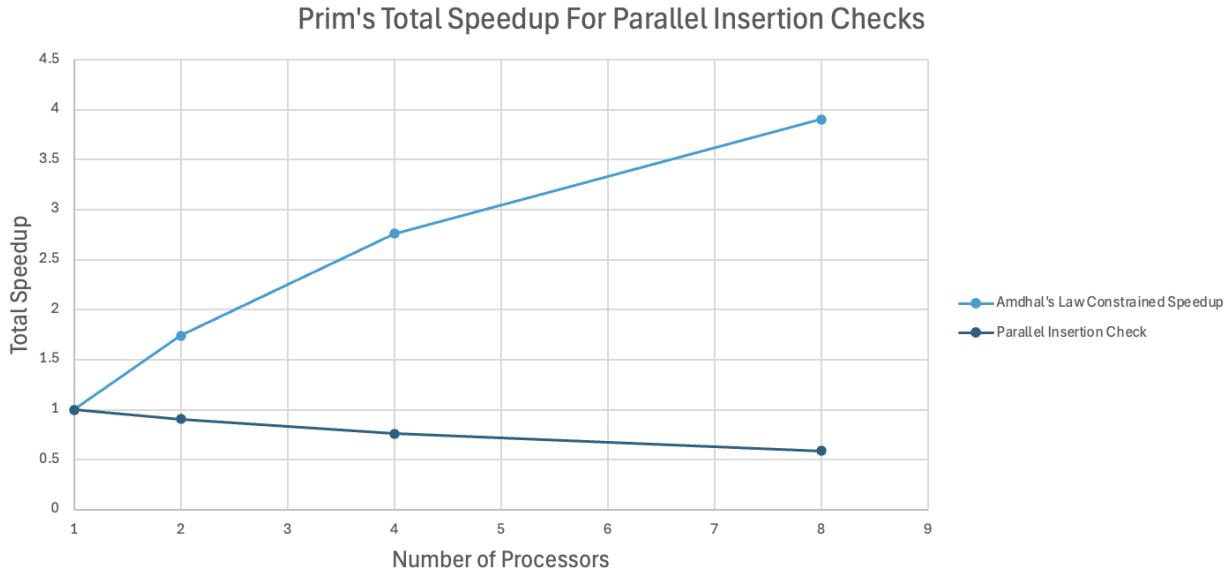

Figure 2: Empirically, we measure about 85% of the runtime is spent inserting elements into the priority queue. Thus, we use Amdhal's Law to account for this limitation of the parallel speedup. Parallelizing the loop over a neighbor list to determine candidates for priority queue insertion is constrained by contention on the priority queue and overhead in managing threads. We see negative benefit in more threads with this approach. Results for the speedup graph were generated on GHC56 using a graph with $N = 2000$ and $M = 1200000$

As shown in Figure 2, this implementation performs very poorly, never achieving runtimes better than the sequential runtime on our test cases. As we increase the threshold for allowing the parallel code to run at all, the performance improves becoming closer to the sequential runtime. This suggests that the bottleneck is contention with priority queue insertion combined with overhead of managing multiple threads.

To reduce contention on the priority queue, we instead implement one priority queue per available thread. This means that we can insert neighbors into the priority queue completely in parallel, with no contention

since each thread has its own data structure. As a tradeoff, we have to inspect the top of all the priority queues before determining what the true minimum is.

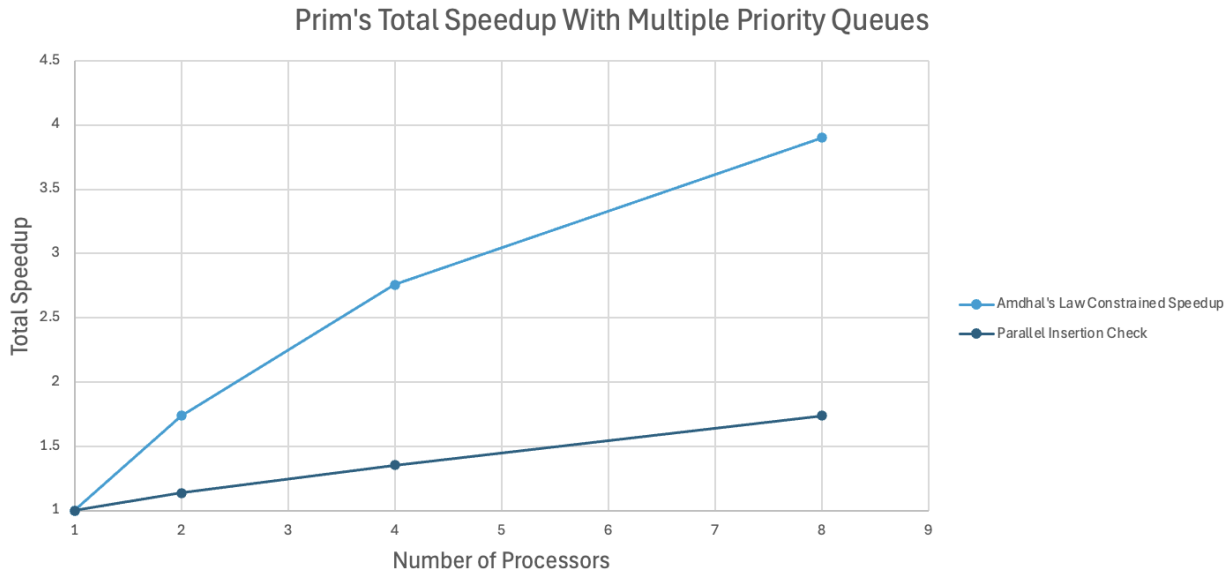

Figure 3: Empirically, we measure about 85% of the runtime is spent inserting elements into the priority queue. Thus, we use Amdhal's Law to account for this limitation of the parallel speedup. To avoid contention on priority queue insertion, each processor maintains its own priority queue. Then, parallelizing the loop over a neighbor list is completely independent. Results for the speedup graph were generated on GHC56 using a graph with $N = 2000$ and $M = 1200000$

This performs much better than our previous implementation. As we reach higher thread counts, the cost of finding the minimum edge in the frontier– the "lightest edge"– increases. This is because we need to inspect the minimum element out of across all the priority queues, which increases as the number of threads increases. This is not factored into the computation of the constrained speedup, but it has a limiting effect on the speedup.

4 Kruskal's Algorithm

Kruskal's 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, as in Prim's algorithm, Kruskal's algorithm builds up multiple connected components. The addition of each edge joins two connected components until every edge has been considered or the result is connected.

In practice, this algorithm is implemented with a union-find data structure for keeping track of connected components, which allows for $O(\log n)$ complexity for finding the connected component of a vertex and for combining two connected components. For an input graph of n vertices and m edges, the loop runs m times. The asymptotic complexity is dominated by the sort step, which is $O(m \log m)$.

4.1 Available Parallelism in Kruskal's Algorithm

There is a large proportion of inherently sequential work in Kruskal's Algorithm. In particular, the inner loop traversing the sorted list of edges must be run sequentially, because the addition of an edge during some

Algorithm 2 Kruskal's Algorithm

```
1: function KRUSKAL(G = (V, E)):
2:     T = { }
3:     Q ← sort(E)
4:     for (a,b) ∈ Q (lightest to heaviest) do:
5:         if a and b are not in the same connected component then T = T ∪ {(a, b)}
6:         end if
7:     end for
8: return T
9: end function
```

iteration will affect whether later edges will be included in the same connected component or not. Thus, edges must be checked in weight priority order for the algorithm to produce correct results.

For the sort step, we implement merge sort due to its capacity for parallelization. However, that too has sequential limitations. In merge sort, we can separately sort two halves of the list, but merging the list together must be done sequentially.

To determine what effect the sequential limitations will have on the capacity for speedup, we inserted timing code to compute the percentage of the total compute time spent on work that has the capacity to be parallelized. While the actual percentage is a function of the number of vertices and edges in the graph, we found the empirical timing of our benchmarks to be a good approximation of the percentage. Then, we used Amdhal's Law to visualize the limitations on speedup, shown in Figure 4.

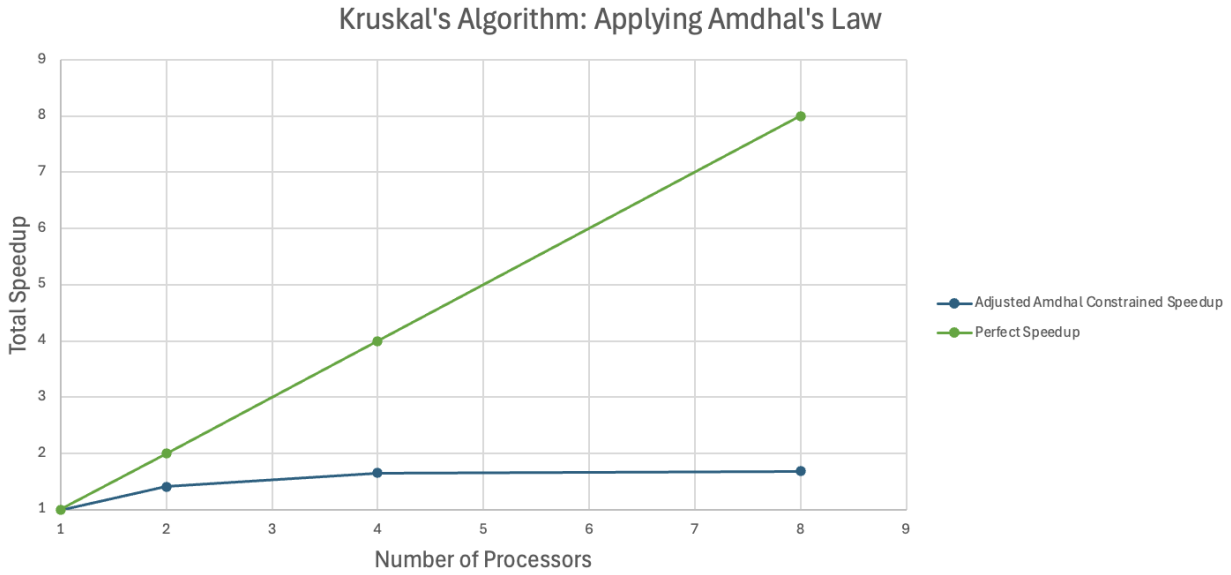

Figure 4: The proportion of parallelizable work in our implementation of Kruskal's Algorithm averages around 0.55, after timing parallelizable runtime of several of our benchmark tests. When applying Amdhal's Law, this significantly limits the speedup it is possible to achieve (compared to the "perfect", linear speedup).

4.2 Parallelizing the Sorting Step

First, we attempt to parallelize Kruskal's Algorithm by taking advantage of the available independence in the sorting step. Our initial attempt at parallelization includes a parallel merge sort function with a given allocation of N threads available. That function makes two recursive calls to parallel merge sort with $N/2$ threads available. When a call is made with only 1 thread available, the function defaults to a sequential sorting algorithm.

The parallelism was implemented with OpenMP tasks, where each recursive call was designated as a task. Once both tasks finished, the program continues with the merge step. As shown by the green line in Figure 5, this implementation worked well at low thread counts, but scaled poorly to 8 threads. This is partially because OpenMP tasks use a task scheduling system to organize the computation of tasks, which would allow for smarter scheduling with many tasks but is overly complicated for our needs.

In this case, the parallelism we want is very explicit: the two recursive calls should be computed in parallel. Since they are about the same length, both calls should be equal in work, and our implementation only attempts to parallelize as many recursive calls as we have threads. Because of this, we can bypass the scheduling overhead of OpenMP tasks by using explicit forks and joins. Using the C++ `<thread>` library, we create a new thread to sort the right half of the array, while the existing thread sorts the left half of the array. The threads join once the two halves are both sorted.

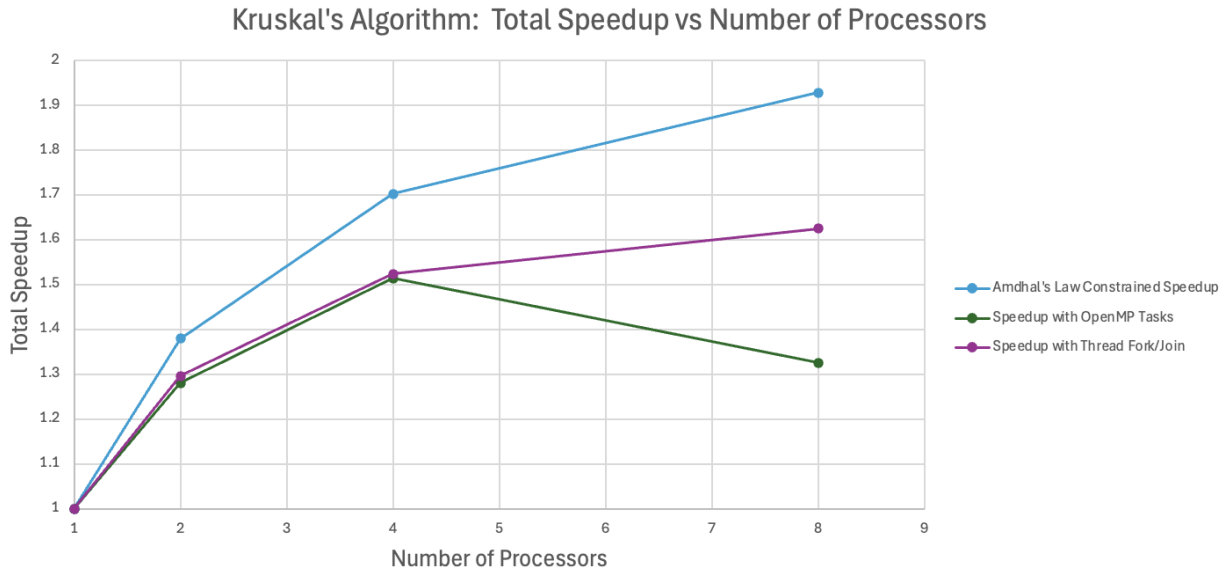

Figure 5: We tried two methods of parallelizing the merge sort step in Kruskal's Algorithm. The first (green), using OpenMP tasks, suffers from overhead at high thread counts. The second (purple), using the C++ thread library, scales better. Results for this graph were generated on GHC71, on a benchmark test with $N = 3000$ and $M = 1200000$.

Now, the implementation scales much better. We don't get quite perfect speedup, due to some imbalance in the cost of sorting subproblems, leading to extra synchronization cost at the join barrier.

4.3 Parallelizing with Helper Threads

There is not anything in the sequential version of Kruskal's Algorithm to be parallelized in the inner loop due to a correctness dependence on the order that edges are considered. However, the algorithm claim relies on the cycle property of MSTs: the heaviest edge in a cycle is excluded from the MST. This suggests some data independence in the algorithm–as edges are added to the MST, some edges later in the list (and therefore heavier) will complete a cycle in the partial result, so we know they can be eliminated. Further, this is a one-way transition: once an edge completes a cycle with the existing tree, it will never need to be considered again.

Our next implementation applies this idea with a helper threads model. While one main thread processes edges for inclusion in the MST, other threads check remaining edges for exclusion in the MST[1]. We implement this with a shared array of booleans for edges, storing whether they can definitely be excluded from the MST

[1]This implementation was inspired by this paper: A. Katsigiannis, N. Anastopoulos, K. Nikas and N. Koziris, "An Approach to Parallelize Kruskal's Algorithm Using Helper Threads," 2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum, Shanghai, China, 2012, pp. 1601-1610, doi: 10.1109/IPDPSW.2012.201.

or whether they still need to be considered. The main thread checks that helper threads have not removed an edge from consideration before doing its own inclusion checks. Meanwhile, helper threads repeatedly cycle through the edges to determine which edges can be excluded by the current state of the MST.

The helper threads only write to the array of booleans, but read from data structures like the MST and the current connected components of the graph. However, because it is a one way transition from considered to definitely excluded, the synchronization in this implementation is quite lightweight. It is okay, for example, if the main thread and helper thread both check an edge at the same time– in this case, they might both determine that the edge should be excluded. However, the amount of work that might be duplicated in this case is very small, and does not warrant the overhead or contention that might be introduced with more controlled read access to the tree.

For now, we assign work to helper threads by splitting the list of edges into equal components. Every helper thread repeatedly iterates through their assigned edges to determine if cycles are created. This means that as the main thread passes through the list, it makes whole threads redundant: in the four threads case, for example, once the main thread gets through the first third of the edge list, the work of the helper thread assigned to the first third of the list becomes useless, since we have already entirely determined what is in the MST from that section. A better approach would be to re-allocate sections of the edge list occasionally based on an updated measure of how far the main thread has progressed through the list to reduce wasted work.

This method gives us substantial performance improvement as shown in Figure 6 For comparison purposes, we have still included the same line for our estimate of maximum speedup due to Amdhal's Law, as in Figure 5. We are able to exceed that measurement because our parallelization with helper threads approach adds work that was not in the original algorithm, which is not accounted for by Amdhal's Law.

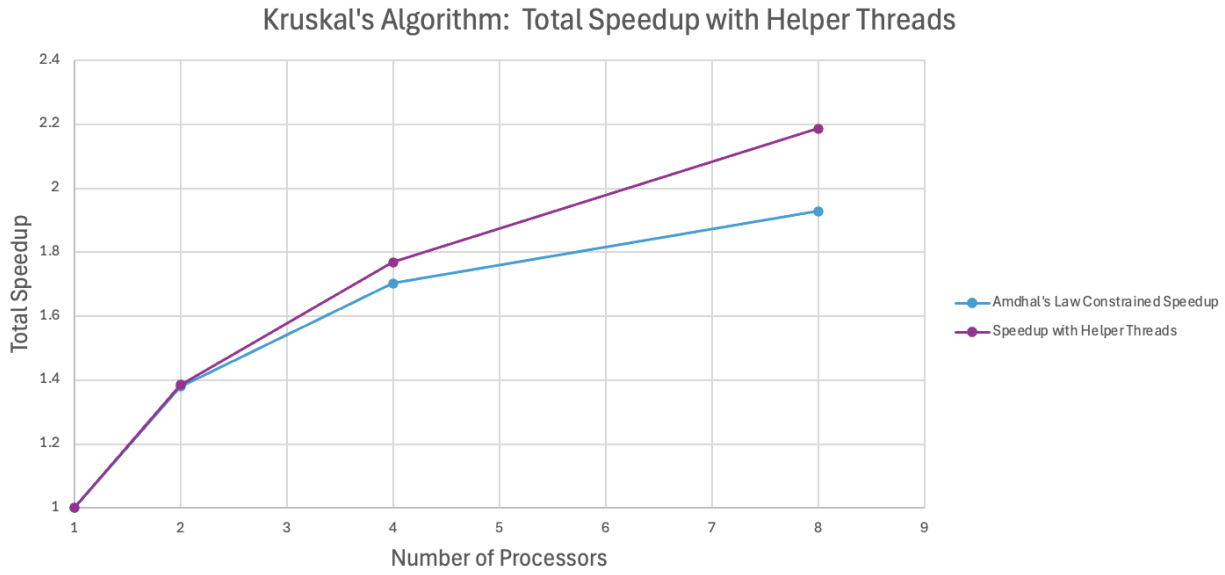

Figure 6: With parallelism through helper threads changing the work that is being done to Kruskal's Algortithm, we are able to exceed our original belief of the sequential limitations of the algorithm. Here, we display the total speedup achieved with n threads (of which $n-1$ are helper threads). Helper threads are able to flag a large percentage of the edges for exclusion: 47% with one helper thread, 67% with three helper threads, and 86% with seven helper threads. Results for this graph were generated on GHC71, on a benchmark test with $N = 3000$ and $M = 1200000$.

5 Boruvka's Algorithm

Boruvka's algorithm uses iteratively finds graph contraction to eliminate edges that would form cycles. Unlike the other two algorithms described, it computes multiple light edges per iteration. The approach overall is still greedy: the algorithm chooses the lightest outgoing edge from each connected component. Like Prim's algorithm, this is based on the MST cut property, which guarantees that for any cut in the graph (in this case, the cut contains the entire connected component) the lightest edge crossing it will be in the MST. Multiple connected components are joined on each iteration, and it is even possible for one connected component to be joined to multiple others in the same iteration. If an edge spans two connected components, it may be the lightest outgoing edge for one but not the other.

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
```

The inner for loop of the algorithm alternates between searching through all the edges and contracting to remove some edges from future consideration. In implementation, we use a union-find data structure to keep track of connected components, as graph contraction works by treating each component connected in the MST as a single vertex in the original graph. The contraction step does present challenges to parallelizing this algorithm: while contracting vertices in union-find is relatively cheap, modifying the representation of edges in G can be quite expensive, especially when other threads are searching through the edges. Nevertheless contraction for this algorithm decreases the size of an arbitrary graph by about half after we have iterated through all the vertices, and it is important for good performance to reduce the search space of edges for future iterations of light-edge computation.

5.1 Available parallelism in Boruvka's algorithm

The first operation in the algorithm—finding the lightest outgoing edge from a vertex v—reads from the graph's edge list and updates a per-vertex minimum weight. A sequential implementation may repeat passes through essentially the same edge list; it may have only had a few edges contracted since another vertex looked through for a minimum. The second step—updating T—involves updating the union-find data structure, as well as adding edges to an output list. Finally, the third (contraction) step involves reads from the union-find data structure and appropriate updates to the edge list. Importantly, each step is almost always reading from one data structure and writing to another. Parallelizing across vertices directly is difficult, as we expect that synchronizing the memory accesses among threads at different points of the three steps will have significant overhead. However, more opportunities for parallelism are present once we restructured the algorithm to make a greater distinction between each step.

First, we find the lightest outgoing edge *for each* vertex v in G. The only writes in this step are happening to a stored local minimum, so we can parallelize this step as long as we can properly synchronize these updates.

Parallelizing edge contraction is possible, but more complicated. If we contract only after completing the previous step for all vertices then there is a higher volume of edges that need updating that would happen otherwise—some graphs we tested decreased the number of edges they had to consider by a factor of 10 every iteration until the last. Our most successful approach involved creating a new edge list, determining (or approximating) which edges in the original were still relevant, and copying them into the new list. We implemented this filter-like operation with a parallel prefix sum.

The main operations which remain sequential are the insertion into the output tree and the memory allocation needed to maintain auxiliary information that supports parallel operations like the prefix sum. Additionally, as discussed below, we implemented a threshold below which we compute sequentially. The reason for this is that because the graph size decreases roughly logarithmically, the last few iterations are attempting to parallelize over arrays that are short relative to the number of threads: each thread might have to make updates to less than five array indices. At this point the parallelism is not worth the overhead.

5.2 Parallelizing light edge computations

Parallelizing the light edge computations involves finding, for each vertex, the lightest of the multiedges which connects it to another vertex/connected component. We read from two shared data structures—the union-find storing the connected components and the edge list of the graph—but the primary concern is synchronization in storing global lightest edges for each vertex.

Our first attempt at parallelizing this section used a fine-grained atomic array, using the primitives of an `atomic` pointer type. This allows atomic loads and stores to or from each array element, as well as an atomic swap of one pointer for another within the array. With an atomic check-and-swap, we could appropriately synchronize the minimum updates by swapping only if the minimum weight was still larger than our current edge's weight. Performance-wise, this approach required at least one dynamic allocation per attempt to update the global minimum for a particular vertex so that we can update all fields of the minimum edge struct in just one pointer swap. This increases the overhead of attempting an atomic swap, so we also tested the performance of a check-and-check-and-swap.

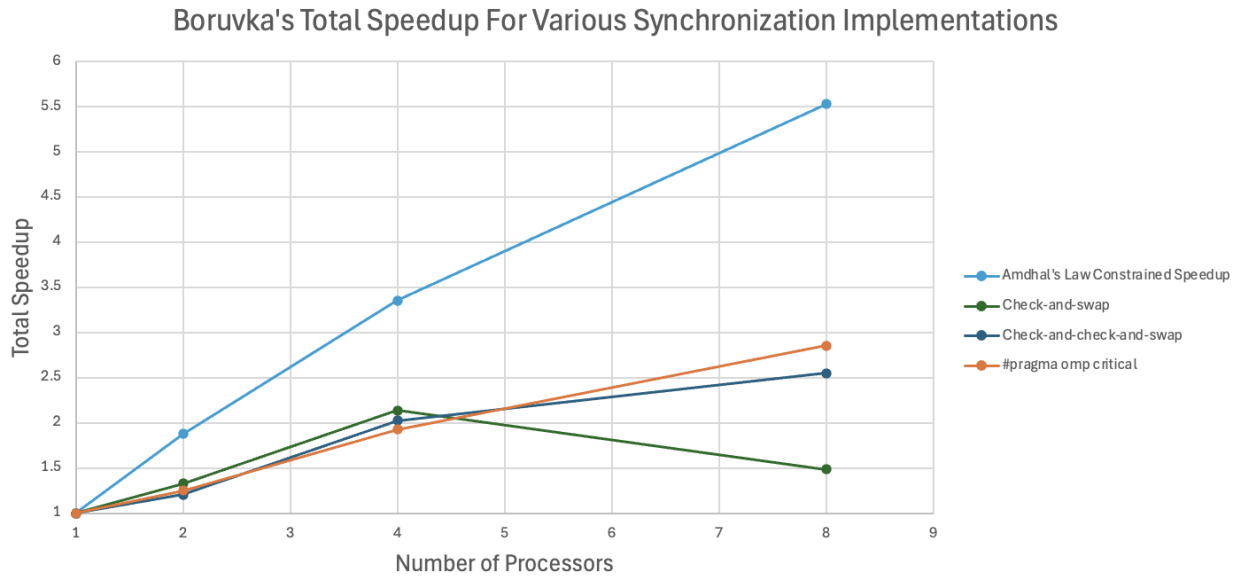

Figure 7: Comparison of synchronization methods on parallelizing the process of finding lightest outgoing edges for each vertex in the graph. Results for this graph were generated on GHC56.

We plotted the speedup achieved by these synchronization methods for protecting the global minimum values. This is displayed in figure 7. We noticed that the check-and-swap implementation's performance degraded past 4 threads. This was due to the increased contention on the atomic operations when there are multiple threads, and the fact that on each retry we must `malloc` space for the edge you attempt to swap. The speedup of the check-and-check-and-swap implementation performed better. When tested on a graph with 2000 vertices and 1200000 edges we found that there were only about fifteen swap misses per run of the algorithm. We thus explored other locking approaches, which trade off contention for overhead.

We then switched to protecting the minimum edge updates with a `#pragma omp critical` directive. This explicitly prevents any other thread from entering the critical region when another thread is updating it. This approach will create more contention than necessary, because if one thread is in the critical section

to update the minimum for a particular vertex then it is safe for a different thread to be in the critical section as long as it is trying to do updates for a different vertex. However, it avoids much of the overhead of our previous fine-grained approach, as once we are in the critical section we can do all updates in place. This is reflected in the plot comparing synchronization methods, where the `#pragma omp critical` version achieves the most linear speedup of each of the synchronization methods.

5.3 Parallelizing edge contraction

The goal of the edge contraction phase is to produce a smaller graph by eliminating edges that are irrelevant based on what has been selected so far for the MST. Though this is generally an expensive operation, it should speed up future iterations as it reduces the search space of edges which might be lightest. Because contraction works on multigraphs, there are multiple implementations of contraction that all yield correct results: they just differ in how many edges remain in the representation of the graph. This trades off decreased cost in the light-edge phase (by reducing the size of the search space) for increased cost in computing which edges are still relevant and making appropriate updates to the algorithm's local representation of the graph. We compared the performance of three different contraction implementations.

The first implementation does not change the graph representation; this approach yields the lowest cost for the contraction phase but does not decrease the size of the graph at all. We then add two edge elimination passes. The first eliminates self-loops (edges that connect two vertices now in the same connected component). The second aims to reduce the set of edges connecting the same two connected components to just a single edge, as only the cheapest among them could be in the MST. We describe our implementations of these elimination heuristics below.

Overall, we saw significant decrease in the number of edges throughout iterations of the algorithm. In the graph below, we show the edge decrease at each of the five iterations of contraction for a particular test case. This test starts with a graph of 2000 vertices and 1200000 edges.

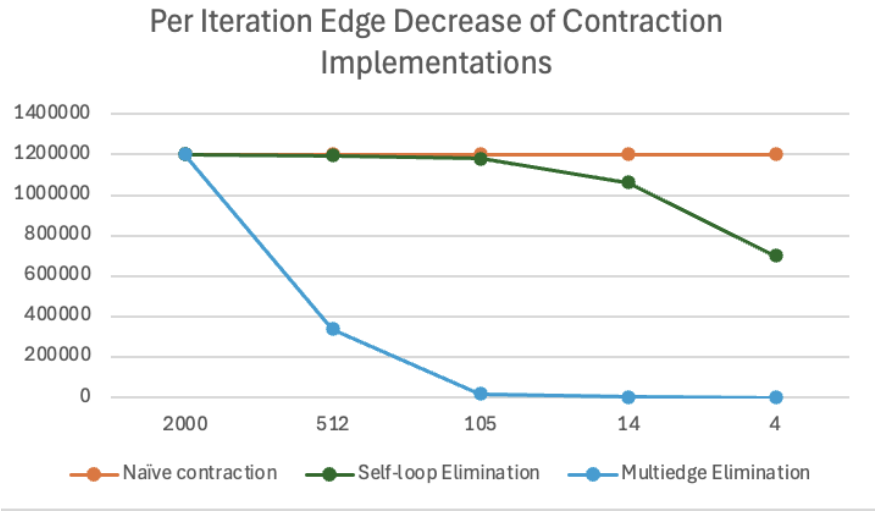

Figure 8: Edge decrease of the three levels of contraction algorithms. The number of vertices and edges left at each iteration is deterministic. As heuristics eliminate more edges the contraction phase gets more expensive. The biggest jump is between a naive algorithm and any elimination algorithm, as elimination algorithms introduce a need to modify significant proportions of the graph at each iteration.

Self-loop elimination is essentially a filter on whether or not edges span connected components. The main challenge in parallelizing the creation of this array is to determine what element of the smaller array to write to given that we have an edge in the old array that may still be relevant in the future. We implemented this using a parallel prefix sum over a boolean list, which computes the number of relevant edges preceding a

particular one, as well as the overall total. This does have some spatial overhead because we have to maintain a boolean array and an int array of the same size as the current number of edges, but this is less information than we were storing about the original edges anyway. Additionally, much of the work (computing whether an edge is relevant, and the prefix sum) can be parallelized over the edges.

In addition to self-loops, contraction also creates multiedges: edges of different weights that connect the same two connected components. For the purposes of an MST, these edges are the same. If we take any of them, we will take the lightest one. Thus, finding light edges will be less expensive if we could eliminate all but the lightest of the equivalent edges. However, we don't want to add an extra pass to compute a minimum of the multiedges, especially since, as we described in the previous section, synchronization in computing global minimums in parallel is a concern. Instead, we developed a heuristic to approximate the minimum while doing the relevancy pass. If a particular edge (that spans two connected components) is heavier than the stored minimum connecting the same ones it is irrelevant. If it is lighter, we will try to update the stored minimum. We don't really need any synchronization on this minimum update *because we are only trying to approximate it*: if the update of the actual minimum is lost because of a concurrent update, that is fine, because we can guarantee that the actual minimum is *no larger* than what we stored. Our primary concern, and indeed our guarantee, is that no edge is ever incorrectly marked irrelevant. Adding this check to eliminate some heavy multiedges significantly decreases the sizes of the subgraphs we have to work with in subsequent iterations.

With the performance we saw as a result of improving our graph contraction representation, certain parallelism techniques we had originally intended to try were no longer feasible. For example, we had originally considered implementing Boruvka's algorithm using MPI, because parallelism in computing a minimum weight edge for each vertex does not require much communication between threads until the end, when they share information to determine a global minimum. When the edge list is static, this still works in theory. However, the scale of updates we ended up doing to the edge list would be unrealistic to match in efficiency without a shared address space.

5.4 Combining two phases of parallelism

This method of implementing contraction that we described in the previous iteration, while it creates further opportunity for parallelism, adds more work in the contraction phase than we would do otherwise. As such, the relative speedup that we saw from parallelizing only the light edge computation was much lower than in a naive (non-parallelizable) implementation of contraction. However, when parallelized we achieved better speedup (and better raw runtime on larger numbers of threads) than previous implementations.

The improvement is highlighted in figure 9. Introducing the edge-elimination strategies made the light-edge phase cheaper and the contraction phase more expensive. Thus (compared to our initial parallelization), the speedup from just the light-edge phase is less significant. Parallelizing the contraction phase, on the other hand, as well as a conjunction of the two approaches achieved much better speedups. In terms of raw runtime, the trade-off in favor of edge elimination was noticeably better. On the same test case, running on 8 threads, the implementation which parallelized over both the light-edge and the contraction phase of the algorithm ran twice as fast as that which parallelized over light-edge computation and used the cheapest contraction possible!

6 Evaluation

6.1 Benchmark Design

We will test the effect of changing different graph parameters on the resulting speedup achieved by our different algorithms. To make these comparisons, we have written infrastructure to create random graphs with specified parameters.

First, we create a set of benchmark tests with a fixed number of vertices and a varying number of edges. This allows us to compare the effect of graph density on speedup.

We also create a set of benchmark tests with a varying number of vertices and edges, but where the density of the graph is always at 25% saturated (i.e. if there are N vertices it would take $\frac{N(N-1)}{2}$ edges to make a clique, so we create a graph with $\frac{N(N-1)}{2}$ edges). Since both vertices and edges scale proportionally,

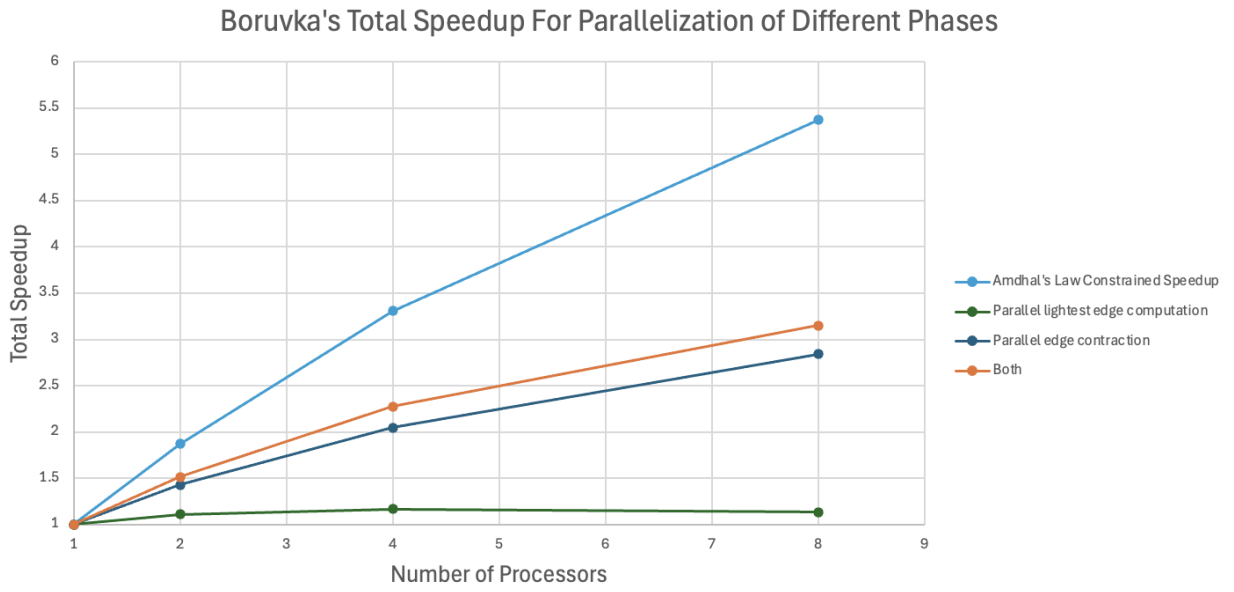

Figure 9: Comparison of speedup for Boruvka's algorithm, when introducing parallelism in each of the two phases of the algorithm. Note that all implementations displayed in this graph use light-edge elimination and multiedge-elimination techniques described in the previous section. Results for this graph were generated on GHC56.

we will use the number of vertices to represent the graph size. We will use these to compare the effect of size on speedup.

6.2 Effect of Varying Graph Size

We compare the speedup of our implementations of all three algorithms, on benchmark test cases of varying size in Figure 10 (Prim's algorithm), Figure 11 (Kruskal's algorithm), and Figure 12 (Boruvka's algorithm).

For Prim's and Kruskal's, varying the graph size does not have a significant effect on the speedup. With Prim's algorithm, the speedup is poor on small graph sizes, but relatively similar on the larger sizes. This is likely because the parallelism in Prim's algorithm comes from vertices having lots of neighbors, which is less true in smaller graphs. With Kruskal's algorithm, the speedup is very consistent, regardless of graph size. With Boruvka's algorithm, there is a positive correlation between graph size and speedup. This algorithm is particularly well-suited to speeding up computation on larger graphs because the graph size decreases through each iteration.

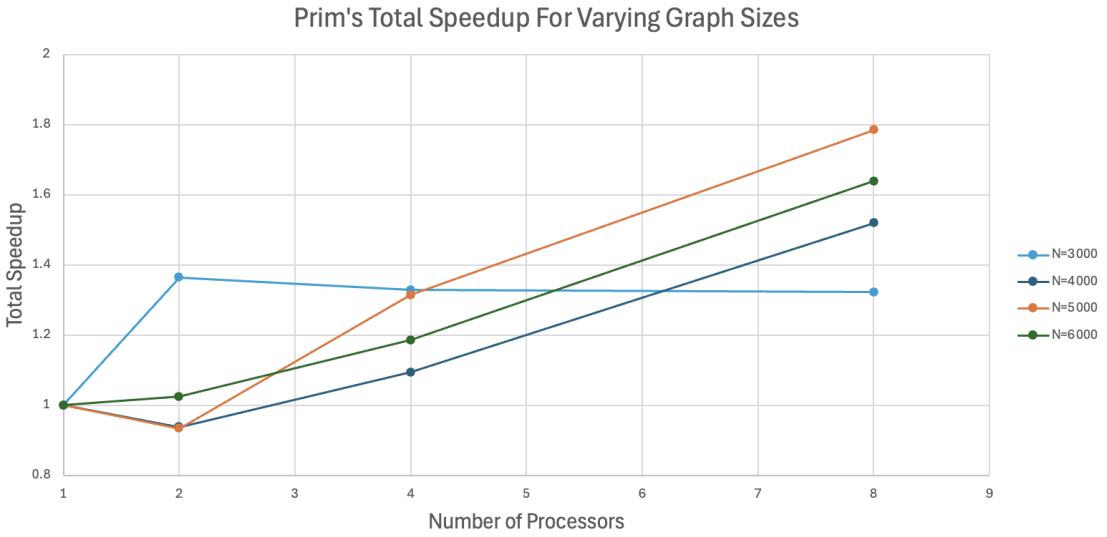

Figure 10: The effect of varying graph size on the speedup obtained from our parallel implementation of Prim's algorithm. Size is measured by the number of nodes, density remains constant across these benchmarks. Results for this graph were generated on GHC71.

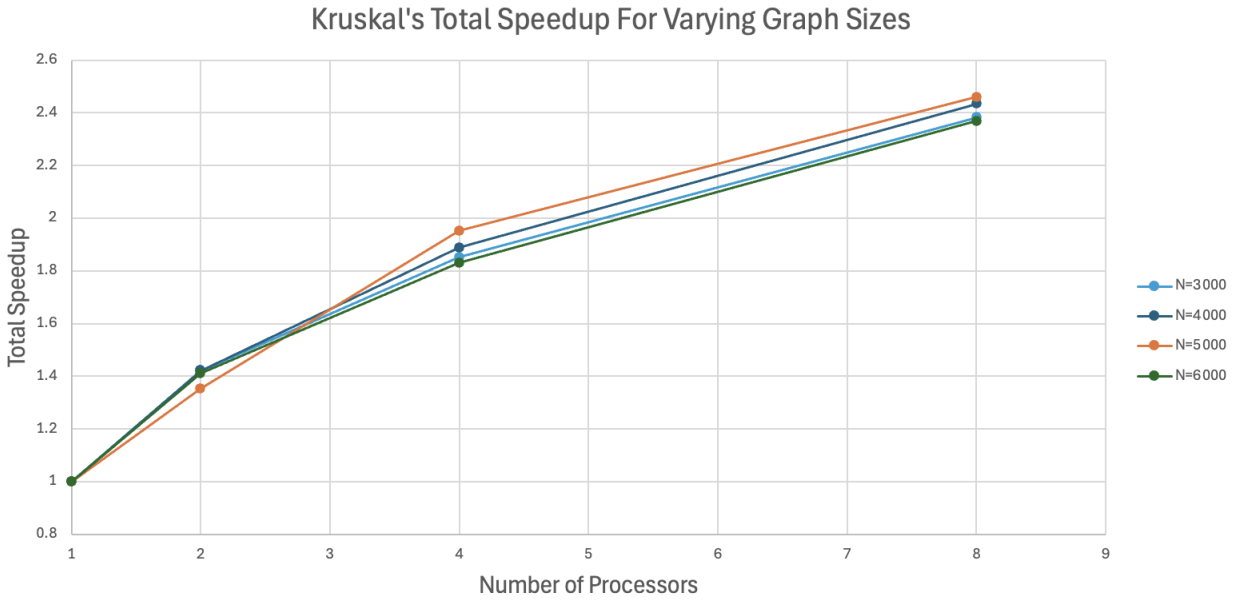

Figure 11: The effect of varying graph size on the speedup obtained from our parallel implementation of Kruskal's algorithm. Size is measured by the number of nodes, density remains constant across these benchmarks. Results for this graph were generated on GHC71.

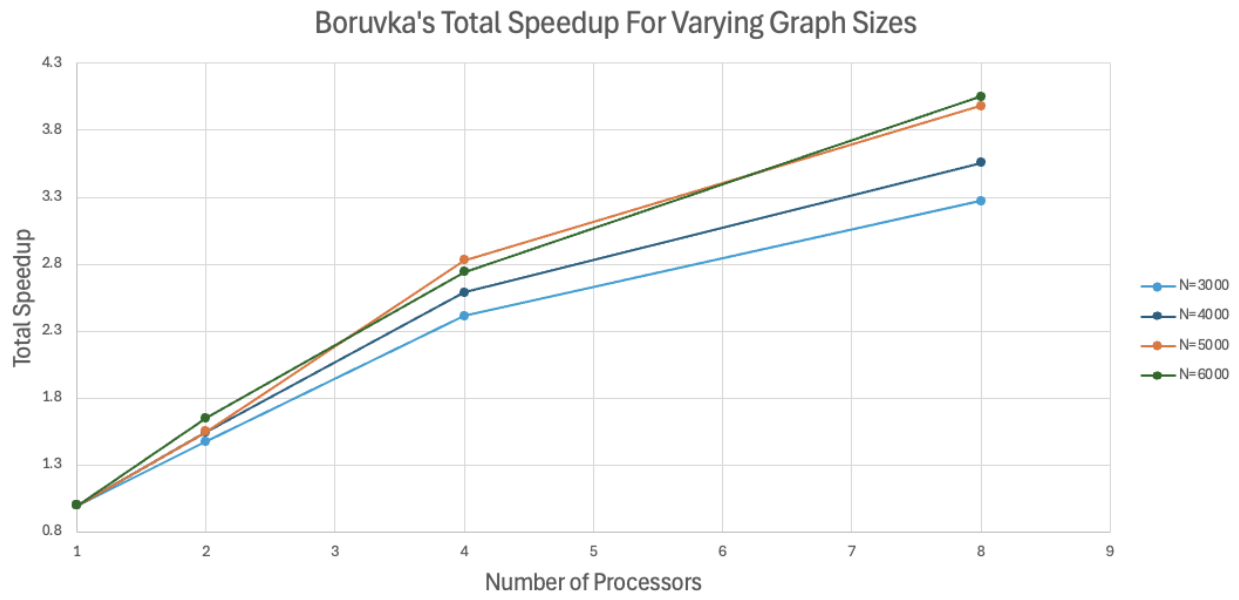

Figure 12: The effect of varying graph size on the speedup obtained from our parallel implementation of Boruvka's algorithm. Size is measured by the number of nodes, density remains constant across these benchmarks. Results for this graph were generated on GHC71.

6.3 Effect of Varying Graph Density

Next, we compare the speedup of our implementations of all three algorithms, on benchmark test cases of varying density in Figure 13 (Prim's algorithm), Figure 14 (Kruskal's algorithm), and Figure 15 (Boruvka's algorithm).

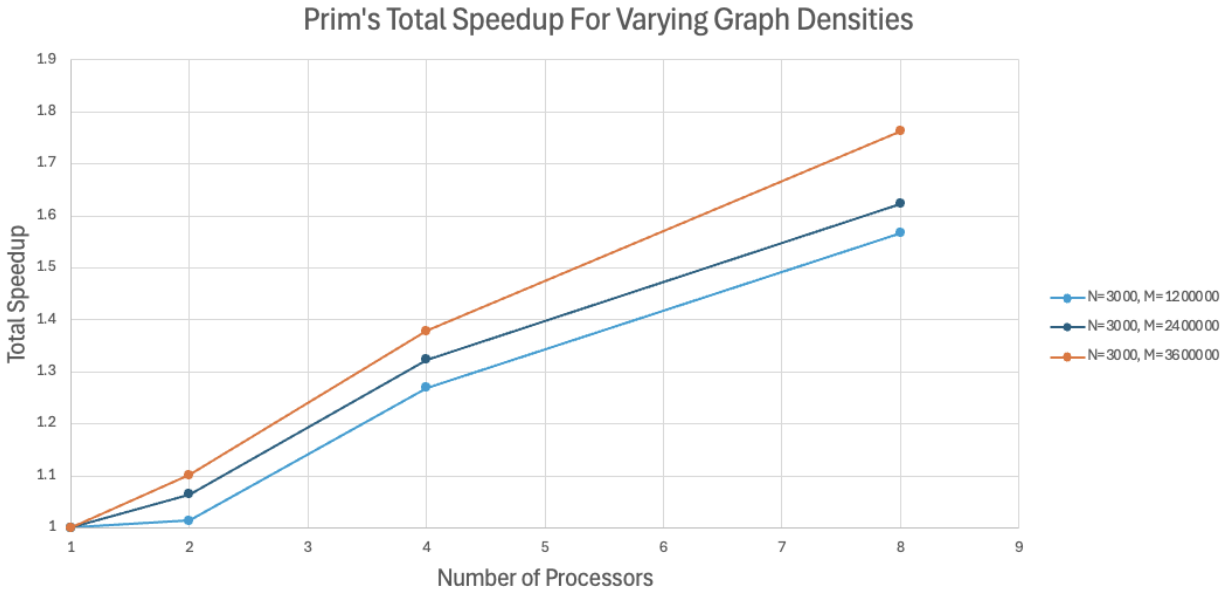

Figure 13: The effect of varying graph density on the speedup obtained from our parallel implementation of Prim's algorithm. Density is determined by the proportion of edges to vertices. Results for this graph were generated on GHC71.

For Kruskal's and Boruvka's algorithm, the speedup is relatively unchanged by changing the density of the graph.

However, there is a positive relationship between the density of the graph and the speedup achieved by Prim's algorithm. Since the parallelism in Prim's algorithm is related to the number of neighbors that a vertex has, we expect dense graphs to perform better, since each vertex likely has more neighbors. Indeed, dense graphs achieve consistently better speedup in our implementation.

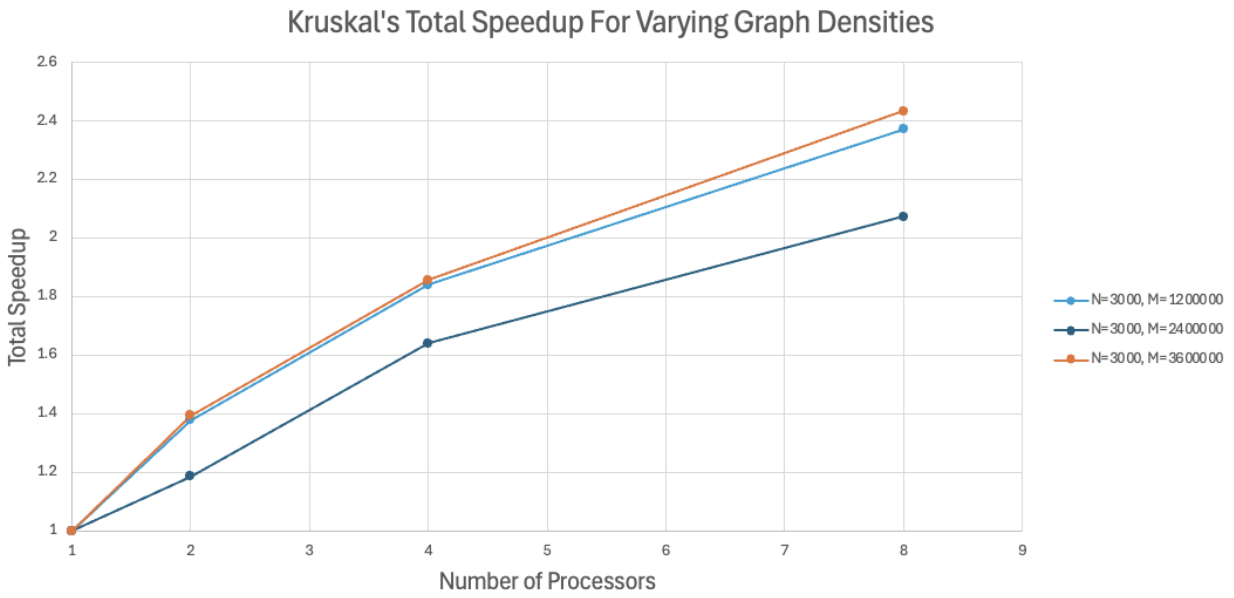

Figure 14: The effect of varying graph density on the speedup obtained from our parallel implementation of Kruskal's algorithm. Density is determined by the proportion of edges to vertices. Results for this graph were generated on GHC71.

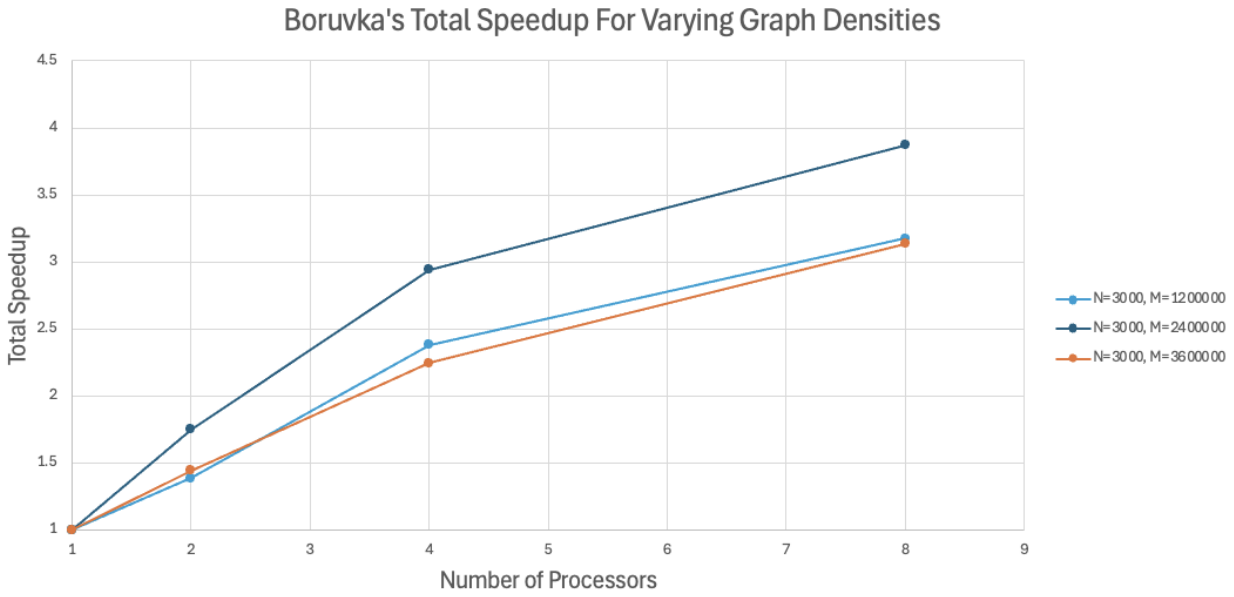

Figure 15: The effect of varying graph density on the speedup obtained from our parallel implementation of Boruvka's algorithm. Density is determined by the proportion of edges to vertices. Results for this graph were generated on GHC71.

7 Algorithm Comparison

Although Prim's algorithm and Kruskal's algorithm are often called "sequential" MST algorithms due to some sequential limitations, while Boruvka's algorithm is often called a "parallel" MST algorithm, all three algorithms have room for some amount of parallelism. We have achieved varying degrees of success exploiting different forms of parallelism in these algorithms.

Ultimately, the strategies that were most successful for parallelizing the algorithms vary.

In Prim's algorithm, we achieve the best results through parallelizing the traversal of a vertex's neighbor list, which is used to expand the current frontier in searching for the lightest edge.

In Kruskal's algorithm, we were most successful in parallelizing the sorting step–the first phase of the algorithm. We additionally saw gains in this algorithm by incorporating helper threads during the traversal of the edge list, computing extra information to aid the construction that is not done in the sequential form of the algorithm. With this algorithmic change, we were able to exceed the maximum speedup that we computed based on Amdhal's Law and the structure of the original algorithm.

In Boruvka's algorithm, we experimented with tradeoffs between parallelizing two halves of the algorithm. The first phase of finding light edges was relatively straightforward to parallelize. For the second phase—edge contraction and potentially edge elimination—we found that expensive implementations, which eliminate more edges, could be parallelized. This made more expensive contraction implementations worthwhile to do, for the added benefit of greater decreases in graph size at each iteration.

Because the three algorithms apply parallelism in different ways, they behave differently to specific graph properties. For example, because Prim's algorithm is related to the number of neighbors each vertex has, it achieves better speedup on dense graphs than on sparser graphs. Because Boruvka's algorithm is able to parallelize the extra work it takes to eliminate contracted self-loops and multiedges, it performs particularly well on larger graphs.

In conclusion, although our parallel implementation of Boruvka's achieved the best speedup overall, the implementations of Prim's algorithm and Kruskal's algorithm were also surprisingly effective. The process of developing parallel implementations of these algorithms led to interesting considerations of data dependence in MST creation, and shows that the nature of the parallelism can have a noticeable effect on the performance of different kinds of graphs.