PARALLEL IMPLEMENTATION OF A MINIMUM SPANNING TREE ALGORITHM

by

Jarim Seo, B.A., M.A.

A thesis submitted to the Graduate Council of
Texas State University in partial fulfillment
of the requirements for the degree of
Master of Science
with a Major in Computer Science
December 2020

Committee Members:

Martin Burtscher, Chair

Apan Qasem

Byron Gao

FAIR USE AND AUTHOR'S PERMISSION STATEMENT

Fair Use

Duplication Permission

ACKNOWLEDGEMENTS

The reason I decided to join the Computer Science department at Texas State University was to learn my major in more depth. I was able to not only learn from all the wonderful professors but also improve my skill as a researcher. I had a wonderful time during my two years at Texas State University. I would like to take this opportunity to thank many people who made this possible.

First, I would like to thank my advisor Martin Burtscher. He is a wonderful advisor as well as a teacher. He thought me so much about research and his passion always inspires me to work harder. I would also like to thank my thesis committee members: Apan Qasem and Byron Gao. Additionally, I would like to thank my family. Especially, my parents and my husband for always supporting me.

TABLE OF CONTENTS

Page

LIST OF TABLES

Table **Page**

LIST OF FIGURES

Figure **Page**

ABSTRACT

Building a Minimum Spanning Tree (MST) of a weighted undirected graph is an important step in various applications, including circuit design, and it is desirable to build the MST quickly. However, current well-known approaches for building MSTs are not highly parallel. In this thesis, I propose a new algorithm to build MSTs in parallel. My new algorithm can be divided into two main steps: adding the 'obvious edges' of the MST and connecting the components produced by the first step. I parallelized both steps using various atomic operations. Two serial and two parallel MST implementations were used for performance comparisons. Based on the geometric mean over 14 graphs, my MST implementation is faster than the Edge Pruning MST approach but slower than Boost. Moreover, when run in parallel with 16 threads, my code is slower than both Galois and the Problem Based Benchmark Suite implementation.

1. INTRODUCTION

The Minimum Spanning Tree (MST) of a weighted graph, as seen in Figure 1.1, is a set of edges that does not form a cycle, has all the vertices connected and has the minimum total weight.

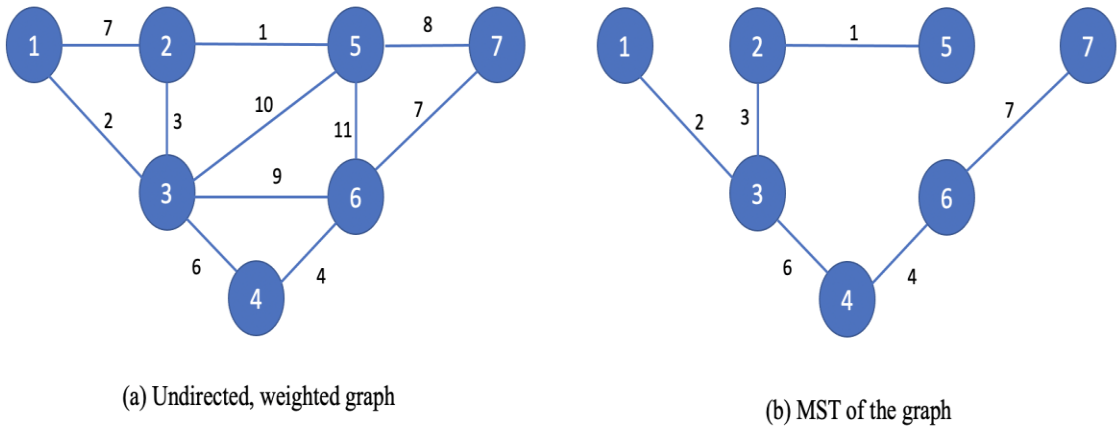


Figure 1.1: Minimum Spanning Tree of a graph

1.1 MST Usage

Finding a Minimum Spanning Tree (MST) of a weighted undirected graph is an important computation used in many domains. The most obvious usage will be network designs, such as designing roads and circuit construction [6]. The purpose of utilizing the MST for roads would be to reach every house within the city but using minimum construction in order to cut the cost. It is also used to approximate algorithms with NP-Hard problems, like the traveling salesperson problem and Steiner tree [6]. Interestingly, it is even used in cancer research to describe the arrangement of nuclei in the epithelium. In many cases, it is desirable to build the MST in a time-efficient manner [6].

1.2 Problem

Building the MST in a time-efficient manner can be achieved by parallelizing the algorithm. The most well-known serial MST algorithms are Prim's, Boruvka's, and

Kruskal's [1][4][7]. However, these algorithms only exhibit a limited amount of parallelism for much of their computation and one of the main reasons is because they are greedy algorithms [6]. In the case of Kruskal's Algorithm, it adds the least weighted edge and discards the edge if it creates a cycle with already added edges. This computation is hard to achieve in parallel because checking for cycles need the edges to be added sequentially. There have been many attempts to parallelize those popular algorithms, but parallelism in the code was limited [3]. For example, in an attempt to parallelize Boruvka's algorithm, the parallelization, in the beginning, is high due because edge-contraction can be performed independently. However, as the graph got denser, the parallelism decreases exponentially [4]. Therefore, a more effective method is required to parallelize MST algorithms.

1.3 MyMST Algorithm

For my thesis, I propose a new way of calculating the MST. I wanted to approach the graph in a new way so that the algorithm is easier to parallelize. It consists of two main steps: adding the edges that are obvious members of the MST, which are the least weighted outgoing edges of each vertex. Also, as an enhancement, the algorithm adds the edges of any neighbors with a degree of one as well as "linked-list-like" chains started by these vertices. This is because MST needs to connect all the vertices of the graph which means the only outgoing edge of a vertex needs to be part of the MST. At the end of the calculation, the vertices will not be all connected, which means the graph will have multiple connected components that need to be connected in order to make the MST. To do this, the algorithm connects the connected components with the least weighted edges to make a complete graph.

1.4 Contribution

These are the steps I took to develop and test my algorithm.

1. Develop the algorithm with high parallelization in mind.
2. Parallelize the algorithm using OpenMP.
3. Run my code using a different number of threads.
4. Compared the results with both existing serial and parallel codes such as Boost, Galois, and PBBS benchmark.
5. Analyzed my algorithm further for future improvements.

1.5 Outline

In the following sections, I will discuss the following: in Sections 2 and 3, I will discuss the classical MST algorithms and attempt to parallelize them. In Section 3, I will introduce my MST algorithm and my attempts to parallelize it, following Section 4 will explain the evaluation methods, Section 6 will show the test results, and finally, Section 7 will conclude with a summary and plans for future work.

2. BACKGROUND

The Minimum Spanning Tree (MST) of a weighted graph has some unique characteristics and several classic algorithms compute it [5].

2.1 Characteristics of MST

Several characteristics define the MST. First, the MST has n-1 edges where n is the number of vertices in a graph. Secondly, if the weight of all the edges in the graph is unique, the MST will be unique as well. However, if there are multiple edges with the same weight, a graph can have multiple MSTs with the same weight, as seen in Figure 2.1. Finally, if an edge e is a unique, least weighted edge in the graph, it has to be part of the MST [6].

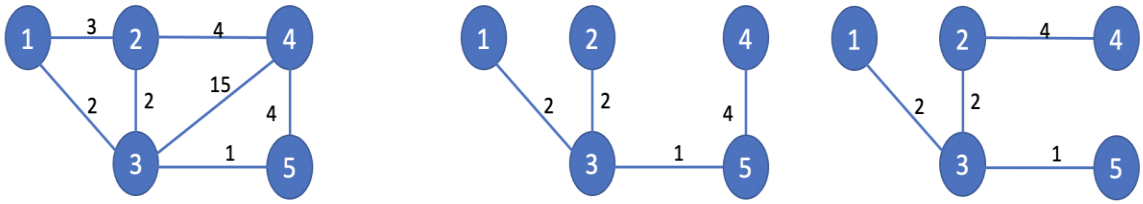


Figure 2.1: Multiple MSTs for a single graph

2.2 Classic MST Algorithms

There are three main classical MST algorithms, which are all greedy algorithms. It means that they pick the best answer and this has to be done sequentially. In the case of calculating the MST, these algorithms need to consider all the edges one by one. The algorithms are Kruskal's, Boruvka's, and Prim's Algorithm [1][4][7].

First, Kruskal's algorithm is an algorithm that looks for an edge with the least possible weight, which means the algorithm orders all the edges according to their weight. Before adding the least weighted edge from the ordered list to the MST, the

algorithm checks for any cycles with the formed spanning tree so far. If it does not form a cycle, the edge is added to the MST, otherwise, the edge is discarded and will not be a part of the MST. The algorithm runs until there are n – 1 edges in the MST where n is the number of vertices in the graph. The time complexity of this algorithm is O(E log V) where E represents the number of edges and V represents the number of vertices [3].

Second, is Boruvka's Algorithm which is published in 1926 [1]. This algorithm looks at each vertex. Initially, the MST is empty and there is a list of the graph's vertices. The algorithm looks at the first vertex and finds the least weighted edge that connects it to some other vertex. The cheapest edge and the connected vertex are added. The algorithm moves on to the newly added vertex and carries on adding the least weighted edge from the current vertex until all the vertices are in the MST. Boruvka's Algorithm also has a time complexity of O(E log V). However, the algorithm can have linear complexity if all the edges are removed except the least weighted edges between vertices after each iteration [1].

Lastly, Prim's Algorithm, similar to Boruvka's, looks at each vertex. It first starts by creating a set that keeps track of which vertices are part of the MST and it assigns the value of infinity to all the vertices except the starting vertex x, which will be assigned a value of 0. To start the algorithm, pick the vertex that has the least value and includes it in the set that keeps track of which vertices are in the MST. Then, it checks any adjacent vertices and compares their value and the weight of the edge connecting two vertices. If the value is less, the value of the adjacent vertex is updated and the least weighted edge is added as part of the MST. These steps run in a loop until all the vertices are part of the MST. The complexity of Prim's Algorithm depends on what data structure is used to store

the least weighted edge. Using adjacency matrix gives the time complexity of O(V2), binary heap gives complexity of O(E log V), and Fibonacci heap gives O(E + V log V) [7].

3. RELATED WORK

In the previous section, I introduced the three most well-known MST algorithms. In this section, I will introduce some MST algorithms that were developed based on the three classic algorithms. I will also introduce various researches that have attempted to parallelize MST algorithms.

3.1 Serial MSTs

One of the algorithms based on Boruvka's is developed by Cheriton and Tarjan [8]. The algorithm initially treats each vertex like a connected component and each connected component maintains a normal queue for vertices and a priority queue for the edges. In each iteration, one vertex is popped from the queue and the least weighted edge is retrieved from the priority queue. The algorithm keeps retrieving the least weighted edge until it finds an edge that connects the vertex in the current connected component to another. When it is found, the edge is recorded as part of the MST and the two connected components' queues are merged. The steps are repeated until there is one connected component.

Some researchers tried to improve Kruskal's Algorithm by including demand sorting [9]. Because Kruskal's Algorithm has sorting of the edges, it makes the algorithm not as efficient. Instead, the authors use a priority queue to sort the edges.

In my experiments, I compared my algorithm to Boost Graph library, based on Prim’s Algorithm, and Edge Pruned Minimum Spanning Tree (EPMST) proposed by Maum and Abdullah-Al [10][11]. EPMST is an algorithm based on both Kruskal’s and Prim’s Algorithm.

3.2 Parallel MSTs

In addition to developing serial algorithms, many researchers parallelized classic algorithms.

First, as mentioned before, Kruskal's Algorithm initially sorts the edges and picks the least weighted edge one step at a time. The initial attempt was to parallelize this algorithm was just parallelizing the sorting and sequentially add those sorted edges. A similar but more complex attempt was called Filter-Kruskal [7]. This is partitioning the edges similar to quicksort and filtering out the 'obvious' edges that are not part of MST. This was more suited for parallelization because the partitioned edges can be distributed among threads. An approach by Katsigiannis and Anastasios introduces "helper threads" [3]. The authors took the idea that if an edge that will be examined in the future might create a cycle within the current MST, that edge can be exempt from future examination. So, in their new algorithm, they run the regular Kruskal's algorithm and have several helper threads examine if edges with higher weights create cycles in the current MST. Whenever cycles are detected, those edges are marked and discarded. One problem with this approach is that it potentially performs a lot of work that is not used as the main thread usually does not need to visit all edges.

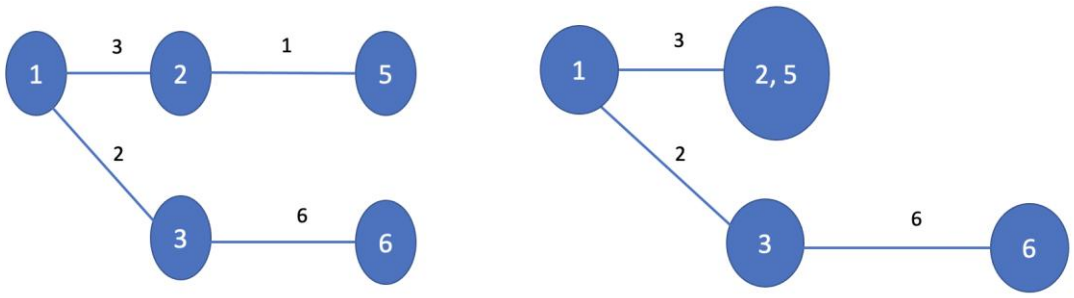


Figure 3.1: Edge contraction

Next, Boruvka's algorithm is parallelized by introducing edge-contraction by ISS Group at the University of Texas [4]. This operation is where an edge is chosen, and a new vertex is formed by carrying out a union of the connectivity among the two end vertices of the chosen edge, which can be seen in Figure 3.1. Vertex 2 and vertex 5 are contracted together and the contracted vertices are pushed into a worklist to be contracted again. However, there are two problems with this parallel version. First is that the edge-contraction operation is costly to implement. Second is that, while there is high parallelism in the beginning because edge-contraction can be performed independently, the parallelism decreases exponentially as the graph becomes denser.

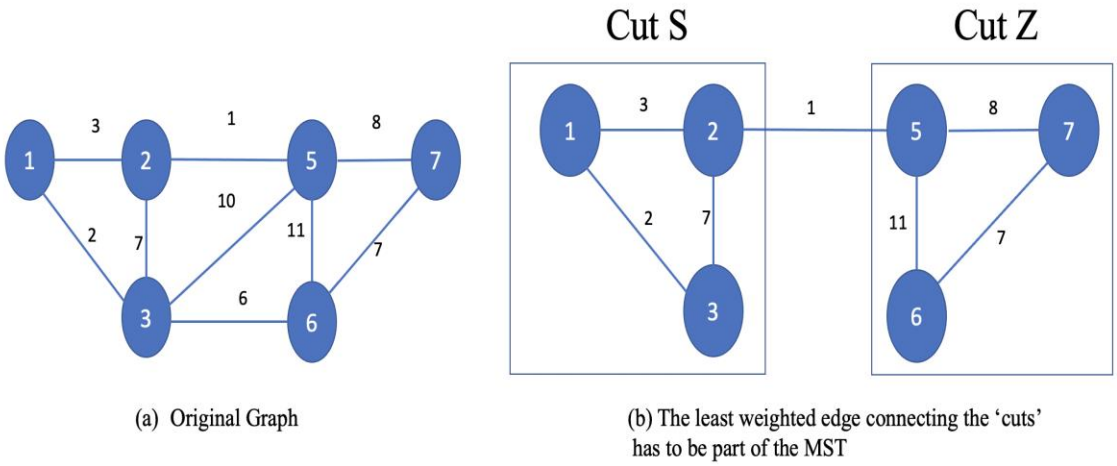


Figure 3.2: Cut-Property

Lastly, to parallelize Prim's algorithm, the cut-property is used. Cut-property is if a graph is cut into subgraphs the least weighted edge that connects those two subgraphs has to be part of the MST, which can be seen in Figure 3.2 [6]. Cut C and Cut Z are connected by three different edges and the edge with the least weight that connects these two cuts has to be part of the MST. The algorithm uses this property in order to split up the work among the cuts and find the minimum edge to connect these cuts which will

result in the MST [7]. It assigns each thread a portion and waits for all the other threads to finish their work. Like in the other attempts, only a relatively small part of the algorithm is parallelized, and the parallelism is initially very small.

For my experiment, I compared my parallel algorithm with Lonestar and Problem Based Benchmark Suite (PBBS) [5][12]. Lonestar is built on Galois runtime and the MST algorithm is based on Boruvka's Algorithm. PBBS includes a parallelized MST algorithm based on Filtered Kruskal Algorithm.

4. APPROACH

Because the classic MST algorithms are difficult to parallelize, I want to develop a new algorithm that could be easily parallelized. In this section, I will explain my new MST algorithm and how it is parallelized using OpenMP.

4.1 MyMST Algorithm

My algorithm consists of two main separate loops: adding the edges that are obvious members of the MST and connecting connected components that were produced from the first part of the algorithm.

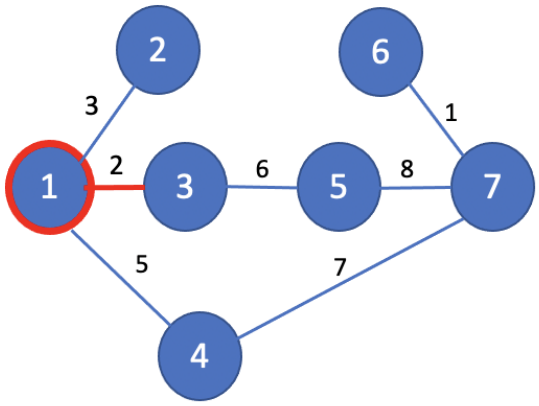


Figure 4.1: Adding the least weighted edge

The first loop is adding the edges that are obvious members of MST, which are two types of edges. The members consist of the least weighted edge of each node and the edges of nodes that have degree one. Before starting the computation, the algorithm assumes that all the nodes from the graph are independent components. In the loop, the algorithm iterates over all the nodes looks at all the outgoing edges the node has to find the least weighted edge and makes it part of the MST. This is because the least weighted edges of each node must be a member of the MST as seen in Figure 4.1.

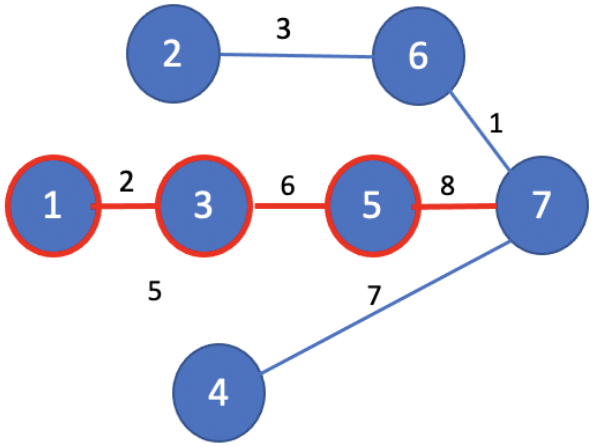
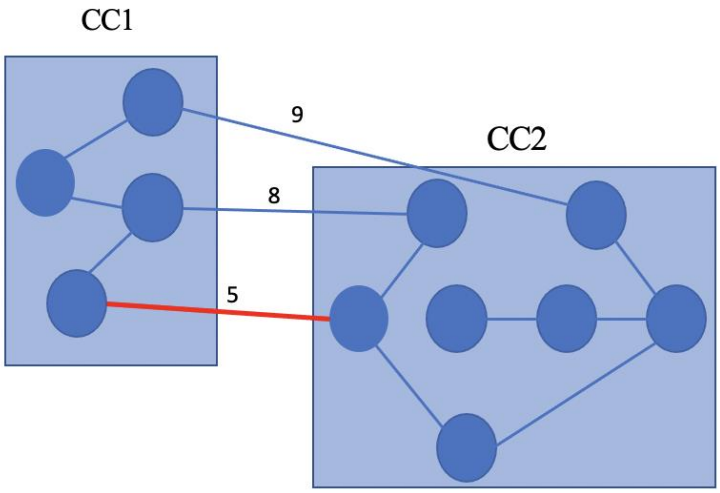
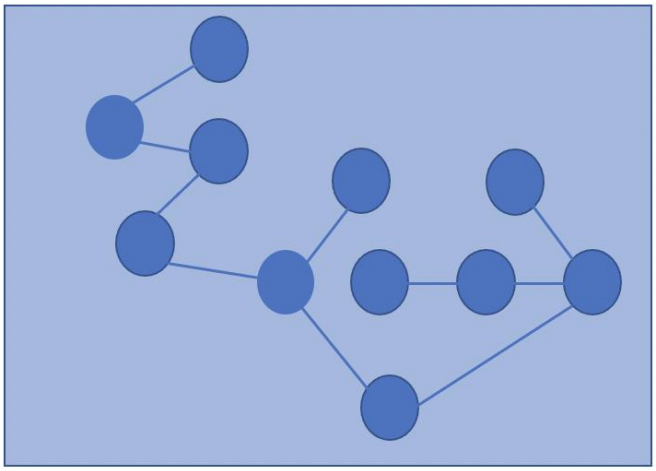


Figure 4.2: Adding the edges of neighbor nodes with a degree of one

Then, in the same loop, the algorithm iterates over the neighboring node and calculates the degree of the node. If the node has a degree of one, that one edge must be part of the MST. The loop continues to iterate over the neighbor nodes of the previous node that had a degree of one. If the subsequent neighbor has a degree of two, the edge is added to the MST and the loop continues until it reaches a node that has a degree higher than two as seen in Figure 4.2. Because vertex 7 has a degree of 3, only vertices 3 and 5 which have a degree of 2 are added to the MST. Because the second loop of the algorithm is where most of the execution time is, adding these edges will help to speed up the algorithm.



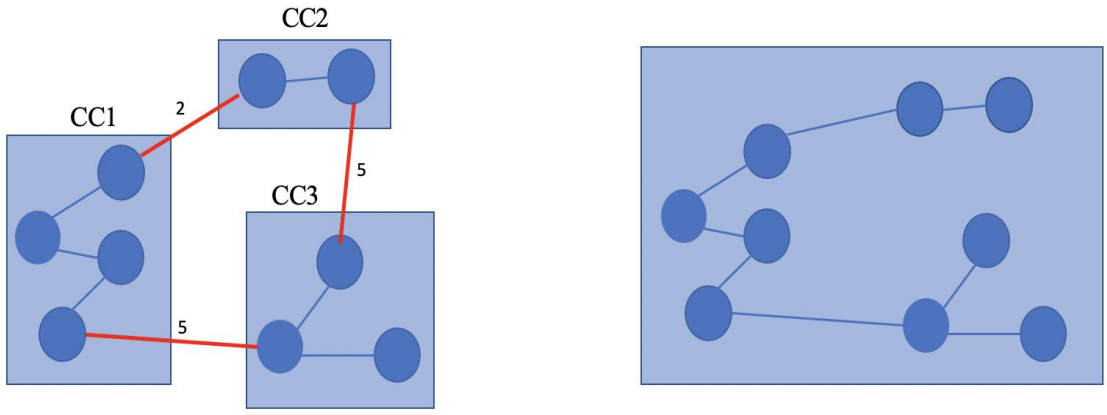
(a) Picking the least weighted edge from CC1



(b) Becomes one connected component

Figure 4.3: Combining two connected components

After processing the first loop, there will be multiple connected components, which can be seen in Figure 4.3. In the second loop, the algorithm iterates over each connected component to find the least weighted outgoing edge that is not already part of the MST. When looking for the least weighted edge, the algorithm checks the root of the current vertex and the neighbor vertex to make sure two vertices are part of different connected components. Then, the least weighted edge is included in the MST, turning two connected components into one.



(a) Least weighted edges from each CC

(b) Becomes one component by discarding one of the edge

Figure 4.4: Discarding one of the least weighted edges

During this process, the algorithm does not blindly add the least weighted edge, but checks to see if two components are still different as seen in Figure 4.4. Connected component 1 and 2 are connected by an edge weighted 2 and connected component 1 and 3 are connected by an edge weighted 5. The edge weighing 5 that connects connected component 3 and 2 can be discarded since they are already in the same component. The algorithm repeats this step until there is only one connected component left. The result will be the minimum spanning tree of the graph.

4.2 Parallelization

The motivation for devising a new MST algorithm was because the classical MST algorithms are very difficult to parallelize. In order to address this problem, I modified the algorithm so that it can be more easily parallelized. To parallelize my algorithm, I am using OpenMP. I have two main loops: the first loop, which is adding the least weighted edges and edges of degree one, is parallelized altogether and in the second loop, I have two inner loops, which are parallelized separately.

In the first loop, I use an atomic decrement whenever the number of connected

components is decreasing. This is because as the threads are adding the edges, the vertices are combined into one connected component. The number of connected components decreases whenever an edge is added to the MST. Because keeping track of which edge belongs to the MST is write-only, it did not need to be atomically modified.

Before starting the second loop, I parallelize two loops to get the list of the roots of the connected components and a list of remaining edges that were not added to MST yet. In both of the loops, I used atomic capture to keep track of the list size. I use atomic capture to ensure that different threads do not insert an element at the same time.

In the second loop, I parallelize the two parts separately. In the first inner loop, I go over all the edges that are not part of MST and find the least weighted edge of each connected component. To achieve parallelization, I use an atomic 'compare and swap'(CAS) to find the least weighted edge of each connected component. Because this process to find the least weighted edge can cause data races, I used an atomicMin operation implemented using CAS. By using CAS, it ensures that only one thread is comparing the value and possibly swapping the value if the edge weight is less than what it previously had. Also, atomic capture is used to keep track of the size of edges remaining. The size of edges remaining will be used in the next iteration of the outer loop and the use of atomic capture will make sure that only one thread is increasing the number of remaining edges. In the second inner loop, I iterate over each connected component and connect them using the least weighted edge found from the first part of the loop.

To parallelize this section, I use an atomic decrement in order to keep the number of connected components valid and, within the same loop, keep track of the root of new

connected components to be used in the next iteration of the outer loop. I use atomic capture to atomically decrement and capture the value before the decrement. This allows me to know where to insert the element in the lists of roots and edges.

In order to make the algorithm more efficient, I keep two copies of the lists: the list of roots of the connected components and the list of edges that are not yet part of the MST. As the algorithm was going through the two inner loops of the second loop, the second lists are being filled for the next iteration while the first lists are being drained. At the end of the loop, the algorithm swaps the newly filled lists with the lists that are being drained the calculation. This is faster because list swapping is just swapping two pointers whereas the alternative would require copying the contents of one list over into the other, which would be much slower.

5. METHODOLOGY

For my thesis, I compared my code with the serial and parallel CPU codes from the literature that are listed in Table 5.1. I converted the input graphs for each algorithm in advance and only measured the time each code took to compute the MST. I ran the experiment three times and report the best runtime.

Table 5.1: Evaluated codes

Name	Version	Source	serial/parallel
My MST	1.0	my code	parallel
Boost	1.74	[10]	serial
EPMST	1.0	[11]	serial
Galois	3.0	[5]	parallel
PBBS	2020	[12]	parallel

To test with the different number of threads, I tested on two systems. The first system I use for the measurements is based on an AMD Ryzen Threadripper 2950X CPU with 16 cores. Hyper-threading is enabled, i.e., the cores can simultaneously run 32 threads. The main memory has a capacity of 64 GB. The operating system is Fedora 29. The second system I used is based on two Intel Xeon E-5-2687W CPU with 20 cores. Hyper-threading is enabled, i.e., the cores can simultaneously run 40 threads. The main memory has a capacity of 128GB. I will compile the codes with gcc/g++ 8.3.1 using the "-O3 -march=native" optimization flags. To run MyMST using OpenMP, I used the flag "-fopenmp".

The codes are evaluated on the 14 graphs, listed in Table 5.2, which are road-map, random, and grid graphs. Road-map graphs have long diameters and random and grid graphs are balanced graphs. They all have a weighted edge. They are obtained from the Center for Discrete Mathematics and Theoretical Computer Science at the University of Rome (Dismacs) [13] and the Galois framework [5]. The graphs are in CSR format.

Table 5.2: List of evaluated graphs

Name	**Type**	**Origin**	**vertices**	**edges**	**d_{mix}**	**d_{max}**	**d_{avg}**
2d-2e20.sym.egr	grid	Galois	1,048,576	4,190,208	2	4	3.8
r4-2e23.sym.egr	random	Galois	8,388,608	67,108,846	2	26	7.9
USA-road-d.BAY.egr	road map	Dimacs	321,270	794,830	1	7	2.4
USA-road-d.CAL.egr	road map	Dimacs	1,890,815	4,630,444	1	7	2.4
USA-road-d.COL.egr	road map	Dimacs	435,666	1,042,400	1	8	2.3
USA-road-d.CTR.egr	road map	Dimacs	14,081,816	33,866,826	1	8	2.4
USA-road-d.E.egr	road map	Dimacs	3,598,623	8,708,058	1	9	2.4
USA-road-d.FLA.egr	road map	Dimacs	1,070,376	2,687,902	1	8	2.5
USA-road-d.LKS.egr	road map	Dimacs	2,758,119	6,794,808	1	8	2.5
USA-road-d.NE.egr	road map	Dimacs	1,524,453	3,868,020	1	9	2.5
USA-road-d.NW.egr	road map	Dimacs	1,207,945	2,820,774	1	9	2.3
USA-road-d.NY.egr	road map	Dimacs	264,346	730,100	1	8	2.8
USA-road-d.USA.egr	road map	Dimacs	23,947,347	57,708,624	1	9	2.4
USA-road-d.W.egr	road map	Dimacs	6,262,104	15,119,284	1	9	2.4

6. RESULTS

In this section, I compared MyMST with two serial implementations, Boost based on Prim and Edge Pruning MST [10][11]. As for parallel comparison, I used Galois based on Bourvka and the Problem Based Benchmark Suite based on Filter Kruskal [5][12].

6.1 Serial Runtime and Throughput

First, I ran the MyMST code with a single thread, Boost, and Edge Pruning MST (EPMST).

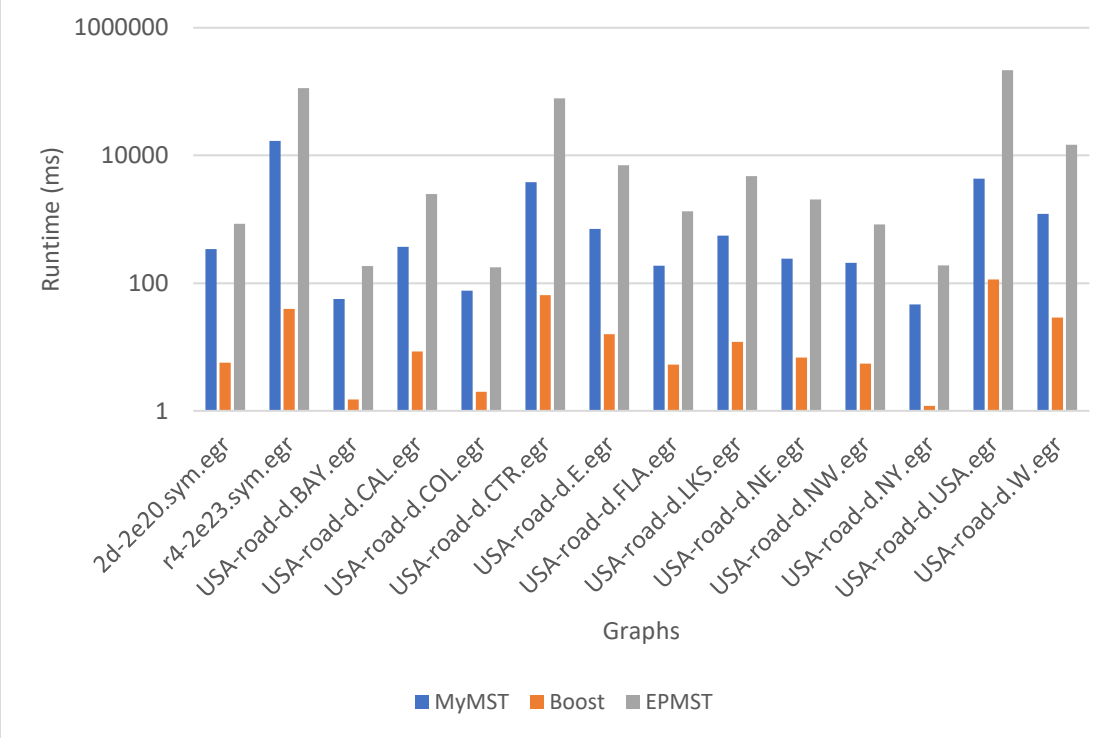


Figure 6.1: Serial runtime

As seen in Figure 6.1, there are drastic differences in runtime. Boost had the lowest runtime throughout all the graphs and EPMST had the slowest runtime overall. All the codes ran fastest on USA-road-d.NY.egr due to the size of the graph. On Boost and EPMST, they both had the highest runtime on USA-road-d.USA.egr, which was the biggest graph in terms of the number of vertices in the graph. However, on My MST,

r4-2e3.sym.egr has the longest runtime, because the graph had the highest degree and the most edges out of all the graphs.

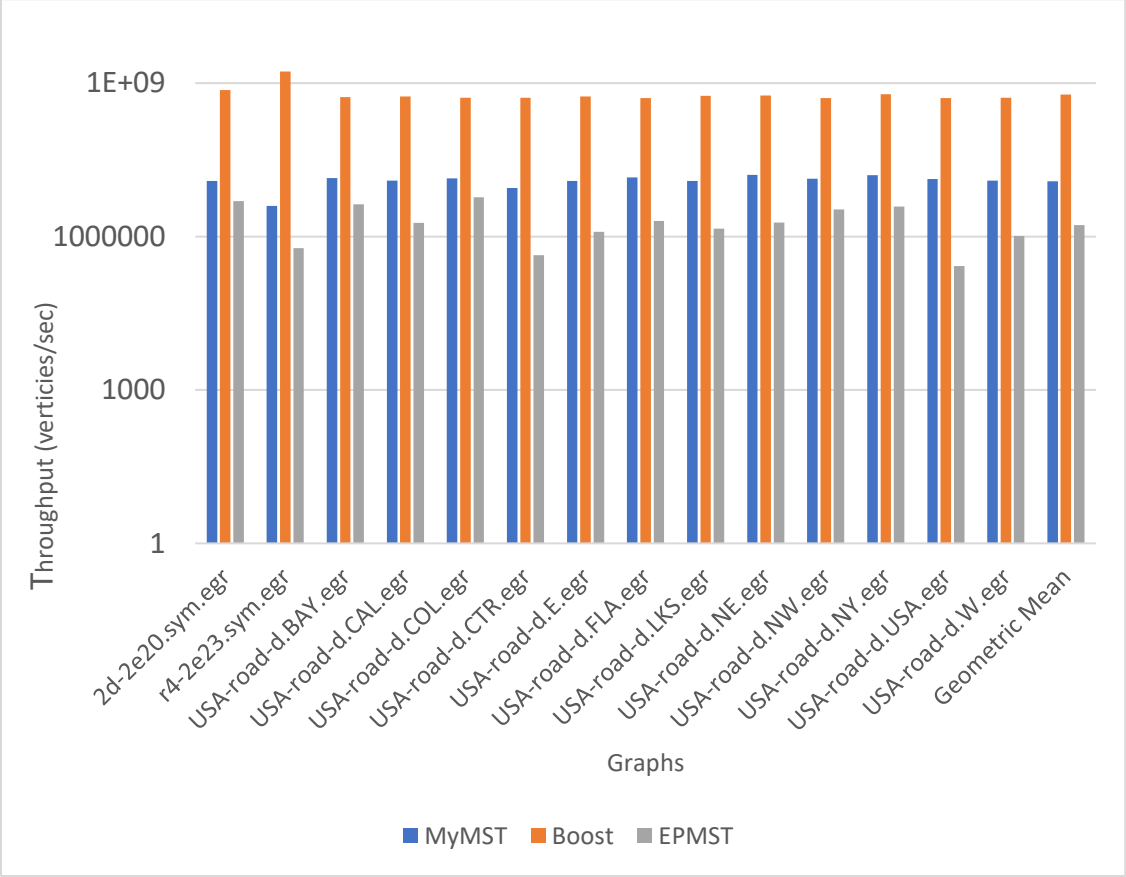
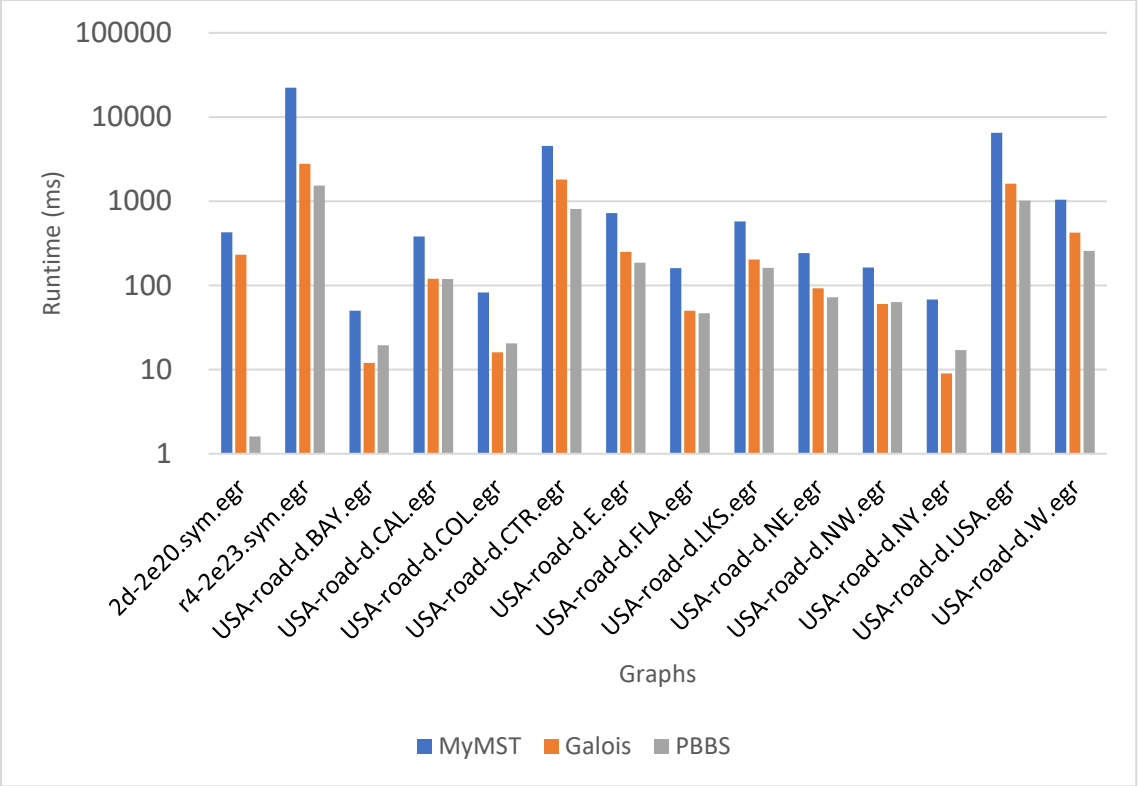


Figure 6.2: Serial throughput

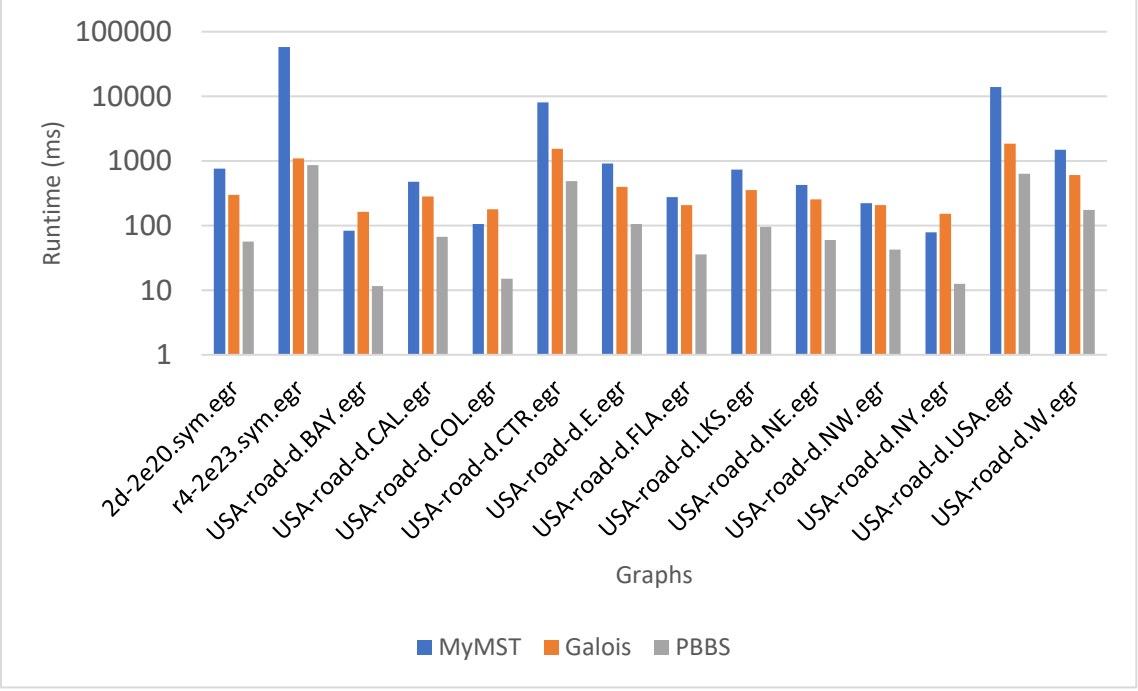
Figure 6.2 shows the throughput, which is the number of vertices divided by the runtime. Throughput generally shows the rate at which something is processed and higher throughput means the algorithm is more efficient. Similar to the runtime, Boost had the best throughput and My MST had the second-best and EPMST had the worst throughput. The geometric mean, which is the type of average usually used for growth rate, shows a similar result. In terms of the geometric mean of throughput, MyMST compared to EPMST could process about 7 times more edges when MyMST was running serially.

6.2 Parallel Runtime and Throughput

In order to test the parallel codes, I decided to use two different machines. One machine had 20 cores and the other had 16 cores. In this section, I will show both of the results and compare them.



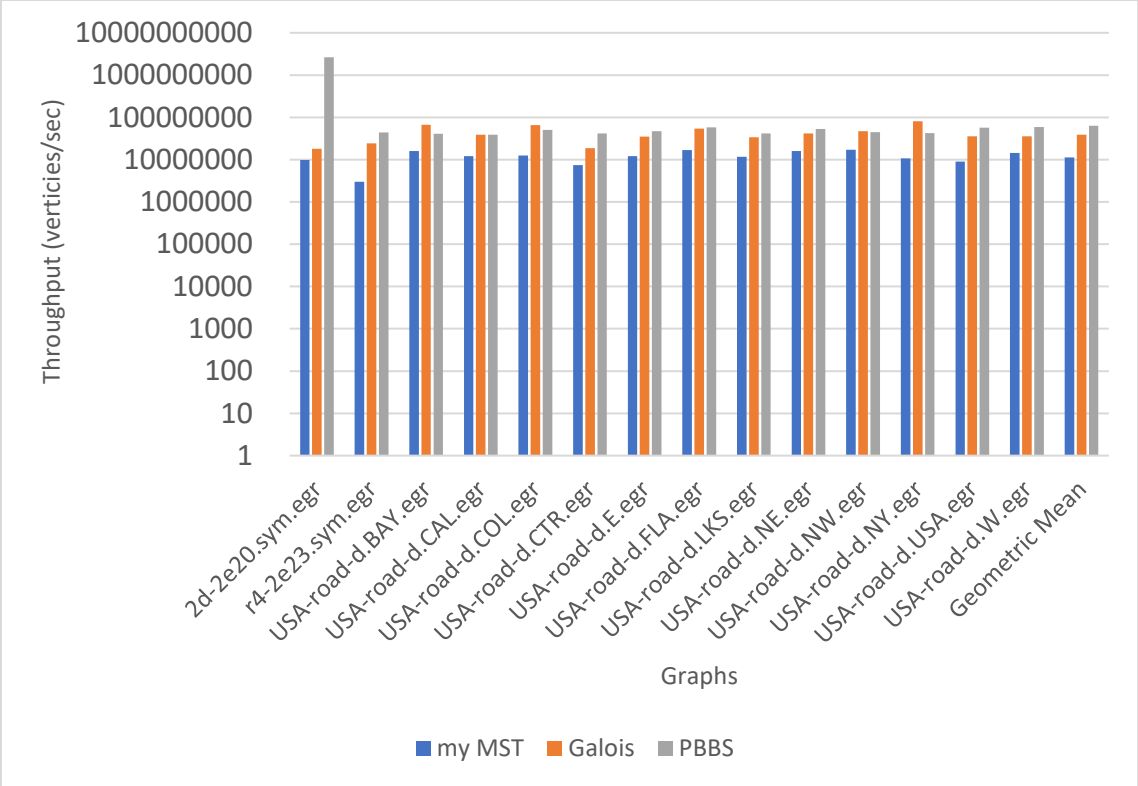
(a) Parallel runtime with 16 threads on the 16-core machine



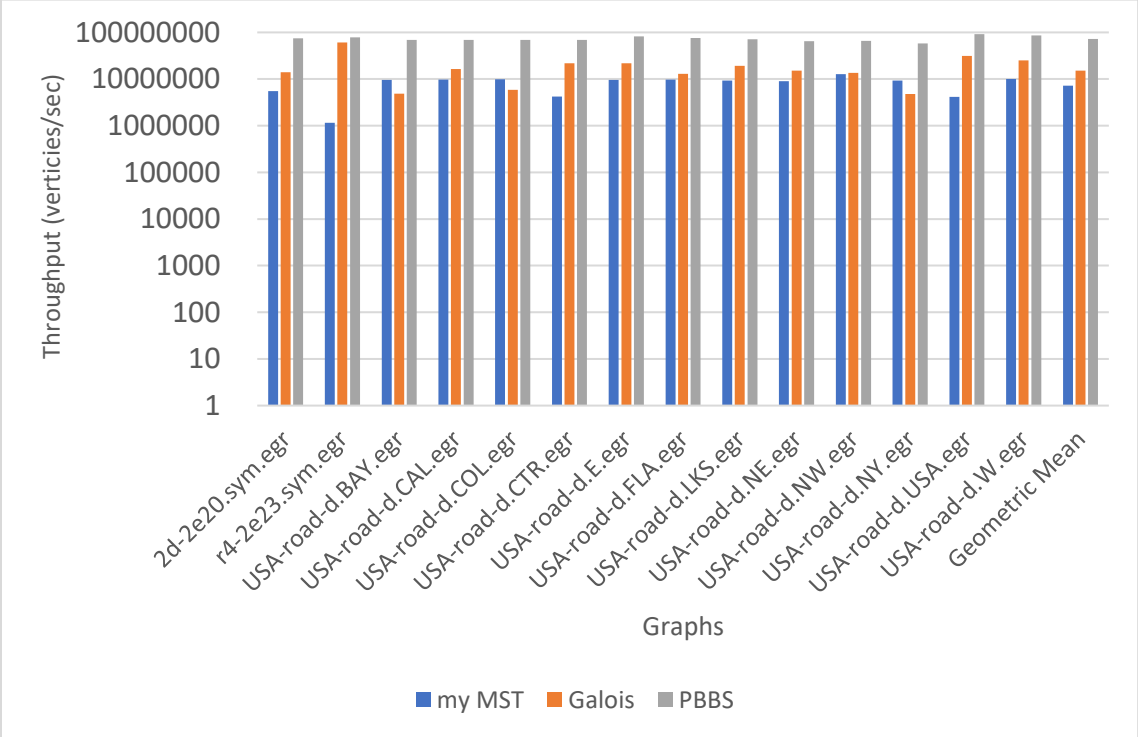
(b) Parallel runtime with 20 threads on the 20-core machine

Figure 6.3: Parallel runtime

Overall, all the codes performed better on the machine with 16 cores and PBBS had the best runtime, then Galois, My MST was slowest, as shown in Figure 6.3. PBBS ran very fast even on the two biggest two graphs, r4-2e23.sym.egr and USA-road-d.USA.egr, whereas My MST and Galois had the slowest runtimes.



(a) Parallel throughput with 16 threads on the 16-core machine



(b) Parallel runtime with 20 threads on the 20-core machine

Figure 6.4: Parallel throughput

Figure 6.4 shows the throughput of each run and it is calculated by dividing the number of vertices by runtime. By looking at the geometric mean, PBBS had the best throughout. My MST did have comparable throughput to Galois on some of the road graphs such as USA-road-d.NW.egr. In terms of geometric mean, Galois could process about 2 times more edges than MyMST, whereas PBBS could process about 10 times more edges than MyMST. Even though My MST is highly parallel, it was slower than the implementations that were based on greedy algorithms, and in the next section, I will present some ways to possibly improve my code.

6.3 Analysis of MyMST

MyMST consist of two main steps: adding the 'obvious edges' of the MST and connecting the components created from the first step. Both steps were parallelized using various atomic operations. However, the code was not as fast as I hoped, and performed two analyses in order to find out why.

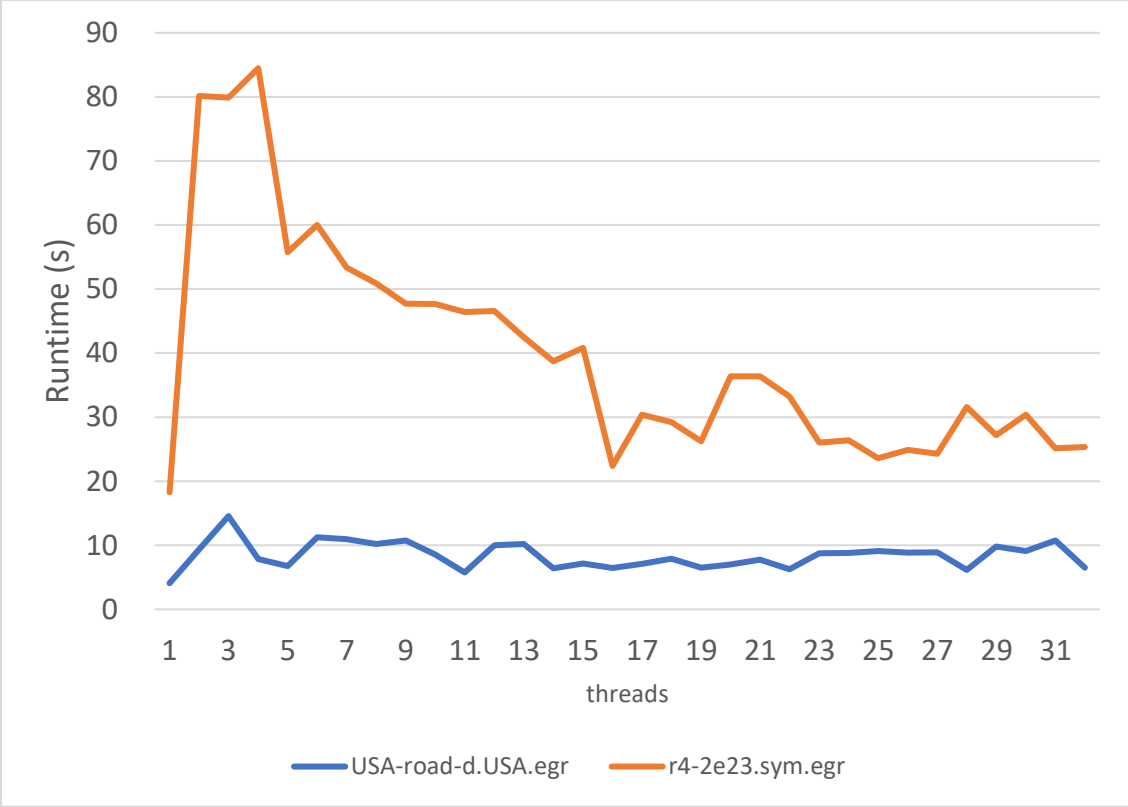


Figure 6.5: Scalability of MyMST

First, I measured the scalability of MyMST. I ran the code using 1 to 32 threads on the machine that has 16 cores and enabled hyperthreading. I ran it on the two largest graphs which are USA-road-d.USA.egr and r4-2e23.sym.egr. As seen in Figure 6.5, the code is not scaling well. The runtime increased drastically when using just a few threads. As the number of threads increased, it did show similar runtime as the single thread. This means that the threads are slowing down the code rather than effectively distributing the work among them.

Table 6.1: Serial analysis of MyMST

	First Step		Second Step	
Graph	time	iterations	time	iterations
2d-2e20.sym.egr	0.0314	1048576	0.4322	10
r4-2e23.sym.egr	0.5355	8388608	15.1246	8
USA-road-d.BAY.egr	0.0059	321270	0.0371	9
USA-road-d.CAL.egr	0.0398	1890815	0.2358	10
USA-road-d.COL.egr	0.0084	435666	0.0485	8
USA-road-d.CTR.egr	0.5343	14081816	2.6029	11
USA-road-d.E.egr	0.0787	3598623	0.4550	9
USA-road-d.FLA.egr	0.0204	1070376	0.1313	9
USA-road-d.LKS.egr	0.0602	2758119	0.3625	10
USA-road-d.NE.egr	0.0310	1524453	0.1994	9
USA-road-d.NW.egr	0.0280	1207945	0.1723	9
USA-road-d.NY.egr	0.0050	264346	0.0403	8
USA-road-d.USA.egr	0.6639	23947347	3.5007	11
USA-road-d.W.egr	0.1698	6262104	0.9937	10

Second, I analyzed My MST serially and in parallel. First, seen in Table 6.1, I measured the serial runtime and how many iterations the two steps took. In this table, one iteration for the first loop is the processing of one vertex to find the least weighted edge and adding it to the MST and adding the edge of neighbors with a degree of 1. On the second loop, one iteration is finding the least weighted edge that connects two different connected components and connecting those components. The first step was where most of the edges belonging to the MST were added and the second step did not go through many iterations but took most of the computation time. For example, on 2d-2e20.sym.egr, about 93.2% of the computation time was spent in the second step, but it only went through 10 iterations whereas 6.8% of the runtime went through 1,048,576 iterations.

Table 6.2: Parallel analysis of MyMST with 16 threads

	First Step		Second Step	
Graph	time	iterations	time	iterations
2d-2e20.sym.egr	0.0146	1048576	0.4711	10
r4-2e23.sym.egr	0.1185	8388608	29.7089	8
USA-road-d.BAY.egr	0.0046	321270	0.0412	9
USA-road-d.CAL.egr	0.0209	1890815	0.3182	10
USA-road-d.COL.egr	0.0059	435666	0.0583	8
USA-road-d.CTR.egr	0.1631	14081816	3.4180	11
USA-road-d.E.egr	0.0375	3598623	0.5980	9
USA-road-d.FLA.egr	0.0124	1070376	0.1480	9
USA-road-d.LKS.egr	0.0298	2758119	0.5077	10
USA-road-d.NE.egr	0.0182	1524453	0.2875	9
USA-road-d.NW.egr	0.0146	1207945	0.1888	9
USA-road-d.NY.egr	0.0036	264346	0.0608	8
USA-road-d.USA.egr	0.2330	23947347	3.9368	11
USA-road-d.W.egr	0.0614	6262104	1.0534	10

I also ran the same analysis in parallel with 16 threads. As seen in Table 6.2, the overall runtime of the first loop did improve, whereas the runtime of the second loop was significantly longer. For example, in graph r4-2e23.sym.egr, there was about 77.9% decrease in runtime for the first loop. However, on the second loop, the runtime increased by about 49%. This means that the first loop does scale as the number of the threads increase, but the second loop was not parallelized very well and it made the runtime worse.

These two analyses show two important facts about MyMST. MyMST does not exhibit very good scalability when the number of threads increases and the operation of connecting components is very expensive.

7. SUMMARY

Computing the Minimum Spanning Tree of a graph is an important computation because it is used in different fields such as circuit design and cancer research. It is desirable to build the MST in a timely manner, but the three classical MST algorithms are difficult to parallelize. In this thesis, I propose a new MST Algorithm that is highly parallelizable.

The algorithm, MyMST, consists of two main steps. The first step is to add the 'obvious edges' of the MST. The 'obvious edges' are the least weighted edge of each vertex and the edges of the neighbors with a degree of 1. After the first step is computed, there are multiple connected components. The second step is to connect these components with the least weighted edges. The second step repeats until there is only one connected component left which will be the MST of the graph. I parallelized the two steps separately. I parallelized the first step as a whole and the second step was divided into two sections which are finding the least weighted edge that connects two different components and connecting those components.

To test MyMST, I found two serial and two parallel implementations. The two serial implementations are Boost, based on Prim's Algorithm, and Edge Pruning MST. The two parallel implementations are Galois, based on Bourvka's Algorithm, and the Problem Based Benchmark Suite, based on Filter Kruskal. I used 14 graphs to test my code and they consist of three types of graphs: random, grid, and road map. The biggest graph was r4-2e23.sym.egr with 67,108,846 edges. For the serial comparison, I ran my code with a single thread. Based on the geometric mean over the graphs, MyMST was faster than the Edge Pruning MST approach but slower than Boost. Moreover, when run

in parallel with 16 threads, my code is slower than both Galois and the Problem Based Benchmark Suite (PBBS) implementation. I measured throughputs of each, which were vertices divided by runtime. Galois could process about 2.1 times more vertices compared to MyMST and PBBS could process about 10 times more edges than MyMST.

To better understand why MyMST was slower than the two parallel implementations even though it is highly parallelizable, I ran two additional analyses. I observed that MyMST was not scalable in terms of the number of threads and the second part of the code took more than 90% of the computation time. Also, with an increase of threads, the first loop did have runtime improvement whereas the second loop resulted in a slower runtime.

7.1 Future Work

I believe with modifications to MyMST, it could run comparable to the two parallel implementations I compared with. First, I will optimize the second step of the code. This will help with not only the parallelization but also will improve the serial runtime. Second, I want to use different parallelization methods to improve the scalability of the code. Lastly, I want to implement the GPU version of MyMST.

REFERENCES

[1] “Boruvka’s Algorithm: Greedy Algo-9.” *GeeksforGeeks*, 14 Aug. 2018, www.geeksforgeeks.org/boruvkas-algorithm-greedy-algo-9/.

[2] “Connected Components in an Undirected Graph.” *GeeksforGeeks*, 2 July 2020, www.geeksforgeeks.org/connected-components-in-an-undirected-graph/.

[3] Katsigiannis, Anastasios, et al. “An Approach to Parallelize Kruskal’s Algorithm Using Helper Threads.” *2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum*, 2012, doi:10.1109/ipdpsw.2012.201.

[4] “Kruskal’s Algorithm.” *Wikipedia*, Wikimedia Foundation, 8 Oct. 2019, en.wikipedia.org/wiki/Kruskal%27s_algorithm.

[5] “Minimum Weight Spanning Tree.” *ISS Group at the University of Texas*, iss.oden.utexas.edu/?p=projects%2Fgalois%2Fbenchmarks%2Fmst.

[6] “Minimum Spanning Tree.” *Wikipedia*, Wikimedia Foundation, 8 Sept. 2020, en.wikipedia.org/wiki/Minimum_spanning_tree.

[7] “Prim’s Minimum Spanning Tree (MST): Greedy Algo-5.” *GeeksforGeeks*, 7 Aug. 2019, www.geeksforgeeks.org/prims-minimum-spanning-tree-mst-greedy-algo-5/.

[8] D. Cheriton and R. E. Tarjan, “Finding minimum spanning trees,” *SIAM Journal on Computing*, vol. 5, no. 4, pp. 724–742, 1976.

[9] B. M. Moret and H. D. Shapiro, “An empirical analysis of algorithms for constructing a minimum spanning tree,” in *Workshop on Algorithms and Data Structures*. Springer, 1991, pp. 400–411.

[10] Mamun, Abdullah-Al, and Sanguthevar Rajasekaran. “An Efficient Minimum Spanning Tree Algorithm.” *2016 IEEE Symposium on Computers and Communication (ISCC)*, 2016, doi:10.1109/iscc.2016.7543874.

[11] Siek, Jeremy. “prim_minimum_spanning_tree.” *Boost C++ Libraries*, 2001, www.boost.org/doc/libs/1_55_0/libs/graph/doc/prim_minimum_spanning_tree.html.

[12] “Problem Based Benchmark Suite (2020).” *Problem Based Benchmark Suite*, 2020, www.cs.cmu.edu/~pbbs/.

[13] DIMACS (2010). Center for discrete mathematics and theoretical computer science. http: //www.dis.uniroma1.it/challenge9/download.shtml. Accessed: 4/30/2020.