Minimum Spanning Tree

徐聖哲
資訊工程學系
國立陽明交通大學
rickydanzel@gmail.com

呂則諺
資訊工程學系
國立陽明交通大學
ab28870540@gmail.com

楊傑宇
資訊工程學系
國立陽明交通大學
jerrychild999922@gmail.com

ABSTRACT

Minimum spanning tree is a common tree. Thus, If we can successfully address this issue, we can gain significant benefits. We find three commonly used algorithms for solving MST tasks including Kruskal, Prim, and Borůvka algorithms and evaluate them on three different kinds of dataset. In our experiments,we demonstrated the successful parallelization of three algorithms.and conducted an analysis on each of them.

1 Introduction

Parallel algorithms for MST have applications in various fields, including network design, transportation planning, and clustering analysis. There has been many parallel mst method today, and paralleling mst algorithm do have its importance. We list two reasons below. First, parallel MST algorithms are crucial for processing large-scale graphs efficiently. Traditional sequential algorithms may become impractical for graphs with millions or billions of vertices and edges. Second, Parallel MST algorithms find applications in scientific simulations and engineering analyses where complex models are represented as graphs, and efficient computation of MSTs contributes to overall simulation speed. Moreover, the main reason and motivation for this topic is we want to implement what we have learned from this course to those well-known algorithm. Under parallelization, maybe some algorithm will be faster than others , and change how people design algorithm. If algorithm can be designed to be well parallelize, some problem may become easy to overcome.

2 Proposed Solution

At this stage, we will individually introduce the algorithms we used and discuss the parallelization strategies we plan to adopt based on our observations and analysis.

2.1 Kruskal’s Algorithm

Kruskal’s algorithm is a mst algorithm which uses greedy methods to find the most. The greedy method is to sort the edges in the graph and choose the edge with least weight in the sorted edge order. When edge is chosen, check if the vertex of the edge is already in the most, if not then union the vertex with the most.

2.1.1 Observation

We observe the original serial version of Kruskal's algorithm. We find out that the sort part accounts for almost 90% of execution time. So we choose a parallel quicksort function and analyze the performance.

2.1.2 Parallel implement

We choose two ways to parallel Kruskal's algorithm- openmp and pthread. We choose a pivot and use openmp parallel when split the data to two parts by pivot.

2.1.2.1 Openmp

we use openmp to parallel the sort function and we try to parallel the partition part of quick sort.

2.1.2.2 Pthread

we try to parallel the partition part of quick sort. In pthread method, we limited the thread number with 4. First, we choose a pivot and partition the data to two part. Each part will create a thread. We named the two threads with thread_0 and thread_1. Entering thread_0, we split data to two part again and create another thread_2 to sort first part . Meanwhile , the original thread_0 will sort the second part. Same method will go through thread_1 and create thread_3 to sort data. This way, we can fully adopt the power of four thread and simultaneously sort data in 4 part.

2.2 Prim's Algorithm

Prim's algorithm is a greedy algorithm used for finding the minimum spanning tree of a connected, undirected graph. It starts with an arbitrary node and repeatedly adds the shortest edge that connects a vertex in the growing tree to a vertex outside the tree. This process continues until all vertices are included in the tree, forming a minimum-weight connected subgraph. Prim's algorithm ensures that the resulting tree spans all the nodes with the minimum possible total edge weight. It is efficient and commonly used in network design and clustering applications.

2.2.1 Observation

When profiling the execution of the program, we observed that Prim's algorithm can be parallelized, with the most crucial parallelization occurring during the process of finding the shortest edge connecting a vertex in the growing tree to a vertex outside the tree.

2.2.2 Parallel implement

Our parallelization approach involves evenly distributing nodes connected to the external environment within the current spanned tree to threads. Each thread is assigned an array to record results. Finally, mutual lock mechanisms are employed to prevent race conditions and ensure the correctness of the results. The method to parallelization includes:

1. Pthread.
2. *OpenMP*

2.3 Boruvka's Algorithm

Borůvka's algorithm starts by treating each point as an independent component. For each component, it selects the minimum edge leading out of the component, ensuring that no loop paths are formed. The selected edges are then connected to form new components, continuing this process until only one component remains, completing the algorithm. Therefore, the algorithm primarily consists of two parts: 1. Finding the minimum outgoing edge for each component, and 2. Connecting these selected edges.

2.3.1 Observation

We primarily aim to parallelize step one, as we believe step two has a significant chance of encountering a race condition. Therefore, we intend to leverage threads to explore whether it is possible to simultaneously complete the search for multiple components.

2.3.2 Parallel implement

In this algorithmn, we will only employ one method-OpenMP to enhance the speed. The parallelization steps are also quite straightforward. To ensure correctness, it is necessary to treat each component as a unit for searching. Compared to the serial version, I must first identify the parent of each component before proceeding. Therefore, we just need to add an extra loop to the original serial version to determine whether to start searching for edges from the parent. With this added loop, we can directly leverage OpenMP for loop acceleration.

3 Experimental Methodology

At this stage, we will provide a detailed explanation of the experimental methods and the configuration of the dataset we employed.

3.1 Experiment Setting

To assess the effectiveness of our parallelization, we will initially implement serialized versions of each algorithm. Subsequently, we will employ our parallelization method and evaluate its performance in comparison.

3.2 Dataset setting

We utilized three distinct datasets, and within each dataset, we replaced varying numbers of nodes which is 1000, 2000, and 5000. The following are the description about three kinds of datasets:

- Random: In this dataset, the probability of edges between nodes is set at 0.8. We employed this dataset to represent a graph with high edge density.
- Sparse: In this dataset, the degree of nodes does not exceed a constant value. We utilize this dataset to represent a graph with sparse edge graphs.
- Power: In this dataset, the probability of edges between nodes follows an exponential decay. We utilize this dataset to represent a graph with extremely sparse edge graphs.

4 Experimental Results

Next, we will provide individual introductions to the results of our experiments for each algorithm, along with our discussions and hypotheses regarding the observed outcomes.

4.1 Kruskal's Algorithm

We experiment the parallel performance with three different graphs and observe the execution time between different graphs and parallelization method.

4.1.1 Compared Parallel version with serial version

We compare sort function execution time with serial, openmp and pthread versions. Figure 1 , 2 and 3 illustrated the performance on random, sparse and power graphs. In Figure 1, the performance of openmp and pthread is better than serial. Because the dense graph is relatively large, the parallel version can have better execution time than the serial version. However, Figure 2 and 3 have different results. When the node number is small, the parallel version may not perform better than the serial version. The reason may be the overhead of parallelization, since data is small thus creating thread may influence performance and slower total execution time.

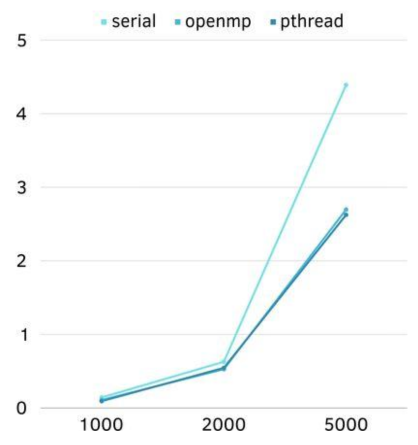

Figure 1: sort function execution time(second) under dense graph with serial, openmp and pthread version

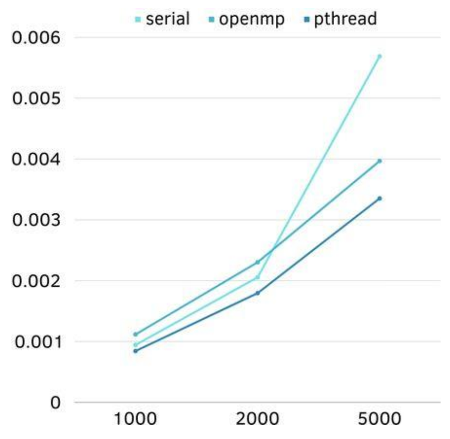

Figure 2: **sort function execution time(second) under sparse graph with serial, openmp and pthread version**

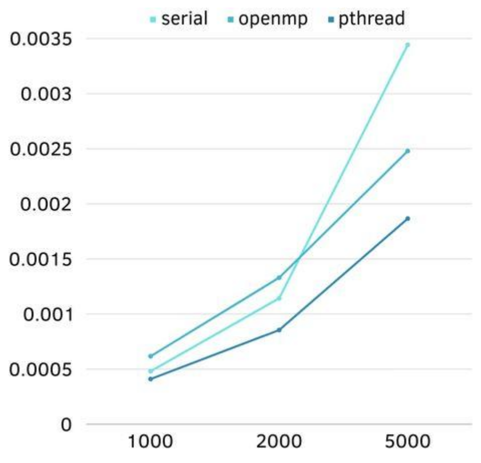

Figure 3: **sort function execution time(second) under highly sparse graph with serial, openmp and pthread version**

4.1.2 Compared Openmp with Pthread

We can find that Pthread has better performance than openmp. The reason behind this result is the property between openmp and pthread. When writing the Pthread version of Quick sort everything could be controlled, where the parallel sections is, what functions and data that should be parallelised and how. This gives the programmer a lot of control and he can really fine-tune the code, this also means that the number of threads that are spawned for a certain task can be controlled. For openmp, The problem with OpenMP is that it has problem to express parallelism within recursion and in an algorithm like a Quick Sort this becomes a problem since it depends on recursion heavily. It spend too much time creating thread. Therefore, Pthread wiil outperform openmp.

Speedup	openmp	Pthread
dense	1.63	1.83
sparse	1.43	1.69
power	1.31	1.55

Table 1: **Speedup between openmp and Pthread**

4.2 **Prim's Algorithm**

We experimented with two common implementations of the Prim's algorithm on three different datasets. One involves preprocessing the edge arrangement, while the other does not.

4.2.1 Compared Parallel version with serial version

This version does not involve preprocessing the edge arragement. Figure 4 , 5 and 6 illustrated the performance on random, sparse and power graphs. After observing the results from the three charts, we noticed that parallelization performs better in edge-dense scenarios. I believe this is because we parallelized the process of finding edges, so in scenarios with fewer edges, the parallelization effect is not as pronounced.

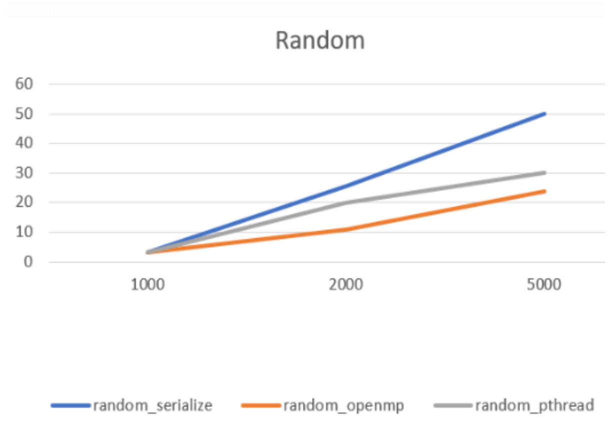

Figure 4: **Execution time(second) under dense graph with serial, OpenMP and Pthread version**

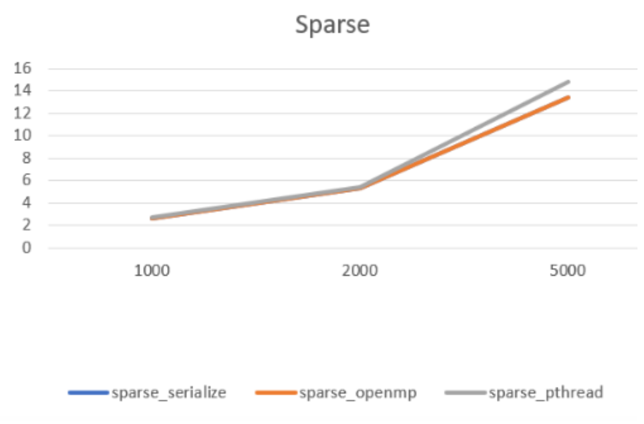

Figure 5: **Execution time(second) under sparse graph with serial, OpenMP and Pthread version**

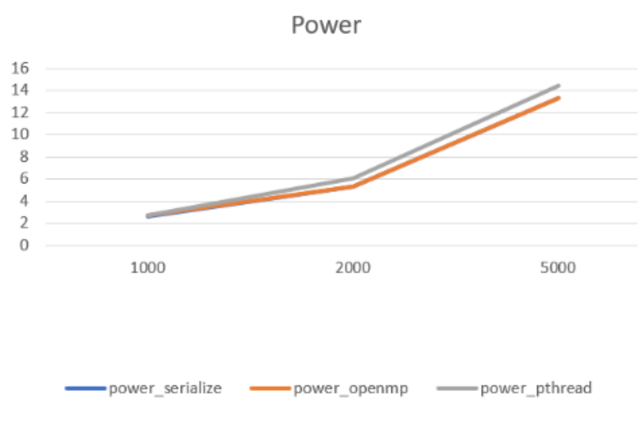

Figure 6: **Execution time(second) under highly sparse graph with serial, OpenMP and Pthread version**

4.2.2 Compared Parallel version with serial version

This version does involve preprocessing the edge arragement. Figure 5 , 6 and 7 illustrated the performance on random, sparse and power graphs. We observed that in the preprocessed version, the parallelization effect is less significant as the original edge-finding time has been compressed.

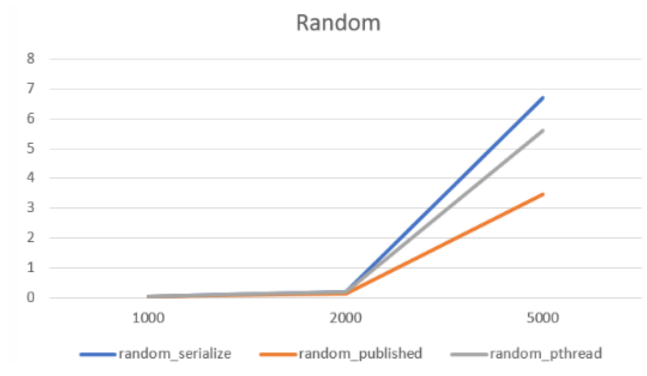

Figure 7: **Execution time(second) under dense graph with serial, OpenMP and Pthread version**

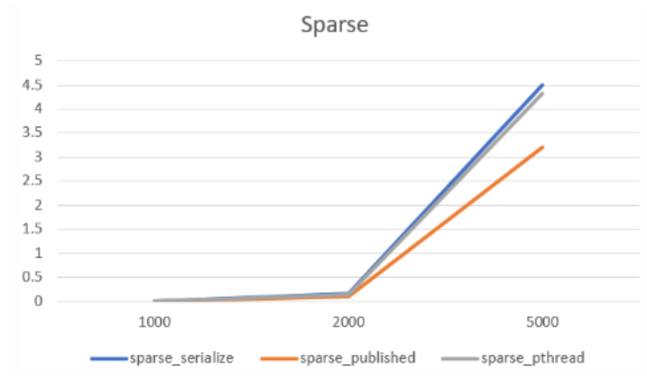

Figure 8: **Execution time(second) under sparse graph with serial, OpenMP and Pthread version**

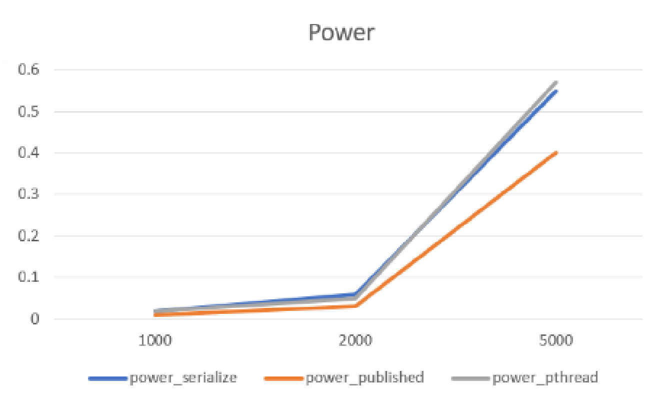

Figure 9: **Execution time(second) under highly sparse graph with serial, OpenMP and Pthread version**

4.2.3 Compared Openmp with Pthread without preprocessing the edge arrangement

Here, we have listed the most significant performance improvements for each different dataset, making it easier for everyone to compare the differences in performance.

Speedup	OpenMP	Pthread
dense	2.18	1.67
sparse	1	0.91
power	1	0.91

Table 2: **Speedup between OpenMP and Pthread in Prim's algorithm without preprocessing the edge arrangement**

4.2.4 Compared Openmp with Pthread with preprocessing the edge arrangement

Here, we observe a slight decrease in the parallelization effect compared to the version without edge arrangement. I believe this is due to the compressed execution time of parallelization.

Speedup	OpenMP	Pthread
dense	1.94	1.19
sparse	1.4	1.04
power	1	0.96

Table 3: **Speedup between OpenMP and Pthread in Prim's algorithm with preprocessing the edge arrangement**

4.3 **Boruvka's Algorithm**

We experiment the parallel performance with three different graphs and observe the execution time between different graphs .

4.3.1 Compared Parallel version with serial version

We compare execution time with serial and openmp versions. Figure 10 , 11 and 12 illustrated the performance on random, sparse and power graphs. In Figure 10, it can be observed that the performance of OpenMP is superior to serial execution when the number of nodes is low. However, in Figures 11 and 12, the performance of OpenMP is not better than serial execution. I speculate that this is due to the need to add an additional loop in our parallelization approach to search for parents. Consequently, the time complexity increases from the original O(logv) to O(ev/klogv). However, in a random graph, where there are numerous edges between each pair of nodes, the Minimum Spanning Tree (MST) is completed in just one iteration. Therefore, I do not incur an additional time cost proportional to the number of vertices (v).

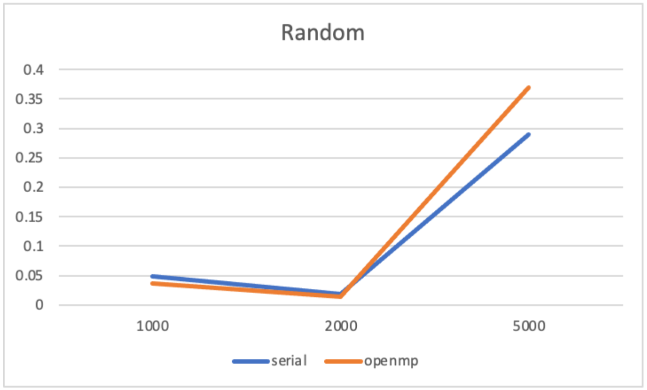

Figure 10: **Execution time(second) under dense graph with serial and OpenMP version**

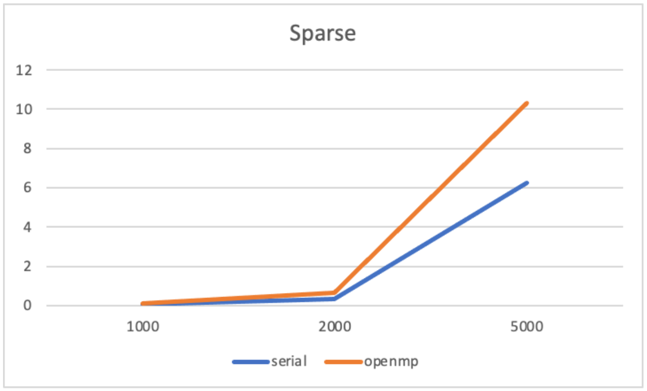

Figure 11: **Execution time(second) under sparse graph with serial and OpenMP version**

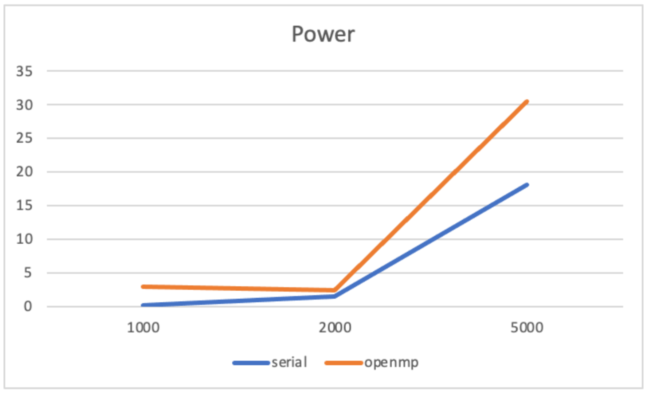

Figure 12: **Execution time(second) under highly sparse graph with serial and OpenMP version**

Speedup	openmp
dense	1.29
sparse	0.45
power	0.05

Table 4: **Speedup using OpenMP in Boruvka's Algorithm**

5 Related work

5.1 Kruskal's Algorithm

In [2], author takes the advantage of union vertex. They use a main thread for serial kruskal's algorithm and run multiple helper threads for checking edges. If the two vertex of the edge is already in the mst, then the helper threads can discard the edges. This way it can reduce the edges number to go through, which significantly speedup the execution time.

5.2 Prim's Algorithm

In [1], parallelization is implemented using OpenMP.They accelerated not only the edge-finding part but also parallelized the comparison process. They accelerated not only the edge-finding part but also parallelized the comparison process. Figures 7, 8, and 9 compare my implementation with [1]. It is evident that their implementation performs slightly better than mine across all comparisons.

5.3 Boruvka's Algorithm

In [3], the author employs wait-free Disjoint Set Union (DSU) techniques to implement a parallelized version of Boruvka's algorithm. By breaking down the Minimum Spanning Tree (MST) into smaller graphs and independently processing them, the author then merges the MSTs from each processor. In my computer-based experiments, I found that this version of Boruvka's algorithm outperforms the serial version, particularly in the case of larger graphs.

6 Conclusion

In our current project, we explored different algorithms and parallelization methods to optimize the Minimum Spanning Tree (MST). The results indicate that, in dense graphs, Boruvka's algorithm outperforms the other two, even when parallelizing the other two algorithms, they still cannot surpass Boruvka's algorithm. However, in sparse graphs, Kruskal's Algorithm and Prim's Algorithm show better performance, which is related to the nature of the algorithms themselves.

Regarding parallelization performance, Kruskal's Algorithm exhibits the most noticeable acceleration, followed by Prim's Algorithm. Boruvka's algorithm, on the other hand, performs the worst and, in some cases, even worse than its original serial version. We believe there is still significant research potential in MST, including the possibility of detecting graph types to select suitable algorithms before applying parallelization versions to minimize running time. Alternatively, exploring different parallelization methods could also contribute to reducing running time.

7 REFERENCES

[1] Erik Lopez, 2016. Parallel Programming with Prim's Algorithm

[2] An approach to parallelize Kruskal's algorithm using Helper Threads

[3] parallel version of the Boruvka's algorithm using wait-free DSU