15-418/618, Parallel Computer Architecture and Programming

Final Project: Parallel Minimum Spanning Tree Algorithms

Xuren Zhou (xurenz), Wenting Ye(wye2)

December 10, 2019

1 URL

GitHub Page: https://allenchou.github.io/CMU-15618-Final-Project/.

2 Summary

In this project, We parallelize the Kruskal's algorithms for minimum spanning tree. We first profile the program and notice that the sorting contributes the most time consuming. Consequently, we investigate different parallel sorting algorithms to speed up the algorithm, including enumeration sort, parallel quicksort, and sample sort. Parallel sample sort achieves the best speed up (4.35x) compared to our sequential baseline implementation with 8 threads on GHC machine.

3 Background

In a connected, edge-weighted undirected graph, a *minimum spanning tree* (MST) is a subset of edges that connects all nodes with the smallest total weights (Figure 1). It has been well-studied for nearly a century and can be solved in polynomial time with different greedy algorithms. For example, Prim's algorithm [1] uses the *cut-property* of MST, and constructs the MST by adding the smallest edge that connects the current nodes set and the remaining nodes set. However, Prim's algorithm is hard to parallelize in nature because each step depends on the current sub-graph built on previous steps.

In this project, we are going to focus on Kruskal's algorithm [2]. Kruskal's algorithm utilizes the *cycles-property* and build the MST by adding the smallest edge that does not create a cycle. These two steps in Kruskal's algorithm: sorting and merging. In sorting step, we sort all edges by their weight in ascending order. In merging step, we

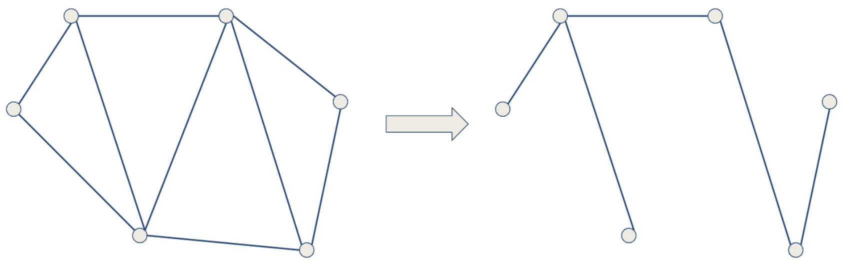


Figure 1: Minimum spanning tree.

iteratively add the smallest edge to our output graph if it does not create a cycle in it. The algorithm is illustrated in Figure 2.

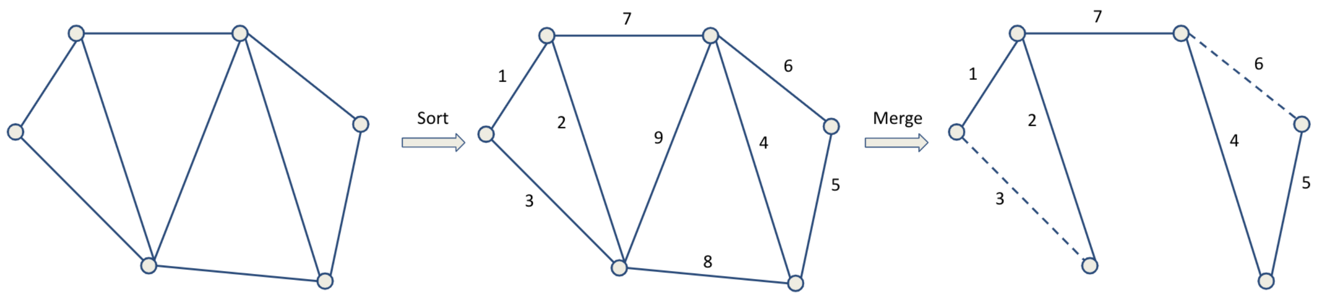


Figure 2: Kruskal's algorithm.

3.1 Data structures

There are two key data structures used in our project: undirected graph and disjoint-set [3]. We use the undirected graph to represent the input and output graph information and use disjoint-set to justify whether it is valid to add a candidate edge to our final output.

3.1.1 Undirected graph

Kruskal's algorithm is an *edge-centric* algorithm. Therefore, we represent the undirected graph as a set of undirected edges. More specifically, `Graph` class has two member variables:

- `int num_vertices`: The number of vertices.
- `std::vector<Edge> edges`: An array of all undirected edges. Each `Edge` has three variables: two integers, `from` and `to`, represent its vertices indices and one float, `weight`, represents its weight. Because our graph is undirected, we require `from` $<$ `to` and there is no duplicated edge in `edges`.

There is no specific key operation on `Graph` because a single edge has already contained all the necessary information for Kruskal's algorithm. Kruskal's algorithm requires to sort all edges based on their weights. For our serial implementation, we use `C++` STL sorting with lambda formula to implement this step.

3.1.2 Disjoint-set

Kruskal's algorithm adds a candidate edge $e = (u, v)$ to the current output set T if there is no cycle after adding e, namely u and v are not connected in current T. Disjoint-set is a very efficient data structure to justify whether u and v are connected and update their connectivity after adding e in $O(\alpha(n))$. We implement this data structure from scratch with two main optimizations, path compression and union by rank, which will be explained in key operations.

Disjoint-set is usually represented as a forest. Two node u and v is contained in the same set if and only if they are located in the same tree. Here the tree is an abstract data structure of the Disjoint-set to represent a set, which is not the tree we need to find in our MST algorithm. Because we use index to represent the vertex and all vertices will be connected in the output MST, we can still use index to represent them in our disjoint-set. More specifically, `DiskjointSet` class has three member variables:

- `int num_nodes`: The number of vertices.
- `std::vector<int> ranks`: An rank array. `ranks[i]` represents the rank of node i.
- `std::vector<int> parents`: An parent array. `parents[i]` represents the parent of node i. If node i is a root, its parent is itself.

There are three key operations on `Disjoint-set`:

- `find(u)`: return the root r of the tree where u is located at. This function traverses the path from u to r to find the root r. To accelerate the later access of u, the parents of all nodes on the path are updated to r. This side effect (optimization) is called as *path compression*.
- `belongSameSet(u, v)`: return true if u and v are located in the same tree. This function is implemented via `find`.
- `unionSet(u, v)`: This function merges the trees of u and v as one tree if u and v are located at different trees. The parent of the root with smaller rank is updated to the root with larger rank and the larger rank is incremented by 1. This optimization is called as *union by rank*. The rank is similar to the depth but it is not because it is not updated during path compression.

We use this data structure to implement the merging step of our serial Kruskal's algorithm and construct the final minimum spanning tree.

3.2 Inputs and outputs

The input and output are both undirected graphs. To reduce the time of loading the input and saving the output, we dump the unidrected graph into binary form. The first 8 bytes is the number of vertices, followed by 8 bytes representing the number of edges and then followed by the edge array:

```
| #vertices (8 bytes) | #edges (8 bytes) | edge[0] (16 bytes) | edge[1] (16 bytes) | ...
```

We implement the loading and the saving as two methods in `Graph`: `loadGraph` and `saveGraph`.

3.3 Workload breakdown

Kruskal's algorithm can be broken down into sorting step and merging step.

The merging step needs the candidate edges added from the smaller weight to larger weight, therefore it needs to wait for the finishing of sorting step. Meanwhile, the validity of a candidate edge depends on the current output forest, which means that it depends on the validity of all previous candidate edges. Therefore, it is hard to parallelize the merging step.

Meanwhile, parallel sorting has been studied for a long time. It is a challenging problem but there is indeed a large potential for parallelism. The most directly is to parallelize QuickSort. QuickSort is an in-place sorting algorithm. It first picks up a pivot and use it to partition elements into two parts and recursively apply the sorting algorithm to these two parts. The partition makes sure the elements will be finally located at its part, which is the data locality. The recursive sorting implies a SIMD execution. Therefore, we can use shared-memory parallelism to speedup the sorting step.

Theoretically, the edge sorting takes $O(|E|\log|E|)$ operations and building the MST by merging takes $O(|E|\alpha(|V|))$ operations, where α is the inverse Ackermann function [4]. In practice, $\alpha(|V|)$ grows extremely slow and hence the overall complexity is $O(|E|\log|E|)$. In the later result section, you will see that sorting step is much more computational dense than merge step and parallel sorting can provide quite good speedup for Kruskal's algorithm.

4 Approach

4.1 Kruskal's algorithm

Recap our serial solution of minimum spanning tree: Kruskal's algorithm involves two main parts.

1. Sorting: sort all edges in the graph by its weight;
2. Merging: add the smallest edge that does not create a cycle to the answer sub-graph until there are $v-1$ edges in the sub-graph.

The edge sorting takes $O(|E|\log|E|)$ operations and building the MST takes $O(|E|\alpha(|V|))$ operations, where α is the inverse Ackermann function. Since in merging part, each step depends on the sub-graph built previously, it's highly dependent and hard to parallelize. Besides, sorting has higher time complexity than the merging part. Hence, in this project, we will focus on parallel sorting algorithms.

4.2 Parallel sorting algorithms

In this section, we will assume that we have p processors.

4.2.1 Enumeration Sort

Enumeration sort is straight-forward and can be parallelized easily. The procedure is as follows:

1. Calculate the final index in sorted array within its partition by counting the number of elements that is smaller than it;
2. Re-map the array to get the sorted array.

The operation is perfectly perallelized, however, it requires $O(n^2)$ time complexity. In our experiment, even for a graph with 5,000 nodes, we find out it cannot obtain the answer in a reasonable amount of time. Hence, we decide to move forward with other approaches.

4.2.2 Parallel quicksort

The classic quicksort algorithm is defined as the following recursive procedure:

1. Randomly select an element x as pivot;
2. Put x in its correct position by placing all smaller elements before it and all bigger elements after it;
3. Sort the two partitions using quicksort separately.

The quicksort is a recursive in-place algorithm, which indicates a SIMD execution and data locality. To parallelize the recursive calls of our quicksort, we adopt OpenMP task parallelism.

4.2.3 Parallel sample sort

The sample sort is a generalization of quicksort. Quicksort divides the array by 2 parts, however, the sample sort divides the array into p partitions, and then each partition can be sorted separately [5,6]. The procedure is described as followed:

1. Select p pivots randomly from the original array;
2. Partition the array by p pivots selected before;
3. Sort each partition.

The run-time of the algorithm depends on the largest partition, hence it's ideal to create balanced p partitions. A technique called *oversampling* is used. In order to get p pivots, we first randomly select rp pivots and sort them.

Then, r-th, $2 \cdot r$-th, $\dots$, and $p \cdot r$-th pivots are used to partition the data. Here r is the *oversampling ratio*, and larger r yields a more balanced distribution. We found this technique improves the balances of partititon significantly.

The parallel sample sort is described as below:

1. Select p pivots randomly from the original array by oversampling ($O(rp \cdot \log(rp))$). This is done by one processor;
2. Each processor counts the size of each bucket in its partition ($O(n/p \cdot \log p)$);
3. Generate the start index for each bucket and partition ($O(p^2)$). This is done by one processor;
4. Each processor maps its elements to its bucket ($O(n/p \cdot \log p)$);
5. Each processor sorts its partitions in the new bucket ($O(n/p \cdot \log(n/p))$).

4.3 Implementation details

We target a single shared-memory machine with multiple cores. `C++` is used to implement the algorithm and OpenMP to parallelize our workload. In our sequential baseline algorithm, we use `std::sort()` and union find with path compression and union by rank.

5 Dataset

The computational complexity of Kruskal's algorithm is in term of $|E|$ instead of $|V|$. Therefore we fix the number of vertices to 40000 in our evaluation. We tried to use a larger number but limited quota of AFS does not allow us to do that.

The minimal spanning tree of an undirected graph is decided by the order of edge weights and the graph topology. To generate a random edge permutation, we uniformly sampling the weight within a fixed range. As we mentioned before, the weight does not matter but the order of weights matters. In our evaluation, we fix the weight range between 1.0 and 10.0.

We explore three kinds of graphs to evaluate the performance of our parallel MST algorithms on different graph topologies:

- **Random graph**: For any two distinguished vertices u and v, edge (u, v) is in the graph with probability $p \in \{0.01, 0.05, 0.1\}$.
- **Sparse graph**: For any vertex u, $d \in \{10, 50, 100\}$ distinguished vertices v are connected to u. After removing duplicated edges, the degree of each vertex is bounded between d and $2d$.
- **Power-law graph**: The vertex degree d satisfies the power-law distribution $p(d) \sim d^{-r}, r \in \{1.5, 2.5, 3.5\}$. Each vertex u is connected with d distinguished vertices v and duplicated edges are removed.

6 Results

6.1 Correctness

We first verity the correctness of our program by comparing with the `kruskal_minimum_spanning_tree()` provided in Boost [7]. We test 5 different graphs with 20,000 nodes generated under "random" setting (edge probability = 0.1) and shows that our program produces the same output as the reference boost library. It proves that our code is correct.

6.2 Breakdown of time consumption

In this experiment, we generate random graphs of different nodes number with edge probability 0.1.

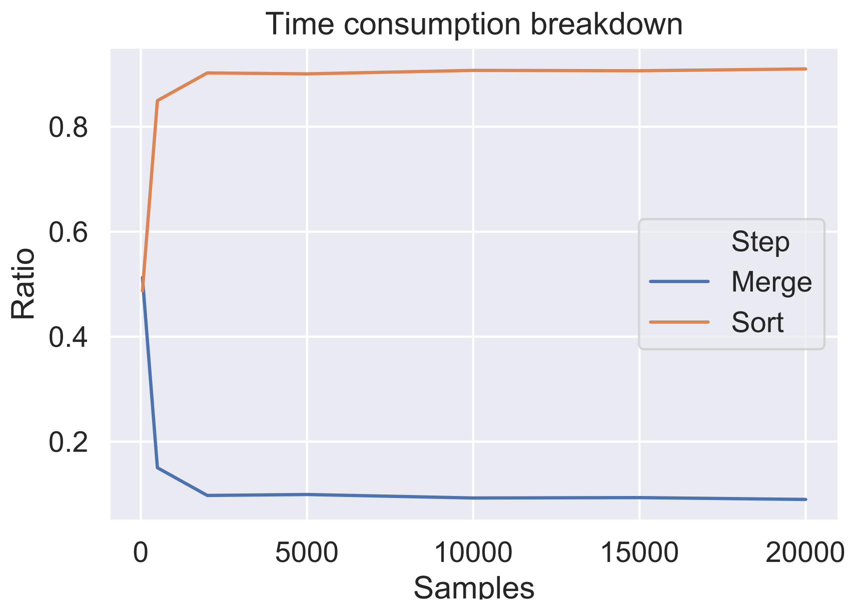


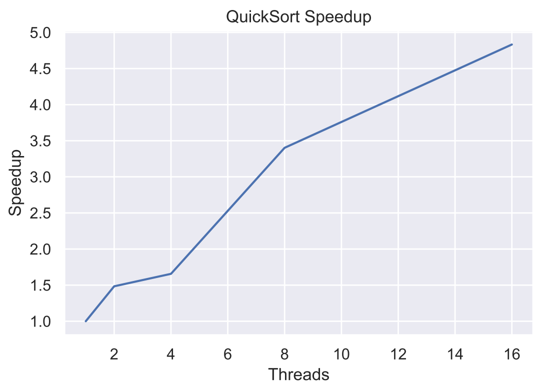
Figure 3: Breakdown of time consumption of our sequential Kruskal's algorithm

We profile our sequential Kruskal's algorithm. With insufficient data, sorting and merging takes the roughly same amount of time. But in the cases where we have a large amount of data, sorting dominates the overhead and takes 90% of run-time.

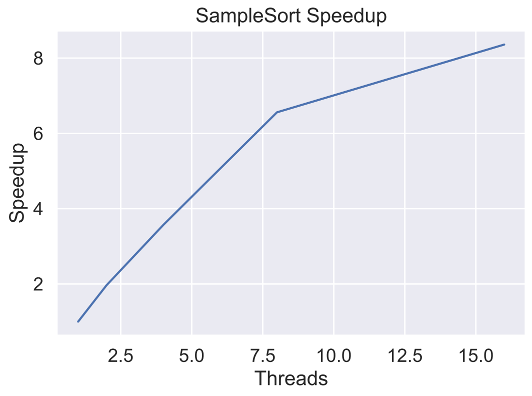
6.3 Sorting algorithm speedup

In this section, we want to investigate the speedup for each parallel sorting algorithm. To this end, we generate a graph with 40,000 nodes under a "random" setting with edge probability 0.1. The baseline run with one thread.

As shown in Figure 4, we can see that sample sort achieves better speedup compared to quicksort. We observe that the slope decrease when the number of threads is larger than 8, which is caused by hyper-threading. Speedup curve for quick sort is less linear. We hypothesize the reason is that it's parallelized based on dynamic scheduling



(a) Quick sort speedup



(b) Sample sort speedup

Figure 4: Sorting algorithm speedup

and each task has different workload. Besides, it takes more time to generate enough task to utilize all processors as the number of threads increases. Quicksort also limited by the larger dependency between different tasks. Sample sort has better performance over all cases.

6.4 Kruskal's algorithm speedup

Then we investigate the speedup for Kruskal's algorithm by parallelizing the sorting. To this end, we first evaluate the algorithm on a graph with 40,000 nodes generated under a "random" setting with edge probability 0.1. The speedup curve is shown as Figure 5.

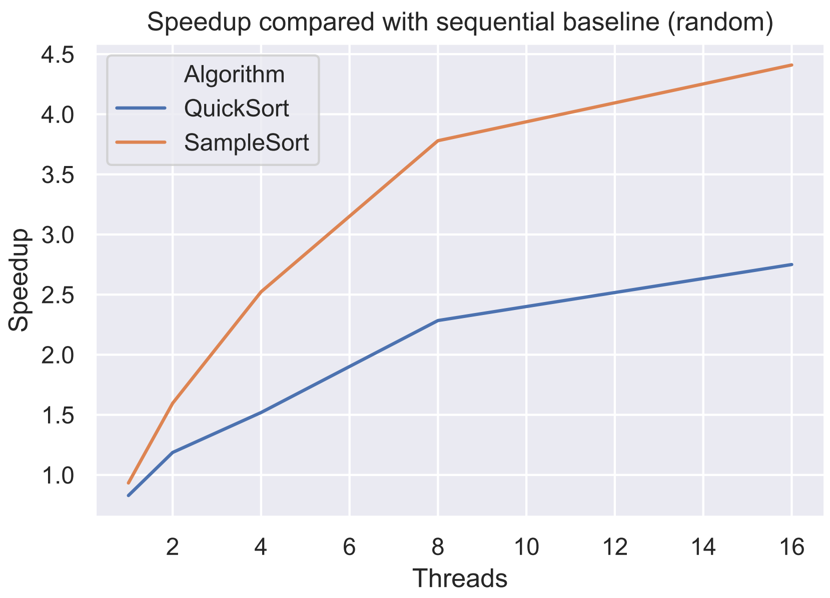
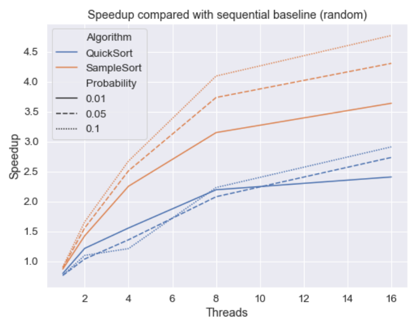


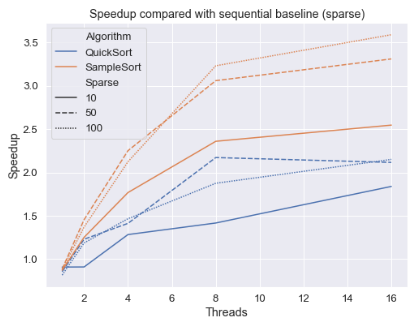
Figure 5: Speedup compared with sequential baseline

Firstly, we notice that although our sample sort achieves more than 8x speedup in sorting, it only achieves around 4.3x speedup in the end. The reason is that sorting only occupies 90% of the time, and such a degradation is caused by Amdahl's law. Under the circumstance with single thread, both algorithms performs worse than baseline because baseline uses STL sorting with sophisticated optimization. Parallel sample sort consistently provide better performance over parallel quicksort.

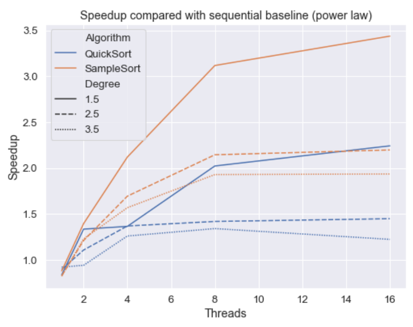
Additionally, we also evaluate the performance of the algorithm with different graphs, and the results are shown in Figure 6.



(a) Random graph



(b) Sparse graph



(c) Power law graph

Figure 6: Speedup achieved on different graphs

As shown in Figure 6(a) and Figure 6(b), the achieved speedup increases when the number of edges increases. It meets our expectations since sorting algorithm will take relatively more than merging with more edges. When the number of edges is small, due to the Amdahl's law, the speedup stops increasing quickly in Figure 6(c).

7 Work Distribution

Wenting Ye implemented the sequential Kruskal's algorithm, enumeration sort, parallel sampling sort, checker program using Boost. Xuren Zhou implemented the parallel quick sort, graph generator, proposal, and website. The final report and poster are finished by both. We contribute 50%-50% to this project.

References

[1] R. C. Prim, "Shortest connection networks and some generalizations," *The Bell System Technical Journal*, vol. 36, no. 6, pp. 1389–1401, 1957.

[2] J. B. Kruskal, "On the shortest spanning subtree of a graph and the traveling salesman problem," *Proceedings of the American Mathematical society*, vol. 7, no. 1, pp. 48–50, 1956.

[3] B. A. Galler and M. J. Fisher, "An improved equivalence algorithm," *Communications of the ACM*, vol. 7, no. 5, pp. 301–303, 1964.

[4] R. E. Tarjan and J. Van Leeuwen, "Worst-case analysis of set union algorithms," *Journal of the ACM (JACM)*, vol. 31, no. 2, pp. 245–281, 1984.

[5] W. D. Frazer and A. McKellar, "Samplesort: A sampling approach to minimal storage tree sorting," *Journal of the ACM (JACM)*, vol. 17, no. 3, pp. 496–507, 1970.

[6] J. Huang and Y. Chow, "Parallel sorting and data partitioning by sampling," 1983.

[7] B. Schäling, *The boost C++ libraries.* Boris Schäling, 2011.