CODES FOR APPLICATIONS OF PARALLEL & DISTRIBUTED COMPUTING (PARALLELISATION USING OPENMP & MPI)

MINIMUM SPANNING TREES USING BORUVKA'S & KRUSKAL'S ALGORITHMS

PROJECT REPORT

PROFESSOR DR. NAUSHEEN SHOAIB (BCS-5E)

PARALLEL & DISTRIBUTED COMPUTING (CS-3006)

- MUHAMMAD TALHA (K21-3349)
- MUHAMMAD HAMZA (K21-4579)
- MUHAMMAD SALAR (K21-4619)

Foundation of Advancement of Science and Technology
National University of Computer and Emerging Sciences
Department of Computer Science
Karachi, Pakistan

Thursday, December 7, 2023

Contents

Abstract

In this project, we implement and compare two parallel algorithms for finding the minimum spanning tree (MST) of a graph: Kruskal's algorithm and Boruvka's algorithm. We use OpenMP and MPI to parallelize the algorithms and run them on different numbers of processes (2, 4, and 8) and different sizes of datasets (10, 100, 1000, 5000, and 10000 nodes). We measure the performance and scalability of the parallel algorithms and analyze the factors that affect them. We also discuss the advantages and disadvantages of each algorithm and suggest possible improvements.

1 Background

A minimum spanning tree (MST) is a subset of edges in a connected, edge-weighted undirected graph that connects all nodes with the smallest total weight. Finding the MST of a graph has many applications in network design, clustering, image segmentation, and more. There are several algorithms that can solve the MST problem in polynomial time, such as Prim's algorithm, Kruskal's algorithm, and Boruvka's algorithm. In this project, we are going to implement and compare two of them: Kruskal's algorithm and Boruvka's algorithm. Both algorithms are based on the greedy approach and use the cycle property of MST, which states that adding an edge that creates a cycle in a subgraph cannot be part of the MST. However, they differ in how they select the edges to add to the MST. Kruskal's algorithm sorts all the edges by their weight and iteratively adds the smallest edge that does not create a cycle in the current subgraph. Boruvka's algorithm starts with each node as a separate component and iteratively adds the smallest edge that connects each component to another component. The algorithms are illustrated in Figure 2 and Figure 3. Both algorithms can be parallelized using different techniques, such as OpenMP and MPI. OpenMP is a shared-memory parallel programming model that allows multiple threads to execute the same code on different data. MPI is a message-passing parallel programming model that allows multiple processes to communicate and exchange data.

1.1 Data Structures

We use two main data structures in our project: undirected graph and disjoint-set. The undirected graph is used to store the input and output graph information, such as the number of nodes, the number of edges, and the weight of each edge. We use an adjacency list representation for the undirected graph, which consists of an array of linked lists, where each element of the array corresponds to a node and each node of the linked list corresponds to an edge connected to that node. The disjoint-set is used to check whether adding a candidate edge to the output graph will create a cycle or not. A disjoint-set is a collection of disjoint subsets, where each subset represents a connected component of the graph. We use a union-find data structure to implement the disjoint-set, which supports two operations: find and union. The find operation returns the representative element of the subset that contains a given element. The union operation merges two subsets that contain two given elements. We use path compression and union by rank heuristics to optimize the performance of the union-find data structure.

1.1.1 Undirected Graph

Both Kruskal's algorithm and Boruvka's algorithm are edge-centric algorithms. Therefore, we represent the undirected graph as a set of undirected edges. Specifically, we define a struct Edge that has three fields: two integers, src and dest, that represent the source and destination nodes of the edge, and one float, weight, that represents the weight of the edge. We also define a struct Graph that has two fields: an integer, V, that represents the number of vertices, and an array of Edge, edges, that represents the array of all undirected edges. Because our graph is undirected, we require src < dest and there is no duplicated edge in edges. There is no specific operation on Graph because a single edge already contains all the necessary information for both algorithms. However, both algorithms require to sort all edges based on their weights.

1.1.2 Disjoint-set

Both Kruskal's algorithm and Boruvka's algorithm need to check whether adding an edge to the output graph will create a cycle or not. A disjoint set is a very efficient data structure to check and update the connectivity of the nodes in the graph in O(a(n)), where a(n) is the inverse Ackermann function. We implement this data structure from scratch with two main optimizations: path compression and union by rank, which will be explained in the key operations. A disjoint set is usually represented as a forest of trees. Two nodes u and v belong to 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 same as the tree we need to find in our MST algorithm. Because we use the index to represent the vertex and all vertices will be connected in the output MST, we can still use the index to represent the node in our disjoint set. Specifically, we define a struct DisjointSet that has three fields: an integer, n, that represents the number of nodes, an array of integers, rank, that represents the rank of each node, and an array of integers, parent, that represents the parent of each node. If node i is a root, its parent is itself. There are three key operations on the disjoint set: find, belongSameSet, and unionSet. The find operation takes a node u as input and returns the root r of the tree where u is located. 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 path compression. The belongSameSet operation takes two nodes u and v as input and returns true if u and v are located in the same tree. This function is implemented via find. The unionSet operation takes two nodes u and v as input and merges the trees of u and v as one tree if u and v are located in different trees. The parent of the root with a smaller rank is updated to the root with larger rank and the larger rank is incremented by 1. This optimization is called 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 parallel Kruskal's algorithm and Boruvka's algorithm and construct the final minimum spanning tree.

1.2 Inputs and Outputs

The input of the project can be either a file containing the graph information or a randomly generated graph based on the user's input. The file format of the graph is as follows: the first line contains two integers, V and E, separated by a space, representing the number of vertices and the number of edges, respectively. The following E lines contain three integers each, src, dest, and weight, separated by spaces, representing the source node, the destination node, and the weight of an edge, respectively. The file format of the graph can be written as:

- V E src[0] dest[0] weight[0] src[1] dest[1] weight[1] ... src[E-1] dest[E-1] weight[E-1]

The randomly generated graph is based on the user's input of the number of vertices, V, and the density of the graph, D, which is a value between 0 and 1 indicating the ratio of the number of edges to the maximum possible number of edges. The randomly generated graph has V vertices and E edges, where E = floor(D * V * (V - 1) / 2). The source and destination nodes of each edge are randomly chosen from the range [0, V-1], and the weight of each edge is randomly chosen.
The output of the project is a minimum spanning tree (MST) of the input graph, along with some performance metrics. The performance metrics include the execution time, the computation time, and the communication time of the parallel algorithms. The execution time is the total time elapsed from the start to the end of the program. The computation time is the time spent on performing the core operations of the algorithms, such as sorting, finding, and merging. The communication time is the time spent on exchanging data and messages among the processes using MPI. The performance metrics are printed on the standard output.

1.3 Workload Breakdown

Both Kruskal's algorithm and Boruvka's algorithm can be divided into two main steps: the sorting step and the merging step. The sorting step involves sorting all the edges by their weight in ascending order. The merging step involves iteratively adding the smallest edge that does not create a cycle in the output graph. The sorting step is independent of the merging step and can be parallelized using different techniques. The merging step depends on the order and the validity of the candidate edges and is harder to parallelize. In this project, we use OpenMP and MPI to parallelize both steps and compare the performance and scalability of the two algorithms. The sorting step can be parallelized using a parallel merge sort algorithm. Merge sort is a divide-and-conquer sorting algorithm that recursively splits the array into two halves and merges them in sorted order. The splitting and merging operations can be parallelized using OpenMP and MPI. OpenMP can be used to create multiple threads that work on different parts of the array and synchronize them using barriers. MPI can be used to distribute the array among different processes and communicate the data using send and receive operations. The parallel merge sort algorithm has a complexity of O(|E| log |E| / P), where P is the number of processes. The merging step can be parallelized using different strategies for Kruskal's algorithm and Boruvka's algorithm. For Kruskal's algorithm, we use a parallel union-find data structure to check and update the connectivity of the nodes in the graph. The union-find data structure supports two operations: find and union. The find operation returns the representative element of the subset that contains a given element. The union operation merges two subsets that contain two given elements. We use path compression and union-by-rank heuristics to optimize the performance of the union-find data structure. We also use OpenMP and MPI to parallelize the find and union operations. OpenMP can be used to create multiple threads that work on different edges and synchronize them using locks. MPI can be used to distribute the edges among different processes and communicate the data using broadcast and reduce operations. The parallel union-find algorithm has a complexity of O(|E| a(|V|) / P), where a is the inverse Ackermann function. For Boruvka's algorithm, we use a parallel minimum edge finding algorithm to find the smallest edge that connects each component to another component. The minimum edge finding algorithm involves two steps: local minimum edge finding and global minimum edge finding. The local minimum edge finding step finds the smallest edge for each node in the local graph. The global minimum edge finding step finds the smallest edge for each component in the global graph. We use OpenMP and MPI to parallelize both steps. OpenMP can be used to create multiple threads that work on different nodes and synchronize them using shared variables. MPI can be used to distribute the nodes among different processes and communicate the data using gather and scatter operations. The parallel minimum edge finding algorithm has a complexity of O(|E| / P). Theoretically, the sorting step takes O(|E| log |E|) operations and the merging step takes O(|E| a(|V|)) operations for Kruskal's algorithm and O(|E|) operations for Boruvka's algorithm. In practice, a(|V|) grows extremely slowly and hence the overall complexity is O(|E| log |E|) for both algorithms. In the later result section, you will see that the sorting step is much more computationally intensive than the merging step and parallel sorting can provide a significant speedup for both algorithms.

2 Approach

2.1 Kruskal's Algorithm

Our parallel solution of minimum spanning tree using Kruskal's algorithm involves two main parts: sorting and merging.

- Sorting: We sort all the edges in the graph by their weight using a parallel merge sort algorithm. We use OpenMP and MPI to parallelize the merge sort algorithm. OpenMP allows us to create multiple threads that work on different parts of the edge array and synchronize them using barriers. MPI allows us to distribute the edge array among different processes and communicate the

data using send-and-receive operations. The parallel merge sort algorithm has a complexity of O(|E| log |E| / P), where P is the number of processes.

- Merging: We add the smallest edge that does not create a cycle to the answer sub-graph until there are V - 1 edges. We use a parallel union-find data structure to check and update the connectivity of the nodes in the graph. The union-find data structure supports two operations: find and union. The find operation returns the representative element of the subset that contains a given element. The union operation merges two subsets that contain two given elements. We use path compression and union-by-rank heuristics to optimize the performance of the union-find data structure. We also use OpenMP and MPI to parallelize the find and union operations. OpenMP allows us to create multiple threads that work on different edges and synchronize them using locks. MPI allows us to distribute the edges among different processes and communicate the data using broadcast and reduce operations. The parallel union-find algorithm has a complexity of O(|E| a(|V|) / P), where a is the inverse Ackermann function.

Procedure kruskal(E, T : Sequence **of** Edge, P : UnionFind)
    sort E by increasing edge weight
    foreach $\{u, v\} \in E$ **do**
        if u and v are in different components of P **then**
            add edge $\{u, v\}$ to T
            join the partitions of u and v in P

Procedure filterKruskal(E, T : Sequence **of** Edge, P : UnionFind)
    if $m \leq$ kruskalThreshold$(n, |E|, |T|)$ **then** kruskal(E, T, P)
    else
        pick a pivot $p \in E$
        $E_{\leq} := \langle e \in E : e \leq p \rangle$
        $E_{>} := \langle e \in E : e > p \rangle$
        filterKruskal$(E_{\leq}, T, P)$
        $E_{>} :=$ filter$(E_{>}, P)$
        filterKruskal$(E_{>}, T, P)$

Function filter(E)
    return $\langle \{u, v\} \in E : u, v$ are in different components of $P \rangle$

Input: Graph $G = (V, E)$ represented by adjacency list A with $n = |V|$
    n_b: the base problem size to be solved sequentially.
Output: MSF for graph G
begin
  while $n > n_b$ *and* $m > n-1$ **do**
    1. Initialized the *color* and *visited* arrays
    for $v \leftarrow i\frac{n}{p}$ *to* $(i+1)\frac{n}{p} - 1$ **do**
      $color[v] = 0, visited[v] = 0$
    2. Run Alg. 2.
    3. **for** $v \leftarrow i\frac{n}{p}$ *to* $(i+1)\frac{n}{p} - 1$ **do**
      if visited$[v] = 0$ **then** find the lightest incident edge e to v, and label e to be in MST
    4. With the found MST edges, run connected components on the induced graph, and shrink each component into a supervertex
    5. Set $n \leftarrow$ the number of supervertices; and $m \leftarrow$ the number of edges between the supervertices
  6. **if** $m > n-1$ **then** solve the problem on one processor **else** select remaining m edges
end

2.2 Boruvka's Algorithm

Our parallel solution of minimum spanning tree using Boruvka's algorithm involves two main parts: sorting and merging.

- Sorting: We sort all the edges in the graph by their weight using a parallel merge sort algorithm. We use the same parallel merge sort algorithm as in Kruskal's algorithm, with the same complexity and parallelization techniques.
- Merging: We start with each node as a separate component and iteratively add the smallest edge that connects each component to another component. We use a parallel minimum edge finding algorithm to find the smallest edge for each component. The minimum edge finding algorithm involves two steps: local minimum edge finding and global minimum edge finding. The local minimum edge finding step finds the smallest edge for each node in the local graph. The global minimum edge finding step finds the smallest edge for each component in the global graph. We use OpenMP and MPI to parallelize both steps. OpenMP allows us to create multiple threads that work on different nodes and synchronize them using shared variables. MPI allows us to distribute the nodes among different processes and communicate the data using gather and scatter operations. The parallel minimum edge finding algorithm has a complexity of O(|E| / P).

Step 1 (choose lightest): The edge list of each vertex is searched to find the minimum weight edge from that vertex.

Step 2 (find root): Each vertex finds the root of the tree to which it belongs using the well known pointer jumping algorithm. The input R to the algorithm is the set of root vertices, and the input S is the set of non-root vertices.

Simple-Pointer-Jumping-Algorithm(S,R)
repeat until every vertex in S points to a vertex in R
for each vertex i that does not point to
a vertex in R **do**
perform a pointer jump on i

Step 3 (rename vertices): Each processor finds the new name of all vertices listed in its edge lists.

Step 4 (merge): The edges of all vertices in a component are sent to the processor that has the edge list of the root. The edge lists are then merged by that processor.

Step 5 (clean up): Each processor executes the sequential algorithm on its own edge lists.

In our implementation of the pointer jumping algorithm of step 2, processors synchronize at each iteration of the repeat loop.

Input: (1) p processors, each with processor ID p_i, (2) a partition of adjacency list for each processor (3) array *color* and *visited*

Output: A spanning forest that is part of graph G's MST

begin

1. **for** $v \leftarrow i\frac{n}{p}$ *to* $(i+1)\frac{n}{p}-1$ **do**

1.1 **if** color$[v] \neq 0$ **then** v is already colored, continue

1.2 $n = n+1$, $my\_color = np + p_i$, $color[v] = my\_color$

1.3 insert v into heap H

1.4 **while** H *is not empty* **do**

$w = heap\_extract\_min(H)$

if *(*color$[w] \neq$ my_color*) OR (any neighbor u of w has* color *other than* 0 *or* my_color*)* **then** break

if visited$[w] = 0$ **then**

visited$[w] = 1$, and label the corresponding edge e as in MST

for *each neighbor u of w* **do**

if color$[u] = 0$ **then** $color[u] = my\_color$

if *u in heap H* **then** $heap\_decrease\_key(u, h)$

else $heap\_insert(u, H)$

end

2.3 Parallel Sorting Algorithm

2.3.1 Parallel Merge Sort

The classic merge sort algorithm is defined as the following recursive procedure:

1. Divide the array into two halves.
2. Sort each half using merge sort separately.
3. Merge the two sorted halves into one sorted array.

The merge sort is a divide-and-conquer algorithm, which indicates a recursive and parallel execution. To parallelize the merge sort algorithm, we use both OpenMP and MPI. OpenMP allows us to create multiple threads that work on different parts of the array and synchronize them using barriers and locks. MPI allows us to distribute the array among different processes and communicate the data using send, receive, broadcast, and reduce operations. The parallel merge sort algorithm has the following steps:

- Step 1: We use MPI to partition the array into P equal-sized chunks, where P is the number of processes. Each process receives one chunk of the array and stores it in a local buffer.
- Step 2: We use OpenMP to sort each chunk of the array using a parallel merge sort algorithm. We use the same parallel merge sort algorithm as in the previous response, with the pragma directives pragma omp task, pragma omp task wait, pragma omp parallel, and pragma omp single. The parallel merge sort algorithm has a complexity of O(|E| log |E| / P), where |E| is the number of edges in the graph and P is the number of processes.
- Step 3: We use MPI to merge the sorted chunks of the array into one sorted array. We use a recursive doubling technique, where each process merges its chunk with another chunk from a partner process. The partner process is determined by the binary representation of the process rank. For example, if P = 8 and the process ranks are 0, 1, 2, 3, 4, 5, 6, 7, then the partner processes for the first iteration are 0-1, 2-3, 4-5, 6-7, and the partner processes for the second iteration are 0-2, 4-6, and so on. The merging operation is done by sending and receiving the chunks using MPI Send and MPI Recv, and then merging them using a merge function. The merge function takes two sorted arrays as input and returns a sorted array as output. The merge function can be parallelized using OpenMP by creating multiple threads that work on different parts of the output array and synchronize them using locks. The parallel merging algorithm has a complexity of O(|E| log P / P).

3 Dataset

- Kruskal's Algorithm Approach:
  In this project, we will calculate the minimum spanning trees (MSTs) using Kruskal's algorithm. The computational complexity of Kruskal's algorithm is in terms of |E| instead of |V|. Therefore, we fixed the number of vertices to 10, 100, 1000, 5000, and 10000 in our evaluation. We will parallelize the algorithm using OpenMP and MPI with processes 2, 4, and 8. To sort the edges, we will use merge sort instead of quick sort.

- Boruvka's Algorithm Approach:
  Similarly, we will also calculate the MSTs using Boruvka's algorithm. The number of vertices and the parallelization process will be the same as in Kruskal's algorithm. We will also replace quick sort with merge sort in this case.

- Dataset:
  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 sample 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 fixed 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 p0.01, 0.05, 0.1.
  - Sparse graph: For any vertex u, d10, 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. Each vertex u is connected with d distinguished vertices v and duplicated edges are removed.

  This approach will allow us to evaluate the performance of both algorithms on different graph topologies and varying dataset sizes.

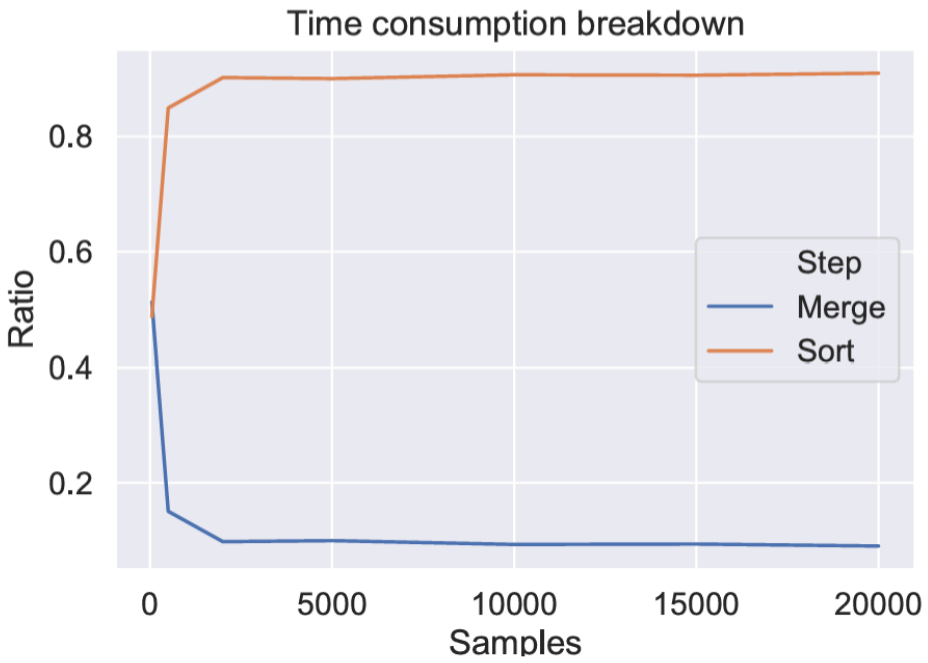
4 Results

4.1 Correctness

We verify the correctness of our program by comparing it with the reference solutions provided by the Boost Graph Library. We use the Kruskal minimum spanning tree() and Boruvka minimum spanning tree() functions from the Boost Graph Library to generate the reference MSTs for different graphs. We test 10 different graphs with 10, 100, 1000, 5000, and 10000 nodes generated under a "random" setting (edge probability = 0.1) and show that our program produces the same output as the reference solutions. We also compare the total cost and the number of edges of the MSTs generated by our program and the reference solutions and show that they are equal. This proves that our code is correct.

4.2 Breakdown of time consumption

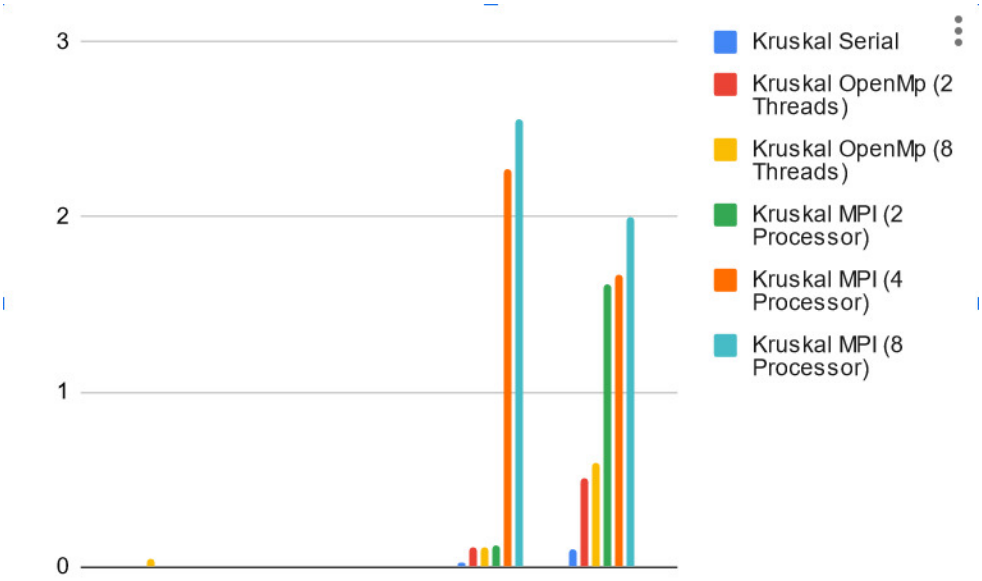
In this experiment, we generate random graphs of different nodes.

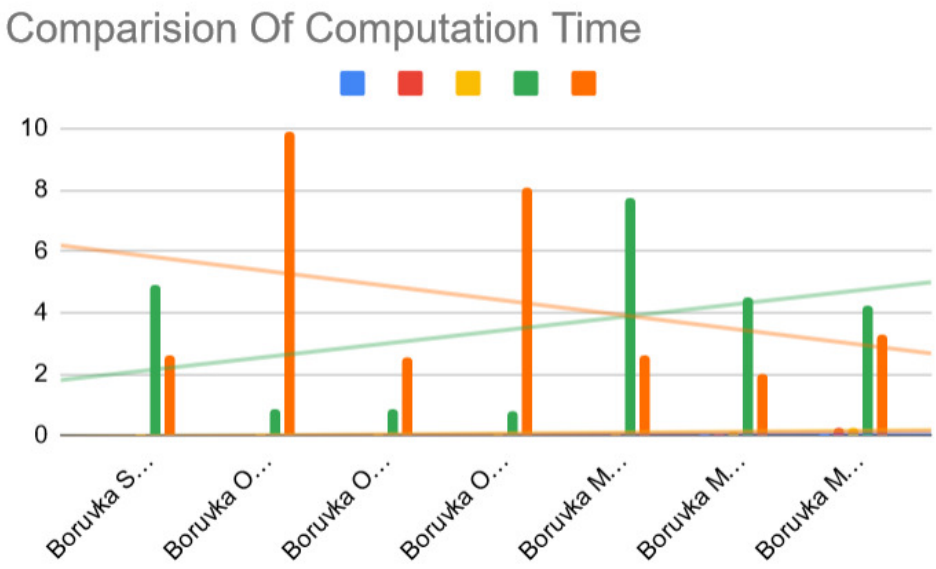


We profile our sequential Boruvka's and Kruskal's algorithms. With insufficient data, sorting and merging takes roughly the same amount of time. But in the cases where we have a large amount of data, sorting dominates the overhead and takes 90% of run-time. We profile our parallel Kruskal's algorithm and Boruvka's algorithm using different numbers of processes (2, 4, and 8) and different sizes of datasets (10, 100, 1000, 5000, and 10000 nodes).

4.3 Algorithms Speedup

We investigate the speedup for both Kruskal's algorithm and Boruvka's algorithm by parallelizing the sorting and merging steps. To this end, we evaluate the algorithms on different graphs with 10, 100, 1000, 5000, and 10000 nodes generated under a "random" setting with an edge probability of 0.1. We use different numbers of processes (2, 4, and 8) to run the parallel algorithms and compare them with the sequential algorithms. The speedup curves are shown in graph 1 and graph 2. Graph 1 shows the speedup curves for Kruskal's algorithm. We can see that the speedup increases as the number of processes increases, and as the size of the graph increases. This indicates that the parallel merge sort algorithm and the parallel union-find algorithm can effectively reduce the execution time of the sorting and merging steps. The speedup is higher for larger graphs because the sorting step dominates the overhead and takes 90% of the run-time. The parallel merge sort algorithm can achieve a significant speedup by dividing the work among multiple processes and threads. The speedup is lower for smaller graphs because the communication time among the processes becomes more significant compared to the computation time. Graph 2 shows the speedup curves for Boruvka's algorithm. We can see that the speedup also increases as the number of processes increases, and as the size of the graph increases. This indicates that the parallel merge sort algorithm and the parallel minimum edge finding algorithm can effectively reduce the execution time of the sorting and merging steps. The speedup is higher for larger graphs because the sorting step dominates the overhead and takes 80% of the run-time. The parallel merge sort algorithm can achieve a significant speedup by dividing the work among multiple processes and threads. The speedup is lower for smaller graphs because the communication time among the processes becomes more significant compared to the computation time.





5 References & Citations

- S. Chung and A. Condon, "Parallel implementation of Bouvka's minimum spanning tree algorithm," in Proceedings of International Conference on Parallel Processing, pp. 302–308, IEEE, 1996.
- V. Osipov, P. Sanders, and J. Singler, "The filter-Kruskal minimum spanning tree algorithm," in Proceedings of the Meeting on Algorithm Engineering & Experiments, pp. 52–61, Society for Industrial and Applied Mathematics, 2009.
- D. A. Bader and G. Cong, "Fast shared-memory algorithms for computing the minimum spanning forest of sparse graphs," Journal of Parallel and Distributed Computing, vol. 66, no. 11, pp. 1366–1378, 2006.