Parallel Calculation to find Minimum Spanning Tree

Said Kazimov
Institute of Computer Science
University of Tartu
Supervised by Mohammad Anagreh
said.kazimov@ut.ee

Abstract

This paper reviews the benefits of Parallelizing the Minimum spanning tree algorithm using Message Passing Interface. In this work, we perform parallel computing for the Kruskal’s algorithm for finding the minimum spanning tree with an adjacency list as a data structure. Overall, the results showed that by running the algorithm in a distributed manner we achieve efficiency improvement in its running time in comparison with standard case.

I. Introduction

A graph is a non-linear data structure consisting of nodes and edges. There are many use cases of graphs in our everyday life. One of these representations is a network, which includes paths in a city, telephone networks or even the websites, which have the networks to show recommendations to a user. Real-world networks are often very large in size resulting in graphs with several hundreds of thousands to billions of vertices and edges. Processing such large-scale graphs is challenging and require effectively harnessing the power of multiple nodes and devices [1].

The problem of optimizing the algorithms is a crucial part of Computer Science and many researchers devote their lives in this field. One possible solution is to run those algorithms in a distributed system, but it also depends on which type of algorithm is used. The algorithms with sequential operations are not suitable. To run the algorithm in parallel the operations should be performed separately, several at once.

In this paper, the graph algorithm that was chosen to be optimized is Kruskal’s Minimum Spanning Tree algorithm. Kruskal’s algorithm is a minimum-spanning-tree algorithm which finds an edge of the least possible weight that connects any two trees in the forest. It is a greedy algorithm in graph theory as it finds a minimum spanning tree for a connected weighted graph adding increasing cost arcs at each step. This means it finds a subset of the edges that forms a tree that includes every vertex, where the total weight of all the edges in the tree is minimized. If the graph is not connected, then it finds a minimum spanning forest (a minimum spanning tree for each connected component) [2].

For finding MST on large graphs, multiple nodes with distributed memory parallelism have to be employed in which the graph is partitioned across the nodes.

The parallelization of minimum spanning tree algorithms can be achieved by simply dividing the graph into small subgraphs, where the algorithm is applied separately to each of them and then the result is gathered [3].

In this paper as an MST algorithm, the Kruskal’s was chosen and was performed in a parallel way using Message Passing Interface.
II. Related work

The parallelization technique can be applied to different graph algorithms. The interesting example was proposed in [3], where for finding a minimum spanning tree the graph was divided to smaller parts and sent as an input to multiple nodes where the independent Burovka’s algorithm was performed. Then, the results from nodes were merged with the use of a novel hybrid merging algorithm. This experiment showed 24-88% performance improvement compared to ordinary Burovka’s algorithm which is implemented to a graph without partitioning.

This technique was also implemented in algorithms for finding the shortest path in a graph. In [4], to find the shortest path the Dijkstra algorithm was applied. For parallelizing the task and for realizing the distribution of various processes MapReduce programming model was used and different datasets were tested on the Hadoop cluster to get the running time and efficiency. By comparing with the traditional serial Dijkstra, the performance of the parallel Dijkstra algorithm achieved an evident improvement.

In [5] the researchers applied the parallelization to the breadth-first search and as a result, they got a generic algorithm that is unchanged from the sequential algorithm, requiring only the introduction of external (distributed) data structures for parallel execution. The generic approach helped to characterize the extension for distributed algorithms and define the patterns for using (and reusing) generic, object-oriented parallel software libraries.

The investigation in this paper is highly related to what was proposed in [3], as it is also about the parallelizing the graph algorithm. The main goal is also to calculate how much improvement can be obtained by parallelizing MST algorithm and is it efficient in terms of hardware usage.

III. Parallel Kruskal’s algorithm

The experiment that would be described below was implemented on a machine with Inter Core i5 processor that has 1.60 GHz speed and 16 GB of installed random access memory.

The MST algorithm that was used in research was Kruskal’s algorithm. It is an algorithm which finds a minimum spanning subtree of a weighted graph. Kruskal’s algorithm follows a greedy approach which finds an optimum solution at every stage instead of focusing on a global optimum [2]. The pseudocode for this algorithm is provided below:

```
Figure 1: Pseudocode of Kruskal’s algorithm [2]
```

The worst case complexity to run Kruskal’s is $O(E \log E)$ or $O(E \log V)$. Sorting of edges takes $O(E \log E)$ time. After sorting, we iterate through all edges and apply find-union algorithm. The find and union operations can take at most $O(\log V)$ time. So overall complexity is $O(E \log E + E \log V)$ time. The value of $E$ can be at most $O(V^2)$, so $O(\log V)$ are $O(\log E)$ same. Therefore, overall time complexity is $O(E \log E)$ or $O(E \log V)$.

To divide the algorithm to run in parallel processes the Message Passing Interface was used. MPI is a specification for the developers and users of message passing libraries. MPI primarily addresses the message-passing parallel programming
model: data is moved from the address space of one process to that of another process through cooperative operations on each process.

Below you can see the architecture of MPI which can handle different interconnections and protocols.

![Figure 2. MPI architecture [6]](image)

MPI provides collective communication pattern which was the main reason to be used in this experiment, because Collective communications allow us to more easily exchange information across all processes. Particularly, in my experiment, the graph was divided to subgraphs using MPI’s Scatter method which in a root process partitions the data to send pieces to every other process. Then, in the end to assemble the results in one process the Gather method was applied.

As a data structure the adjacency list was used. There are several differences between adjacency matrix and adjacency list, such as [7]:

I. Adjacency matrix:
- Uses $O(n^2)$ memory
- Check for presence or absence: $O(1)$
- Slow to iterate over all edges
- Adding or deleting a node: $O(n^2)$
- Adding a new edge: $O(1)$

II. Adjacency list:
- Memory usage depends on the number of edges, which might save a lot of memory if the adjacency matric is sparse
- Check for presence or absence: $O(k)$, where $k$ is the number of neighbor nodes
- Fast to iterate over all edges, as any node can be accessed directly
- Adding or deleting a node faster as we store only connected edges between the nodes.
- Adding a new edge: $O(1)$

The majority of the real world graph implications are sparse, which means each node has few edges and the graph is not complete, and in order to obtain the efficiency in memory, the adjacency list was chosen.

IV. Results and discussion

For running the algorithm, the pseudocode from Figure 1 was used, but with splitting the input graph to 3 subgraphs which then were broadcasted to 3 different processes. After that, the Kruskal’s algorithm was applied separately to each of the subgraphs. The expected result was to decrease the running time 3 times as the number of process were 3, but this result was not achieved because of the several overheads. Firstly, there are interprocess interactions which occur during running time of the code. In our case, this happens when the graphs are splitted into the 3 subgraphs using Scatter method and then, in the end, when we assemble the partitions using Gather method. In addition, idling may occur because of load imbalance, synchronization, or serial components.

In Figure 3 the difference between running the Kruskal’s algorithm in sequential and parallel manner can be observed. In the beginning when the amount of nodes is small the results are almost similar due to the overheads that was discussed earlier.
With increasing the number of nodes, the difference starts to grow too. The experiment with 600 nodes showed that time for finding MST for sequential algorithm takes in average 0.001 seconds per node, whereas for parallel it was roughly 0.0005 seconds per node. It was possible to decrease the running time twice by using the parallelization technique.

![Figure 3. The results from running the algorithm in a sequential and parallel manner](image)

### IV. Conclusion and Future work

This paper presents parallel calculation to find minimum spanning tree by applying Kruskal’s algorithm. The experiments showed that using this technique helps to decrease the running time of an algorithm almost twice. The results obtained from the experiment proves that using parallelization in graph algorithms is efficient as the running time of an algorithm can be reduced several times, depending on the number of processes applied for the distribution.

### References


