REPORT

Parallel and Concurrent Course Programming
Course Project Submission

Abstract

This project implements concurrent algorithms for finding Minimum Spanning Tree (MST) in a connected undirected graph. Two well known sequential algorithms “Prim's Algorithm” & “Boruvka Algorithm“ are state-of-the-art algorithms for finding the Minimum Spanning Tree. But these algorithms do not scale well for large number of nodes. That’s where concurrency comes into the picture. For this project we implemented concurrent “Prim's Algorithm” & “Boruvka Algorithm“ for finding the MST and analyzed their performance with their sequential counterparts and also with each other.

Introduction:

Minimum Spanning Tree (MST) problem is defined as follows: Given an undirected connected graph G with n vertices and m edges, the minimum spanning tree (MST) problem finds a spanning tree with the minimum sum of edge weights.

There are various algorithms to find minimum spanning tree in a graph. In our project we focused on two algorithms namely:

- **Prim’s Algorithm**
- **Boruvka’s Algorithm**

Parallel Prim’s Implementation:

Prim’s algorithms is a greedy algorithm. It starts with empty tree and grows till all the vertices are covered. It maintains two sets, one to store unvisited nodes and other which are already included in the tree. At each step, one vertex is selected and all the adjacent vertices to that vertex are updated in the array. Then finally the minimum value is selected from the array based on the weights of edges.

We implemented two ways to parallelize prim's algorithm using priority queues. In this approach we used fine-grained priority queues to insert and remove minimum element. Priority queues follow the property of heap, therefore the time complexity of insertion and deletion is $O(log V)$ where V is the number of nodes being added.

Overview of the implementation:

We used adjacency list to represent the graph. Each node consists of the list of adjacent vertices. Then we remove the minimum element from the priority queue using removeMin function which is sequential. As at a time we are removing only one element from the queue, we can do it sequentially in $O(1)$ because the minimum element of the queue is always the ROOT node. Later we iterate through all the adjacent vertices of ROOT node and calculate the distances parallely. The important part in minimum spanning tree is to find the least distance. We parallelized this by creating 'V' threads where $V = Number\ of\ Vertices\ in\ the\ graph$. Then for each vertex we add it's adjacent nodes parallely in the priority queue. In the priority queue, we used two kinds of locks, heap lock and node lock. Heap lock is used when we have to calculate the position of next node. Then for adding any node, we use locks for the node and it's parent node. We are using array based queue implementation therefore the child node can directly access it's variables without having to traverse through head node. For updating child and parent nodes, we used node locks, which locks only two nodes and therefore helps in reducing the contention significantly.

Time Complexities:

- The time complexity of priority queue insert is

  $O(log\ V)\ where\ V = Number\ of\ vertices$

- The time complexity of priority queue removeMin is $O(1)$

- The time complexity of MST is

  $O((V/p\ +\ E)\ logV)\ where\ p = \#threads,\ V\ = \#Vertices\ and\ E\ = \#Edges$ because the time complexity of insertion is $O(logV)$ and we are creating p threads.

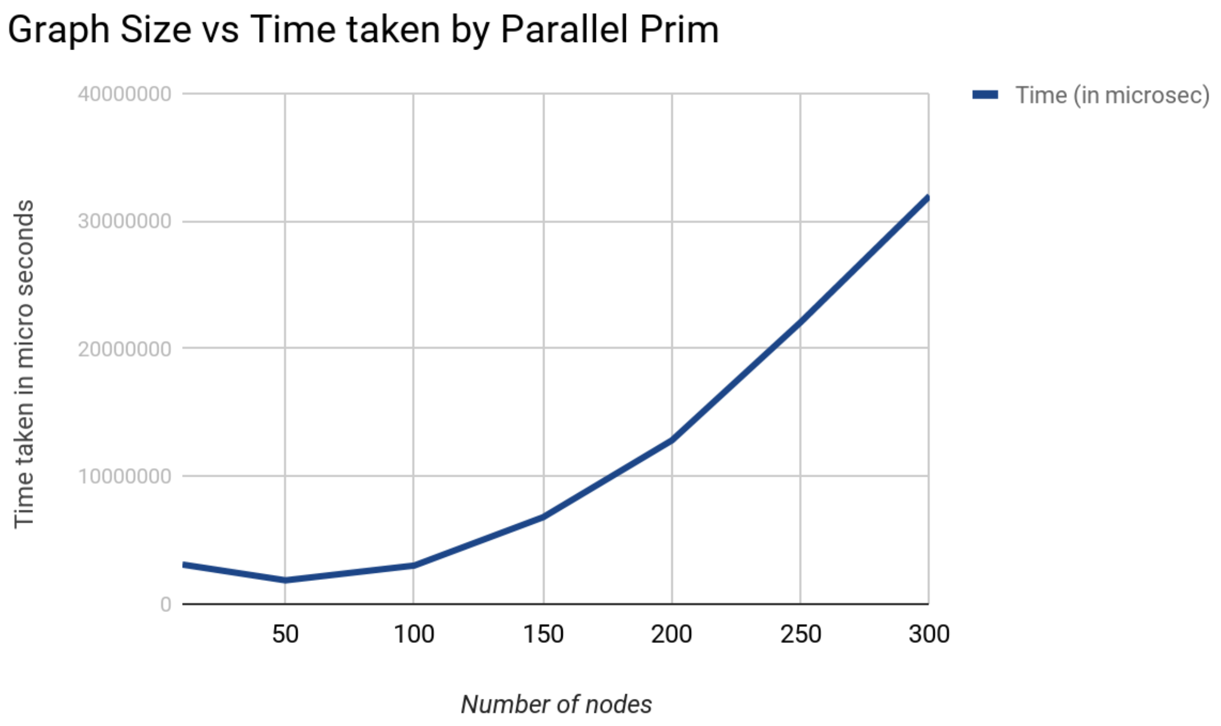
Analysis:

For testing purpose the graphs which were taken were complete graphs, therefore for each node N, there will be N*(N-1)/2 edges. All the results are based on complete graphs with varying number of vertices.

Parameters:

- Number of threads in Parallel Prim's = Number of vertices
- Testing was done on Intel i8 Processor
- Time is calculated using 'chrono high resolution clock' in C++
- Mutex Locks in C++ were used for locking.

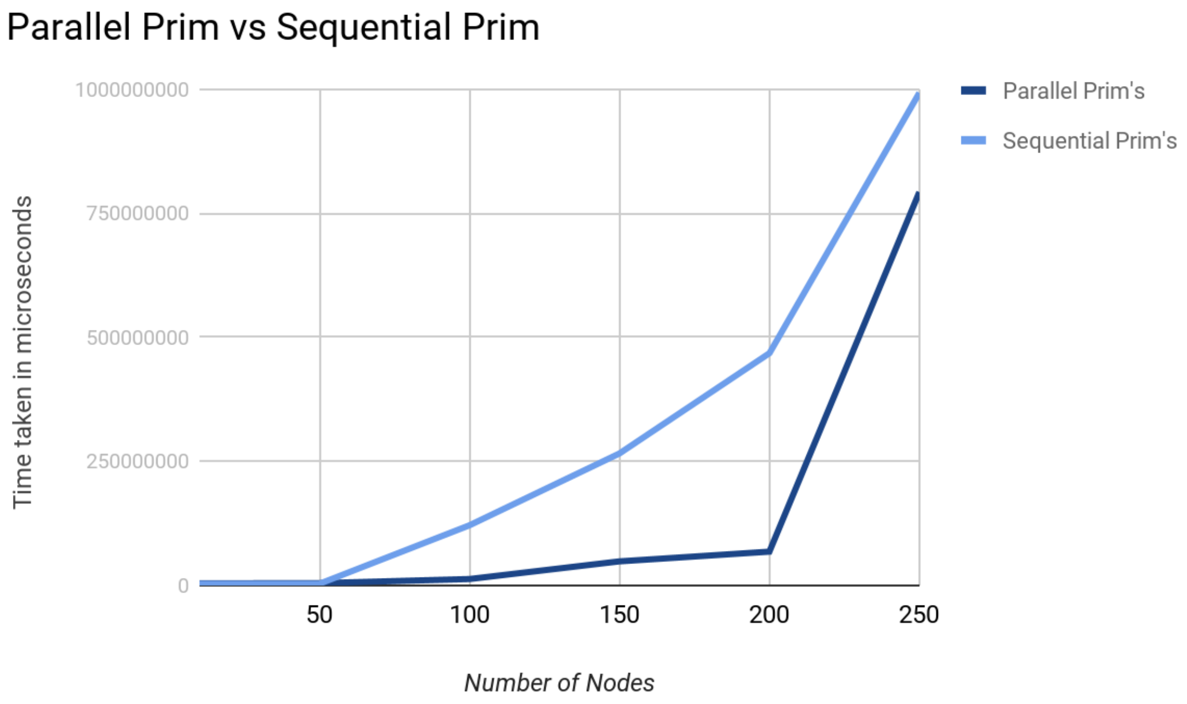
1. Graph size vs Time



Observations:

- As the number of nodes increase, the time taken also increases.
- The graph grows exponentially because more number of nodes result in more number of threading which leads to more overhead for the OS.

2. Time taken by Parallel Prim's vs Sequential Prim's Algorithm



Observations:

- As the number of nodes increase, the time taken by both the algorithms increase.

- When the number of nodes is smaller, the time taken by sequential code is less than or equal to the time taken by the parallel implementation. The reason for this can be the overhead caused due to creating threads. In all cases the number of threads created in parallel implementation is equal to the number of vertices in the graph.
- When the number of nodes is high, parallel implementation is significantly faster than the sequential implementation. This can be justified as the number of nodes increases, sequential execution will take time, whereas in parallel implementation the work is divided among say 'p' threads and the overhead caused by maintaining the threads is less than the improvement in performance.

Parallel Boruvka's Implementation:

What is Boruvka?

Boruvka's algorithm is also a Greedy algorithm. The algorithm begins by finding the minimum-weight edge incident to each vertex of the graph, and adding all of those edges to the forest. Then, it repeats a similar process of finding the minimum-weight edge from each tree constructed so far to a different tree, and adding all of those edges to the forest. Each repetition of this process reduces the number of trees, within each connected component of the graph, to at most half of this former value, so after logarithmically many repetitions the process finishes. When it does, the set of edges it has added forms the minimum spanning forest.

Overview of the implementation

We used adjacency list to represent the graph. Each node consists of the list of adjacent vertices. Initially each vertex is a component. Boruvka is divided into 3 parts

1) **Find-Min** : The algorithm begins by finding the minimum-weight edge incident to each component of the graph. This has been implemented parallely. When the program starts , the remaining components are divided equally among the threads and each thread update a vertex **cheapest** and stores the least weighted incoming edge for each component . In this particular function , finding the minimum edge among a set of edges for a particular node has also been done parallely using async and future.

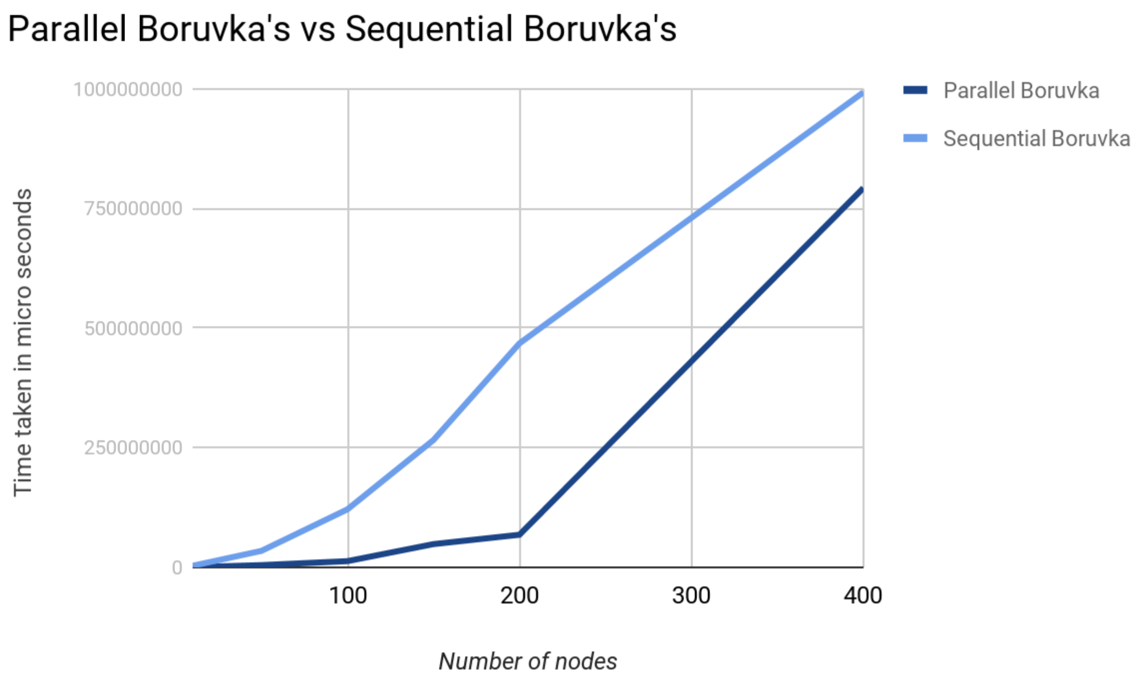
2) **Connected-Components** : From the induced subgraph of the edges found in step 1, find all the connected components and merge them into one using union find data structure. Since this involves many threads updating and modifying the union-find data structure simultaneously, this is implemented sequentially.
3) **Restructure graph :** Based on the components found in step 2 all the duplicated edges from the nodes in the components and all cycles are removed. This is executed parallely with each thread updating the structure of some predefined vertices.

This process is repeated till only one is component left and from that we will get the MST edges.

Analysis:

Parameters:

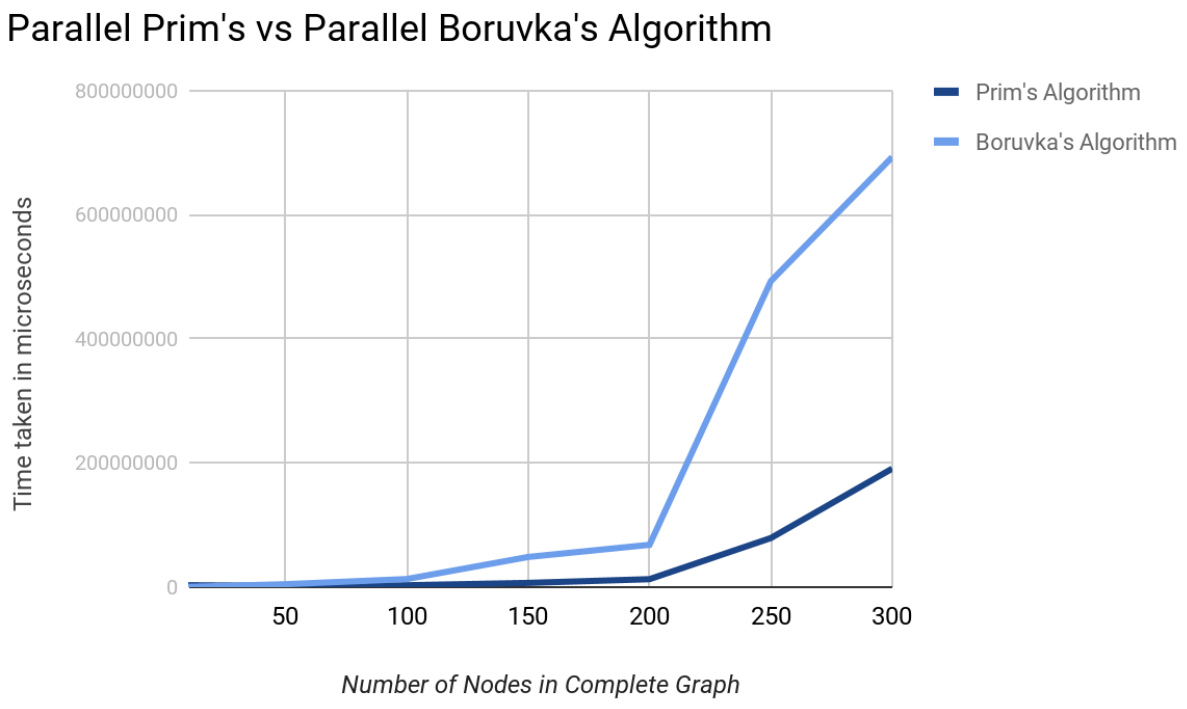
- Number of threads in Parallel Boruvka = N
- Testing was done on Intel i8 Processor
- Time is calculated using 'chrono high resolution clock' in C++
- Mutex and future<int> in C++ were using for synchronization.



Observations:

- As the number of nodes increase, the time taken by both the algorithms increase.
- When the number of nodes is smaller, the time taken by sequential code is less than or equal to the time taken by the parallel implementation. The reason for this can be the overhead caused due to creating threads. In boruvka, threads keep getting created and joined on the fly. There is no fixed number of threads in the beginning
- When the number of nodes is high, parallel implementation is significantly faster than the sequential implementation. This can be justified as the number of nodes increases, sequential execution will take time, whereas in parallel implementation the work is divided among say 'p' threads and the overhead caused by maintaining the threads is less than the improvement in performance.
- For more number of vertices , the bottleneck is at the finding connected-components part.

Comparison between Boruvka's Algorithm and Prim's Algorithm



Observations:

- The time taken by Prim’s algorithm is much less than the time taken by Boruvka’s Algorithm. A possible reason for that can be the way the algorithms are implemented. Prim’s algorithm focuses to parallelize the major chunk of data whereas in Boruvka’s algorithm two parts are executed parallelly and connected-component part is executed sequentially which gives rise to some amount of bottleneck.
- Another observation that can be made is fine-grain locks perform better than coarse grain. Although the algorithms are different, there is a significant difference in performance. Prim’s algorithm uses fine-grained locks whereas Boruvka’s algorithm uses coarse grained locks.
- The difference in runtime increases as the number of nodes increases.

References:

- https://www.cc.gatech.edu/~bader/papers/MST-JPDC.pdf
- https://hipcor.fatcow.com/hipc2009/documents/HIPCSS09Papers/1569250351.pdf
- https://en.wikipedia.org/wiki/Minimum_spanning_tree
- http://www.cs.tau.ac.il/~mad/publications/opodis2015-heap.pdf
- https://drum.lib.umd.edu/handle/1903/903