Parallel Minimum Spanning Tree

Edward Tallentire (edward.tallentire@postgrad.manchester.ac.uk)
MSc. Advanced Computer Science

***Abstract*—This paper will attempt to create an optimal parallelised Minimum Spanning Tree (MST) in Java. After a short literature review an algorithm will be selected for the MST design and implemented iteratively. Timing results will be obtained by running the Java program on the University of Manchester's mcore48 machine. The resulting values including speedup will be analysed for a greater understanding of the problematic overheads limiting execution time reduction.**

I. Introduction

Finding the minimum spanning tree of a graph is a problem that has been analysed for decades spouting many famous algorithms such as Kruskal's, Prim's and Boruvka's. Its real world uses are endless and a lot of money has been spent, within this paper is an attempt at designing and implementing a Minimum Spanning Tree program based on one of the above 3 algorithms. To decide which one, each algorithm will be shortly analysed for its potential for parallelisation and ease of parallelisation. This paper will not be a comprehensive parallel implementation of an entire algorithm due to certain constraints but can be used as a starting point for your own potential implementation.

II. Relevant Literature

This section contains a short investigation between 3 popular Minimum Spanning Tree algorithms and their suitability for parallelisation. The 3 algorithms in question are Prim's, Kruskal's and Boruvka's.

A. Prim

Prim's algorithm is stated as being greedy, meaning it always chooses the best possible option presented to itself at the current time, in an MST this means the shortest edge. An edge connects two nodes (vertices in a graph). The simple implementation of the Prim's algorithm has an overall complexity with adjacency list representation of $O(|V|^2)$ where V is the number of Vertices. (Medak, 2018)
This time complexity means that as the number of vertices increase, the time taken to calculate the minimum spanning tree increases extremely quickly. Parallelising this algorithm is tricky because the main loop of Prim's algorithm that checks against a list to see if an edge is already added, must be completed sequentially and therefore is unparallelisable.
This creates the problem covered in Hill and Marty (2008) whereby the amount of sequential code that is unparallelisable restricts potential speedup of the overall program. Because a main part of this algorithm is unparallelisable any optimisable part will have limited effects on runtime.

B. Kruskal

Kruskal's algorithm chooses the smallest edge that does not form a cycle and adds it to the MST, because of this it is also greedy.
Due to the checking of the smallest edges not forming a cycle, this algorithm also shows that a main part of it is inefficiently parallelisable as two edges cannot be added to a minimum edges set in case one of them forms a cycle with the other, the synchronisation needed to preserve correctness would cause a large overhead. Regarding the time complexity of this algorithm, Theorem 4.1 from (Karger, Klein, & Tarjan, 1995) proves the worst-case running time of the minimum-spanning-forest algorithm is $0(min\{n2, m \cdot log(n)\})$, the same as the bound for Boruvka's algorithm, where m is the number of edges in the graph and n is the number of vertices. If we assume the average amount of edges a vertex has is 2.5 then the time taken as more vertices are added increases much slower than the simplified Prim's algorithm.

C. Boruvka

Boruvka's algorithm is greedy because each node within the graph calculates its closest neighbour in the graph, then adds this to a set where repeated edges are deleted, each connected node within this set is treated as one node and then it loops again. Regarding the time complexity of this algorithm, (Nešetřil, Milková, & Nešetřilová, 2001) states that 'It is easy to implement the algorithm so that its complexity will be bounded by $C \cdot m \cdot log(n)$ (where m is the number of edges and C is a constant).'
This algorithms time complexity is the same as Kruskal's but unlike Prim and Kruskal, this algorithm shows signs of having high parallelisability.

III. Design & Implementation Phase

In this section we talk about designing and implementing the Sequential & Parallel MST.
Verified test data (Graph) was taken from GeeksForGeeks. [1] Taking the relevant literature analysis into consideration, Boruvka's algorithm will be used due to its lower time complexity and high proportion of parallelisability. The pseudocode for this algorithm can be seen on page 3 as Listing 1. During this stage a predefined tree (Fig 3 in appendix) was used, this allows the correctness of the MST to be verified. For recording results each program will be run 10 times, the mean will be taken and speedup calculated. The results will be viewable in the tables proceeding this. The graph G will be connected and undirected within development, this assumes the graph will also be connected and undirected in future cases.

A. Sequential Iteration Phase 1

The first real implementation used Boruvka's algorithm as a loose guideline for a primitive MST attempt, each node and edge were hard coded together with each node being designated a letter as their key for the map, this led to problems with iterating through the node maps when searching for nodes with related edges and was later changed. The average sequential execution time recorded in this iteration was 0.003132122

B. Final Sequential Iteration Phase

The final sequential program can be found in the MSTreeFinalSeq file.
The problem of assigning a "String letter" to each node as the key was fixed by assigning them String numbers instead, this was done by using the Integer.toString() method to assign their keys in a for loop, for instance NodeA is now thought of as Node1. Assigning edges to nodes was made simpler by adding edges to a list called 'mstEdges' with the method 'addEdges' in Main, this method takes "String start, String end, int weight" as constructors and then adds the edge to the shared 'mstEdges' list, the edges in this list are then, through the use of 'withEdges' method in MSTreeFinalSeq, added to a shared Map to for all methods to access. So if for instance we want to find the edge that connects two nodes (Node1 & Node2), we can iterate through this list to find an edge with start node1 and end node2 or start node2 and end node1, as this edge is the same the iteration will only return one edge. When accessing this edge the methods in MSTEdge allow us to find out many things about it such as the values stated above or the weight.

The method trimTree in Listing 2 is the major currently sequential method that executes Boruvkas MST algorithm. It creates a List of Minimum Edges with the findMinEdges method, then the returned list is passed into the getNodesForMinEdges method to create a Set of nodes that are merged together. After this the shared old Minimum Edges list from the last iteration is appended alongside the newly created Minimum Edges to a new list of edges called combinedMinEdges. This method returns a new MSTreeFinalSeq that contains the Merged Nodes and Combined Minimum Edges.
The average sequential execution time recorded in this iteration was 0.0061371891, 0.003s slower than the original sequential result, but for the amount of progress made towards a professional looking MST implementation this result is acceptable.

C. Parallel Iteration Phase 1

Table 1 documents the performance results of Minimum Spanning Tree Iteration Phase 1.
The first and most obvious part to parallelise about a MST is minimum edge finding, in this program that is done by the findMinEdges method in MSTree. The findMinEdges method creates a threadpool and assigns that thread pool a limit of threads that corresponds to the input taken in main.
This method loops through all the nodes in the Set of nodes, and for each node the threadpool assigns a task to a thread through the use of the submit function which creates a new runnable task for each thread, this task compares the edges for each node and returns the minimum one to a shared list, synchronisation of writing to the list is not needed as each thread is assigned one node, no nodes are ever repeated *in one iteration* and duplicate edges are removed. This is the only parallelisation implemented for the first phase.

D. Parallel Iteration Phase 2

The biggest part of the algorithm that is parallelisable starts on line 6 of Listing 1: "For each edge uv of G:".
This can be accomplished like so, you have a component class and at the start a list of components are initilized, one for each node, these components are then merged and their edges retained only to components outside of them.
This changes the existing tree so that all components that are merged after a step are like one "node".

[1] https://www.geeksforgeeks.org/boruvkas-algorithm-greedy-algo-9/

```
Input: A graph G whose edges have distinct weights
Initialize a forest F to be a set of one-vertex trees, one for each vertex of the graph.
While F has more than one component:
  Find the connected components of F and label each vertex of G by its component
  Initialize the cheapest edge for each component to "None"
  For each edge uv of G:
    If u and v have different component labels:
      If uv is cheaper than the cheapest edge for the component of u:
        Set uv as the cheapest edge for the component of u
      If uv is cheaper than the cheapest edge for the component of v:
        Set uv as the cheapest edge for the component of v
   For each component whose cheapest edge is not "None":
     Add its cheapest edge to F
 Output: F is the minimum spanning forest of G.
```

Listing 1. Boruvka's Algorithm Pseudocode (*Borůvka's algorithm*, 2018)

This is then repeated while 'mstTree.getNodes().size() != 1', the number of nodes inside the merged Tree does not equal 1, meaning until all components/nodes are connected continue iterations. Implementing this was harder than expected but can be seen in the method withUpdateEdgesThread in the MSTree class with use of the WithUpdateEdgesThread class. This did introduce a problem into the program.

Correctness Problem:
After parallelising this section of the algorithm incorrect tree builds started to occur. This could be because of the order in which the minEdges get processed. So if we take the graph in Figure 3 for instance, we have a link between 8, 2 and 3 and 4, 2 and 3, if the edges are in the right order (which is a random chance) it creates one node for 8 and 2, then adds 2 and 3, then adds 3 and 4, but if they are in the order as above it will create one node for 8 and 2 then one for 3 and 4 and then add the 2 and 3 to each of these nodes instead of, effectively merging these two nodes together like it ought to. The two code snippets in Listings 3 & 4 show how this problem was tackled, but the re-implemented method was unable to entirely fix the problem, only decrease the occurrence of a incorrect result.

Table 2 documents the performance Results of Minimum Spanning Tree Iteration Phase 2.

E. Implementation of random graph

The code for the implementation of a random graph can be seen in listing 5. The random graph takes a number of edges and nodes, the number of edges cannot be less than the number of nodes -1, as the graph cannot be unconnected. There is one loop for creating new nodes and one loop for adding edges with use of the addEdges method.
One thing to note is that if the number of nodes inputted is below a 1/4 of the amount of edges, then the minimum edges all become 1's or 2's.

IV. Performance Results & Analysis

The sequential runtimes for increasing edges can be seen in the appendix as Table III. These sequential runtimes are used to calculate the respective speedups in Tables I & II. In table I the most encouraging sign is the increase in speedup as the number of edges increase, whilst having a speedup of lower than 1 is not ideal it shows the cost of creating a new Runnable class for 8 threads quite well. Due to the nature of this algorithm an optimal number of threads may be suggested that could potentially raise the speedup to above 1, this number could be calculated by dividing the number of nodes by 4 as for each node within the MST a minimum edge is calculated, then these nodes are joined... this rapid decrease in the amount of nodes in the graph is why dividing by 4 was suggested but this number is arbitrary. Quartering the amount of nodes for a number of threads is probably not optimal but a good mid point to potentially increase speedup whilst decreasing overheads such as idle thread time, time spend creating threads and scheduling threads.

It is important to note that this parallel version provides a correct minimum spanning tree 100% at the time of writing and nothing has currently been found that disproves this, the next version does not give a correct minimum spanning tree 100% of the time.

TABLE I
PERFORMANCE RESULTS FOR THE FINAL VERSION OF PARALLEL ITERATION 1 ON A RANDOM GRAPH OF INCREASING EDGES

Edges	Run time avg	Speedup
50	0.025015649	0.50961668
400	0.174060319	0.53709091
800	0.199373100	0.71780748
1200	0.270882075	0.72520364
1600	0.317552803	0.76845206
2000	0.345988465	0.84929502
2400	0.392105598	0.87942478
2800	0.446264837	0.87639078

TABLE II
PERFORMANCE RESULTS FOR THE FINAL VERSION OF PARALLEL ITERATION 2 ON A RANDOM GRAPH OF INCREASING EDGES

Edges	Run time avg	Speedup
50	0.03223470420	0.395486551
400	0.1689246305	0.55341968
800	0.2913974880	0.491121265
1200	0.35595927725	0.551873994
1600	0.7411506695	0.329250335
2000	0.6997204555	0.419948112
2400	0.76667639625	0.449769136
2800	0.907119664	0.431147516

In Table II the speedup shown is less encouraging, coupled with the frequency to get an incorrect MST, means that this parallel version needs to be reworked in the future with a potentially new perspective to parallelisation.

Fig. 1.

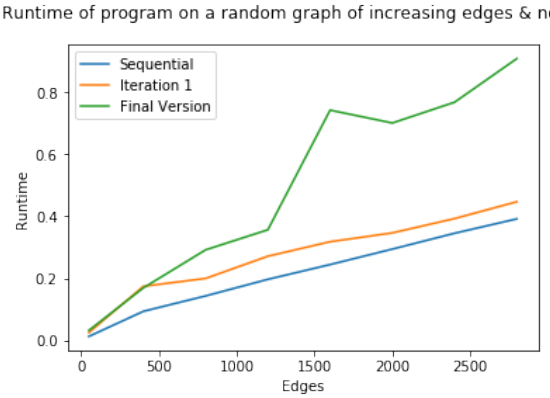
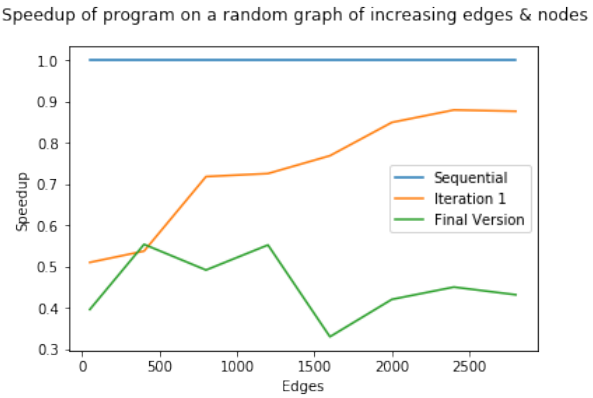


Figure 1 also gives a good indication of the overheads that arrive due to parallelisation of the getMinEdgeForNode method. Iteration 1 and the Sequential version are almost parallel lines. Unlike the final version where the synchronisation of updating edges clearly takes its toll giving an unpredictable line.

Fig. 2.



Given that the speedup was steadily increasing as the number of edges was increasing Fig 2. dampers the spirits as the speedup line for Iteration 1 can be seen to gradually stop increasing, the final two values in the table show a -0.0031 speedup. Load imbalance between created threads isn't much of an issue with this algorithm as each calculation is relatively simple and the main thread does merging in serial. Some load imbalance that hasn't been dealt with is when for instance only 3 nodes are left within the graph, 8 threads may be created but only 3 threads will be doing any work whilst the other 5 remain idle.

V. SUMMARY

In summary three relevant minimum spanning tree algorithms were reviewed and one was chosen due to its ease and depth of parallelisability. Then a minimum spanning tree based on this algorithm (Boruvka) was designed & implemented with a random set up, the first parallel iteration showed the most promise with the second and final iteration falling short. Any future work should continue on from Parallel Iteration 1 whilst taking into account the lessons learnt from implementing Parallel Iteration 2.

REFERENCES

Boruvka's algorithm. (2018, Aug). Retrieved from https://www.geeksforgeeks.org/boruvkas-algorithm-greedy-algo-9

Borůvka's algorithm. (2018, Oct). Wikimedia Foundation. Retrieved from https://en.wikipedia.org/wiki/Borůvka's_algorithm

Hill, M. D., & Marty, M. R. (2008, July). Amdahl's law in the multicore era. *Computer*, *41*(7), 33-38. doi: 10.1109/MC.2008.209

Karger, D. R., Klein, P. N., & Tarjan, R. E. (1995, Jan). A randomized linear-time algorithm to find

minimum spanning trees. *Journal of the ACM*, *42*(2), 321–328. doi: 10.1145/201019.201022

Medak, J. (2018, Apr). Review and analysis of minimum spanning tree using prim's algorithm. *International Journal of Computer Science Trends and Technology (IJCST)*, *6*(2), 34–39.

Nešetřil, J., Milková, E., & Nešetřilová, H. (2001). Otakar borůvka on minimum spanning tree problem translation of both the 1926 papers, comments, history. *Discrete Mathematics*, *233*(1-3), 3–36. doi: 10.1016/s0012-365x(00)00224-7

APPENDIX

How to run the jar files:

The inputted Edge Number must be more than Node Number -1.

For the sequential program the command to run is:
java -jar MiniProjectSeqFinal.jar 1 (Edge number) (Node Number)

For the Parallel Iteration 1 (working version):
java -jar MiniProjectIter1Final.jar 1 (Edge number) (Node Number)

For the Final Parallel Version:
java -jar MiniProjectFinal.jar 1 (Edge number) (Node Number)

Fig. 3. Boruvka Tree Used for algorithm (*Boruvka's algorithm*, 2018)

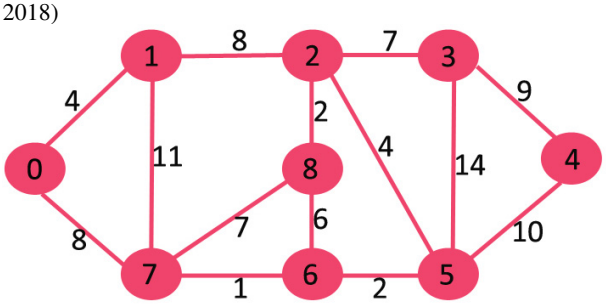


TABLE III
PERFORMANCE RESULTS FOR THE FINAL SEQUENTIAL VERSION ON A RANDOM GRAPH OF INCREASING EDGES

Edges	Run time avg
50	0.012748392
400	0.093486215
800	0.143111503
1200	0.196444668
1600	0.244024106
2000	0.293846284
2400	0.344827380
2800	0.391102390

```
public MSTreeSeqWorkingCopy trimTree() {
  List<MSTEdge> minEdges = findMinEdges();
  Set<MSTNode> mergedNodes = getNodesForMinEdges(minEdges);

  List<MSTEdge> combinedMinEdges = new ArrayList<>();
  combinedMinEdges.addAll(this.minEdges);
  combinedMinEdges.addAll(minEdges);

  return new MSTreeSeqWorkingCopy(mergedNodes, combinedMinEdges).withUpdateEdges(this.
  nodes2Edges);
}
```

Listing 2. trimTree

```
private Set<MSTNode> getNodesForMinEdges(List<MSTEdge> edges) {
  Set<MSTNode> mstNodes = new HashSet<>();

  for (MSTEdge edge : edges) {
      MSTNode mstNode = null;
      for(MSTNode node:mstNodes) {
          if(node.getNodes().contains(edge.getStartNode())
              || node.getNodes().contains(edge.getEndNode())) {
              mstNode = node;
              break;
          }
      }
      if(mstNode == null) {
          mstNode = new MSTNode();
          mstNodes.add(mstNode);
      }
      mstNode.addNode(edge.getStartNode());
      mstNode.addNode(edge.getEndNode());
  }
  return mstNodes;
}
```

Listing 3. Correctness Problem

```
private Set<MSTNode> getNodesForMinEdges(List<MSTEdge> edges) {
    List<MSTEdge> mstEdges = new ArrayList<>(edges);

    Set<MSTNode> mstNodes = new HashSet<>();

    while(mstEdges.size() > 0) {
      MSTEdge edge = mstEdges.get(0);
      mstEdges.remove(0);

      MSTNode mstNode = new MSTNode();
      mstNode.addNode(edge.getStartNode());
      mstNode.addNode(edge.getEndNode());

      mergeEdgesIntoNode(mstNode, mstEdges);

      mstNodes.add(mstNode);
    }

    return mstNodes;
  }

  private MSTNode mergeEdgesIntoNode(MSTNode mstNode, List<MSTEdge> edges) {
    boolean hadChanges = false;

    List<MSTEdge> toBeRemoved = new ArrayList<>();

    for(MSTEdge edge: edges) {
      if(mstNode.getNodes().contains(edge.getStartNode()) || mstNode.getNodes().contains(
    edge.getEndNode())) {
        mstNode.addNode(edge.getStartNode());
        mstNode.addNode(edge.getEndNode());

        toBeRemoved.add(edge);
        hadChanges = true;
      }
    }

    edges.removeAll(toBeRemoved);

    if(hadChanges) {
      return mergeEdgesIntoNode(mstNode, edges);
    } else {
      return mstNode;
    }
  }
```

Listing 4. Correctness Solution

```
  private MSTree initStartMSTree(int numEdges, int numNodes) {
    Set<MSTNode> mstNodes = new HashSet<>();
    List<MSTEdge> mstEdges = new ArrayList<>();

    for (int i = 0; i < numNodes; i++) {
      mstNodes.add(new MSTNode(String.valueOf(i)));
    }

    Random random = new Random();
    for(int i = 0; i<numEdges; i++) {
      addEdges(mstEdges, Integer.toString(random.nextInt(numNodes)),
            Integer.toString(random.nextInt(numNodes)), random.nextInt(14));
    }

    return new MSTree(mstNodes).withEdges(mstEdges);
  }
```

Listing 5. Random Graph Generator