

UNIVERSITY OF LJUBLJANA  
FACULTY OF COMPUTER AND INFORMATION SCIENCE

BELLMAN FORD ALGORITHM USING HADOOP  
MAP REDUCE

Author: Luka Golinar

Mentor: Matevž Jekovec

Date: 25.09.2014

## 1. INTRODUCTION

The technology of today is all about fast data transfer with as less effort and time as possible. We strive to make programs that are as responsive as possible and make for a great end-user experience. Imagine you having a bunch of computer servers and would like to know which one is the closest, what route to take to get to each and every one of them the fastest and cheapest way possible. How can we accomplish this? Introducing the Bellman-Ford (Sometimes also called Bellman-Ford-Moore) algorithm. Similar to Dijkstra's algorithm, it calculates the shortest path from a source vertex to all the other vertices in a weighted, directional graph. It runs slower than Dijkstra, but it's more versatile, as it enables negative circle detection.

Worst case performance:

| Dijkstra       | Bellman-Ford            |
|----------------|-------------------------|
| $O( V  *  E )$ | $O( E  +  V  \log  V )$ |

But can we make a better, faster algorithm, or improve the existing ones?

## 2. BELLMAN-FORD

First, I'd like to say a word or two about the algorithm itself. Bellman-Ford uses the technique of relaxation. First we expand all the nodes and set the value of the source node to 0 and all the rest to infinity (or a max possible value in our observatory environment). Then we relax all the edges repeatedly from the source vertex, until all the nodes are visited. We expand each node and set it's value depending on the value of the edge and the value from the node we were expanding. The code below is a pseudo code demonstrating the algorithm in action.

```

function BellmanFord(list vertices, list edges, vertex source)

::weight[],predecessor[]

// This implementation takes in a graph, represented as
// lists of vertices and edges, and fills two arrays
// (weight and predecessor) with shortest-path
// (less cost/weight/metric) information

// Step 1: initialize graph
for each vertex v in vertices: do
    if v is source then
        | weight[v] := 0;
    else
        | weight[v] := infinity
    end
    predecessor[v] := null;
end
// Step 2: relax edges repeatedly
for i from 1 to size(vertices)-1: do
    for each edge (u, v) with weight w in edges: do
        if weight[u] + w < weight[v]: then
            | weight[v] := weight[u] + w;
            | predecessor[v] := u;
        else
            end
        end
    end
// Step 3: check for negative-weight cycles
for each edge (u, v) with weight w in edges: do
    if weight[u] + w < weight[v]: then
        | error "Graph contains a negative-weight cycle";
    else
        end
    end
end
return weight[], predecessor[];

```

### 3. INTRODUCING HADOOP MAP REDUCE

Imagine having tens of thousands of nodes. Computing the paths from each of them would take a lot of time. Thus, map reduce! I tried Bellman Ford algorithm with both sequential version and the map reduce version. I used the Hadoop implementation of the technique.

So how does map reduce actually work? Map reduce fits into Big data concept of saving a huge amount of data, replacing the standard SQL table oriented database system. Map reduce has a few stages: Splitting, mapping, shuffling and reducing. Mapping and reducing are the two stages that are going to be important for us. The below diagram shows a word count program making use of map reduce.

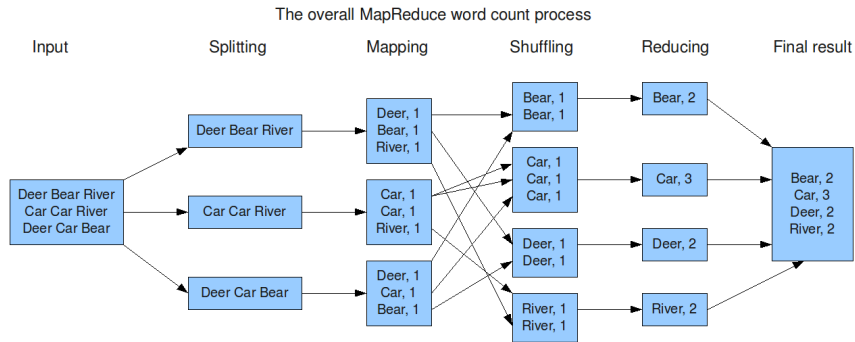


Figure 1: The process of counting words using map reduce

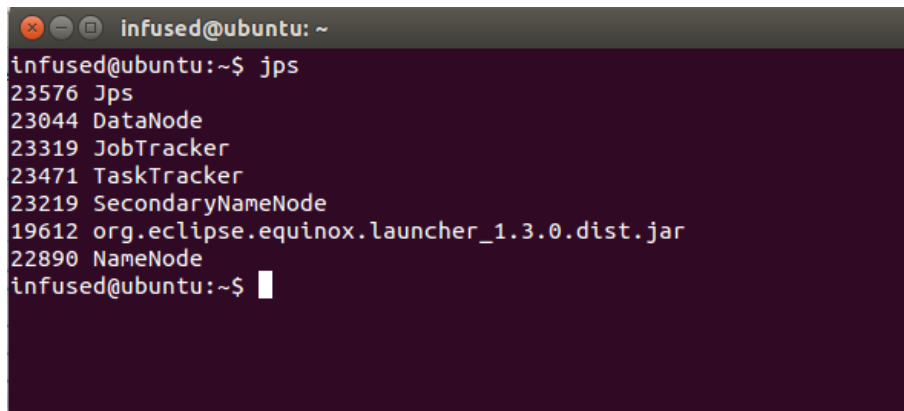
## 4. HADDOOP AND BELLMAN-FORD

Implementing a program to work along side Hadoop is pretty easy. We will be using Single Node setup. This means everything is running on one local machine. There are several other possibilities of using hadoop, like cluster setup (the main idea behind hadoop). This means, that we have one machine designated as the NameNode and another machine as the Resouce-Manager. These are the masters. The rest of the machines in the cluster are slaves and act both as DataNodes and NodeManager. I, however, won't be going too much into details about cluster setup.

Lets start implementing BF into hadoop.

- Requirements:
  - I used ubuntu 14.04 as my OS, but any UNIX based system will do
  - Java SDK with Eclipse installed
  - Hadoop up and running (single node setup, of course)

I will not be going into details of how to install hadoop and eclipse with Java onto your machine, as there are several tutorials on the web (link in the reference page). Once you have all the requirements, you are all set to go. Hadoop-a-loop!

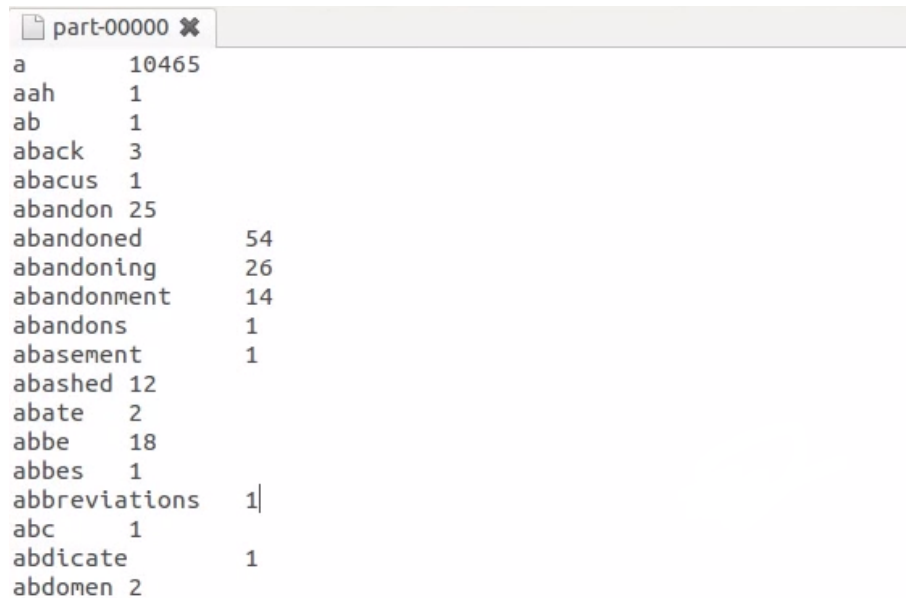
A terminal window titled 'infused@ubuntu: ~' showing the output of the 'jps' command. The output lists several Hadoop daemons running on the local machine: Jps (PID 23576), DataNode (PID 23044), JobTracker (PID 23319), TaskTracker (PID 23471), SecondaryNameNode (PID 23219), org.eclipse.equinox.launcher\_1.3.0.dist.jar (PID 19612), and NameNode (PID 22890). The prompt 'infused@ubuntu:~\$' is visible at the end of the output.

```
infused@ubuntu:~$ jps
23576 Jps
23044 DataNode
23319 JobTracker
23471 TaskTracker
23219 SecondaryNameNode
19612 org.eclipse.equinox.launcher_1.3.0.dist.jar
22890 NameNode
infused@ubuntu:~$
```

Figure 2: Local daemons running.

## 4.1 PROBLEMS

The hadoop official site gives us an implementation of a simple program called WordCount, simulating Figure 1. But how does it work? We take a text file (a book, for example) and in the map stage, we read line after line. Afterwards we send every read word along side its hard coded value of 1 to the reducer stage, which just increments the values. The communication between the mapper and the reducer is always in form K, V. K standing for key and V for value.



```
part-00000 ✕
a      10465
aah    1
ab     1
aback  3
abacus 1
abandon 25
abandoned 54
abandoning 26
abandonment 14
abandons 1
abasement 1
abashed 12
abate  2
abbe   18
abbes  1
abbreviations 1|
abc    1
abdicate 1
abdomen 2
```

Figure 3: The output file after our sample word count program.

So then how can we represent a graph data structure in such a way?

## 4.2 SOLUTION: ITERATIVE HADOOP MAP REDUCE

So the idea is this: For each node expansion mentioned earlier in the BF algorithm, we will make use of one iteration of map and reduce (and all the other not mentioned stages). We will represent our graph data in the following way:

```
ID | EDGES | EDGES_WEIGHT | DISTANCE_FROM_SOURCE | NODE_COLOR
```

Our key in this case is going to be the ID of the node and the value a Text type of everything else in the line. So let's say we have the following structure:

```
0 1,2,3 5,8,-4 0 GRAY
1 0 -2 Integer.MAX_VALUE WHITE
2 3,1 9,-3 Integer.MAX_VALUE WHITE
3 1,4 7,2 Integer.MAX_VALUE WHITE
4 2,0 7,6 Integer.MAX_VALUE WHITE
```

This would translate in the graph shown below:

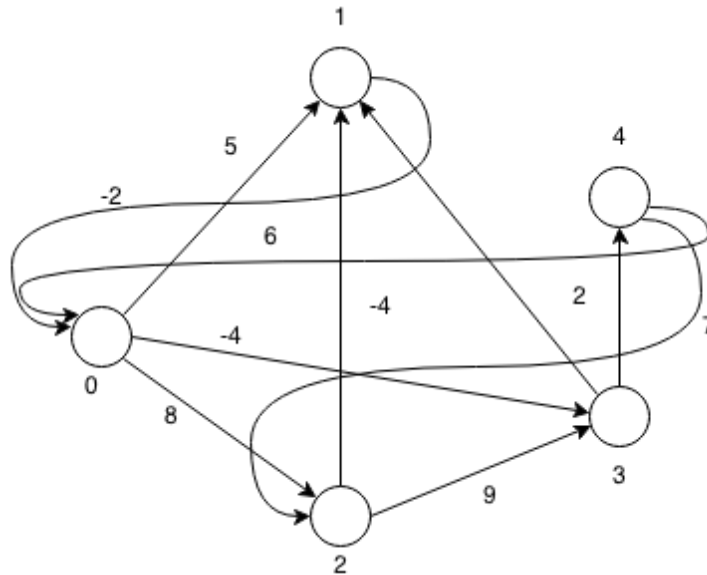


Figure 4: Graph visualization



Like mentioned before, the first column is the ID, separated by a tab are the edges connected to our node. The next value, this time separated by one space character are the weights corresponding by position for each of the edges in column 2. The next value is the max value we assigns every value. This is the infinity value I was talking about before. Lastly, the color tells us whether or not the node has been visited yet.

- WHITE: Not yet visited node.
- GRAY: Already visited but not expanded node.
- BLACK: We expanded this node and are now done with it.

So if we take our graph and run it in hadoop we will need 3 iterations for hadoop to correctly calculate the shortest paths. This basically means, that we will gradually expand node after node. First, we will look at every node with color GRAY, set all it's edges equally to color GRAY and send the result (once again, ID as the key and all the rest as a Text value) to the reduce map. The mapper emits two nodes with the same ID. One that has NULL edges and one that does. The reducer job is then to merge these two nodes into one node with not NULL edges and the right distance. The result is then written to a text output. If we simulate this on a real life graph, each iteration can represent a node expansion. So in this case, we will expand the vertex 0 and set its edges to color GRAY.

The first iteration then gives us the following result:

```
0 1,2,3 5,8,-4 0 BLACK
1 0 -2 5 GRAY
2 3,1 9,-3 8 GRAY
3 1,4 7,2 -4 GRAY
4 2,0 7,6 2147483647 WHITE
```

We can see, that node 1, 2 and 3 are coloured GRAY, because they are connected to node 0. In the second iteration we repeat step one, described above. We read line by line of this new file that was given as an output of the first iteration and check for any GRAY nodes. We then emit all the GRAY nodes as BLACK nodes and expand their edges setting the colour from WHITE to GRAY. If the nodes colour to be expanded is set to BLACK, we only look at the distance and change it accordingly. This repeats until all the nodes are either coloured BLACK or WHITE(if the graph is not connected).

---

```

/* If the node is GREY, we emit all of the node edges */
if (node.getColor() == Node.Color.GRAY) {
    int counter = 0;
    for (int v : node.getEdges()) {
        Node vnode = new Node(v);
        int w_e = node.getWeights().get(counter);

        /* Set the distance of the edges -- Main BellmanFord
           Algorithm */
        if(vnode.getDistance() > node.getDistance() + w_e)
            vnode.setDistance(node.getDistance() + w_e);
        counter++;

        /* Set node as visited */
        vnode.setColor(Node.Color.GRAY);
        output.collect(new IntWritable(vnode.getId()), new
            Text(vnode.toString()));
    }
    /* We're done with this node now, color it BLACK */
    node.setColor(Node.Color.BLACK);
}

/* We always emit the input node. If it was GREY, we color it
   BLACK */
output.collect(new IntWritable(node.getId()), new
    Text(node.toString()));

```

---

After the second and the third iteration we get the following result.

```

0 1,2,3 5,8,-4 0 BLACK
1 0 -2 3 BLACK
2 3,1 9,-3 5 BLACK
3 1,4 7,2 -4 BLACK
4 2,0 7,6 -2 BLACK

```

Meaning the distances from node 0 are 1:3, 2:5, 3:-4 and 4:-2. If we do a quick calculation by hand on the following graph, we can see that the results are indeed correct.

I have used a hard-coded variable to tell hadoop how many times it has to iterate, but since a recent patch, we can implement global counters that do tell hadoop when to stop. I have yet to make use of that feature in my program.

## 5. RESULTS

In this task, we've tried to make use of parallel processing to try and speed-up the calculation of shortest paths Bellman Ford does. If we have a bunch of computers that all together do data computation we ought to, in the end, get a much faster calculation of really big graphs. But is that really the case?

First, I tried to run the BF algorithm with the standard sequential version and got the following results:

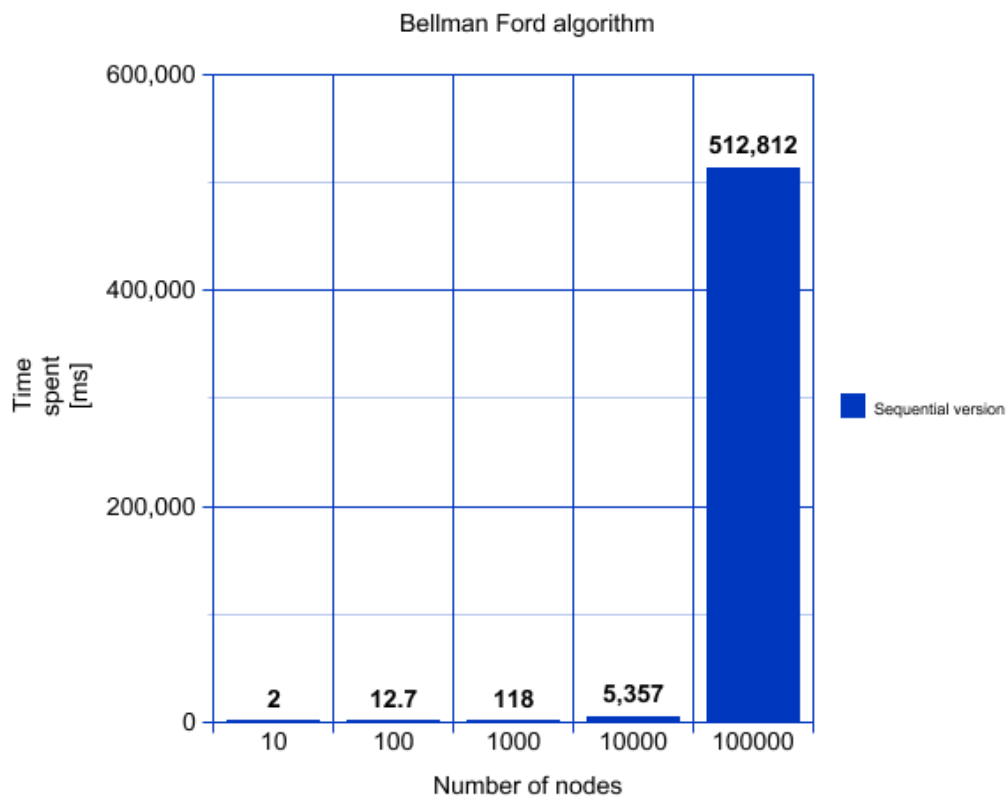


Figure 5: Sequential version measurements.

We can clearly see that time gradually grows with the number of given nodes. With 100K nodes it takes a very long time, to calculate the shortest paths to all of them.

Now let's take a look at the measurements in parallel version. There are several things we can measure in hadoop. If we take a look at CPU time, shown in the picture below, we can see that it's clearly a lot more than the standard version.

| Kind    | Total Tasks(successful+failed+killed) | Successful tasks | Failed tasks | Killed tasks | Start Time           | Finish Time                 |
|---------|---------------------------------------|------------------|--------------|--------------|----------------------|-----------------------------|
| Setup   | 1                                     | 1                | 0            | 0            | 27-Sep-2014 12:45:14 | 27-Sep-2014 12:45:15 (1sec) |
| Map     | 2                                     | 2                | 0            | 0            | 27-Sep-2014 12:45:15 | 27-Sep-2014 12:45:18 (2sec) |
| Reduce  | 1                                     | 1                | 0            | 0            | 27-Sep-2014 12:45:18 | 27-Sep-2014 12:45:27 (8sec) |
| Cleanup | 1                                     | 1                | 0            | 0            | 27-Sep-2014 12:45:27 | 27-Sep-2014 12:45:28 (1sec) |

|                             | Counter  | Map         | Reduce     | Total       |
|-----------------------------|--|-------------|------------|-------------|
| Job Counters                | Launched reduce tasks  | 0           | 0          | 1           |
|                             | SLOTS_MILLIS_MAPS  | 0           | 0          | 8,080       |
|                             | Total time spent by all reduces waiting after reserving slots (ms) | 0           | 0          | 0           |
|                             | Total time spent by all maps waiting after reserving slots (ms)    | 0           | 0          | 0           |
|                             | Launched map tasks   | 0           | 0          | 2           |
|                             | Data-local map tasks   | 0           | 0          | 2           |
|                             | SLOTS_MILLIS_REDUCEs   | 0           | 0          | 8,730       |
| File Input Format Counters  | Bytes Read   | 5,051       | 0          | 5,051       |
| File Output Format Counters | Bytes Written  | 0           | 3,367      | 3,367       |
| FileSystemCounters          | FILE_BYTES_READ  | 0           | 3,685      | 3,685       |
|                             | HDFS_BYTES_READ  | 5,259       | 0          | 5,259       |
|                             | FILE_BYTES_WRITTEN   | 116,867     | 60,155     | 177,022     |
|                             | HDFS_BYTES_WRITTEN   | 0           | 3,367      | 3,367       |
| Map-Reduce Framework        | Map output materialized bytes                                      | 3,691       | 0          | 3,691       |
|                             | Map input records  | 101         | 0          | 101         |
|                             | Reduce shuffle bytes   | 0           | 3,691      | 3,691       |
|                             | Spilled Records  | 101         | 101        | 202         |
|                             | Map output bytes   | 3,477       | 0          | 3,477       |
|                             | Total committed heap usage (bytes)                                 | 317,718,528 | 62,390,272 | 380,108,800 |
|                             | CPU time spent (ms)  | 370         | 950        | 1,320       |
|                             | Map input bytes  | 3,367       | 0          | 3,367       |
|                             | SPLIT_RAW_BYTES  | 208         | 0          | 208         |

Figure 6: An example of the HDFS GUI data interface.

But we should take note, that CPU time consists with both the reducer stage and the map stage (which basically reads all the data from a input file line by line and then passes it to the Reduce stage). The main algorithm happens in the latter. So how much is the real time used to calculate all these paths?

The graph in Figure 7. shows the average times of both the map and reduce function and their total.

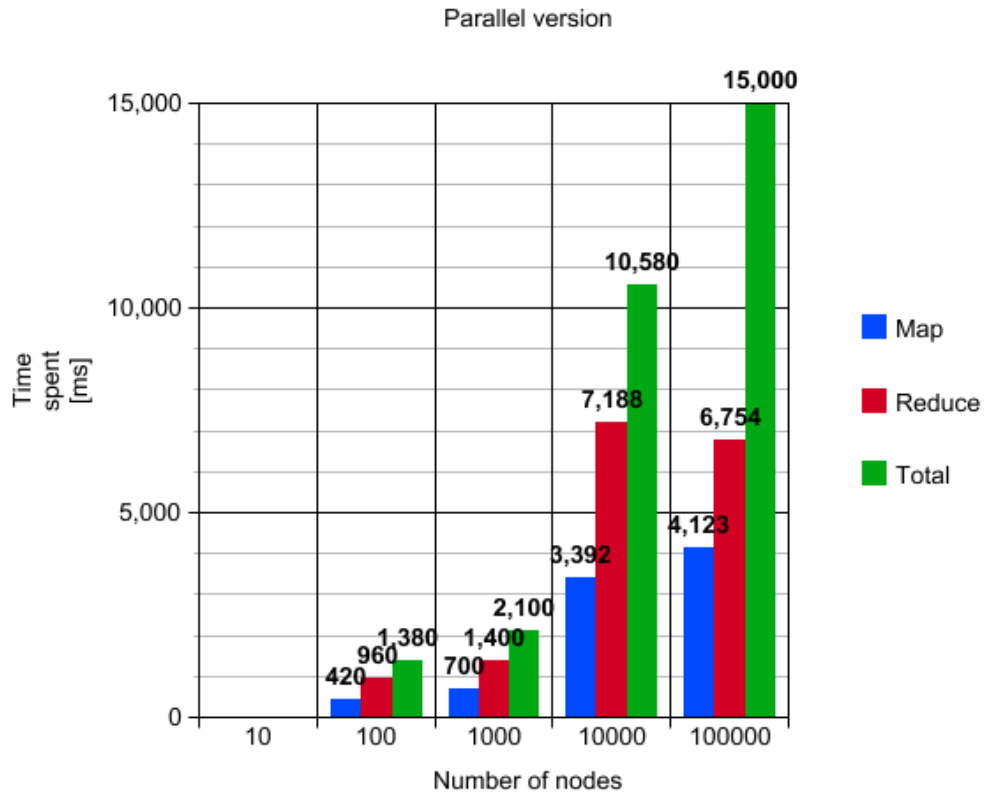


Figure 7: One iteration of parallel processing.

I didn't display the measurements for 10 nodes, as I found them to be quite similar to 100 or even slower, which I find pretty peculiar. We can see, that the more nodes we have, the bigger the difference between the first and the second graph. An astute reader would've noticed, that the last two numbers of map and reduce don't quite add up to the total sum. The reason behind this is, that map always makes two tasks. Like I explained earlier we always emit two nodes with the same ID to the reduce stage. Therefore, the time is doubled.

## COMPARISON

A more detailed comparison between the two methods. I used the total sum of both map and reduce stage to represent the parallel, hadoop version.

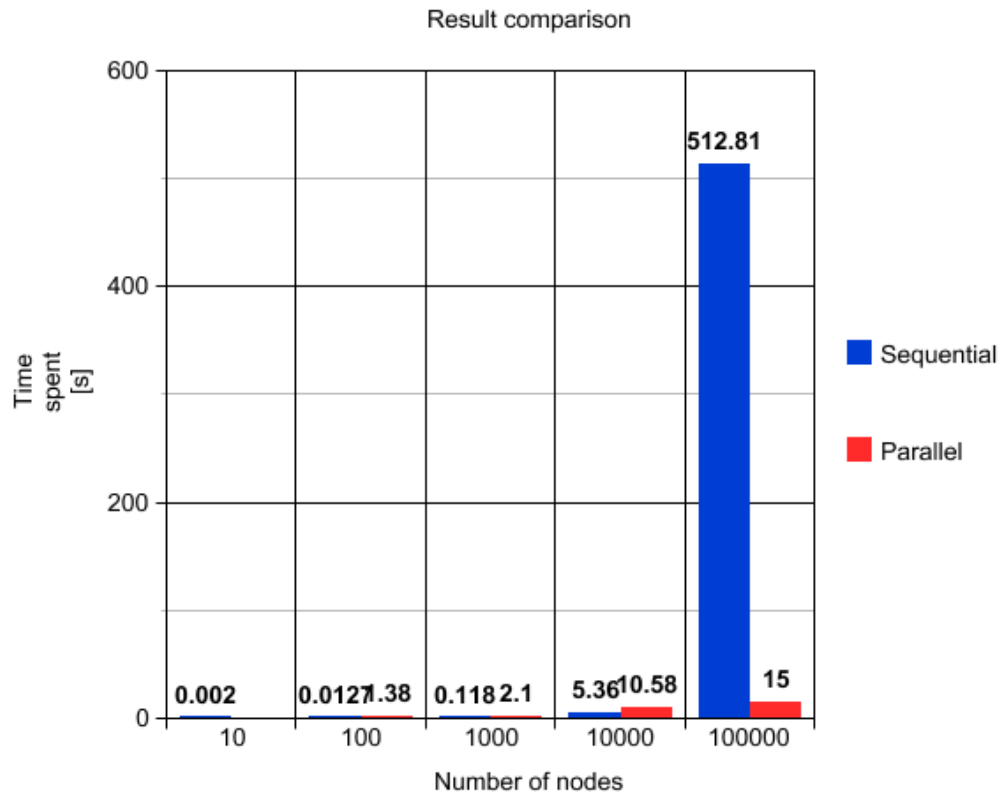


Figure 8: Comparison between the two methods.

## 5. CONCLUSION

With the implementation of the sequential version, the Bellman Ford algorithm took a pretty decent time to complete all the path calculations. I find it really weird, though, that the time it took for the parallel version to complete blew sky-high. It took nearly five minutes to calculate a graph with 50 nodes. The measurements in Figure 8 demonstrate one sole iteration of our algorithm. Here we do have to take note, that for every iteration, the algorithm goes through the whole file and checks if there are any non visited nodes (nodes coloured GRAY), which it can expand. In conclusion, I can not firmly state that the parallel version is either slower nor faster than the sequential. Further testing would be needed.

## 6. REFERENCES

Ubuntu download page

<http://www.ubuntu.com/>

Eclipse and Java SDK

<https://www.eclipse.org/>

<http://www.oracle.com/technetwork/java/javase/downloads/>

Hadoop

Great hadoop tutorials with some basic informations, how to setup hadoop and how to develop applications.

<https://www.youtube.com/channel/UCnuZg1aSJxROZRgNvftqCyA>

Hadoop counters:

<http://hadoop.apache.org/docs/r2.2.0/api/org/apache/hadoop/mapred/Counters.html>

<http://hadooptutorial.wikispaces.com/Iterative+MapReduce+and+Counters>