Parallel Boruvka

Final project for the SPM course 2020/21

Matteo De Francesco

Contents

1 Introduction

In this report we will analyze a parallel implementation of the Boruvka algorithm.
Boruvka algorithm is used to discover the **MST** (*Minimum Spanning Tree*) of a given graph.
It proceeds in the following way:

Algorithm 1 Boruvka Algorithm

```
1:  function BORUVKA ALGORITHM(V, E)
2:      Comp = [ ]                                  ▷ Initialize empty components
3:      for each v ∈ V do
4:          add v to Comp                           ▷ Add each vertex to the Components
5:      end for
6:      while |V| > 1 do                            ▷ Until there is more than one parent node left
7:          V' = getMinEdges(c, V, E)
8:          Contract components
9:          Update V
10:         Update E
11:     end while
12:     return MST
13: end function
```

and the function `getMinEdges` does the following

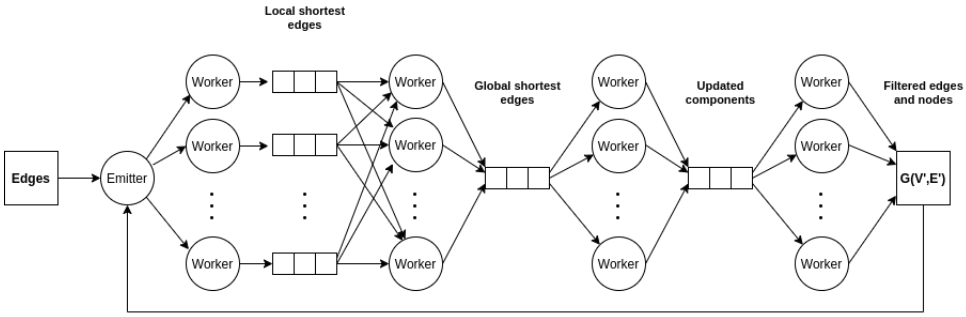
Algorithm 2 Get min edges function

```
1: function GETMINEDGES(c, V, E)
2:     E_new = [ ]
3:     for each e ∈ E do          ▷ Iterate through all the edges in the remaining graph
4:         Set E_new [e.from] = e  ▷ Find the minimum outgoing edge from each vertex
5:     end for
6:     return E_new
7: end function
```

Basically the algorithm starts by initializing each component with a node. Then for each edge in the graph, Boruvka updates that minimum outgoing edge if it found it with a lesser weight.
After the minimum edges are found, components are merged together, using the `UNION-FIND` data structure [1], a lock-free structure commonly used when dealing with disjoint sets.
This is repeated until there is only one node left.

2 Parallel Implementation

As we can see from the above algorithm, the code can be parallelized with a `map-reduce` approach.
Indeed, the line 7 can be easily parallelized by distributing the edges among the different computational resources we have, each one computing its assigned shortest local edges. Then we can merge together this local shortest edges found by distributing again the computation between the different resources, always avoiding concurrency between the threads.
Then, in the line 8, a `reduce` approach can be used to merge the components using the selected edges. We use the `UNION-FIND` data structure to keep track of the nodes, parents and rank, by using the appropriate operations of `unite` and `same`.
A possible schema of our parallel application can be the following:



We can see the typical schema of a pipeline. As edges arrives, the *Emitter* splits the workload among different workers. Each one of them apply the map function and create a local copy of shortest edges, which is stored in the queue. As soon as all the items are processed, the local shorted edges are merged together to create a global shortest edges array, distributing the indexes among the next workers. Then using these edges, we update the `UNION-FIND` data structure, we filter the sets of edges and nodes, and finally we sent them back to the *Emitter* and the cycle repeats, until there is only one node left.
Multiple graphs[1] were tested.

[1]Taken from `https://networkrepository.com/networks.php`

References

[1] Richard J. Anderson and Heather Woll. Wait-free parallel algorithms for the union-find problem. In *STOC '91*, 1991.