

Parallel Boruvka

Final project for the SPM course 2020/21

Matteo De Francesco

Contents

1	Introduction	1
2	Parallel Implementation	1

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  $v \in V$  do
4:     add  $v$  to  $Comp$  ▷ Add each vertex to the Components
5:   end for
6:    $MST = \{\}$  ▷ Initialize empty MST
7:   while  $|Comp| > 1$  do ▷ Until there is more than one component left
8:     for each  $c \in Comp$  do
9:        $E' = \text{getMinEdges}(c, V, E)$ 
10:      add min  $e \in E'$  to  $MST$ 
11:    end for
12:    Merge components
13:  end while
14:  return  $MST$ 
15: end function
```

and the function `getMinEdges` does the following

Algorithm 2 Get min edges function

```
1: function GETMINEDGES( $c, V, E$ )
2:    $E_{tot} = []$ 
3:   for each  $v \in c.vertices$  do ▷ Iterate through all the vertices in the component
4:      $E' = \text{Find all edges involving } v$  ▷ Find all edges where  $v$  is present
5:      $e.val = +Inf$  ▷ Minimum node found, set starting value to  $\infty$ 
6:     for each  $e' \in E'$  do
7:        $\#$  Check if  $v$  is not linked to another  $v$  in the same component
8:       if  $e'.destNode \notin c.vertices$  and  $e'.val < e.val$  then
9:          $e = e'$  ▷ Update  $e$ 
10:      end if
11:    end for
12:    Add  $e$  to  $E_{tot}$ 
13:  end for
14:  return  $E_{tot}$ 
15: end function
```

Basically the algorithm starts by initializing each component with a vertex. For each component, the algorithm look for the minimum edge that links it with a different component, and update the MST with it.

After the minimum edges are found, the algorithm merge the components together based on the edges present in the MST . When the component is composed by more than one vertex, we look for the minimum edge by repeating the procedure on each node in the component, always satisfying the condition that the other node must not be in the same component (otherwise we create a cycle).

This is repeated until there is only one component left, which is the desired minimum spanning tree.

2 Parallel Implementation

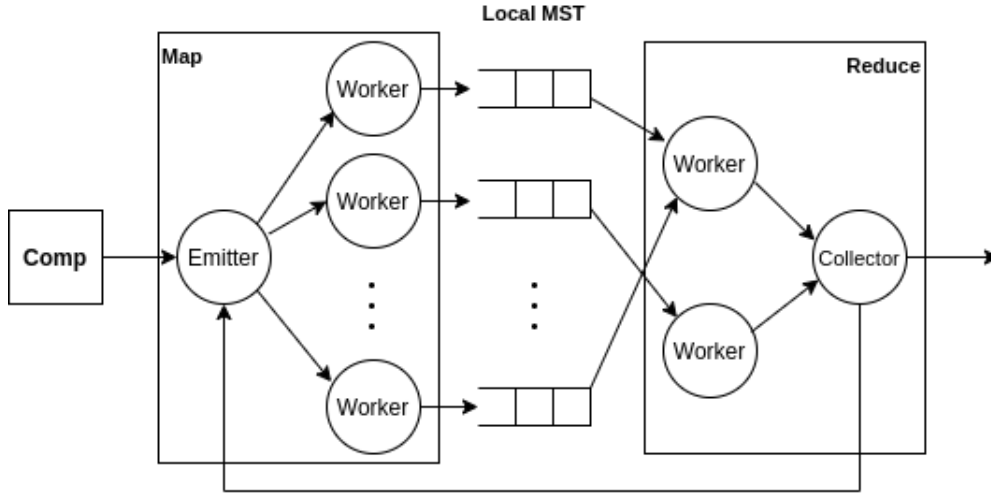
As we can see from the above algorithm, the code can be easily parallelized with a **map-reduce** approach.

Indeed, the line 8 can be easily parallelized by distributing the components among the different computational resources we have, each one computing the same function (which coincide with the loop body). Then, in the line 12, a **reduce** approach can be used to merge the components using the selected edges in *MST*.

We can notice also how there is no condition when for each component we add the minimum edge to the *MST* (two components can add also the same edge, which is exactly the link between the two components). This implies that we have no need for synchronization over the *MST*. However, since each thread will execute on a different core, we can assume that each cache level of each core will store the global variable *MST*. If we keep it global, we don't need synchronization among threads, but the cache coherency protocol need to update *MST* in each cache whenever this is modified by a thread.

A better solution is to keep a local copy of *MST* per each thread and at the end merge them all together in the global *MST*.

A possible schema of our parallel application can be the following:



As components arrives, the *Emitter* splits the workload among different workers. Each worker apply the map function and create a local copy of *MST*, which is stored in the queue. As soon as all the components are processed, the *reduce* part begins and the different edges in the local *MST* are connected together. In this way we update the global *MST* and obtain new components, which are sent back from the *Collector* to the *Emitter* and the cycle repeats, until there is only one component left.