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

```
function BORUVKA ALGORITHM(V, E)
    Comp = [ ]                              ▷ Initialize empty components
    for each v ∈ V do
        add v to Comp                       ▷ Add each vertex to the Components
    end for
    MST = { }                               ▷ Initialize empty MST
    while |Comp| > 1 do                     ▷ Until there is more than one component left
        for each c ∈ Comp do
            E' = getMinEdges(c, V, E)
            add min e ∈ E' to MST
        end for
        Merge components
    end while
    return MST
end function
```

and the function `getMinEdges` does the following

Algorithm 2 Get min edges function

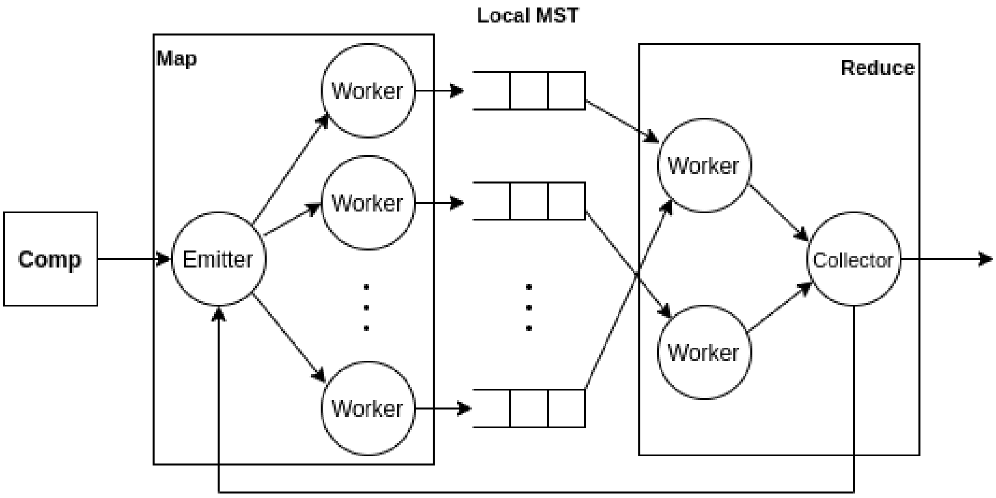
```
function GETMINEDGES(c, V, E)
    E_tot = [ ]
    for each v ∈ c.vertices do              ▷ Iterate through all the vertices in the component
        E' = Find all edges involving v     ▷ Find all edges where v is present
        e.val = +Inf                        ▷ Minimum node found, set starting value to ∞
        for each e' ∈ E' do
            # Check if v is not linked to another v in the same component
            if e'.destNode ∉ c.vertices and e'.val < e.val then
                e = e'                      ▷ Update e
            end if
        end for
        Add e to E_tot
    end for
    return E_tot
end function
```

Basically the algorithm starts by initializing each component with a node. For each one of this, Boruvka looks for the minimum edge that links it with a different component, and update the MST with it.
After the minimum edges are found, the obtained edges in the MST are merged together to update the components. When these latter are made up by more than one node, we look for the minimum edge by repeating the procedure on each node, 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 parallelized with a `map-reduce` approach.

Indeed, the line 8 can be easily parallelized by distributing the nodes 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 them using the selected edges in MST.
We can notice also how there is no condition when we add the minimum edge to the MST (two components can add also the same edge, which is exactly the link between them). This implies that we have no need for synchronization over the MST. However, since each computational resource will execute on a different core, we can assume that each first cache level 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 one of them apply the map function and create a local copy of MST, which is stored in the queue. As soon as all the items 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 left.
Multiple graphs[1] were tested.

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