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 $v \in V$ do
        add $v$ to $Comp$                          ▷ Add each vertex to the Components
    end for
    $V' = \{\}$                                    ▷ Empty set of size $V$
    while $|V| > 1$ do                             ▷ Until there is more than one parent node left
        $V' =$ getMinEdges($c, V, E$)
        Contract components
        Update $V$
        Update $E$
    end while
end function
```

and the function `getMinEdges` does the following

Algorithm 2 Get min edges function

```
function GETMINEDGES($c, V, E$)
    $E_{new} = [\,]$
    for each $e \in E$ do                          ▷ Iterate through all the edges in the remaining graph
        Set $E_{new}\,[e.from] = e$                ▷ Find the minimum outgoing edge from each vertex
    end for
    return $E_{new}$
end function
```

The algorithm starts by initializing each component with a node. Then, for each edge in the graph, Boruvka update the minimum outgoing edge if it is the smallest found until that moment.
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.
Finally, the edges E of the original graph are filtered by removing those that lie in the same components (no cycle are allowed) and the nodes V by removing the non-root nodes of the component data structure.
This is repeated until there is only one node left.

2 Parallel Implementation

As we can see from the above algorithms, the application is not embarassingly parallel, but code can be parallelized with a proper `map-reduce` approach.
The line 8 can be parallelized by distributing the edges among the different computational resources we have, each one computing its shortest local edges using the given edges. Then, the local shortest edges found can be merged by distributing again the computation between the different resources, always avoiding concurrency between the threads.
Then, at line 9, 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:

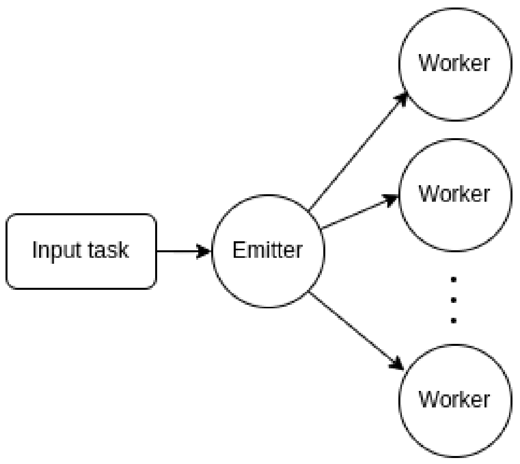

Figure 1: Parallel schema

We can see the typical schema of a farm. As edges arrives, the *Emitter* splits the workload among the different workers. Each one of them apply the map function and create a local copy of shortest edges, which is saved for the next task. 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 workers. Then, using these edges, we update the `UNION-FIND` data structure, we filter the sets of edges and nodes, and the cycle repeats, until there is only one node left.

2.1 Measures used

2.2 `UNION-FIND` data structure

The `UNION-FIND` data structure is a typical used data structure when dealing with disjoint sets. To have an efficient implementation and support a lock-free architecture, the structure simply consists in a single array of *atomic* types. Assuming we have N nodes in our graph, we represent these by using the indexes, ranging from $[0, N-1]$. Then, we store in the relative position of the array the parent of that index node. Lastly, we use the bitwise operation `AND` to keep track of the rank of each node.
In this way, we do not need a global lock to access the array since we represent values using *atomic* types.
The implemented operations are:

- `FIND(`id`)`: find the parent of a given index node id;
- `SAME(`$id1$`, `$id2$`)`: check if $id1$ and $id2$ have the same parent;
- `UNITE(`$id1$`, `$id2$`)`: connect the two disjoint trees where $id1$ and $id2$ belongs to;
- `PARENT(`id`)`: return the parent of the given id;

It's worthwhile to mention that in the latter two operations, we loop through the structure until we reach the objective, because having *atomic* entries these can be changed at anytime by another resource.

2.3 Parallel steps

Before going on, we need to clarify better the steps that are done at each iteration.
We can identify 4 steps done at each iteration:

1. In the first step, we create a copy of shortest local edges for each resource we have. We initialize this with size the number of nodes V, and we store then a "fake" edge having starting node 0, ending node 0 and weight 10, the maximum allowed. The resource receive a pair of indexes, and look for all the edges E in the grah for the given indexes. For each one of them, if it has a lesser weight than the current one, then updates its

local shortest edge at the index corresponding to the starting node of the current edge. We decided to keep a different copy for each worker otherwise we would have concurrent accesses to the shortest edges.

2. In the second step, all the local shortest edges must be merged together to have a unique copy of the shortest edges found. In this phase each resource receive a pair of indexes contained in the rang $[0, V-1]$. Then, the current worker loop through all the obtained local shortest edges in the previous step and update the global shortest edge array in the given pair positions. Distirbuting the indexes in this way we are guaranteed of not having concurrent accesses to the same indexes and so we don't need a lock mechanism neither here.

3. In the phase 3, we distribute the indexes of the global shortest edges array found among the workers. Each one then loop through the given positions and update the `UNION-FIND` structure.

4,5 In this latter two phases, we filter the edges and nodes of the current graph. Again, we distribute the indexes (respectively of edges and nodes) among the workers and we access the disjoint structure to check:

- In the first case, we use the operation `SAME` to check if the starting and ending node of the current edge are in the same tree. It it is so, we discard this edge.
- In the second case, we use the operation `PARENT` to check if the given node is parent of himself. If it is so, we keep this node.

3 Implementation

3.1 Standard Thread implementation

3.2 Fastflow implementation

References

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