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 $v \in V$ do
4:          add $v$ to $Comp$                ▷ Add each vertex to the Components
5:      end for
6:      $V' = \{\}$                          ▷ Empty set of size $V$
7:      while $|V| > 1$ do                   ▷ Until there is more than one parent node left
8:          $V' =$ getMinEdges($c, V, E$)
9:          Contract components
10:         Update $V$
11:         Update $E$
12:     end while
13: 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 the algorithm cycles through all the edges and select the smallest one not belonging to the same component (otherwise we create cycles).
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 (as we said, no cycles 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 approach.
Operations inside the `while` loop can be parallelized using different strategies. Line 8 can be tackled 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.
At line 9, we can 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.
Finally, also lines 10 and 11 can be executed distributing the set of nodes V and edges E among the resources and merging the results.
A possible schema of our parallel application can be the following:

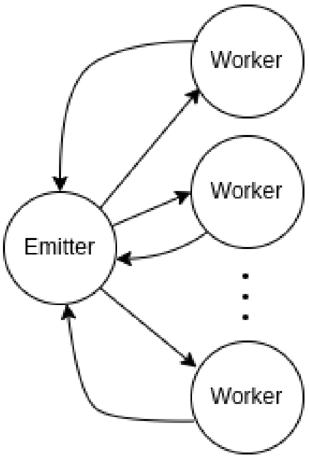


Figure 1: Parallel schema

We can see the typical schema of a master-worker farm. The *Emitter* splits the workload among the different workers. Each one of them apply the required 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, by 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 `UNION-FIND` data structure

The `UNION-FIND` data structure is a typically used data structure when dealing with disjoint sets. To have an efficient implementation and support a lock-free architecture, the structure 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:

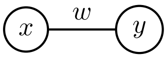
- `FIND(`id`)`: find the parent of a given index node id;
- `PARENT(`id`)`: return the parent of the given id;
- `SAME(`$id1$`, `$id2$`)`: check if $id1$ and $id2$ have the same parent;
- `UNITE(`$id1$`, `$id2$`)`: connect the two disjoint trees where $id1$ and $id2$ belongs to;

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.

2.2 Parallel steps

Before going on, we need to clarify better the steps that are done at each iteration.
We can identify 5 phases 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 as the number of nodes $|V|$, and we initialize all the positions with a "fake" edge having starting node 0, ending node 0 and weight 10, the maximum allowed. Each resource receives a pair of indexes, and look for all the edges E in the graph for the given indexes. For each one of them, if it has a lesser weight than the current one, updates its local shortest edge array at the index corresponding to the starting node of the current edge.
   For example, given the shortest edge array of resource i E_i, and the following edge:



If $E_i\left[x\right].weight > w$, then we update $E_i\left[x\right] = \{x, y, w\}$ storing the current edge. We decided to keep a different copy for each worker to avoid concurrent accesses.

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 range $[0, V-1]$. Then, the current worker i loop through all the local shortest edges E_i obtained in the previous step and update the global shortest edge array in the same indexes. By distributing the indexes in this way, we are guaranteed of avoiding concurrent accesses to the same indexes, so we don't need a lock mechanism.

3. In the phase 3, we distribute the indexes of the global shortest edges array found among the workers. Each one loop through the given positions and update the `UNION-FIND` structure, by calling the `UNITE` operations over the starting and ending node of the current minimum edge.

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

- For the edges, we use the operation `SAME` to check if the starting and ending node of the current edge aren't in the same tree. It it is so, we keep this edge.
- For the nodes, we use the operation `PARENT` to check if the given node is parent of himself. If it is so, we keep this node.

In addition, we left out as a sequential operation a *final filtering* phase where we merge the result of phase 4 and 5 and we update the graph with the remaining edges and nodes.

2.3 Measures used

To evaluate the performances of our application, we use the typical metrics for a parallel application

$$sp(n) = \frac{T_{seq}}{T_{par}(n)} \text{ (\textit{Speedup})}$$

$$sc(n) = \frac{T_{par}(1)}{T_{par}(n)} \text{ (\textit{Scalability})}$$

$$ef(n) = \frac{sp(n)}{n} \text{ (\textit{Efficiency})}$$

The timing required by our application will be computed according to the steps described in the previous section

$$T_{total} = T_{read} + T_{boruvka} + T_{reload}$$

Since we are interested in optimizing the time of our algorithm, we will not consider T_{read}, the time needed to read the graph, and T_{reload}, the time needed to restore the original graph for the next try.
We will focus on optimizing $T_{boruvka}$, which according to the above steps is given by

$$T_{boruvka} = T_{map} + T_{merge} + T_{contraction} + T_{filter\_edges} + T_{filter\_nodes} + T_{final\_filtering}$$

where:

- T_{map}: coincides with the phase 1;
- T_{merge}: coincides with the phase 2;

- $T_{contraction}$: coincides with phase 3;
- T_{filter_edges} and T_{filter_nodes}: coincides with phase 4 and 5;
- $T_{final\_filtering}$: coincides with the final sequential operation;

2.4 Graph generation

The graph generation is a simple function that create a graph with given number of edges and nodes. When the provided number of nodes is smaller than the edges, it should return in most cases an undirected connected graph, which is necessary for Boruvka to converge.
The algorithm used is the following:

Algorithm 3 Graph generation

```
function GRAPH GENERATION($|V|, |E|$)
    $Edges = Set\{\}$
    $Nodes = Set\{\}$
    while $Edges.size < |E|$ do
        $x =$ random mod $|V|$
        $y =$ random mod $|V|$
        if $y < x$ then
            $weight =$ generate random weight $\in [1, 10]$
            add $\{x, y, weight\}$ to $Edges$
            add $\{y, x, weight\}$ to $Edges$
            add $x$ to $Nodes$
            add $y$ to $Nodes$
        end if
    end while
    return $G = (Nodes, Edges)$
end function
```

3 Implementation

As required by the project guidelines, we parallelized Boruvka algorithm by using the standard thread library of C++ and the Fastflow [2] library.
Both the standard thread implementation and the fastflow version take as input

- `nw`: the number of workers;
- `filename`: the filename storing the graph;
- `nodes`, `edges`: the number of nodes and edges to generate the graph, ignored if `filename` is specified;
- `tries`: the number of execution to do for each number of worker;

We designed the code by first loading the graph from a given file or by generating it. Since we used big graphs to perform the experiments and gaining some speed up, we load the graph only once at start. Then, for each number of workers from 1 until `nw`, we perform `tries` executions to measure the mean time $T_{boruvka}$.
As stated before, T_{read} is the time needed to read/generate the graph at start, which is executed only one time.
Since we modify the graph at each iteration, after each `trie` we need to restore the original graph. This is done by creating a copy of the loaded graph after the reading, and used later to restore the graph for the next try. Indeed T_{reload} is the time needed to restore it.
We report here a pseudocode representing the implemented algorithm:

Algorithm 4 Boruvka Parallel Algorithm

```
function Boruvka Parallel Algorithm(nw, filename, nodes, edges, tries)
    $G$ = load graph from filename, otherwise generate(nodes, edges)
    $G' = G$                                              ▷ Create copy of $G$
    for $i = 1$ until nw do
        Create threadpool(nw) or ParallelFor(nw)
        $Comp = init()$                                   ▷ Initialize components
        while tries $> 0$ do
            while $|V| > 1$ do
                Compute each phase
                Update $V$ and $E$ of $G$
            end while
            Restore $G = G'$
            Decrement tries
        end while
    end for
end function
```

3.1 Standard Thread implementation

For the standard thread implementation, we started by designing a farm using an emitter to dispatch the indexes, and a different worker implementation for each of the required phases. Then in each loop we instantiate a farm for each of the phases.
To reduce the overhead of this implementation (creating a farm for each phase in each while loop) we refactored the implementation, creating a *threadpool*. In this way, we start the pool at each iteration and then we enqueue tasks, in the mostly possible general way.
To achieve this, our *threadpool* spawns the given number of threads at start, waiting for task to be putted in the queue. We use the `std::packaged_task` library functionality to allow enqueueing of general functions, which are then executed by the threads, and `std::future` return type to check for termination, so we can start the next phase. We have 5 different tasks that are "packaged" and stored in the *threadpool* queue:

- `mapwork(local_edges, graph, chunk_indexes, index)`
  Each thread modify the `local_edges[index]`, taking the edges of `graph` at the given positions `chunk_indexes`;

- `mergework(local_edges, graph, chunk_indexes)`
  Each thread loop through all the `local_edges` with the given indexes of `chunk_indexes`, and modify the `global_edges` in the same positions;

- `contractionwork(global_edges, initialComponents, graph, chunk_indexes)`
  Each worker loop through `global_edges` at the indexes specified by `chunk_indexes`, and unify the given edges by using the `UNITE` operation of `initialComponents`;

- `filteringedgework(remaining_edges, initialComponents, graph, chunk_indexes, index)`
  Each worker loop through `graph`'s edges at given positions `chunk_indexes`, then check each edge's nodes using the operation `SAME` of `initialComponents`, and update `remaining_edges[index]` accordingly;

- `filteringnodework(remaining_nodes, initialComponents, graph, chunk_indexes, index)`
  Each worker loop through `graph`'s nodes at given positions `chunk_indexes`, then check each node using the operation `PARENT` of `initialComponents`, and update `remaining_nodes[index]` accordingly;

The functions above are basically the implementation of the phases described in the previous section.

3.2 Fastflow implementation

Also for the Fastflow implementation, we firstly used a *master-worker* approach, designing emitter and workers for the specific tasks. Using this approach we spawn a `ff_Farm` object for each phase in the while loop, which as happened for the threads, introduced a lot of overhead.
To cope with this, we decided then to use the more general `ParallelFor` utility. We create a `ParallelFor` with the given number of threads, and then we execute each phase without the need of respawning the threads.
Each phase is implemented in the same way of the above described functions, but executed in the lambda closure of the `ParallelFor`.

3.3 Normal Form and alternative implementation

The tree of computation described above can be represented in the following way:

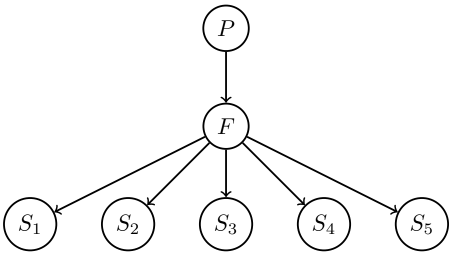


Figure 2: Original tree

$\Longrightarrow$

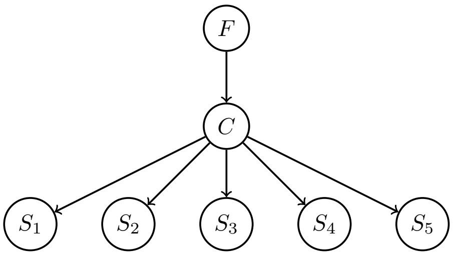


Figure 3: **Normal Form**

As we can see in the **Normal Form**, we could turn the entire implementation into a single farm executing the stage sequentially. However, since we have dependencies between threads (i.e. each thread needs to wait the completion of all the other before passing to the next stage), we decided to stick with the current implementation.
An alternative could have been using a barrier between threads to synchronize them at the end of each stage.

4 Experiments and Results

Experiments were conducted on the remotely available XEON PHI machine in the facility of San Piero a Grado. Test were executed by simply running the three different implementation:

- `boruvka_sequential`, for comparison;
- `boruvka_thread`, the version using the threadpool;
- `boruvka_ff`, the version using fastflow;

To compile everything run

```
make all -j4
```

To reproduce the experiments (omit `nw` for `boruvka_sequential`), simply run

```
./build/executable nw n_nodes n_edges filename tries
```

Also, we use the library `jemalloc` to speedup the allocation process. This is stored in the root of the remote machine and exported in the according file `.bashrc`.
We observed that almost in all the settings, the fastflow version holds quite stable when increasing the number of threads.
Instead, for the threadpool version, we observed that after reaching the number of physical cores the speedup start decreasing, and we consequently lose the advantage of using parallelism.
We used random generated graphs whose magnitudes can be summarized in the following table:

$\lvert V \rvert$	$\lvert E \rvert$
$100K$	$2M$
	$5M$
	$10M$
	$15M$
$500K$	$30M$

Table 1: Experiment sizes

Then, we can show timing results of the tested version together with some plots of the most interesting parallel measures used.

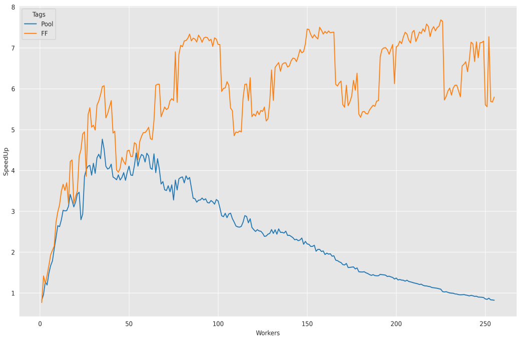


Figure 4: Speedup 15M edges

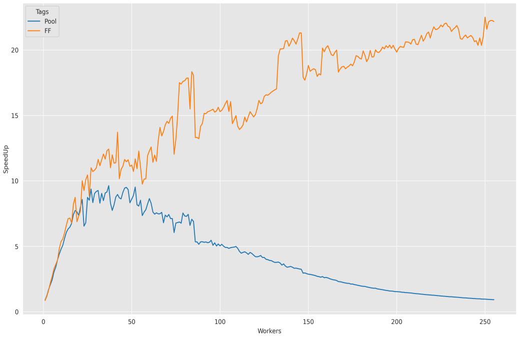


Figure 5: Speedup 30M edges

The above speedups confirm our previous claim. The Fastflow version is almost stable with some peaks when reaching the maximum amount of threads, bringing a speedup of around 8 and 20 in the above examples. Instead for the version using the standard thread library, we reach a speedup of around 5 and 10 when using a number of threads < 64, which coincides with the number of physical cores. After that, we observe a decreasing trend of speedup for our application, meaning the overhead is becoming too much.

We explain this behavior with the maximum reaching of the physical cores, the current implementation of the threadpool, which make use of `async` and `packaged_task` functionality, which introduce overhead in our implementation. Also, for the standard thread implementation the thread mapping to the physical cores was not performed, while in the fastflow version the provided script execute automatically the PIN of threads onto cores.

We tried also performing the mapping of threads on the physical cores, but this does not bring any advantages.

We can see also how the magnitude of the graph impacts on the problem: the more the graph is bigger, the more the problem scales better (*Gustafson Law*).

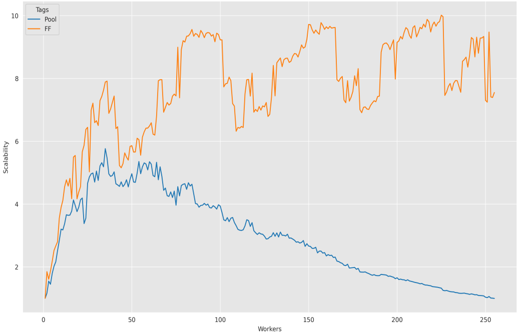


Figure 6: Scalability 15M edges

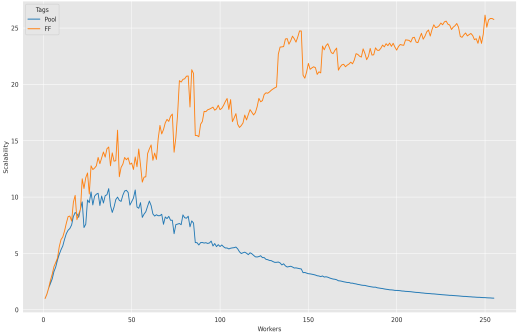


Figure 7: Scalability 30M edges

The same holds also for the scalability. We can see overall that the trend is almost the same as what happened for the speedup, with fastflow retaining an high value of sc until the end while for the thread version we observe a decrease in the performance.

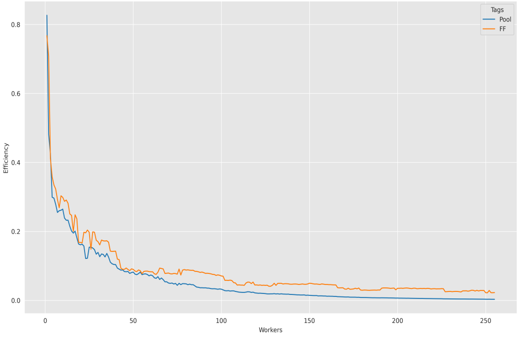


Figure 8: Efficiency 15M edges

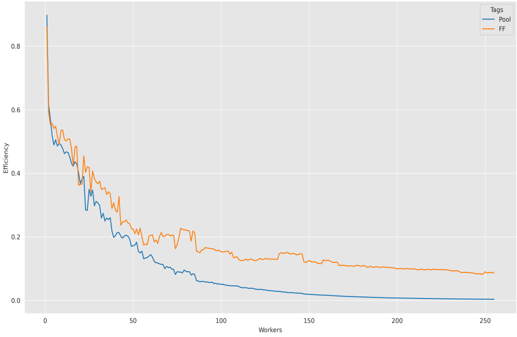


Figure 9: Efficiency 30M edges

Finally, from the efficiency measure we can see at impact the different between two different graphs. The one reported on the left 8 is drastically different for the example over 30 million edges 9.

We can infer that increasing the size of the problem brings more benefits in both the parallel version of Fastflow and the standard library. We have an higher speedup, a better scalability and also an efficient result until the number of physical cores.

Regarding the timings, we have the following results:

`Degree`	FF $T_{boruvka}$ (μsec)	Thread $T_{boruvka}$ (μsec)
`sequential`	2578422.8	2578422.8
1	3360022.6	3119345.6
2	1819374.6	2679723.3
4	1800767.6	2155530
8	1199759	1239372.3
16	805364	826165.6
32	460589.3	598506.6
64	493515	585244.3
128	489832.6	1056917.3
256	447482	3168656
`Bests`	225 `nw`, 335486.6 μsec	35 `nw`, 517920 μsec

Table 2: Results for $15M$ edges graph

Degree	FF $T_{boruvka}$ (μsec)	Thread $T_{boruvka}$ (μsec)
sequential	13224496.6	13224496.6
1	15355088.6	14744939.3
2	11150112.3	10742981.6
4	5934893.6	6366435.3
8	3355254.3	3346484.6
16	1952591.3	1959520.6
32	1185156.6	1594288.6
64	1151006.6	1734905.6
128	794541.3	3374470.6
256	602700.6	14522257.3
Bests	250 nw, 587527 μsec	37 nw, 1372857 μsec

Table 3: Results for $30M$ edges graph

References

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

[2] Marco Aldinucci, Marco Danelutto, Peter Kilpatrick, and Massimo Torquati. *Fastflow: High-Level and Efficient Streaming on Multicore*, chapter 13, pages 261–280. John Wiley & Sons, Ltd, 2017.