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:      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

```
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
```

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

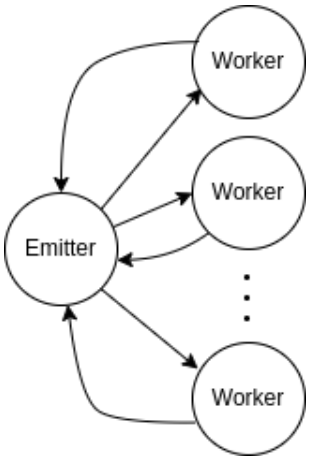

Figure 1: Parallel schema

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 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*;
- `PARENT(`*id*`)`: return the parent of the given *id*;
- `SAME(`*id*`1, `*id*`2)`: check if *id*1 and *id*2 have the same parent;
- `UNITE(`*id*`1, `*id*`2)`: connect the two disjoint trees where *id*1 and *id*2 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.
   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 loop through all the obtained local shortest edges in the previous step and update the global shortest edge array in the received indexes. By distributing the indexes in this way we are guaranteed of avoiding concurrent accesses to the same indexes and 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:

   - 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.

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

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

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

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

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;

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 huge graphs to perform the experimentations and gain 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 given the long amount of time required for huge graphs.
Since we modify the graph at each iteration, after each try 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.

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 threadpool at each iteration and then we enqueue tasks, in the mostly possible general way.
To achieve this, our threadpool implementation 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` of index `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` at the given `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` at the given `index` accordingly;

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 expected introduce 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`.

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.