Distributed REM Algorithm

Fredrik Manne, Rmi Dupr

August 18, 2018

1 REM

Algorithm 1 Union-find operation through REM algorithm

```
function REM-UNION-FIND($x$, $y$, $p$)
    $r_x \leftarrow x$
    $r_y \leftarrow y$

    while $p(r_x) \neq p(r_y)$ do
        if $p(r_x) > p(r_y) >$ then
            if $p(r_x) = r_x$ then
                $p(r_x) \leftarrow p(r_y)$
                return false
            else
                $p_{r_x} \leftarrow p(r_x)$
                $p(r_x) \leftarrow p(r_y)$
                $r_x \leftarrow p_{r_x}$
        else
            if $p(r_y) = r_y$ then
                $p(r_y) \leftarrow p(r_x)$
                return false
            else
                $p_{r_y} \leftarrow p(r_y)$
                $p(r_y) \leftarrow p(r_x)$
                $r_y \leftarrow p_{r_y}$

    return true
```

2 Distributed algorithm

Assumptions:

- $\forall x, p(x) \leq x$.
- Each process can determine the owner of any node.

Algorithm 2 Prepare distributed algorithm

```
1: $tasks = \emptyset$
2: for $i = 0$ to $n$ do
3:     $p(i) \leftarrow i$                                   ▷ Main disjoint set structure
4:     $p'(i) \leftarrow i$                                  ▷ Disjoint set used to filter edges to keep
5:
6: for all $(x, y)$ in $G$ do
7:     if $owner(y) = process$ then                          ▷ We assumed that $owner(x) = process$
8:         if not REM-UNION-FIND($x$, $y$, $p$) then
9:             REM-UNINON-FIND($x$, $y$, $p'$)
10:    else if REM-UNION-FIND($x$, $y$, $p'$) then
11:        $tasks \leftarrow tasks \cup \{(x, y)\}$
```

Algorithm 3 Handle one request (r_x, r_y) on $process = owner(r_x)$

```
1: function HANDLE($r_x$, $r_y$, $p$)
2:     $r_x \leftarrow$ LOCAL-ROOT($r_x$, $p$)
3:
4:     if $p(r_x) < r_y$ then
5:         send $(r_y, p(r_x))$ to $owner(r_y)$
6:     else if $p(r_x) > r_y$ then
7:         if $p(r_x) = r_x$ then
8:             $p(r_x) \leftarrow r_y$
9:         else
10:            $p_{r_x} \leftarrow p(r_x)$
11:            $p(r_x) \leftarrow r_y$
12:            send $(p_{r_x}, r_y)$ to $owner(z)$
```

3 Correctness

3.1 Notations

In order to make the proof clearer, some of the structures will be simplified. The state of the algorithm can be sumed up with following structures:

dset is the forest holding the disjoint set structure representing p. Thus we can use two kind of notations: $p(x) = y \Leftrightarrow (x, y) \in dset$ and $(y, x) \in dset$. Notice that as vertices are partitioned between process, it is not necessary to split this structure into one per process, in the actual algorithm however, only the process owning x knows the value of $p(x)$.

tasks is a graph containing all the pairs (r_x, r_y) the algorithm needs to handle. Such a pair can be an edge of the original graph, or a task sent by another process. As the proof will not need to order how the tasks are handled, it is enough to only consider the union all tasks received by all process.

union_graph is the graph of all edges that the algorithm still has in memory. $union\_graph = dset \cup tasks$.

equivalence a relation $\sim$ is defined as $G_1 \sim G_2 \Leftrightarrow G_1$ and G_2 have the same components.

A step of the algorithm consists in poping a task (r_x, r_y) from the graph *tasks* and then by applying algorithm 3 the states in *dset* and *tasks* is modified. Given a run, we will denotate the state of the algorithm after t iterations by $dset_t$ and $tasks_t$ (cf. algorithm 4).

Algorithm 4 Construction of $dset_{t+1}$ and $tasks_{t+1}$

```
1: $dset_{t+1} = dset_t$
2: $tasks_{t+1} = tasks_t \setminus (r_x, r_y)$ for a $(r_x, r_y) \in tasks_t$
3: Apply algorithm 3 on $owner(r_x)$.
```

The algorithm is initialised with state $dset_0 = \{(i, i) \ / \ i \in V\}$ and $tasks = G$.

3.2 Proof

Lemma 3.1. *At any step t of algorithm 3, $union\_graph_t \sim union\_graph_{t+1}$.*

Proof. At any step t, the algorithm pops an edge $(r_x, r_y) \in tasks_t$. We need to check that r_x and r_y are still in the same component of $union\_graph_{t+1}$ and that no other component is merged.

- If $p_t(r_x) = r_y$, algorithm 3 doesn't change $dset_{t+1}$ or $tasks_{t+1}$.
  Then $(r_x, r_y) \in dset_t = dset_{t+1}$, then $union\_graph_t = union\_graph_{t+1}$.
- If $p_t(r_x) < r_y$, then $(r_y, p_t(r_x))$ is inserted in $tasks_{t+1}$.
  Thus on any path containing $(r_x, r_y) \in tasks_t$, the edge (r_x, r_y) can be replace by edges $(r_x, p_t(r_x)) \in dset_t = dset_{t+1}$ and $(p_t(r_x), r_y) \in tasks_{t+1}$. No separate component are merged as $p_t(r_x)$ and r_x were in the same component.
- If $p_t(r_x) > r_y$:

– If $p_t(r_x) = r_x$, then (r_x, r_y) is added to $dset_{t+1}$.
  Then $union\_graph_t = union\_graph_{t+1}$ as $tasks_t \setminus tasks_{t+1} = (r_x, r_y)$.

– If $p_t(r_x) \neq r_x$, (r_x, r_y) replaces $(r_x, p_t(r_x))$ in $dset_{t+1}$ and $(p_t(r_x), r_y)$ is added to $tasks_{t+1}$.
  All theses nodes were part of the same component in $union\_graph_t$, no components are merged. Moreover, (r_x, r_y) is added to $dset_{t+1}$, not path using this edge is broken. Finaly, there is a new path from r_x to $p_t(r_x)$: $(r_x, r_y)(r_y, p_t(r_x))$ with $(r_x, r_y) \in dset_{t+1}$ and $(r_y, p_t(r_x)) \in tasks_{t+1}$.

□

Lemma 3.2. *Runing algorithm 4 until there is no task will halt.*

Proof. Let's assume that initially, $\forall (r_x, r_y) \in tasks, r_x > r_y$. Notice that new tasks created by algorithm 3 will always fit to this criteria.
When a task (r_x, r_y) is removed from $tasks$, either no new task is pushed to it, either a new task is pushed where this task is lower than the one removed in the meaning of total order $<_{lexrev}$ where $\forall (x_1, y_1), (x_2, y_2) \in V^2$:

$$(x_1, y_1) <_{lexrev} (x_2, y_2) \iff y_1 < y_2 \ \vee \ (y_1 = y_2 \ \wedge \ x_1 < x_2)$$

. □

Theorem 3.3 (Correctness)**.** *Runing algorithm 3 until there is no more tasks will result in dset beiing a spaning forest of the initial graph.*

Proof. Lemma 3.2 ensures that there is $t \in \mathbb{N}$ such that $tasks_t = \emptyset$. Finally, using lemma 3.1 we have:

$$G \sim tasks_0 = union\_graph_0 \sim union\_graph_1 \sim ... \sim union\_graph_t = dset_t$$

. □

4 Results

4.1 Observations

4.1.1 Performance loss for a low number of nodes

This algorithm is usually slower when one to two nodes are used to run it, whereas the only one comparison is added for each edge (checking if the second vertex of each edge belongs to the current process, it results in one modulus operation and one comparison per edge).

It seems that checking this condition in an empty loop over all edges takes more than one third of the total time taken by the local step of the algorithm.

4.2 Last run