Internship report

Rmi Dupr

August 18, 2018

Contents

1 Introduction

2 Context and state of the art

2.1 Union-Find algorithms

2.1.1 Disjoint set structure

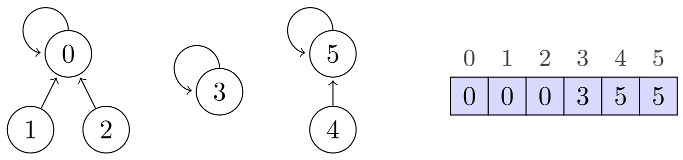
The disjoint set structure is a very classical structure that represents a partition of a finite set $X = \biguplus_{i \in I} S_i$. It is meant to allow three fast queries:

- $\textsc{makeset}(x)$: add the element x to the structure, initially in a singleton.
- $\textsc{Union}(x, y)$: alter the structure to merge the set x belongs to and the set y belongs to. After such an operation, $\exists! i \in I \ / \ x \in S_i \wedge y \in S_i$.
- $\textsc{find}(x)$: give a unique representative of the set x belongs to. It means that $\forall x, y \in X, \ \textsc{find}(x) = \textsc{find}(y) \Leftrightarrow \exists! i \in I \ / \ x \in S_i \wedge y \in S_i$.

In practice, it is usually represented by a forest of the elements of X, MAKESET builds a tree containing only its root, UNION merges two trees and FIND returns the root of a tree given one of its nodes. In a program, such a forest is represented by an array of size $|X|$, assuming that $X = [\![0, |X| - 1]\!]$, the element of index x will be y if y is the parent of x in a tree, or x if x is a root.

In order to lighten notations, whenever such a structure is used, we note the parent of any node x as $p(x)$.

Figure 1: A disjoint set structure representing $\{\{0, 1, 2\}, \{3\}, \{4, 5\}\}$



2.1.2 Classical union-find algorithm

A very typical application of this structure is to find a spaning forest for an undirected graph. A general algorithm used to answer this problem is called *union-find* (alg. 1).

Typically, FIND runs over the tree until it reaches the root, keeps track of every nodes on the path and finally set their parent to be the root. In this case UNION doesn't need to compress the trees and will just make one of the roots be parent of the other, the new root can be chosen arbitrary or given a criteria (index, rank . . .).

Algorithm 1 General structure of Union-Find

```
$F \leftarrow \emptyset$
for all $x \in V$ do
    MAKESET(x)
for all $(x, y) \in E$ do
    if FIND($x$) $\neq$ FIND($y$) then
        UNION($x, y$)
        $F \leftarrow F \cup \{x, y\}$
```

This algorithm can actualy be implemented with many variations (Patwary, Blair, and Manne (2010)), where a typical goal is to make sure that trees never get too high, thus compressing the path leading to the root while processing FIND operation.

2.1.3 REM algorithm

Add a reference to *Dijkstra, E.W.: A Discipline of Programming*?

A familly of variation, called *interleaved algorithms* in Patwary et al. (2010) replaces the two separate find operations at line 5 of Union-Find (alg. 1). This kind of algorithm will instead operate by running over the forest simultaneously from the two starting nodes. The first advantage of this kind of algorithm is that it won't need to reach both roots on every case. In Rem's algorithm, line 5 and 6 of algorithm 1 are replaced by algorithm 2 which handles find operations, merging components if they are disconnected and compression in one loop.

Algorithm 2 Rem's algorithm

```
1: $r_x \leftarrow x, r_y \leftarrow y$
2: while $p(r_x) \neq p(r_y)$ do
3:     if $p(r_x) > p(r_y)$ then
4:         if $r_x = p(r_x)$ then
5:             $p(r_x) \leftarrow p(r_y)$
6:             return false
7:         $p_{r_x} \leftarrow p(r_x)$
8:         $p(r_x) \leftarrow p(r_y)$
9:         $r_x \leftarrow p_{r_x}$
10:    else
11:        if $r_y = p(r_y)$ then
12:            $p(r_y) \leftarrow p(r_x)$
13:            return false
14:        $p_{r_y} \leftarrow p(r_y)$
15:        $p(r_y) \leftarrow p(r_x)$
16:        $r_y \leftarrow p_{r_y}$
17: return true
```

During all of the execution of the algorithm, we assume that the disjoint set structures always have any parent lower than its children (a sequence of nodes from a root to a leaf would be increasing). This hypothesis allows REM algorithm to process a run on each side of a node, and pausing the one that got the lowest index, that way if two nodes have a common ancester, the algorithm will reach a state where both runs just reached it and can conclude that these two nides where previously in the same component.

As searching and merging operation are done as a single operation, there is no problem with merging any subtree of the algorithm while it runs. This algorithm will take benefits of that, at each step it will make sure that given the two current nodes r_x and r_y of both runs, they will both have the lowest parent of the two (for example if $p(r_x) > p(r_y)$, then the new parent of r_x will be $p(r_y)$).

A union-find algorithm can typically be implemented to fit a worst case complexity of $O(n + m \cdot \alpha(m, n))$ where α is the inverse of Ackermann's function, $n = |V|$ and $m = |E|$. Rem's algorithm however, has a worst case complexity in $O(n + m^2)$. Nevertheless, this algorithm shows better performances on practice, I didn't try to proove it but it should have an average complexity that fits to the previous $O(n + m \cdot \alpha(m, n))$.

Reference or proof

2.2 Distributed algorithms

Almost no work seems to previously focus on the issue of distributing the union-find algorithm. This may be due to the fact that this algorithm is close to optimal, as much for its speed as for its memory complexity. It is possible however that because of the size of the input graph, the nodes don't fit in memory, Manne and Patwary (2009) mentioned that an application to Hessian matrices can reach this limit.

2.2.1 Distributed algorithms

A distributed algorithm is a parallel algorithm that doesn't use any shared memory. It means that it can run on a set of interconnected processors which don't need any shared hardware.

In our specific case we will only focus on the specific case where the algorithm can fit to the Bulk model where it runs on the following scheme:

1. Communicate with other processors to get new datas.

2. Process localy, without any communication.
3. Wait for other processes to finish their work. This step is called a barrier.
4. Return to the communication step (1).

Theses steps are called *super steps*, and while describing such an algorithm we want to describe what are the local computations during step (2) and which data will be exchanged between each pair of process during the communication step.

While following this scheme it is not hard to express a theorical asymptotic time complexity in of form like

$$T = W + H.g + S.l$$

where T, W, H and S are functions of the input parameters:

- T is the running time of the algorithm.
- W is the total time spent processing local steps. This is obtained by summing up the time spent by the slowest process at each super step.
- H is the number of messages sent (assuming they are of equal size).
- S is the number of barrier steps.

Finaly, g and l are constants representing the time spent to deliver a message and to execute a barrier step. These two constants are widely depending on the network setup used to executed the algorithm and on the speed of the processor.

Notice that even though this complexity is as simple at it can be (we could split $H.g$ to take the bandwith and the latency into account), it is already realy hard to evaluate each hardware-dependant constants. Thus, the testing phase takes an important place in the conception of a distributed algorithm.

2.2.2 Idea

Manne and Patwary (2009) introduced an parallization of the algorithm, based on an approach close to REM. The same kind of zigzag merge operation is done by comparing the rank of r_x and r_y, where the rank is the depth of a node in its component's tree. This process is extended to a distributed implementation by attributing for each vertex an unique process. Then, a disjoint-set structure will be stored among processes, for any node x, only the owner of x knows the value of $p(x)$. Something close to the sequential algorithm can then be executed by sending messages between processes whenever the current process can't run anymore the algorithm because it doesn't own r_x or r_y depending on their ranks. This algorithm had an issue of clarity as many specific cases had to be handled with a patch. However results showed that it was nicely scalable and where encouraging some refinement.

2.2.3 Repartition

As the operations likely to be the most time-expensive, the main goal is to reduce communication charge. Thus, the actual first step of a distributed union-find algorithm would be to eliminate for each processing node as much edges as possible before making any communication.

2.3 Shared algorithms

Refsnes, Patwary, and Manne (2012) showed that REM was a very convenient algorithm to adapt for parallelized computing. As it only modifies one side of the edge during each step of the merge operation, it can very easily be implemented for several cores using locks. But as locks are slow, the paper also introduce a lock-free approach that is much quicker. This approch processes by inserting edge in the structure in a first step that doesn't try to fix concurency issues. Then, the algorithm checks that each edge that added informations to the structure is still inside a component of the structure, the algorithm restarts with the set of theses node that were not connected as input and halts when this set is empty.

Add this algorithm somewhere

3 Writing REM as a distributed memory algorithm

3.1 Core idea

I started my internship by writing a distributed algorithm based on the REM algorithm. After only a few versions, it appeared that it was possible to write an algorithm very close to the sequential one: executing the same conditions, where half of the decisions are to request another process.

Each node keeps a set of pairs (r_x, r_y), as in the sequential algorithm, this means that r_x and r_y have to be in the same component of resulting tree. This algorithm operate using a supersteps scheme: it will sequentially handle a chunk of edges on each processing node, then will synchronise every node and exchange datas between them and start again.

Algorithm 3 Distributed REM algorithm

```
function HANDLE($r_x$, $r_y$, $p$)
    $r_x \leftarrow$ LOCAL-ROOT($r_x$, $p$)

    if $p(r_x) < r_y$ then
        send $(r_y, p(r_x))$ to $owner(r_y)$
    else if $p(r_x) > r_y$ then
        if $p(r_x) = r_x$ then
            $p(r_x) \leftarrow r_y$
        else
            $p_{r_x} \leftarrow p(r_x)$
            $p(r_x) \leftarrow r_y$
            send $(p_{r_x}, r_y)$ to $owner(p_{r_x})$
```

Algorithm 3 describes how a node can handle a pair (r_x, r_y). It runs on a node that received (r_x, r_y) and expects that the sender made sure that the current node owns r_x. Then the algorithm runs on its knowledge of the tree by processing the "local root" of r_x (ie. the highest parent of r_x in the disjoint set structure maintained by this process). From there, the algorithm can either conclude that this edge is already inside a component of the disjoint-set structure or send a new edge to process to another node.

3.2 Avoiding unecessary communications

As communication between process are very slow compared to the processor's computation, the most communications can be avoided the better it usually is. Thus, if a node has the ability to eliminate an edge, it should be done as soon as possible.

Before any communication a process will isolate every edges (x, y) it received such that it owns both x and y. Then, it will localy process the spaning forest (algo. 2) of these edges, which doesn't require any communication. Thus, the only remaining edges are edges that connect nodes owned by two different process.

Some of the edges linking two process can also be localy filtered. Given two edges (x_1, y) and (x_2, y), assuming that the process owns x_1 and x_2, if it already knows that x_1 and x_2 are in the same component, only one of them is required. After processing local initial components and before any communication, on every edge (x, y) where x is owned by the process, x can be replaced by its local root. Finally, duplicates can be eliminated.

4 Mixing up distributed and shared algorithms

5 Conclusion

6 Appendices

6.1 Correctness and halt of distributed algorithm

As I didn't find a proof for the sequencial algorithm, this proof also stands as my proof of this algorithm, this is the kind of reasoning that helped me understanding it at the begining of my internship. This proof is quite straightforward but it heavily rely on the introduction of a lot of notations.

6.1.1 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 $tasks = G$.

6.1.2 Proof

Lemma 6.1. *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 6.2. *Runing algorithm 4 until there is no task left in $tasks_t$ 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 $\forall (x_1, y_1), (x_2, y_2) \in V^2$:

$$(x_1, y_1) <_{lexrev} (x_2, y_2) \iff y_1 < y_2 \lor (y_1 = y_2 \land x_1 < x_2).$$

□

Theorem 6.3 (Correctness)**.** *Runing algorithm 3 until there is no more tasks will result in dset beiing a spaning forest of the initial graph.*

Proof. Lemma 6.2 ensures that there is $t \in \mathbb{N}$ such that $tasks_t = \emptyset$. Finally, using Lemma 6.1 we have:

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

□

Manne and Patwary (2009) Patwary et al. (2010)

References

Manne, F., & Patwary, M. M. A. (2009, 09). A scalable parallel union-find algorithm for distributed memory computers. , 186-195. http://www.ii.uib.no/~fredrikm/fredrik/papers/PPAM2009.pdf.

Patwary, M. M. A., Blair, J. R. S., & Manne, F. (2010). Experiments on union-find algorithms for the disjoint-set data structure. http://www.ii.uib.no/~fredrikm/fredrik/papers/SEA2010.pdf.

Refsnes, P., Patwary, M. M., & Manne, F. (2012, 05). Multi-core spanning forest algorithms using the disjoint-set data structure. , 827-835. Retrieved from `http://www.ii.uib.no/~fredrikm/fredrik/papers/sm_disjoint.pdf` doi: 10.1109/IPDPS.2012.79