Parallelized Union Find Set, with an Application in Finding Connected Components in a Graph

Zi Yin
Zhiang Hu (Harvy)

June 2, 2015

1 Background

1.1 Union Find Set

Union find set is a data structure that manages disjoint subsets. It can be constructed using a linked forest. Each node will have a pointer pointing to its parent, and the root node's parent will be itself.
The union find set supports two operation:

1. Find. The find operation will return the root of the node being requested. Since each node keeps a pointer to its parent, the algorithm will trace upward along the tree and find the node with itself as its parent, which is the root node.
   An optimization called path compression can be performed to flatten the tree. When finding the root of a node, we can set the parents of the nodes along the way directly as the root node. Hence the depth of the tree is reduced which decreases the time needed for the next find request. Below is the subroutine for find:

```
function find(x){
  if (x.parent != x){
    x.parent = find(x.parent)
  }
  return x.parent
}
```

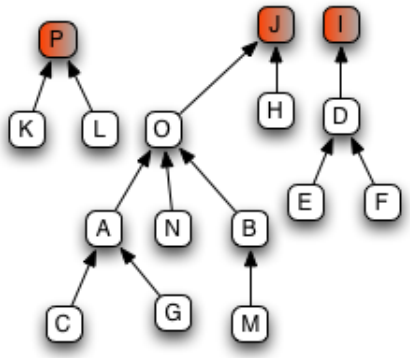

Figure 1: Illustration of a union find set, image courtesy to Mark C. Chu-Carroll

2. Union. When there is a request to union the sets containing node x and node y, the algorithm first find the roots of x and y. If the root of x is different from the root of y, one of their parent will be set to the other.
   An optimization called union by rank can be performed. Each root will keep the current depth of the tree, which is called the rank. When performing a union operation, the root with lower rank will be merged to the one with higher rank. In the illustration above, to merge the set on the left to the set in the middle, the parent of node P will be set to node J. Below is the subroutine for union:

```
function union(x, y){
  root_x = find(x)
  root_y = find(y)
  if (root_x != root_y){
    if (root_x.rank > root_y.rank){
      root_y.parent = root_x
    }
    else if (root_x.rank < root_y.rank){
      root_x.parent = root_y
    }
    else{
      root_x.parent = root_y
      root_y.rank = root_y.rank + 1
    }
}
```

2 Distributed Algorithm

We are going to construct a union find set on a graph $G = (V, E)$, with $n = |V|$ and $m = |E|$. We assume that the number of nodes n fits in a single machine's memory but the number of edges m does not. So we are going to distribute the edges onto k machines, and at the same time each machine will have a copy of the nodes V. Each subset will be a connected component.

At the beginning the parent of each node will be itself. Then we loop through the edges. At each time, the subsets containing the two endpoints of an edge will be merged. On a single machine, this algorithm finishes after going through the edge set once. So the time complexity is approximately $O(m)$.

In distributed settings, find operation can be done in parallel on different machines, since the operations won't interfere with each other, even with path compression. The tricky part is the union operation. Since the algorithm is now distributed, it is possible that simultaneously tow machines will set the parent of a root node to two different roots when performing two union requests. So the algorithm should be designed carefully to prevent this from happening.
Below we describe how to do union function in parallel setting. The input is m request (u_i, v_i) meaning that set containing node u_i and set containing node v_i should be unioned.

```
while there are unprocessed merge  request {
Step 1: for each merge request (u,v){
  Ru = find(u), Rv = find(v);
  if (Ru != Rv) emit root merge request (Ru,Rv)
```

```
 }
Step 2: (Construct Graph) {
2.1: Reduce duplicated root merge request into one,
and emit distinct edges of merging roots
2.2: (Set proper direction of the root merging graph) {
2.2.1 {
  For each root merging edge (Ru, Rv), set an arbitary direction, say Ru -> Rv
  Report number of roots with outer degree at least 1, denote as N1
}
2.2.2 {
 Reverse the root merging edge directions in 2.2.1,
 report number of roots with outer degree at least 1, denote as N2
}
Use the final root direction as 2.2.1 if N1>N2, otherwise, set direction as 2.2.2
}
}
 Step 3: (Merge roots) {
  For each root R with outer degree at least 1 {
  pick up an arbitrary out going edge (R,R0), set Parent(R) <- R0
  emit all other out going edges of R (R,R1)..(R,Rk) ...
  as unmerged request input to Step 1 in next iteration
  }
 }
}
```

3 Analysis

3.1 Parallelized Find

Find can be done in embarrassing parallel since the result change of parent in path compression is to change the parent to the same value, i.e. the node's current root. The amortized time for find operation can still be viewed as O(1) because of the path compression. If n fits in memory, the look up can be done in local machine and the updates of parent pointer need be communicated with a all to all communication with shuffle size O(m_i) where m_i is the number of unhandled request at iteration i.

3.2 Root Merging Graph construction

First we need to reduce the duplicated root merging requests. This can be done by a simple map reduce round. The shuffle size of this step is O(m_i) where m_i is the number of unhandled merging request at iteration i. The time complexity of this step is O($\frac{m}{k}$).

Then, We can construct the graph. Although it can be proven that a graph with edge direction that low degree node goes to high degree node can guarantee that at least half of nodes, we present a simpler version of constructing the graph in this report. As shown in the algorithm, first for each edge in the root merge graph, set an arbitrary direction, then count number of roots with outer degree at least 1 denote as N_1, then reverse the direction and count the non-zero outer degree roots as N_2. Pick the direction with higher number and this guarantees that at least half of nodes have outer degree at least 1. This is because, suppose N_1 is less than half of the engaging roots, then we have in the first graph there are at least half of engaged roots (roots with non-zero degree) have outer degree zero; by reversing the

direction of edges we can have these outer degree zero nodes to be non-zero outer degree. Therefore at least one of N_1 and N_2 is larger than half of engaged root number. This step also has shuffle size O($m_i + n$) and time complexity O($\frac{m}{k} + \frac{n}{k}$).

3.3 Merge

The time complexity of this step is O(min ($\frac{m}{k}$, $\frac{n}{k}$)), and shuffle size for updating parent is O(n), for unhandled requests is O(m_i), thus total shuffle size of this step is O(n+m_i).

3.4 Complexity

First we should notice that the problem size of engaged roots are reduced by at least half after each iteration.

1. **Time Complexity**
   As analysed before the total time complexity for each iteration is $O(\frac{m_i+n}{k})$. And thus the total time is $O(\frac{m}{k} \log m)$, since we have $m > n$.

2. **Shuffle Size and Communication**
   For each step the total shuffle size is O(m_i+n), therefore the total shuffle size is O((m+n) log m). The communication happened in each iteration is all to all communication

3.5 Comparison

As can be seen, the algorithm is off the optimal $O(\frac{m}{k})$ by a fraction of $\log m$ because of the constraint that one root can only change its parent to at most one other root during one iteration, which is the price we need to pay when information is distributed across many different machines.
Remember the distributed algorithm for finding connected components talked during the lecture has time complexity $O(\frac{m}{k})diam(G)$. The guarantee for this algorithm does not depend on the particular graph structure and hence is more stable. It can be faster when $\log(m) < diam(G)$, which is not the rare case. For example, when the graph is a path, the new algorithm can be exponentially faster $O(\log(m))$ versus $O(m)$.