SOMETHING WITH CONNECTED COMPONENTS

Michael Bernasconi, Roman Haag, Giovanni Balduzzi, Lea Fritschi

Department of Computer Science
ETH Zürich
Zürich, Switzerland

ABSTRACT

1. INTRODUCTION

Motivation.

Related work.

Connected components.

For an unordered graph $G = (V, E)$, the connected components are the ensemble of connected subgraphs, where connected means that for any two vertices, there exist a path along the edges connecting them. The straightforward algorithm to find them is to perform either a breath or depth first search from a starting random vertex in V, and give the same label to all the touched vertices. Then repeat the search from an unlabelled vertex until there are no more left. This has a cost in terms of memory accesses of $\Theta(|E|+|V|)$, which turns to be optimal [**?**].

2. PROPOSED ALGORITHM

Unfortunately this algorithm does not parallelize straightforwardly. Pavel Tvrdik [**?**] proposed to cast the problem in terms of the generation of a forest, where the vertices of the same connected comonent belong to the same three, and its root can be used as the representative. We defining a star as a three of height one, a singleton a tree with a single element, and use the variables $n = |V|$ and $m = |E|$. His algorithm can be summarized as:

We defer to [**?**] for a proof of correctness.

After implementing this algorithm we found advantageous to remove the constraint that only singletons and stars can be hooked to another vertex, so that only a single pass through the edge list is required. Extra care is then required during parallel execution: as each vertex has only one outgoing connection, we need to avoid that a process overwrites a connection that has been formed by another one. We therefore need to grow our forest with the following rules:

1. A hook must originate from a vertex id higher than the destination.
2. All edges must generate a connection between the relative vertices, or vertices at an higher level in their three.
3. A hook must originate from a vertex that is currently the root of a tree.

Algorithm 1 Pavel Tvrdik's Connected components

```
1:  procedure HOOK(i, j)
2:      p[p[i]] = p[j]
3:  end procedure
4:  procedure CONNECTEDCOMPONENTS(n, edges)
5:      p[i] = i   ∀i ∈ {1, ··· , n}.        ▷ Initialize a list of parents.
6:      while Elements of p are changed. do
7:          for ⟨i, j⟩ ∈ edges do            ▷ Execute in parallel.
8:              if i ≥ j then Hook(i, j)
9:              if isSingleton(i) then Hook(i, j)
10:         end for
11:         for ⟨i, j⟩ ∈ edges do            ▷ Execute in parallel.
12:             if isStar(i) and i ≠ j then Hook(i, j)
13:         end for
14:         p[i] = root(i)   ∀i ∈ {1, ··· , n}   ▷ Compress the forest in parallel.
15:     end while
16: end procedure
```

The intuitive proof of correctness follows: rule 1 means that the graph gebnerated by the hooks generate a directed graph with no cycles and with at most a single outgoing connection, therefore it must be a forest. Rule 2 and 3 enforce that after processing an edge between two nodes, they belong to the same tree, and rule 3 guarantees that this connection can not be broken by a different edge. At the end of the algorithm, by following the connections from each vertex to the root, we can find a representative for each connected component.

To implement rule 3 in a multithreaded environment, we use an atomic compare and swap. We compare the parent of the hook's origin with its id, if they match it means the vertex is still a root and we hook we hook it to its destina-

tion. It does not matter for correctness if the destination is a root, but we try withouth enforcing to hook to a root to minimize the three height. We found empirically that using `std::atomic_compare_exchange_weak`, compared to `std::atomic_compare_exchange_strong` offers better performance, as we anyway need to loop until a hook is successful.

In pseudocode our algorithm is:

Algorithm 2 Proposed algorithm

```
1:  procedure CONNECTEDCOMPONENTS(n, edges)
2:      p[i] = i   ∀i ∈ {1, ··· , n}.
3:      for ⟨i, j⟩ ∈ edges do                ▷ Execute in parallel.
4:          while hook is not successful. do
5:              from = max(root(i), root(j))
6:              to = mint(root(i), root(j))
7:              atomicHook(from, to)
8:          end while
9:          if !isRoot(i) then p[i] = root(i)
10:         if !isRoot(j) then p[j] = root(j)
11:     end for
12:     p[i] = root(i)   ∀i ∈ {1, ··· , n}    ▷ Compress the
    forest in parallel.
13: end procedure
```

While the step 9 is not necessary for correctness, we found that reusing the already computed vertex's representative leads to a smaller tree height. This and the parallel compression works and was tested to be efficient only on architectures such as x86, where writes to 32 or 64-bits variables storing a label are atomic.

The overall cost of the algorithm is $\Theta((n+m)\langle H\rangle)$, where $\langle H\rangle$ is the average tree height. Theefore $\langle H\rangle = \Theta(1)$ for a subcritical random graph, and on average (relatively to the execution order of the loop) $\langle H\rangle = \Theta(\log(n))$ for a supercrital one [**?**].

Multiple compute nodes.

3. EXPERIMENTAL RESULTS

Results.

4. CONCLUSIONS

5. FURTHER COMMENTS

Here we provide some further tips.