Comparison of Concurrent SCC Search Algorithms

Parv Mor

Indian Institute of Technology, Kanpur

June 2, 2018

Abstract

This text compares two algorithms proposed for concurrent SCC search. The first one is proposed by Robert E. Tarjan while the second one is proposed by Vincent Bloemen *et al.* [**?**]
We refer to the former one by *Algorithm 1* and latter one by *Algorithm 2.*

1 Introduction of Algorithm 1

This introduction is more or less same as one described by Robert E. Tarjan.

The algorithm labels each vertex to be either unvisited, previsited or postvisited. All vertices are seen as a forest of disjoint trees. Initially all vertices are unvisited and hence can be seen as a root with no children. All previsited vertices are either a root or possesses a parent that is previsited. A root is called "idle" if no thread is processing it. An idle thread can grabs a root and starts processing it. If the root was postvisited then do nothing. If that root was unvisited then it is marked as previsited. Next all untraversed outgoing edges of the root are processed. While considering the edge (v, w) we encounter the following cases:

1. w is postvisited: do nothing.
2. v and w are in different trees: w is made parent of v, and marked previsited. Since only roots are eligible for further processing, the thread stops processing v at the moment and becomes "idle".
3. $v = w$: do nothing (Self loops exist in the original graph or can be created by contraction of nodes)
4. v and w are in same tree: Contract all ancestors (including w) into the root v.

If there were no outgoing untraversed edge then all the root is marked postvisited and all its children are made "idle". The algorithm runs until all the vertices are postvisited. Each contracted node represents an SCC.

2 Introduction to Algorithm 2

The algorithm maintains are **shared union find data structure** and each thread maintains a **stack** similar to the one in Tarjan's sequential version. In addition to the usual information each node in union find maintains a bit vector of the threads that are currently processing it. The representative element always contain all the threads that contain any of the nodes in this set in their stack.

Following few observations will be helpful in understanding the algorithm:

1. Call an `SCC`(or node) to be `Dead` if it(or the `SCC` it belongs to) has been completely discovered by the algorithm. If `SCC`s are reported `Dead` in a bottom-up approach then it is sufficient for an `SCC` to be reported `Dead` if: All outgoing arcs from every node in the `SCC` goes to a node that is either in the `SCC` or belongs to a `Dead SCC`.

2. Suppose some thread p traverses the edge (v, w). Let neither w exist in the stack of p nor it be `Dead`. Then, if p exists in the bit vector of representative node of w we have a cycle from $v \to \cdots \to w \to \cdots \to v$. So the thread need not visit w.

What remains is when to mark a node `Dead`?
Following terminologies will be helpful:

1. `GlobalDone`. A node v is said to be `GlobalDone` if all directly neighbouring nodes are either `Dead` or belong to the same set as v.

2. `GlobalDead`. A node v is said to be `GlobalDead` if all nodes in its set are `GlobalDone`. Note that `GlobalDead` implies that node is indeed `Dead`.

To verify if a node is `GlobalDead` we need to iterate over all nodes in the set. As a result a means of exploring all nodes in the set is required and can be achieved by a cyclic linked list. The list is cyclic as representatives can be changed by multiple threads. An node from list is removed if the `GlobalDone` holds for it. Now a verification of `GlobalDead` depends on emptiness of the list. Note that this operations are all constant time.

Hence, each thread keeps on processing the graph until all nodes are marked `GlobalDead`. Each set in the union find represents an `SCC`.

3 Comparison of the algorithms and their implementations

3.1 Experimental comparison

The results excludes the graph loading time. *(Note: Algorithm 1 was ran in Java while Algorithm 2 was ran in C++)*

- Dataset: bakeryN3

Cores	1	2	4	8	16	32
Algorithm 1	-	-	357 sec	-	-	351 sec
Algorithm 2	1366.245000 ms	717.493000 ms	397.887000 ms	231.848000 ms	159.261000 ms	131.375000 ms

- Dataset: tlcN8

Cores	1	2	4	8	16	32
Algorithm 1	-	-	65 sec	-	-	65 sec
Algorithm 2	292.662000 ms	177.975000 ms	104.404000 ms	60.826000 ms	46.014000 ms	35.046000 ms

- Dataset: tlcN9

Cores	1	2	4	8	16	32
Algorithm 1	-	-	303 sec	-	-	309 sec
Algorithm 2	3145.177000 ms	1646.623000 ms	853.695000 ms	452.822000 ms	275.728000 ms	220.425000 ms

- Dataset: smallDG

Cores	1	2	4	8	16	32
Algorithm 1	-	-	≈0 sec	-	-	≈0 sec
Algorithm 2	0.020000 ms	0.031000 ms	0.057000 ms	0.175000 ms	0.427000 ms	0.835000 ms

- Dataset: mediumDG

Cores	1	2	4	8	16	32
Algorithm 1	-	-	≈0 sec	-	-	≈0 sec
Algorithm 2	0.028000 ms	0.029000 ms	0.060000 ms	0.174000 ms	0.546000 ms	0.905000 ms

- Dataset: largeDG

Cores	1	2	4	8	16	32
Algorithm 1	-	-	193 sec	-	-	221 sec
Algorithm 2	753.377000 ms	381.863000 ms	219.540000 ms	117.457000 ms	86.489000 ms	66.603000 ms

3.2 Qualitative comparison

Following might be reasons that might explain the difference in performance of the two:

1. Algorithm 1 was implemented using a **lock-based** design. On the contrary Algorithm 2 uses **lock-less** implementation using `Compare&Swap` operation.

2. Algorithm 1 does not uses path compression to find if two nodes are in the same tree/set, while Algorithm 2 does which can result in poor performance of Algorithm 1.

3. Algorithm 1 allows only "idle" roots to be processed by threads, while Algorithm 2 possess no such restriction.

Incorporating the **shared union find structure with cyclic linked list** in the existing implementation should improve performance of the Algorithm 1. If not, yet another approach would be to adopt a lockless implementation of the same.

References

[1] Vincent Bloemen, Alfons Laarman, and Jaco van de Pol. 2016. Multi-core on-the-fly SCC decomposition. SIGPLAN Not. 51, 8, Article 8 (February 2016), 12 pages. DOI: https://doi.org/10.1145/3016078.2851161