Lessons learned about concurrent SCC search proposed by Robert E. Tarjan

Markus Alexander Kuppe

08/23/2015 (v 0.3)

1 Algorithm (by Robert Tarjan) (Leslie/Markus' amendments in red)

As a basis for a concurrent algorithm I propose the naïve one-way algorithm, using a disjoint-set data structure to keep track of components. This data structure is simple and its time overhead is small, and, as we shall see, the search paths of several threads can merge, resulting in an in-tree (or more generally an in-forest). As a result, I don't see how to use a stack mechanism to keep track of components. (Maybe there is a way to do it, but that is future research; it might not save much time in practice.)
To efficiently keep track of the colors in each component, I propose using a bit vector or something like a Bloom filter. This part of the algorithm is orthogonal to generating the components, so it can be designed separately.

Concurrent algorithm

To explain the algorithm, I'll describe it as contracting cycles immediately, although a good implementation would not do the contractions explicitly (I think). Each vertex is either unvisited, previsited, or postvisited. The unvisited and previsited vertices form an in-forest (a set of in-trees[1]): each unvisited vertex is a root with no children; each previsited vertex is either a root or it has a parent that is previsited. Initially all vertices are unvisited. Only roots are eligible for processing. I'll call a root "idle" if no thread is processing it. An idle thread can grab an idle root and start processing it. To process a root, if it is unvisited, mark it PRE-*visited. Then traverse the next outgoing untraversed arc* of that node*; if there is no such arc, mark the root postvisited and make all its children idle roots. To traverse an arc (v, w), if w is postvisited do nothing; otherwise, if w is in a different tree than v, make w the parent of v and mark w previsited if it is unvisited. Since v is now a child, it is not (for the moment) eligible for further processing. The thread can switch to an arbitrary root, or it can check to see if the root of the tree containing v and w is idle, and switch to this root if*

[1] An in-tree is one in which edges point from a node to its parent, the root having no parent

so. The other possibility is that w is in the same tree as v. If $v = w$, do nothing. (Self-loops can be created by contractions or exist in the original graph.*) If $v \neq w$, contract all the ancestors of w* and w itself *into vertex* v, which is the root of w. *Continue processing this root. (The new root inherits all the outgoing untraversed arcs from the set of contracted vertices). It may be convenient and efficient to view this contraction as contracting all the vertices into v, since v is already a root.*
Continue until all vertices are postvisited. Upon termination, the set of nodes in each contracted node represents maximal SCC of the original graph.

Implementation Details

One can use two disjoint-set data structures here, one to keep track of the partition of vertices into trees, the other to keep track of the partition of vertices into components. These partitions are nested, so one can actually use a single data structure to keep track of both partitions [ ], which may or may not be a good idea. Each component will also need to keep track of the set of vertices in it, in order to find untraversed arcs to traverse.
I can fill in additional details of this algorithm as necessary. I think it should work well in practice, but of course this needs to be tested. There is an existing concurrent algorithm (along with several refinements), but I see no reason to think it would run faster than the algorithm described above, and I suspect it would run considerably slower. I'll add a description of it to these notes. Perhaps surprisingly, the algorithm above seems to be new. At least I have not yet found a description of it in the literature.

Errata note

I overlooked one thing in my high-level description of the proposed algorithm: when a root runs out of outgoing arcs to be traversed and it is marked as post visited, deleting it breaks its tree into s number of new trees, one per child. This means that one cannot use the disjoint-set data structure to keep track of trees, since sets must be broken up as well as combined. There are efficient data structures to solve this more-complicated problem, notably Euler tour trees, which represent a tree by an Euler tour stored in a binary search tree. The time for a query is O(logn), as is the time to add an arc (a link) or break an arc (a cut).
My suggestion would be to implement one of the linear-time sequential algorithms as well as a version of the concurrent algorithm (assuming we nail down the details) that uses naive data structures, and do some experiments. If you don't get the speed you want, we can try a more-sophicticated implementation, such as one using Euler tour trees or maybe even dynamic trees.

2 Prototype

- A greenfield prototype has been implemented to study the characteristics of the algorithm.
- Independent of TLC to be able to rapidly explore ideas and not be constrained by TLC's idiosyncrasies.
- Still resembles TLC data structures, i.e. API of TLC's TableauDiskGraph. TLC port should thus be relatively easy.
  - Think of TableauDiskGraph as an adjacency list.
- Prototype contains an extensive performance test suite.
  - Input graphs are too large for git repository to handle. Download from locations listed in org.kuppe.graphs.tarjan.ConcurrentFastSCCTestFromFile.
- Tests have been executed on artificially created and real-world graphs of various sizes.
- Core count from 1 to 32 cores (32 cores is the maximum count on Azure and Amazon AWS).
- Graph size seems too to small (~10GB) for scalability advantages to be measurable (comparable to sequential algorithm).
  - TLC revealed another performance bottleneck during the creation of even larger liveness graphs.
- Runtime is compared to Robert Tarjan's text-book sequential algorithm and TLC's implementation.

3 Tree algorithm

- Forest of dynamically created trees.
  - For each two nodes (v & w) in the original graph connected via an edge, the SCC search runs queries to determine if two nodes (v & w where w is reachable from v via one of v's out-arcs) happen to be in the same tree or not. A query is a tree traversal from either a leaf or internal node up to the tree's root node. Trees in the forest are created on-the-fly by the SCC search by making v a tree child of w when the query returned that v is not an ancestor (root) of w.
- Initially used an adapted Link-cut tree O(log n) implementation originally authored by [Sleator and Endre Tarjan, 1983].

- Robert Tarjan suggested to use a dynamic tree (Link-cut being such a tree).
    - Abandoned due to locking/concurrency problems caused by explicit tree rotations even for queries.
- Naive tree O(n) now in use.
    - Robert Tarjan's independent measurements [Tarjan and Werneck, 2009] revealed that a naive tree can even be faster than more sophisticated dynamic trees (this result is only valid for non-concurrent/exclusive access).
    - Approximately 1/4 of the runs result in a unbalanced tree of height of > 60.
- Looked into Euler-Tour-Trees (ETT) as a replacement for Naive tree. ETTs are built on top of Red-Black or AVL trees. Both still suffer from rotations during query. Current research is looking into better support for concurrent Red-Black/AVL trees (lazy rotations) [Gramoli, 2015, Morrison, Nurmi and Soisalon-Soininen, 1996, Korenfeld, 2012]. Protocol/Algorithms complex and mostly without ready to use implementations (would have to come up with our own implementation which is a major undertaking on its own).

4 Contractions

- The current prototype uses explicit contraction (Robert Tarjan hinted to use implicit contraction but I do not understand how that works): When a node v is contracted into its tree root w:
    - v's out-arcs are moved (copy & remove) into w.
    - Iff v happens to be the former root of a subset SCC (meaning an earlier contraction of u into v occurred), the subset SCC has to be merged into w.
        - *LL: "I don't understand this. If s, t, and u have been contracted into v, then haven't their out-arcs become out-arcs of v, so their arcs to w become self-loops of w?*
            - Yes, the out-arcs of s, t and u earlier became out-arcs of v and thus - once v gets contracted into w - become self-loops of w.
        - It uses a custom implementation of a linked list that supports merge/append in O(1) (Java's generic one has O(n)).

  - (Eager path contraction) The graph has to be updated so that out-arcs ending at v now point to w (otherwise out-arc exploration from p -> v would incorrectly result in the out-arc being ignored due to v's POST-visited state when it's logically correct state is w's visited state).
  - Iff v happens to be the former root of subset SCCs (meaning earlier contractions of s, t, u into v occurred), the old mappings from s, t, u -> v have to be updated to s, t, u -> w for node p with out-arcs to s correctly checking w's visited state.
    * The set of nested mapping updates grows with each contraction.
    * Statistics show that mapping updates do not significantly add to the algorithms runtime so far.

- ~~Claim: Subset SCCs do not have to be merged into larger SCC when used for real liveness checking. It suffices to check each subset SCC individually if it satisfies the counter-example.~~
  - ~~*LL: For this claim to be correct, the properties that hold in the subset SCC have to be recorded (i.e. "the counterexample must satisfy []<>A /\ []<>B and A is satisfied by one subset SCC and B by the other)".*~~
  - ~~It is open if it is faster/cheaper to just merge the SCCs or record their properties.~~
    * *LL: "This makes no sense to me. We start with a bunch of SCCs–namely, each node with its self-loop is an SCC. If we didn't have to merge SCCs, then there would be no need to do any SCC computation."*

- Contraction moves the out-arcs of the to be contracted nodes into the root node. This leads to redundant out-arc exploration when the sets of out-arc intersects. For space reason explored out-arcs are deleted and subsequently garbage collected. Also, experiments showed that the overhead to sort the out-arcs for efficient set membership queries is significant.

- Alternatively, lazy path contraction in place of eager one would trade repeated graph update cost with more expensive lookups, i.e. follow contractions from u -> v -> w -> .... -> r. More lookups also result in increased graph lock contention and require extra space in each graph node (pointer from contracted node to the one its been contracted into). If the graph is generally sparse, lazy contraction is probably preferable to eager contraction because of the reduced likelihood to have multiple edges going to the same node. If the graph is dense though, eager contraction should yield better performance because out-arc traversal generally requires a single graph lookup.

5 Locking

- Current implementation is based on one RW-lock per node.
  - Causes increased memory requirements per graph node.
- Lock protocol:
  1. v is an idle root eligible for processing by a thread T.
     (a) v is write-locked by T.
     (b) Iff v's write-lock acquisition fails, v's processing is re-scheduled.
  2. The next unexplored out-arc of v is followed to a w.
     (a) w is write-locked by T.
     (b) Iff w's write-lock acquisition fails, v's write lock is released and v's processing is re-scheduled.
  3. The root r of w is queried (compare 3) by traversing the path from w to r.
     (a) r is read-locked by T.
     (b) Iff r's read-lock acquisition fails (or any read-lock acquisition on the path from w to r), read-lock acquisition re-starts at w because the tree has changed and the read parent pointer might be stale.
     (c) (Tree) Traversal uses hand-over-hand locking (only ever holds two locks of the path up to the root).
- Justification for obtaining w's root r: w's root has to be read-locked to prevent concurrent re-parenting of v -> w and w -> v (see figure 5.1).
- Finding "master" lock per tree is a similar problem to find-root/checking if v and w are in an identical tree.
  - If tree height is relatively huge, failed lock acquisition adds notably to overall runtime.
- TODO: Model check lock protocol once runtime of algorithm shows superior performance over sequential algorithm.
- Explore adoption of more sophisticated lock protocols [Golan-Gueta et al., 2011, Lanin and Shasha, 1990].
- Thought about exploring a lock-free approach. Not started yet due to its complexity.

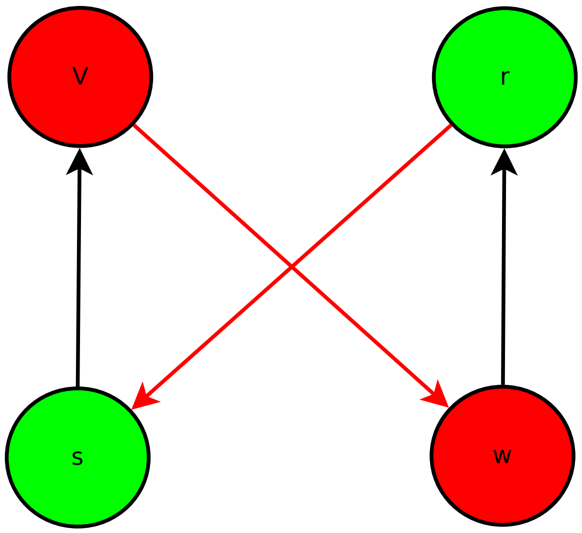

Figure 5.1: Counter-example showing cyclic re-parenting of v to w and r to s when w's root r is not read-locked. Node colors indicate corresponding locks, black marks tree edges, red graph out-arcs.

6 Shortcomings of proposed algorithm

- On 02/20/2015 Robert Tarjan in an email amended the algorithm to account for the case where a tree root's out-arcs have been fully explored and thus becomes ineligible for further processing. Since it is a tree root, its children have to be explored for the algorithm to be complete. However, the algorithm only permits roots to be explored. Thus, the children have to be freed from the fully explored root. This requires that it is possible to navigate from a root to its tree children hence forcing the tree to be bi-directional. The overhead incurred by this requirement is significant because the children set has to be combined during contractions where it can grow to very large sizes.
  - *LL: "This isn't a change to the algorithm, which (quoting from above) says "if there is no such [further outgoing] arc, mark the root postvisited and make all its children idle roots." Do you mean that he observed that implementing this part of the algorithm requires maintaining bi-directional links? This seems to be just a part of the whole problem of how you find nodes that need to be explored."*
    - Yes, implementing it requires bi-directional links in forest trees. Otherwise, I do not see how the implementation would know the set of children to become idle roots.
- The algorithm requires that each node in the graph is explored. Thus,

for each graph node a worker has to be started to explore its out-arcs. However, at a certain point in the search most nodes have already changed into the POST-visited state and thus could be skipped. Scheduling a worker is just overhead.

  - The sequential/traditional algorithm does not mandate that the search is started from each graph node. It suffices if the search starts from all initial nodes (which are known to reach every other (non) initial node in the same behavior).

- If the graph has one or a few large SCCs it tends to reach a state at the end of the search where it runs essentially sequential (see 6.1). This is because the number of forest trees gets down to a very low number possibly a single one and thus only a few root nodes are eligible for processing. Usually these nodes have a huge set of unexplored out-arcs. The out-arcs are explored sequentially.

  - Leslie and I talked about an extensions to the algorithm to explore out-arcs in parallel for as long as the exploration does not lead to re-parenting (contractions and out-arcs pointing to POST-visited nodes can run in parallel). This extension has not been implemented yet due to its extra complexity (especially the case of nested contractions requires clever locking).

7 Advantages of algorithm over TLC's current sequential one

- Incremental SCC search possible due to contraction (don't search for the same SCCs over and over again)

  - Analysis and proof of correctness for this claim still missing, i.e. what happens if an already contracted graph node is updated with a new out-arc? This does not occur with regular breadth-first model-checking where all out-arcs are generated as part of the next state generation, but in simulation mode out-arcs are generated incrementally.

    * Simulation mode is limited to graphs (sequences) of a few hundred or thousand nodes. The sequential SCC search is therefore sufficient.

  - Can an action constraint correctly be checked from a contracted node?

    * Non-contracted node could be P and new out-arc could lead to ¬P, however the P node could be contracted into ¬P and thus incorrectly falsifying/true-ifying an action constraint.

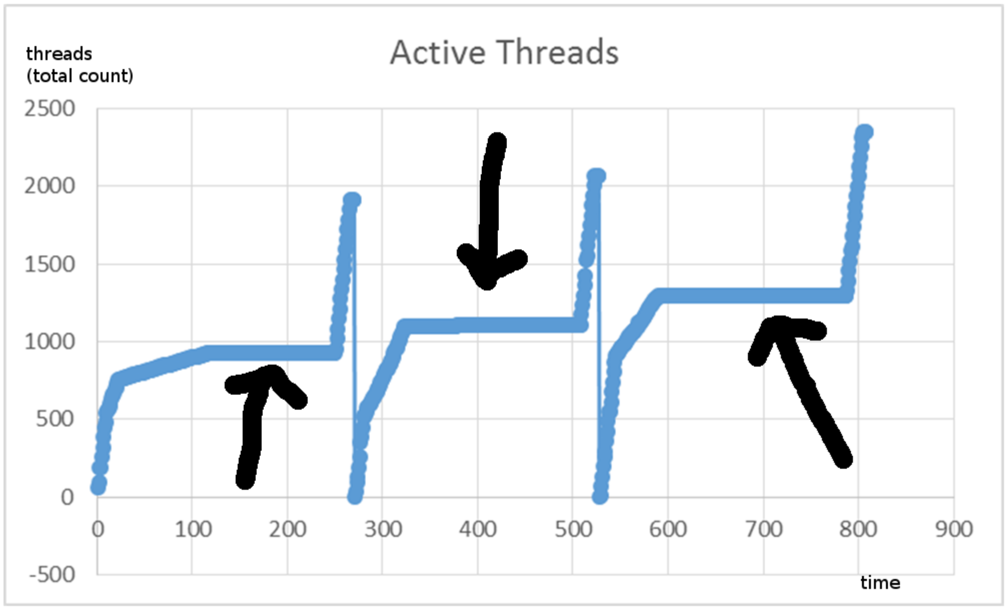

Figure 6.1: Graph shows three consecutive runs of the algorithm. All three of them exhibit extended periods of almost no concurrent node exploration (flat sections pointed to by black arrows). The flat line indicates that no new worker threads are started during that period because the total number of threads does not increase. As we know from OS statistics, all but a single core were idle during that time. These flat periods are the dominant contributors to overall runtime.

      * *LL: "For an action constraint, whether or not the action is satisfied should be attached to the arc, not to a node. That includes self-looping arcs. What you say is true for state-predicate formulas. However, the SCC algorithm is for finding the SCCs. Once you've found them, don't you then have to determine if any of them can be used to provide a counter-example? As I recall, the Manna-Pnueli algorithm for dealing with fairness involves some kind of successive decomposition of the graph. But doesn't that involve finding SCCs at each stage of the decomposition?"*
        - => Needs further investigation (I do not recall any decomposition though)

- SCC search can run in parallel with TLC's check of found components. In TLC's current sequential algorithm the process is two staged. First an SCC is searched until an SCC is found (not necessarily a maximal one). Then, the search is stopped to check the SCC if it satisfies the liveness counter-example. The concurrent algorithm - by design - will continue the search for SCCs with all worker threads except the one checking the counter-example.

8 Random (minor) optimizations

- Under the assumption that SCCs generally are of close graph locality and to reduce lock contention, randomizing and partitioning graph iterators have been implemented. Both attempt to evenly spread the workers across the graph. However, neither produced significant performance improvements. It is unknown if this is due to the underlying assumption being incorrect or due to other factors.

- Stop SCC search the moment all nodes in the graph (remember adjancecy list) are POST-visited. Unexplored arcs cannot end at UN-visited nodes.

- (Generally) run tear-down of forest trees (free'ing) in parallel. Especially at the end of the SCC search when there are no out-arcs left, all forest trees are completely traversed (breadth-first like). At each level the parent-child relationship is removed which returns the former childs to the set of idle roots and thus makes them eligible for out-arc exploration. For each former child a worker thread is started only to determine that the former child has no out-arcs left. It then again removes the next level of the parent-child relationship and schedules workers for each former child.

- (Somehow) bias forest to grow many small trees over a few large ones to increase the likelihood of less lock contention.
    - *LL: "Can this be done by assigning available threads to check unexplored nodes rather than unexplored edges?"*

- ∗ You are saying that the algorithm should randomly select nodes from the set of UN-/PRE-visited ones and only explore a single out-arc before moving on to the next one?
    - · *LL: "I am asking if that would achieve your goal of growing many small trees rather than a few large ones."*
    - · When I looked into the idea of randomly selecting nodes and only processing a single out-arc before selecting the next node, it showed no real performance improvement.

References

Guy Golan-Gueta, Nathan Bronson, Alex Aiken, G. Ramalingam, Mooly Sagiv, and Eran Yahav. Automatic fine-grain locking using shape properties. In *ACM SIGPLAN Notices*, volume 46, pages 225–242. ACM, 2011. URL `http://dl.acm.org/citation.cfm?id=2048086`.

Vincent Gramoli. More than you ever wanted to know about synchronization: synchrobench, measuring the impact of the synchronization on concurrent algorithms. pages 1–10. ACM Press, 2015. ISBN 978-1-4503-3205-7. doi: 10.1145/2688500.2688501. URL `http://dl.acm.org/citation.cfm?doid=2688500.2688501`.

Boris Korenfeld. *CBTree: A Practical Concurrent Self-Adjusting Search Tree.* PhD thesis, Tel Aviv University, 2012. URL `http://www.cs.tau.ac.il/thesis/thesis/Korenfeld.Boris-MSc.thesis.pdf`.

Vladimir Lanin and Dennis Shasha. *Tree locking on changing trees.* New York University, Department of Computer Science, Courant Institute of Mathematical Science, 1990. URL `http://www.cs.nyu.edu/web/Research/TechReports/TR1990-503/TR1990-503.pdf`.

Yehuda Afek Boris Korenfeld Adam Morrison. Concurrent Search Tree by Lazy Splaying. URL `http://www.cs.tau.ac.il/~afek/LazySplaying.pdf`.

Otto Nurmi and Eljas Soisalon-Soininen. Chromatic binary search trees. *Acta Informatica*, 33(6):547–557, September 1996. ISSN 0001-5903, 1432-0525. doi: 10.1007/s002360050057. URL `http://link.springer.com/10.1007/s002360050057`.

Daniel D. Sleator and Robert Endre Tarjan. A data structure for dynamic trees. *Journal of Computer and System Sciences*, 26(3):362–391, June 1983. ISSN 00220000. doi: 10.1016/0022-0000(83)90006-5. URL `http://linkinghub.elsevier.com/retrieve/pii/0022000083900065`.

Robert E. Tarjan and Renato F. Werneck. Dynamic trees in practice. *Journal of Experimental Algorithmics (JEA)*, 14:5, 2009. URL `http://dl.acm.org/citation.cfm?id=1594231`.