Multi-Core On-The-Fly SCC Decomposition

Vincent Bloemen
Formal Methods and Tools,
University of Twente
v.bloemen@utwente.nl

Alfons Laarman
FORSYTE,
Vienna University of Technology
alfons@laarman.com

Jaco van de Pol
Formal Methods and Tools,
University of Twente
j.c.vandepol@utwente.nl

Abstract

The main advantages of Tarjan's strongly connected component (SCC) algorithm are its linear time complexity and ability to return SCCs on-the-fly, while traversing or even generating the graph. Until now, most *parallel* SCC algorithms sacrifice both: they run in quadratic worst-case time and/or require the full graph in advance.

The current paper presents a novel parallel, on-the-fly SCC algorithm. It preserves the linear-time property by letting workers explore the graph randomly while carefully communicating partially completed SCCs. We prove that this strategy is correct. For efficiently communicating partial SCCs, we develop a concurrent, iterable disjoint set structure (combining the union-find data structure with a cyclic list).

We demonstrate scalability on a 64-core machine using 75 real-world graphs (from model checking and explicit data graphs), synthetic graphs (combinations of trees, cycles and linear graphs), and random graphs. Previous work did not show speedups for graphs containing a large SCC. We observe that our parallel algorithm is typically 10-30× faster compared to Tarjan's algorithm for graphs containing a large SCC. Comparable performance (with respect to the current state-of-the-art) is obtained for graphs containing many small SCCs.

* Categories and Subject Descriptors CR-number [*subcategory*]: third-level

* Keywords strongly connected components, SCC, algorithm, graph, digraph, parallel, multi-core, union-find, depth-first search

1. Introduction

Sorting vertices in depth-first search (DFS) postorder has turned out to be important for efficiently solving various graph problems. Tarjan first showed how to use DFS to find biconnected components and SCCs in linear time [52]. Later it was used for planarity [26], spanning trees [53], topological sort [14], fair cycle detection [16] (a problem arising in model checking [57]), vertex covers [47], etc.

Due to irreversible trends in hardware, parallelizing these algorithms has become an urgent issue. The current paper focuses on solving this issue for SCC detection, improving SCC detection for large SCCs. But before we discuss this contribution, we discuss the problem of parallelizing DFS-based algorithms more generally.

PPoPP '16 March 12-16, 2016, Barcelona, Spain
Copyright © 2016 ACM 978-1-4503-4092-2/16/03…$15.00
DOI: http://dx.doi.org/10.1145/2851141.2851161

Traditional parallellization. Direct parallelization of DFS-based algorithms is a challenge. Lexicographical, or ordered DFS is P-complete [45], thus likely not parallelizable as under commonly held assumptions, P-complete and NC problems are disjoint, and the latter class contains all efficiently parallelizable problems.

Therefore, many researchers have diverted to phrasing these graph problems as fix-point problems, which can be solved with highly parallelizable breadth-first search (BFS) algorithms. E.g., we can rephrase the problem of finding all SCCs in $G \stackrel{\text{def}}{=} (V, E)$, to the problem of finding the SCC $S \subseteq V$ to which a vertex $v \in V$ belongs, remove its SCC $G' = G \setminus S$, and reiterate the process on G'. S is equal to the intersection of vertices reachable from v and vertices reachable from v after reversing the edges E (backward reachability). This yields the quadratic ($\mathcal{O}(n \cdot (n+m))$) Forward-Backward (FB) algorithm (here, $n = |V|$ and $m = |E|$). Researchers repeatedly and successfully improved FB [13, 20, 25, 42, 48], which reduced the worst-case complexity to $\mathcal{O}(m \cdot \log n)$.

A negative side effect of the fix-point approach for SCC detection is that the backward search requires storing all edges in the graph. This can be done using e.g., adjacency lists or incidence matrices [9] in various forms [30, Sec. 1.4.3], however always at least takes $\mathcal{O}(m + n)$ memory. On the contrary, Tarjan's algorithm can run *on-the-fly* using an implicit graph definition $I_G = (v_0, \textsc{post}())$, where $v_0 \in V$ is the initial vertex and $\textsc{post}(v) \stackrel{\text{def}}{=} \{v' \in V \mid (v, v') \in E\}$, and requires only $\mathcal{O}(n)$ memory to store visited vertices and associated data. The on-the-fly property is important when handling large graphs that occur in e.g. verification [12], because it may allow the algorithm to terminate early, after processing only a fraction ($\ll n$) of the graph. It also benefits algorithms that rely on SCC detection but do not require an explicit graph representation, e.g. [41].

Parallel Randomized DFS (PRDFS). A novel approach has shown that the DFS-based algorithms can be parallelized more directly, without sacrificing complexity and the on-the-fly property [8, 18, 19, 31–33, 36, 37, 46]. The idea is simple: (1) start from naively running the sequential algorithm on p independent threads, and (2) globally prune parts of the graph where a local search has completed.

For scalability, the PRDFS approach relies on introducing randomness to direct threads to different parts of a large graph. Hence the approach can not be used for algorithms requiring lexicographical, or ordered DFS. But interestingly, none of the algorithms mentioned in the first paragraph require a fixed order on outgoing edges of the vertices (except for topological sort, but for some of its applications the order is also irrelevant [5]), showing that the oft-cited (cf. [4–6, 10, 11, 20]) theoretical result from Reif [45] does not apply directly. In fact, it might even be the case that the more general problem of non-lexicographical DFS is in NC.

For correctness, the pruning process in PRDFS should carefully limit the influence on the search order of other threads. Trivially,

Table 1: Complexities of fix-point (e.g. [13]) and PRDFS solutions (e.g. [46]) for problems taking linear time sequentially: $\mathcal{O}(n+m)$. Here, n and m represent the number of vertices and edges in the graph and p the number of processors.

	Best-case ($\mathcal{O}$)			Worst-case ($\mathcal{O}$)		
	Time	Work	Mem.	Time	Work	Mem.
Traditional	$\frac{n+m}{p}$	$n+m$	$n+m$	$m \cdot \log n$	$m \cdot \log n$	$n+m$
PRDFS	$\frac{n+m}{p}$	$n+m$	n	$n+m$	$p \cdot (n+m)$	$p \cdot n$

in the case of SCCs, a thread running Tarjan can remove an SCC from the graph as soon as it is completely detected [37], as long as done atomically [46]. This would result in limited scalability for graphs consisting of a single SCC [46]. But, more intricate ways of pruning already showed better results in other areas [8, 18, 32, 33].

Time and work tradeoff play an import role in parallelizing many of the above algorithms [50]. In the worst case, e.g. with a linear graph as input, the PRDFS strategy cannot deliver scalability – though neither can a fix-point based approach for such an input. The amount of work performed by the algorithm in such cases is $\mathcal{O}(p \cdot (n+m))$, i.e.: a factor p, i.e. the number of processors, compared to sequential algorithm ($\mathcal{O}(n+m)$). However, the *runtime* never degrades beyond that of the sequential algorithm $\mathcal{O}(n+m)$, under the assumption that the synchronization overhead is limited to a constant factor. This is because the same strategy can be used that makes the sequential algorithm efficient in the first place. Table 1 compares the two parallelization approaches. The hypothesis of the current paper is that a scalable PRDFS-based SCC algorithm can solve parallel on-the-fly SCC decomposition.

Contribution: PRDFS for SCCs. The current paper provides a novel PRDFS algorithm for detecting SCCs capable of pruning partially completed SCCs. Prior works either lose the on-the-fly property, exhibit a quadratic worst-case complexity, or show no scalability for graphs containing a large SCC. Our proof of correctness shows that the algorithm indeed prunes in such a way that the local DFS property is preserved sufficiently. Experiments show good scalability on real-world graphs, obtained by model checking, but also on random and synthetic graphs. We furthermore show practical instances (on explicitly given graphs) for which existing work seems to suffer from the quadratic worst-case complexity.

Our algorithm works by communicating partial SCCs via a shared data structure based on a union-find tree for recording disjoint subsets [54]. In a sense therefore it is based on SCC algorithms before Tarjan's [40, 44]. The overhead however is limited to a factor defined by the inverse Ackermann function α (rarely grows beyond a constant 6), yielding a quasi-linear solution.

We avoid synchronization overhead by designing a new *iterable* union-find data structure that allows for concurrent updates. The subset iterator functions as a queue and allows for elements to be removed from the subset, while at the same time the set can grow (disjoint subsets are merged). This is realized by a separate cyclic linked list, containing the nodes of the union-find tree. Removal from the list is done by collapsing the list like a shutter, as shown in Figure 1. All nodes therefore invariably point to the sublist, while *path compression* makes sure that querying queued vertices always takes an amortized constant time. Multiple workers can concurrently explore and remove nodes from the sublist, or merge disjoint sublists.

The paper is organized as follows: Section 2 provides preliminaries on SCC decomposition and present a sequential set-based SCC algorithm. Section 3 describes our new multi-core SCC algorithm with a proof of correctness. In Section 4, we provide an implementation of our algorithm based on the iterable union-find data structure and discuss its memory usage and runtime complexity. The related work is discussed in Section 5. We discuss an experimental evaluation in Section 6 and provide conclusions in Section 7.

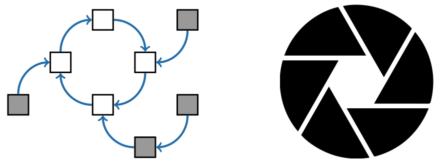
Figure 1: Schematic of the concurrent, iterable queue, which operation resembles a closing camera shutter (on the right). White nodes (invariably on a cycle) are still queued, whereas gray nodes have been dequeued (contracting the cycle), but can nonetheless be used to query queued nodes, as they invariably point to white nodes.

2. Preliminaries

Given a directed graph $G \overset{\text{def}}{=} (V, E)$, we denote an edge $(v, v') \in E$ as $v \rightarrow v'$. A *path* is a sequence of vertices $v_0 \ldots v_k$, s.t. $\forall_{0 \leq i \leq k} : v_i \in V$ and $\forall_{0 \leq i < k} : v_i \rightarrow v_{i+1}$. The transitive closure of E is denoted $v \rightarrow^* w$, i.e. there is some path $v \ldots w \in V^*$. If a vertex reaches itself via a non-empty path, the path is called a *cycle*. Vertices v and w are *strongly connected* iff $v \rightarrow^* w \wedge w \rightarrow^* v$, written as $v \leftrightarrow w$. A *strongly connected component (SCC)* is defined as a maximal vertex set $C \subseteq V$ s.t. $\forall v, w \in C : v \leftrightarrow w$. For the graph's size, we denote $n \overset{\text{def}}{=} |V|$ and $m \overset{\text{def}}{=} |E|$.

Since we focus on on-the-fly graph processing, our algorithms do not have access to the complete graph G, but instead access an implicit definition: $G_I \overset{\text{def}}{=} (v_0, \text{POST}())$, where v_0 is an initial vertex and $\text{POST}(v)$ a *successors function*: $\text{POST}(v) \overset{\text{def}}{=} \{w \mid v \rightarrow w\}$. The structural definition G is however used in our correctness proofs.

A sequential set-based SCC algorithm. One of the early SCC algorithms, before Tarjan's, was developed by Purdom [44], and later optimized by Munro [40]. Like Tarjan's algorithm it uses DFS, but this is not explicitly mentioned (Tarjan was the first to do so). Moreover, instead of keeping vertices on the stack, the set-based algorithm stores partially completed SCCs (originally together with a set of all outgoing vertices of these vertices). These sets can be handled in amortized, quasi- constant time by storing them in a union-find data structure (introduced below).

As a basis for our parallelization, we first generalize Munro's algorithm to store the vertices instead of edges of partially completed SCCs. We also add the ability to collapse cycles into partial SCCs immediately (as in Dijkstra [17]). We refer to this as the *Set-Based* SCC algorithm, which is presented in Algorithm 1. Its essence is to perform a DFS and collapse cycles to *supernodes*, which constitute (partial) strongly connected components.

The strongly connected components are tracked in a collection of disjoint sets, which we represent using a map: $\mathcal{S} \colon V \rightarrow 2^V$ with the invariant: $\forall v, w \in V \colon w \in \mathcal{S}(v) \Leftrightarrow \mathcal{S}(v) = \mathcal{S}(w)$. As a consequence, all the mapped sets disjoint and retrievable in the map via any of its member. This construction allows us later to iterate over the sets. A UNITE function merges two mapped sets, s.t. the invariant is maintained, e.g.: let $\mathcal{S}(v) = \{v\}$ and $\mathcal{S}(w) = \{w, x\}$, then $\text{UNITE}(\mathcal{S}, v, w)$ yields $\mathcal{S}(v) = \mathcal{S}(w) = \mathcal{S}(x) = \{v, w, x\}$ keeping all other mappings the same.

The algorithm's base DFS can be seen on Line 6, Line 8, and Line 10-11 (ignoring the condition at Line 15). The recursive procedure is called for every unvisited successor vertex w from v. Because cycles are collapsed immediately at Line 12–14, the

Algorithm 1 Sequential set-based SCC algorithm

```
1: $\forall v \in V : \mathcal{S}(v) := \{v\}$
2: DEAD := VISITED := $\emptyset$
3: $R := \emptyset$
4: SETBASED($v_0$)
5: procedure SETBASED($v$)
6:     VISITED := VISITED $\cup \{v\}$
7:     $R$.PUSH($v$)
8:     for each $w \in$ POST($v$) do
9:         if $w \in$ DEAD then continue ....... [already completed SCC]
10:        else if $w \notin$ VISITED then ................ [unvisited node $w$]
11:            SETBASED($w$)
12:        else while $\mathcal{S}(v) \neq \mathcal{S}(w)$ do .................. [cycle found]
13:            $r := R$.POP()
14:            UNITE($\mathcal{S}, r, R$.TOP())
15:    if $v = R$.TOP() then ............... [completely explored SCC]
16:        report SCC $S(v)$
17:        DEAD := DEAD $\cup$ $S(v)$
18:        $R$.POP()
```

stack R is maintained independently of the program stack and invariantly contains a subset of the latter (in the same order). Note that the algorithm is called for an initial node v_0 (Line 4), i.e. only vertices reachable from v_0 are considered. W.l.o.g. we assume that all vertices are reachable from v_0 (as defined with G_I).

The algorithm partitions vertices in three sets, s.t. $\forall v \in V$, either:

a. $v \in$ DEAD, implying that v is part of a completely explored (and reported) SCC,
b. $v \notin$ VISITED, (hence also $v \notin$ DEAD) implying that v has not been encountered before by the algorithm, or
c. $v \in$ LIVE, implying that v is part of a partial component, i.e. LIVE $\overset{\text{def}}{=}$ VISITED $\setminus$ DEAD.

It can be shown that the algorithm ensures that all supernodes for the vertices on R are disjoint and together contain all LIVE vertices:

$$\biguplus_{v \in R} \mathcal{S}(v) = \text{LIVE} \quad (1)$$

As a consequence, the LIVE vertices can be reconstructed using the map $\mathcal{S}$ and R, so we do not actually need a VISITED set except for ease of explanation. Furthermore, it can be shown that all LIVE vertices have a unique representation on R:

$$\{\mathcal{S}(v) \cap R \mid v \in \text{LIVE}\} = \{\{r\} \mid r \in R\} \quad (2)$$

Or, for each LIVE vertex v, its mapped supernode $\mathcal{S}(v)$ has exactly one $r \in R \cap \mathcal{S}(v)$ s.t. $\mathcal{S}(v) = \mathcal{S}(r)$. Both equations play a role in ensuring that the algorithm returns complete SCCs, explained next. However, we will also use them in the subsequent section to guide the parallelization of the algorithm.

From the above, we can illustrate how the algorithm decomposes SCCs. If a successor w of v is LIVE, it holds that $\exists r \in R : \mathcal{S}(r) = \mathcal{S}(w)$. Such a LIVE vertex is handled by Line 12-14, where the top two vertices from R are united, until r is encountered, or rather until it holds that $\mathcal{S}(r) = \mathcal{S}(v) = \mathcal{S}(w)$. Because r has a path to v (an inherited invariant of the program stack), all united components are indeed strongly connected, i.e. lie on a cycle. Moreover, because of r's uniqueness (Equation 2), there is no other LIVE component part of this cycle (eventually this guarantees completeness, i.e. that the algorithm reports a *maximal* strongly connected component).

A completed SCC $\mathcal{S}(v)$ is reported and marked DEAD if v remains on top of the R stack at Line 15, indicating that v could not be united with any other vertices in R (lower on R).

Union-find. The *disjoint set union* or *union-find* [54] is a data structure for storing and retrieving a disjoint partition of a set of *nodes*. It also supports the merging of these partitions, allowing an incremental coarsening of the partition. A set is identified by a single node, the *root* or *representative* of the set. Other nodes in the set use a *parent* pointer to direct towards the root (an inverted tree structure). The FIND(a) operation recursively searches for the root of the set and updates the parent pointer of a to directly point to its root (path-compression). The operation UNITE(a, b) involves directing the parent pointer from the root of a to the root of b, or vice versa. By choosing the root with the highest identifier (assuming a uniform distribution) as the 'new' root, the tree remains asymptotically optimally structured [23]. Operations on the union-find take amortized quasi-constant time, bounded by the inverse Ackermann function α, which is practically constant.

Union-find is well-suited to maintain the SCC vertex sets as the algorithm coarsens them (by collapsing cycles). In Algorithm 1, the map $\mathcal{S}$ and the UNITE() operation can be implemented with a union-find structure as shown by [22]. This results in quasi-linear runtime complexity: each vertex is visited once (linear) executing a constant number of union-find operations ('quasi').

3. A Multi-Core SCC Algorithm

The current section describes our multi-core SCC algorithm. The algorithm is based on the PRDFS concept, where $\mathcal{P}$ workers randomly process the graph starting from v_0 and prune each other's search space by communicating parts of the graph that have been processed. (This dynamically, but not perfectly, partitions the graph accross the workers, trading redundancy for communication.) To the best of our knowledge, the current approach of doing this is by 'removing' completed SCCs from the graph once identified by one worker [46]. The random exploration strategy then would take care of work load distribution. However, such a method would not scale for graphs consisting of single large SCC. We demonstrate a method that is able to communicate partially identified SCCs and can therefore scale on more graphs.

The basis of the parallel algorithm is the sequential set-based algorithm, developed in the previous section, which stores strongly connected vertices in supernodes. For the time being, we do not burden ourselves with the exact data structures and focus on solutions that ease explanation. Afterwards, we show how the set operations are implemented.

Communication principles. We assume that each parallel worker p starts a PRDFS search with a local stack R_p and randomly searches the graph independently from v_0. The upcoming algorithm is based on four core principles about vertex communication:

1. If a worker encounters a cycle, it communicates this cycle globally by uniting all vertices on the cycle to one supernode, i.e. $\mathcal{S}$ becomes a shared structure. As a consequence, worker p might unite multiple vertices that remain on the stack $R_{p'}$ of some other worker p', violating Equation 2 for now.
2. Because, the workers can no longer rely on the uniqueness property, the algorithm looses its completeness property: It may report partial SCCs because the collapsing stops early (before the highest connected supernode on the stack is encountered). To remedy this, we iterate over the contents of the supernodes until all of its vertices have been processed.
3. Since other workers constantly grow partial SCCs in the shared $\mathcal{S}$, it is no longer useful to retain a local VISITED set. Equation 1 showed that indeed the LIVE vertices can be deduced from $\mathcal{S}$ and R in the sequential algorithm. We choose to use an implicit definition of LIVE so that a worker p only explores a vertex v if there is no $r \in R_p$, s.t. $v \in \mathcal{S}(r)$, we say v is not *part of an R_p supernode*. Thus vertices connected to a supernode on R_p by some other worker p' can be pruned by p.
4. If a worker p backtracks from a vertex, i.e. it finds that all successors are either DEAD or part of some R_p supernode, it

records said vertex in a global set DONE for all other workers to disregard. Here, a DONE vertex implies that it is fully explored. Once all vertices in an SCC are DONE, the SCC is marked DEAD.

The algorithm. The multi-core SCC algorithm, provided in Algorithm 2, is similar to the sequential set-based one (Algorithm 1) and follows the four discussed principles. Each worker p maintains a local search stack (R_p), while $\mathcal{S}$ is shared among all workers for globally communicating partial SCCs. We also globally share the DONE and DEAD sets. For now we assume that all lines in the algorithm can be executed atomically.

Algorithm 2 The UFSCC algorithm (code for worker p)

```
 1: $\forall v \in V : S(v) := \{v\}$ ........................ [global $S$]
 2: DEAD := DONE := $\emptyset$ ............... [global DEAD and DONE]
 3: $\forall p \in [1 \dots p] : R_p := \emptyset$ .................... [local $R_p$]
 4: UFSCC$_1(v_0) \| \dots \|$UFSCC$_p(v_0)$
 5: procedure UFSCC$_p(v)$
 6:     $R_p$.PUSH($v$)
 7:     while $v' \in S(v) \setminus$ DONE do
 8:         for each $w \in$ RANDOM(POST($v'$)) do
 9:             if $w \in$ DEAD then continue ............... [DEAD]
10:             else if $\nexists w' \in R_p : w \in S(w')$ then ...... [NEW]
11:                 UFSCC$_p(w)$
12:             else while $S(v) \neq S(w)$ do ............... [LIVE]
13:                 $r := R_p$.POP()
14:                 UNITE($S, r, R_p$.TOP())
15:         DONE := DONE $\cup \{v'\}$
16:     if $S(v) \nsubseteq$ DEAD then DEAD := DEAD $\cup\, S(v)$; report SCC $S(v)$
17:     if $v = R_p$.TOP() then $R_p$.POP()
```

First of all, instead of performing only a DFS, the algorithm also iterates over vertices from $\mathcal{S}(v)$, at Line 7-15, until every $v' \in \mathcal{S}(v)$ is marked DONE (principle 2 and 4). Therefore, once this while-loop is finished, we have that $\mathcal{S}(v) \subseteq$ DONE, hence the SCC may be marked DEAD in Line 16. The if-condition at that line succeeds if worker p wins the race to add $\mathcal{S}(v)$ to DEAD, ensuring that only one worker reports the SCC. Thus, when UFSCC(v) terminates, we ensure that $v \in$ DEAD and that v is removed from the local stack R_p (Line 17).

For every $v' \in \mathcal{S}(v) \setminus$ DONE, the successors of v' are considered in a randomized order via the RANDOM() function at Line 8 — according to PRDFS' random search and prune strategy. After the for-loop from Line 8-14, we infer that POST(v') $\subseteq$ (DEAD $\cup$ $\mathcal{S}(v)$) and therefore v' may be marked DONE (principle 4).

Compared to the VISITED set from Algorithm 1, this algorithm uses $\mathcal{S}$ and R_p to derive whether or not a vertex w has been 'visited' before (principle 3). Here, 'visited' implies that there exists a vertex w' on R_p (hence also on the local search path) which is part of the same supernode as w. Section 4 shows how iteration on R_p can be avoided. The recursive procedure for w is called for a successor of v' that is part of the same supernode as v. Assuming that $\forall a, b \in \mathcal{S}(v) : a \leftrightarrow b$ holds, we have that $v \rightarrow v' \rightarrow w$ and w is reachable from v.

Correctness sketch. The soundness defined here implies that reported SCCs are strongly connected, whereas completeness implies that they are *maximal*. We demonstrate soundness (in Theorem 1) by showing that vertices added to $\mathcal{S}(v)$ are always strongly connected with v (or any other vertex in $\mathcal{S}(v)$). Completeness (see Theorem 2) follows from the requirements to mark an SCC DEAD (Lemma 2–4). Complete correctness follows from termination in Theorem 3.

Lemma 1 (Stack). $\forall r \in R_p \colon r \rightarrow^* R_p$.TOP().

Proof. Follows from R_p being a subset of the program stack. □

Theorem 1 (Soundness). *If two vertices are in the same set, they must form a cycle:* $\forall v, w \colon w \in \mathcal{S}(v) \Rightarrow v \rightarrow^* w \rightarrow^* v$.

Proof. Initially the hypothesis holds, since $\forall v \colon \mathcal{S}(v) = \{v\}$. Assuming that the theorem holds for $\mathcal{S}$ before the executing a statement of the algorithm, we show it holds after, considering only the non-trivial case, i.e. Line 14. Before Line 12, we have $\mathcal{S}(v) = \mathcal{S}(R_p.\text{TOP}())$ (either $v = R_p$.TOP() or a recursive procedure has united v with the current R_p.TOP()). The while-loop (Line 12-14) continuously pops the top vertex of R_p and unites it with the new R_p.TOP(). Since $\exists w' \in R_p \colon w \in \mathcal{S}(w')$ (the condition at Line 10 did not hold), the loop ends iff $\mathcal{S}(R.\text{TOP}()) = \mathcal{S}(v) = \mathcal{S}(v') = \mathcal{S}(w) = \mathcal{S}(w')$. Because $v' \rightarrow w$ and $w' \rightarrow^* R$.TOP() (Lemma 1), we have that the cycle $v' \rightarrow^* w \rightarrow^* v'$ is merged, thus preserving the hypothesis in the updated $\mathcal{S}$. □

Lemma 2. *After UFSCC$_p(v)$ terminates,* $v \in$ DEAD.

Proof. This follows directly from Line 16 and the fact that v is never removed from $\mathcal{S}(v)$. □

Lemma 3. *Successors of* DONE *vertices are* DEAD *or in a supernode on R_p:* $\forall v \in$ DONE$\colon$ POST(v) $\subseteq$ (DEAD $\cup$ ($\mathcal{S}(v) \cap R_p$)).

Proof. The only place where DONE is modified is in Line 15. Vertices are never removed from DEAD or $\mathcal{S}(v)$ for any v. In Line 8-14, all successors w of v' are considered separately:

1. $w \in$ DEAD is discarded.
2. w is not in a supernode of R_p and UFSCC is recursively invoked for w. From Lemma 2, we obtain that $w \in$ DEAD upon return of UFSCC$_p(w)$.
3. there is some $w' \in R_p$ s.t. $\mathcal{S}(w) = \mathcal{S}(w')$. Supernodes of vertices on R_p are united until $\mathcal{S}(v) = \mathcal{S}(v') = \mathcal{S}(w)$.

All three cases thus satisfy the conclusion of the hypothesis before v' is added to DONE at Line 15. □

Lemma 4. *Every vertex in a* DEAD *partial SCC set is* DONE*:* $\forall v \in$ DEAD$\colon \forall v' \in \mathcal{S}(v) \colon v' \in$ DONE.

Proof. Follows from the while-loop exit condition at Line 7. □

Theorem 2 (Completeness). *After UFSCC$_p(v)$ finishes, every reachable vertex t (a) is* DEAD *and (b) $\mathcal{S}(t)$ is a maximal SCC.*

Proof. (a. every reachable vertex t is DEAD). Since $v \in$ DEAD (Lemma 2), we have $\forall v' \in \mathcal{S}(v) : v' \in$ DONE (Lemma 4). By Lemma 3, we deduce that every successor of v' is DEAD (as $v' \in \mathcal{S}(v)$ implies $v' \in$ DEAD by Line 16). Therefore, by applying the above reasoning recursively on the successors of v' we obtain that every reachable vertex from v is DEAD.

(b. $\mathcal{S}(t)$ is a maximal SCC). Assume by contradiction that for disjoint DEAD sets $\mathcal{S}(v) \neq \mathcal{S}(w)$ we have $v \rightarrow^* w \rightarrow^* v$. W.l.o.g., we can assume $v \rightarrow w$. Since $\mathcal{S}(w) \neq \mathcal{S}(v)$, we deduce that $w \in$ DEAD when this edge is encountered (Lemma 3). However, since $w \rightarrow^* v$ we must also have $v \in$ DEAD (Theorem 2a), contradicting our assumption. Hence, every DEAD $\mathcal{S}(t)$ is a maximal SCC. □

Theorem 3 (Termination). *Algorithm 2 terminates for finite graphs.*

Proof. The while-loop terminates because vertices can only be added once to $\mathcal{S}(v)$ and due to a strictly increasing DONE in every iteration. The recursion stops because the vertices part of LIVE supernodes in R_p only increase: as per Line 6 and Line 14. □

4. Implementation

For the implementation of Algorithm 2, we require the following:

- Data structures for R_p, $\mathcal{S}$, DEAD and DONE.
- A mechanism to iterate over $v' \in \mathcal{S}(v) \setminus$ DONE (Line 7).
- Means to check if a vertex v is part of $(\mathcal{S}(v) \cap R_p)$ (Line 10).
- An implementation of the UNITE() procedure (Line 14).
- A technique to add vertices to DONE and DEAD (Line 15,16).

We explain how each aspect is implemented. For a detailed version of the complete algorithm, we refer the readers to Appendix A.

R_p can be implemented with a standard stack structure. $\mathcal{S}$ can be implemented with a variation of the wait-free union-find structure [2]. Its implementation employs the atomic *Compare&Swap* (CAS) instruction ($c :=$ CAS(x, a, b) atomically checks $x = a$, and if true updates $x := b$ and $c :=$ TRUE, else just sets $c :=$ FALSE) to update the parent for a vertex without interfering with operations from other workers on the structure (FIND() and UNITE()). We implement this structure in an array, which we index using the (unique) hashed locations of vertices. In a UNITE$(\mathcal{S}, a, b)$ procedure, the new root of $\mathcal{S}(a)$ and $\mathcal{S}(b)$ is determined by selecting either the root of $\mathcal{S}(a)$ or $\mathcal{S}(b)$, whichever has the highest index, i.e.: essentially random in our setting using hashed locations. This mechanism for uniting vertices preserves the quasi-constant time complexity for operations on the union-find structure [23].

The DEAD set is implemented with a status field in the union-find structure. We have that $\mathcal{S}(v)$ is DEAD if the status for the root of v is DEAD, thus marking an SCC DEAD, indeed implementing Line 16 of Algorithm 2 atomically. This marking is achieved in quasi-constant time (a FIND() call with a status update).

Worker set. We show how $\nexists w' \in R_p : \mathcal{S}(w') = \mathcal{S}(w)$ (Line 10) is implemented. In the union-find structure, for each node we keep track of a bitset of size p. This *worker set* contains a bit for each worker and represents which workers are currently exploring a partial SCC. We say that worker p has visited a vertex w if the worker set for the root of $\mathcal{S}(w)$ has the bit for p set to true. If otherwise worker p encounters an unvisited successor vertex w, it sets its worker bit in the root of $\mathcal{S}(w)$ using CAS and recursively calls UFSCC(w) (Line 10). In the UNITE$(\mathcal{S}, a, b)$ procedure, after updating the root, the worker sets for a and b are combined (with a logical OR, implemented with CAS) and stored on the new root to preserve that a worker bit is never removed from a set. The process is repeated if the root is updated while updating the worker set. Since only worker p can set worker bit p (aside from combining worker sets in the UNITE() procedure), only partial SCCs visited by worker p (on R_p) can and will contain the worker bit for p.

As an example to explain the worker set, consider Figure 2. Here, worker B has found the cycle $a \to b \to c \to d \to a$, which are all united: $\mathcal{S}(b) = \{a, b, c, d\}$, and worker R explored $a \to e \to f \to e$ (with $\mathcal{S}(f) = \{e, f\}$). Worker R now considers the edge $f \to c$ and wants to know if it has previously visited a vertex in $\mathcal{S}(c)$ (which would imply a cycle). Because the worker set is managed at the root of $\mathcal{S}(c)$ (which is node b), worker R simply has to check there if its worker bit is set. Since this is the case, the sets $\mathcal{S}(f)$ and $\mathcal{S}(e)$ can be united.

Cyclic linked list. The most challenging aspect of the implementation is to keep track of the DONE vertices and iterate over $\mathcal{S}(v) \setminus$ DONE. We do this by making the union-find iterable by adding a separate cyclic list implementation. That is, we extend the structure with a `next` pointer and a list status flag that indicates if a vertex is either BUSY or DONE (initially, the flag is set to BUSY).

The idea is that the cyclic list contains all BUSY vertices. Vertices marked DONE are skipped for iteration, yet not physically removed from the cycle. Rather, they will be lazily removed once reached from their predecessor on the cycle (this avoids maintaining doubly linked lists). All DONE vertices maintain a sequence of `next` pointers towards the cyclic list, to ensure that the cycle of BUSY vertices is reachable from every vertex $v' \in \mathcal{S}(v)$. If all vertices from $\mathcal{S}(v)$ are marked DONE, the cycle becomes empty (we end up with a DONE vertex directing to itself) and $\mathcal{S}(v)$ can be marked DEAD.

To illustrate the cyclic list structure, consider the example from Figure 3. In the partial SCC $\{a, b, c, d\}$, vertices a and c are marked DONE (gray), and vertices b and d are BUSY (white). We further assume that $\{e\}$ is a DEAD SCC. The vertices a and c are marked DONE since their successors are in (DEAD $\cup\ \mathcal{S}(a)$) (see also Lemma 3). Note that b and d (and f) may not yet be marked DONE because their successors have not been fully explored yet. The right side depicts a possible representation for the cyclic list structure (where the arrows are `next` pointers).

In the algorithm, Line 7 can be implemented by recursively traversing v.`next` to eventually encounter either a vertex $v' \in \mathcal{S}(v) \setminus$ DONE or otherwise it detects that there is no such vertex and the while-loop ends. Line 15 is implemented by setting the status for v' to DONE in the union-find structure, which is lazily removed from the cyclic list.

The UNITE$(\mathcal{S}, a, b)$ procedure is extended to combine two cyclic lists into one (without losing any vertices). We show this procedure using the example from Figure 4. Here UNITE$(\mathcal{S}, a, f)$ is called (b becomes the new root of the partial SCC). The `next` pointers (solid arrows) for a and f are swapped with each other (note that this can not be done atomically), which creates one list containing all vertices. It is important that both a and f are BUSY vertices (as otherwise part of the cycle may be lost), which is ensured by first searching and using light-weight, fine-grained locks (one per node) on BUSY vertices a' and f'. The locks also protect the non-atomic pointer swap.

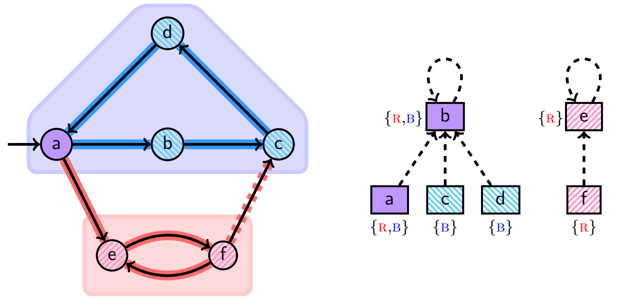

Figure 2: Illustration of the worker set in practice. The left side depicts the graph, the union-find structure with the worker set is shown right for workers R and B.

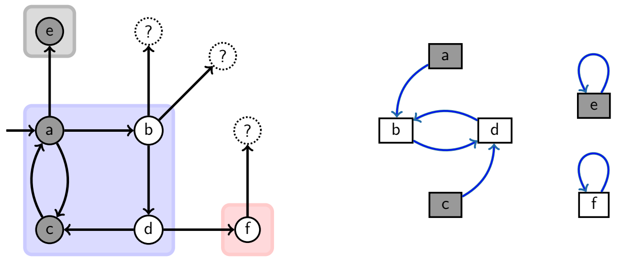

Figure 3: Illustration of the cyclic list in practice. The left side depicts the graph, the right side depicts the corresponding cyclic list structure. White nodes are BUSY and gray nodes are DONE.

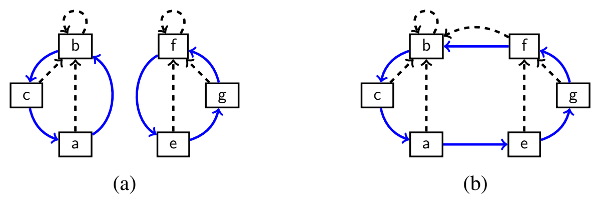

Figure 4: Cyclic list before (a) and after (b) a UNITE$(\mathcal{S}, a, f)$ call. Dashed edges depict parent pointers and solid edges `next` pointers.

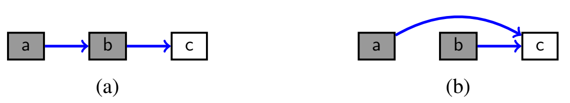

Figure 5: Cyclic list before (a) and after (b) a list 'removal'. Solid edges depict `next` pointers and gray nodes are DONE.

Removing a vertex from the cyclic list is a two-step process. First, the status for the vertex is set to DONE. Then, during a search for BUSY vertices, if two subsequent DONE vertices are encountered, the `next` pointer for the first one is updated (Figure 5). Note that the 'removed' vertex remains pointing to the cyclic list.

We use a lock on the union-find structure (only on the root for which the parent gets updated) and a lock on list nodes encoded as status flags. The structure for union-find nodes is shown in Figure 6. We chose to use fine-grained locking instead of wait-free solution, for two reasons: we do not require the stronger guarantees of the latter, and the pointer indirection needed for a wait-free design would decrease performance compared to the current node layout with its memory locality (as we learned from our previous work).

pointer	`parent`	. . . [union-find parent]
pointer	`next`	. . . . [list next pointer]
bit$[p]$	`worker_set`	. . . . . . [worker set]
bit$[2]$	`uf_status`	$[\in \{$LIVE, LOCK, DEAD$\}]$
bit$[2]$	`list_status`	$[\in \{$BUSY, LOCK, DONE$\}]$

Figure 6: The node record of the iterable union-find data structure.

Runtime/work tradeoff. Our treatment of the complexity here is not as formal as in e.g. [55], and relies on practical assumptions. We argue that, while the theoretical complexity study of parallel algorithms offers invaluable insights, it is also remote from the practical solution that we focus on. E.g. we can not within the foreseeable future obtain n parallel processors for n-sized graphs.

We assume that overhead from synchronization and successor randomization is bound by a constant factor. Because the worker sets need to be updated, UNITE operations take $\mathcal{O}(p)$ time and work (instructions). Assuming again that p is small enough for the worker set to be stored in one or few words (e.g. 64), UNITE becomes quasi constant (now the FIND operation it calls dominates). Finding a BUSY vertex in the cyclic list is bounded by $\mathcal{O}(n)$ (the maximal sequence of DONE vertices before reaching a BUSY vertex). However, we suspect that due to the lazy removal of DONE nodes, similar to path-compression [54], supernode iteration is amortized quasi constant under the condition that $p \ll n$. Though if this not the case, we could always use a more complicated doubly linked list instead [58, p. 63]. So amortized, all supernode operations are bounded by $\mathcal{O}(\alpha)$ time and work. I.e. the inverse Ackermann function representing quasi constant time.

Since every worker may visit every edge and vertex, performing a constant number of supernode operations on the vertex, the worst-case work complexity for the algorithm is bounded by $\mathcal{O}(\alpha \cdot p \cdot (n+m))$. As typically, the inverse Ackermann function is bounded by a constant, we obtain $\mathcal{O}(p \cdot (n+m))$. The runtime, however, remains bounded by $\mathcal{O}(m+n)$ (ignoring the quasi factor), as useless work is done in parallel. The best-case runtimes from Table 1, $\mathcal{O}(\frac{n+m}{p})$, follow when assuming that approximately each vertex is processed by only one worker.[1]

Memory usage. The memory usage of Algorithm 2, using the implementation discussed in the current section, comprises two aspects. First, each worker has a local stack R_p, which contains at most n items (e.g. vertices on a path or within the same SCC when drawn from the iterable union-find structure). Second, the global union-find contains: a parent pointer, a worker set, a `next` pointer, and status fields for both union-find and its internal list (Figure 6). Assume that vertices and pointers in the structure can be stored in constant amount of space, their storage requires 3 units. Also assuming again that $p \ll n$, the worker set takes another unit, e.g. one 64-bit machine word for $p = 64$. Then, the algorithm uses at most $(p \cdot n) + 4n$ units (stacks plus shared data structures).

The worst-case space complexity becomes $\mathcal{O}(p \cdot n)$ (also when p is not constant, e.g. approaching n, and the worker set cannot be stored in a constant anymore). The local replication on the stacks is the main cause of the worst-case memory usage of PRDFS algorithms as shown in Table 1 (other PRDFS algorithms, such as [18, 19, 32, 46], do not require a worker set). In practice however, the replication is proportional to the contention occurring on vertices which is arguably low when the algorithm processes large graphs and $p \ll n$. Hence the actual memory usage will be closer to the best-case complexity that occurs when stacks do not overlap, i.e.: $\mathcal{O}(n)$ (unless, again, p is not constant and the worker set dominates memory usage). Section 6.1 confirms this.

5. Related Work

Parallel on-the-fly algorithms. Renault et al. [46] were the first to present a PRDFS algorithm that spawns multiple instances of Tarjan's algorithm and communicates completely explored SCCs via a shared union-find structure. We improve on this work by communicating partially found SCCs. Lowe [36] runs multiple *synchronized* instances of Tarjan's algorithm, without overlapping stacks. Instead a worker is suspended if it meets another's stack and stacks are merged if necessary. While in the worst case this might lead to a quadratic time complexity, Lowe's experiments show decent speedups on model checking inputs, though not for large SCCs.

Fix-point based algorithms. Section 1 discussed several fix-point based solutions. A notable improvement to FB [20] is the trimming procedure [39], which removes trivial SCC without fix-point recursion. OBF [4] further improves this by subdividing the graph in a number of independent sub-graphs. The Coloring algorithm [42] propagates priorized colors dividing the graph in disconnected subgraphs whose backwards slices identify SCCs. Barnat et al. [6] provide CUDA implementations for FB, OBF, and Coloring. Coppersmith et al. [13] present and prove an $\mathcal{O}(m \cdot \log n)$ serial runtime version of FB, Schudy [48] further extended this to obtain an expected runtime bounded by $\mathcal{O}(\log^2 n)$ reachability queries.

Hong et al. [25] improved FB for practical instances, by also trimming SCCs of size 2. After the first SCC is found (which is assumed to be large in size), the remaining components are detected

[1] We acknowledge that for fair comparison with Table 1, the impact of the worker set (a factor p) cannot be neglected, e.g. for large p approaching n, yielding $\mathcal{O}(p \cdot \max(\alpha, p) \cdot (n+m))$ work and $\mathcal{O}(\max(\alpha, p) \cdot (n+m))$ runtime.

(with Coloring) and decomposed with FB. Slota et al. [49] propose orthogonal optimization heuristics and combine them with Hong's work. Both Hong's and Slota's algorithm operate in quadratic time.

Parallel DFS. Parallel DFS is remotely related. It has long been researched intensively, though there are two types of works [21]: one discussing parallelizing DFS-like searches (often using terms like 'backtracking', load balancing and stack splitting), and theoretical parallelization of lexicographical (strict) DFS, e.g. [1, 55]. Recent work from Träff [56] proposes a strategy to processes incoming edges, allowing them to be handled in parallel. The resulting complexity is $\mathcal{O}(\frac{m}{p} + n)$, providing speedup for dense graphs. Research into the characterization of search orders provides other new venues to analyze these algorithms [7, 15]

Union-find. The investigation of Van Leeuwen and Tarjan settled the question which of many union-find implementations were superior [54]. Goel er al. [23] however later showed that a simpler randomized implementation can also be superior. A wait-free concurrent union-find structure was introduced in [2]. Other versions that support deletion [29]. To the best of our knowledge, we are the first to combine both iteration and removal in the union-find structure. This allows the disjoint sets to be processed as a cyclic queue, which enables concurrent iteration, removal and merging.

6. Experiments

The current section presents an experimental evaluation of UFSCC. Results are evaluated compared to Tarjan's sequential algorithm, another on-the-fly PRDFS algorithm using model checking inputs, synthetic graphs and an offline, fix-point algorithm.

Experimental setup. All experiments were performed on a machine with 4 AMD Opteron$^{\text{TM}}$ 6376 processors, each with 16 cores, forming a total of 64 cores. There is a total of 512GB memory available. We performed every experiment at least 50 times and calculated their means and 95% confidence intervals.

We implemented UFSCC in the LTSMIN model checker [28],[2] which we use for decomposing SCCs on implicitly given graphs. LTSMIN's POST() implementation applies measures to permute the order of a vertex's successors so that each worker visits the successors in a different order. We compare against a sequential version of Tarjan's algorithm and the PRDFS algorithm from Renault et al. [46] (see Section 5), which we refer to as *Renault*. Both are also implemented in LTSMIN to enable a fair comparison. Unfortunately, we were unable to implement Lowe's algorithm [36], nor successful to use its implementation for on-the-fly exploration.

For a direct comparison to offline algorithms, we also implemented UFSCC in the SCC environment provided by Hong et al. [25]. This implementation is compared against benchmarks of Tarjan and Hong's concurrent algorithm (both provided by Hong et al.). While Slota et al. [49] improve upon Hong's algorithm, we did not have access to an implementation. However, considering the fact that both are quadratic in the worst case, we can expect similar performance pitfalls as Hong's algorithm for some inputs.

We chose this more tedious approach because the results from on-the-fly experiments are hard to compare directly against offline implementations. Offline implementations benefit from having an explicit graph representation and can directly index based on the vertex numbering, whereas on-the-fly implementations both generate successor vertices, involving computation, and need to hash the vertices. This results in a factor of ± 25 vertices processed per second for the same graphs using the sequential version of the algorithms: Too much for a meaningful comparison of speedups.

[2] All our implementations, benchmarks, and results are available via https://github.com/utwente-fmt/ppopp16.

Table 2: Graph characteristics for model checking, synthetic and explicit graphs. Here, M denotes millions and columns respectively denote the number of vertices, transitions and SCCs, the maximal SCC size and an approximation of the diameter.

graph	# vertices	# trans	# sccs	max scc	diameter
leader-filters.7	26.3M	91.7M	26.3M	1	71
bakery.6	11.8M	40.4M	2.6M	8.6M	176
cambridge.6	3.4M	9.5M	8,413	3.3M	418
lup.3	1.9M	3.6M	1	1.9M	134
resistance.1	13.8M	32.1M	3	13.8M	60,391
sorter.3	1.3M	2.7M	1.2M	278	300
L1751L1751T1	9.2M	24.5M	3	3.1M	3501
L351L351T4	3.8M	11.3M	31	123,201	704
L5L5T16	3.3M	9.8M	131,071	25	24
Li10Lo200	4.0M	15.2M	100	40,000	416
Li50Lo40	4.0M	15.8M	2,500	1,600	176
Li200Lo10	4.0M	16.0M	40,000	100	416
livej	4.8M	69.0M	971,232	3.8M	19
patent	3.8M	16.5M	3.8M	1	24
pokec	1.6M	30.6M	325,893	1.3M	14
random	10.0M	100.0M	877	10.0M	11
rmat	10.0M	100.0M	2.2M	7.8M	9
ssca2	1.0M	157.9M	31	1.0M	133
wiki-links	5.7M	130.2M	2.0M	3.7M	284

Table 2 summarizes graph sizes from respectively: a selection of graphs from an established benchmark set for the model checker, synthetic graphs generated in the model checker, and explicitly stated graphs consisting of real-world and generated graphs. We also exported both model checking and synthetic graphs to Hong's framework, to use as inputs for the offline algorithms.

We validated the implementation by means of proving the algorithm and thoroughly examining that all invariants are preserved in each implemented sub-procedure. Moreover, obtained information from a graph search and the encountered SCCs were reported and compared with the results from Tarjan's algorithm.

Experiments on model checking graphs. We use model checking problems from the BEEM database [43], which consists of a set of benchmarks that closely resemble real problems in verification. From this suite, we select all inputs (62 in total) large enough for parallelization (with search spaces of at least 10^6 vertices), and small enough to complete within a timeout of 100 seconds using our sequential Tarjan implementation. Though our implementation in reality generates the search space during exploration (on-the-fly), we refer to it simply as 'the graph'.

Figure 7 (left) compares the runtimes for UFSCC on 64 workers against Tarjan's sequential algorithm in a speedup graph. The x-axis represents the total time used for Tarjan's algorithm and the

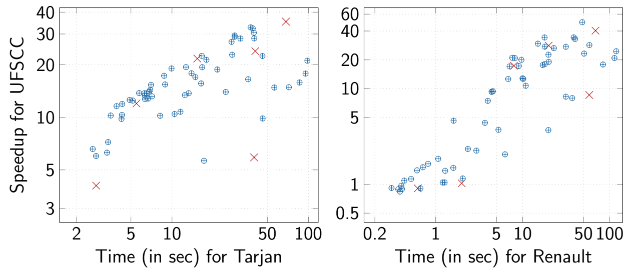

Figure 7: Comparison of time and speedup for 62 model checking graphs for concurrent UFSCC against Tarjan (left) and concurrent Renault (right), where UFSCC and Renault use 64 workers. The red crosses represent the selected BEEM graphs in Figure 8.

Table 3: Comparison of sequential on-the-fly Tarjan with concurrent UFSCC and Renault on 64 cores for a set of model checking inputs and synthetic graphs.

graph	execution time (s) Tarjan	Renault	UFSCC	UFSCC speedup vs Tarjan	Renault
leader_filters.7	68.602	1.995	1.933	35.494	1.032
bakery.6	40.747	68.526	1.464	27.825	46.795
cambridge.6	15.162	19.965	0.716	21.176	27.884
lup.3	5.645	7.941	0.473	11.944	16.803
resistance.1	39.425	58.247	6.258	6.300	9.307
sorter.3	2.808	0.618	0.687	4.088	0.900
L1751L1751T1	28.994	36.088	2.526	11.478	14.287
L351L351T4	12.189	4.793	0.928	13.138	5.166
L5L5T16	8.095	1.255	0.782	10.357	1.606
Li10Lo200	15.144	5.127	1.399	10.828	3.666
Li50Lo40	13.473	3.973	1.892	7.122	2.100
Li200Lo10	12.906	4.438	2.874	4.491	1.544

y-axis represents the observed speedup for the UFSCC algorithm on the same graph instances. We observe that in most cases UFSCC performs at least 10× faster than Tarjan. The performance improvements for all model checking graphs range from 4.1 (`sorter.3`) to 35.5× faster (`leader_filters.7`) and the geometric mean performance increase for UFSCC over Tarjan is 14.84. Compared to the sequential runtime of UFSCC, its multi-core version exhibits a geometric mean speedup of 19.53. We also witness a strong trend towards increasing speedups for larger graphs (dots to the right represent longer runtimes in Tarjan, which is proportional to the size of the search space).

Figure 7 (right) compares UFSCC with Renault on model checking graphs, both using 64 workers. For a number of graphs Renault performs slightly better (up to 1.2× faster than UFSCC). These graphs all contain many small SCCs and thus communicating completed SCCs is effective. Since UFSCC applies a similar communication procedure while also maintaining the cyclic list structure, a slight performance decrease is to be expected. For most other graphs however we see that UFSCC significantly outperforms Renault. These graphs indeed contain large SCCs. In some cases Renault even performs worse than Tarjan's sequential algorithm, which we attribute to contention on the union-find structure (where multiple workers operate on a large SCC at the same time, thus all requiring the root of that SCC). On average, we observe again that UFSCC scales better for larger graphs. The geometric mean performance increase for UFSCC over Renault (considering every examined model checking graph) is 6.42.

We select 6 graphs to examine more closely. We choose the graphs to range from the best and worst results when UFSCC is compared to Tarjan and Renault. We indicated the selected graphs with red squares in Figure 7. Information about these graphs is provided in Table 2 (the first 6 graphs).

The scalability for UFSCC and Renault compared to Tarjan is depicted in Figure 8 and Table 3 quantifies the performance results for 64 workers. In Figure 8, while we generally see a fairly linear performance increase, we notice two peculiarities when the number of workers is increased for UFSCC. First, in some graphs the performance increase levels out (or even drops a bit) for a certain number of workers and thereafter continues in increasing performance (`bakery.6`, `cambridge.6`, `lup.3`). These results occur from high variances in the performance results, e.g. where UFSCC executes in 1 second for 75% of the cases and in 4 seconds for the other 25%. Informal experiments suggest that these results origin from the combination of an unfortunate successor permutation and the specific graph layout.

Another peculiarity is that for most graphs the performance does not increase as much for 32 workers or more (and even drops for

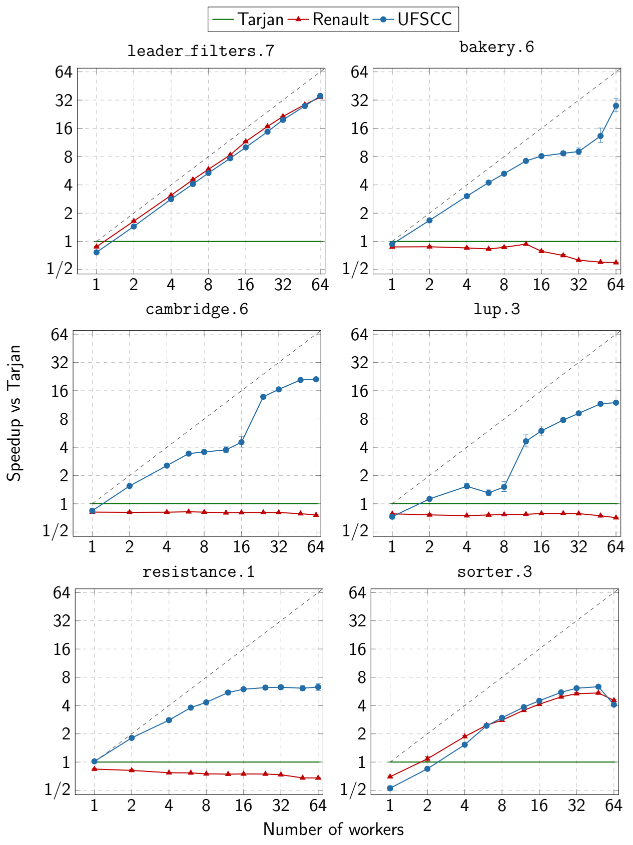

Figure 8: Scalability for the on-the-fly UFSCC and Renault implementations relative to Tarjan's algorithm on a set of model checking inputs.

`sorter.3`). We determined that this effect likely results from a high contention on the internal union-find structure; where a large number of workers try to access the same memory locations (the same reason for why Renault with 64 workers performs worse compared to Tarjan for graphs with large SCCs). In a previous design of UFSCC, which used a more drastic locking scheme, this effect was observed to be more prevalent [8].

For each model checking graph, we compared the total number of visited vertices for UFSCC using 64 workers with the number of unique vertices. We observe that 0.5% (`leader_filters.7`) up to 128% (`resistance.1`) re-explorations occurred in the experiments. With re-exploration we mean the number of times that vertices are explored by more than one worker, which ideally should be minimized. As a geometric mean (over the averaged relative re-exploration for each graph), the total number of explored vertices is 21.1% more than the number of unique vertices, implying that on average each worker only visits $\frac{121.1}{64} \approx 1.9\%$ of the total number of vertices.

Experiments on synthetic graphs. We experimented on synthetically generated graphs (equivalent to the ones used by Barnat et al. [4]) to find out how particular aspects of a graph influence UFSCC's scalability. The first type of graph, called LxLxTy, is the Cartesian product for two cycles of x vertices with a binary tree of depth y (generated by taking the parallel composition of processes with said control flow graphs: $Loop(x)\|Loop(x)\|Tree(y)$). This graph has $2^{y+1} - 1$ components of size x^2.

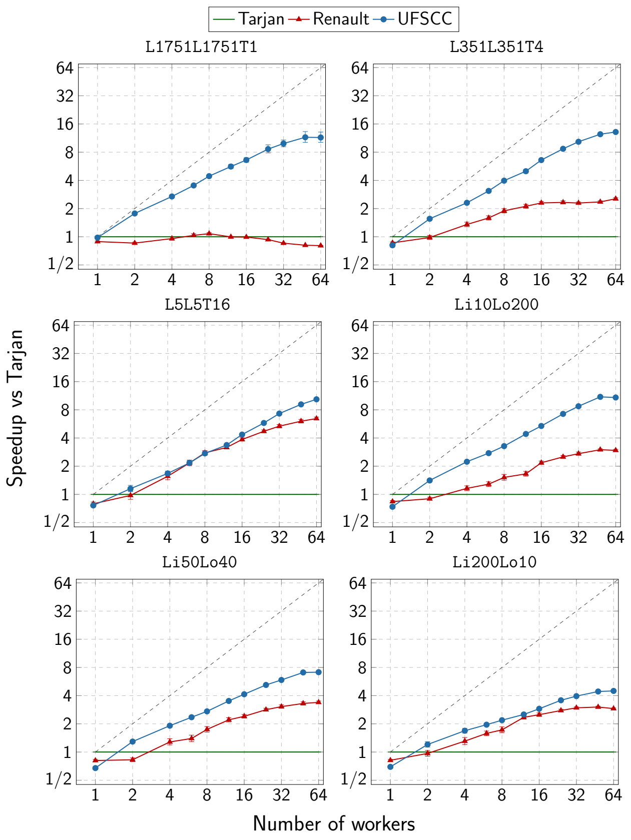

Figure 9: Scalability for selected synthetic graphs of UFSCC and Renault relative to Tarjan's algorithm.

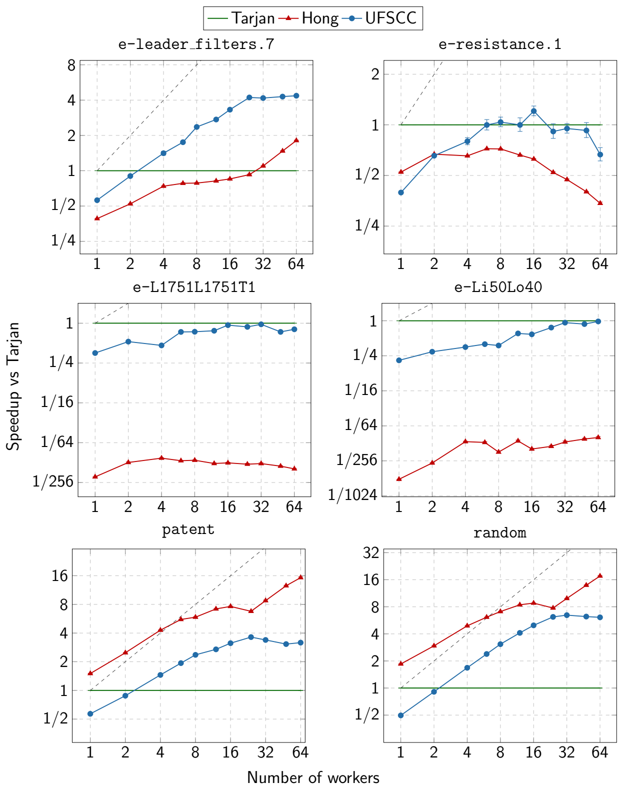

Figure 10: Scalability for the offline UFSCC and Hong implementations relative to Tarjan's algorithm on a set of explicit graphs.

The second type is called LixLoy and is a parallel composition of two sequences of x vertices with two cycles of y vertices ($Line(x)\|Line(x)\|Loop(y)\|Loop(y)$). This graph has x^2 components of size y^2. Table 2 summarizes these graphs. We provide a comparison of Tarjan, Renault and UFSCC in Table 3 and show how UFSCC and Renault scale compares to Tarjan in Figure 9.

In Figure 9, we notice that scalability of UFSCC is more consistent than the results in Figure 8. This likely follows from the even distribution of successors and SCCs in these graphs. While UFSCC's speedup compared to both Tarjan and Renault are not as impressive as we see in the model checking experiments, it still outperforms Tarjan and Renault significantly (though less impressively for L5L5T16 and Li200Lo10, which both consist of many small SCCs). Additional experiments on inputs with increased outdegree (generated by putting more processes in parallel), showed that speedups indeed improved.

Experiments on explicit graphs. We experimented with an offline implementation of UFSCC and compare its results with (an offline implementation of) Tarjan's and Hong's algorithm. The explicit graphs are stored in a CSR adjacency matrix format (c.f. [25]). We benchmarked several real-world and generated graph instances. Information about these examined graphs can be found in Table 2 (the bottom seven graphs). The livej, patent and pokec graphs were obtained from the SNAP [35] database and represent the LiveJournal social network [3], the citation among US patents [34], and the Pokec social network [51]. The graph wiki-links[3] represents Wikipedia's page-to-page links. The random, rmat and ssca2 graphs represent random graphs with real-world characteristics and were generated from the GTGraph [38] suite using default parameters. We also constructed explicit graphs from the selected on-the-fly model checking and synthetic experiments (by storing all edges during an on-the-fly search in CSR format), these graphs are prefixed with "e-" in Table 4. We used the GreenMarl [24] framework to convert the graphs to a binary format suitable for the implementation.

The results of these experiments can be found in Table 4. We provide the speedup graphs for six of these graphs in Figure 10. Again, we stress that the results from Table 3 and Table 4 cannot be compared, since the on-the-fly experiments perform more work due to the dynamic generation of successors. We notice some interesting results from the offline experiments, which we summarize as follows:

- On model checking graphs, UFSCC generally performs best (with two exceptions) and Hong performs slower than Tarjan for 4 of the 6 graphs.
- On synthetic graphs, UFSCC struggles to gain performance over Tarjan but remains to perform similarly, while Hong significantly performs worse (and always crashed on L5L5T16).
- On real-world graphs, UFSCC shows increased performance compared to Tarjan, but Hong clearly outperforms UFSCC.

The synthetic graphs seem to exhibit instances of Hong's quadratic worst-case performance. We also examined that the performance for UFSCC with 1 worker is significantly worse compared to that of Hong (up to a factor of 4), indicating that the overhead for the iterable union-find structure affects the performance on offline graphs.

[3] Obtained from http://haselgrove.id.au/wikipedia.htm.

Table 4: Comparison of sequential offline Tarjan with concurrent UFSCC and Hong on 64 cores for a set of model checking inputs, synthetic graphs and real-world graphs (all explicitly represented in CSR adjacency matrix format).

	execution time (s)			**UFSCC speedup vs**	
graph	**Tarjan**	**Hong**	**UFSCC**	**Tarjan**	**Hong**
`e-leader_filters.7`	2.771	1.532	0.635	4.361	2.411
`e-lup.3`	0.418	0.100	0.167	2.505	0.597
`e-bakery.6`	1.733	5.110	0.760	2.280	6.725
`e-cambridge.6`	0.277	0.330	0.188	1.475	1.752
`e-sorter.3`	0.091	3.710	0.068	1.342	54.419
`e-resistance.1`	2.040	5.974	3.065	0.666	1.949
`e-L1751L1751T1`	0.878	139.636	1.086	0.809	128.631
`e-L351L351T4`	0.247	30.216	0.290	0.854	104.239
`e-L5L5T16`	0.201	-	0.104	1.936	-
`e-Li10Lo200`	0.251	30.627	0.362	0.692	84.548
`e-Li50Lo40`	0.221	22.452	0.227	0.975	98.925
`e-Li200Lo10`	0.223	39.934	0.332	0.671	120.170
`livej`	2.963	0.278	0.809	3.664	0.344
`patent`	0.914	0.060	0.288	3.174	0.208
`pokec`	1.372	0.113	0.338	4.061	0.333
`random`	10.081	0.573	1.650	6.110	0.347
`rmat`	8.327	0.498	1.577	5.279	0.315
`ssca2`	1.657	0.208	0.356	4.651	0.583
`wiki-links`	4.237	0.352	1.017	4.168	0.347

6.1 Memory usage

The memory usage is dominated by the shared union-find structure with linked list and worker set. Its implementation comprises an array of nodes. Each node contains two 64-bit pointers, a bit set of width 64 functioning as the worker set and several flag fields (see Figure 6). For alignment, 64 bits are reserved for the flags, resulting in a total node size of $4 \times 8 = 32$ bytes. Storing a vertex requires an 8-byte entry in a shared hash table and one union-find node. Therefore, our implementation requires at least 40 bytes per vertex.

The only additional memory usage comes from the local stacks, all of which could in the worst case contain all vertices, as explained in Section 4. One stack entry takes 8 bytes. To investigate this worst-case behavior, we plotted the total memory usage

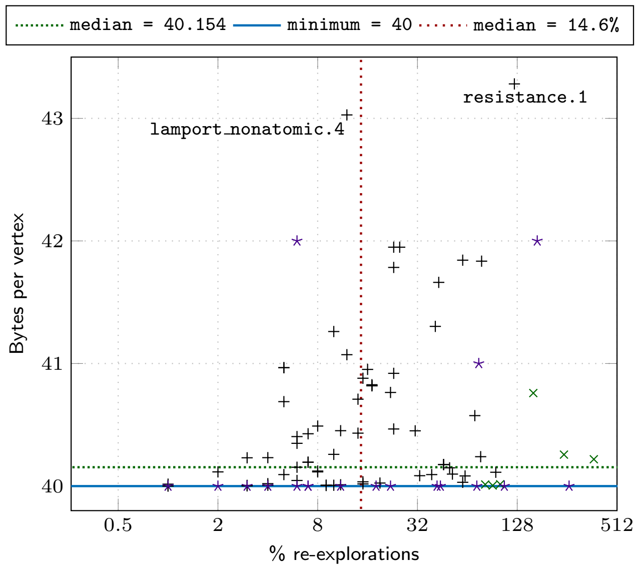

Figure 11: Memory usage in bytes per vertex of UFSCC on 64 cores for all graphs: (on-the-fly) model checking inputs (+) / synthetic graphs (×) and the different types of explicit graphs (⋆).

per vertex of UFSCC on 64 cores against the percentage of re-explorations for all of the above models in Figure 11. The figure shows that is no positive correlation between memory usage and re-explorations, confirming our expectation that stack sizes and re-explorations are independent quantities. The memory usage for most inputs is close to the minimum of 40 bytes, with a median of 40.154 bytes. This demonstrates that the worst-case of $\mathcal{O}(p \times n)$ is far from typical, i.e.: actual memory usage is linear in the number of vertices. The two graphs which resulted in the highest memory usage are both tagged in the figure.

Due the presence of a fixed-sized hash table, we were forced to overestimate the stack sizes using local counters measuring their maximum sizes (one shared total counter would compromise scalability [31, Sec. 1.6.1]). Therefore, in reality the memory use could even be much lower .

7. Conclusions

We presented a new quasi-linear multi-core on-the-fly SCC algorithm. The algorithm communicates partial SCCs using a new *iterable* union-find structure. Its internal cyclic linked list allows for concurrent iteration and removal of nodes – enabling multiple workers to aid each other in decomposing an SCC. Experiments demonstrated a significant speedup (for 64 workers) over the current best known approaches, on a variety of graph instances. Unlike previous work, we retain scalability on graphs containing a large SCC. In future work, we will attempt to reduce the synchronization overhead, and focus on applications of parallel SCC decomposition. We also want to investigate the iterable union-find data structure separately and derive its exact amortized complexity.

Acknowledgments

We are grateful for the helpful comments from Robert Tarjan, Zanbo Zhang and Lawrence Rauchwerger. This work is supported by the Austrian National Research Network S11403-N23 (RiSE) of the Austrian Science Fund (FWF) and by the Vienna Science and Technology Fund (WWTF) through grant VRG11-005. This work is also supported by the 3TU.BSR project.

References

[1] Aggarwal, A., Anderson, R. J., & Kao, M. Y. (1989). Parallel depth-first search in general directed graphs. In Proceedings of the ACM symposium on Theory of computing (pp. 297-308). ACM.

[2] Anderson, R. J., & Woll, H. (1991). Wait-free Parallel Algorithms for the Union-find Problem. In Proceedings of the Twenty-third Annual ACM Symposium on Theory of Computing (pp. 370-380). ACM.

[3] Backstrom, L., Huttenlocher, D., Kleinberg, J., & Lan, X. (2006). Group Formation in Large Social Networks: Membership, Growth, and Evolution. KDD.

[4] Barnat, J., Chaloupka, J., & van de Pol, J. (2009). Distributed algorithms for SCC decomposition. Journal of Logic and Computation.

[5] Barnat, J., Brim, L., & Ročkai, P. (2010). Parallel partial order reduction with topological sort proviso. In Software Engineering and Formal Methods (SEFM), 2010 8th IEEE International Conference on (pp. 222-231). IEEE.

[6] Barnat, J., Bauch, P., Brim, L., & Češka, M. (2011). Computing strongly connected components in parallel on CUDA. in Proc. 25th Int. Parallel and Distributed Processing Symp. (IPDPS). IEEE, pp. 544-555.

[7] Berry, A., Krueger, R., & Simonet, G. (2005). Ultimate generalizations of LexBFS and LEX M. In Graph-theoretic concepts in computer science (pp. 199-213). Springer.

[8] Bloemen, V. (2015). On-The-Fly parallel decomposition of strongly connected components. Master's thesis, University of Twente.

[9] Bondy, J. A., & Murty, U. S. R. (1976). Graph theory with applications (Vol. 290). London: Macmillan.

[10] Brim, L., Černá, I., Krčál, P., & Pelánek, R. (2001). Distributed LTL model checking based on negative cycle detection. In FST TCS 2001: Foundations of Software Technology and Theoretical Computer Science (pp. 96-107). Springer Berlin Heidelberg.

[11] Černá, I., & Pelánek, R. (2003). Distributed explicit fair cycle detection (set based approach). In Model Checking Software (pp. 49-73). Springer Berlin Heidelberg.

[12] Clarke, E. M., Grumberg, O., & Peled, D. (1999). Model checking. MIT press.

[13] Coppersmith, D., Fleischer, L., Hendrickson, B., & Pınar, A. (2003). A divide-and-conquer algorithm for identifying strongly connected components. Technical Report RC23744, IBM Research, 2005.

[14] Cormen, T. H. (2009). Introduction to algorithms. MIT press.

[15] Corneil, D. G., & Krueger, R. M. (2008). A unified view of graph searching. SIAM Journal on Discrete Mathematics, 22(4), 1259-1276.

[16] Courcoubetis, C., Vardi, M., Wolper, P., & Yannakakis, M. (1993). Memory-efficient algorithms for the verification of temporal properties. In Computer-aided Verification (pp. 129-142). Springer US.

[17] Dijkstra, E. W. (1976). A discipline of programming (Vol. 1). Englewood Cliffs: prentice-hall.

[18] Evangelista, S., Laarman, A., Petrucci, L., & van de Pol, J. (2012). Improved multi-core nested depth-first search. In Automated Technology for Verification and Analysis (pp. 269-283). Springer Berlin Heidelberg.

[19] Evangelista, S., Petrucci, L., & Youcef, S. (2011). Parallel nested depth-first searches for LTL model checking. In Automated Technology for Verification and Analysis (pp. 381-396). Springer Berlin Heidelberg.

[20] Fleischer, L. K., Hendrickson, B., & Pınar, A. (2000). On identifying strongly connected components in parallel. In Parallel and Distributed Processing (pp. 505-511). Springer Berlin Heidelberg.

[21] Freeman, J. (1991). Parallel algorithms for depth-first search. Technical Report MS-CIS-91-71, University of Pennsylvania

[22] Gabow, H. N. (2000), Path-based depth-first search for strong and biconnected components. Information Proc. Letters, 74(3-4): 107-114.

[23] Goel, A., Khanna, S., Larkin, D. H., & Tarjan, R. E. (2014). Disjoint set union with randomized linking. In Proceedings of the Twenty-Fifth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '14). SIAM 1005-1017.

[24] Hong, S., Chafi, H., Sedlar, E., & Olukotun, K., (2012). Green-Marl: A DSL for easy and efficient graph analysis, ASPLOS 2012.

[25] Hong, S., Rodia, N. C., & Olukotun, K. (2013). On fast parallel detection of strongly connected components (SCC) in small-world graphs. In High Performance Computing, Networking, Storage and Analysis (SC), 2013 International Conference for (pp. 1-11). IEEE.

[26] Hopcroft, J., & Tarjan, R. E.(1974). Efficient planarity testing. Journal of the ACM (JACM), 21(4), 549-568.

[27] Kahn, A. B. (1962). Topological sorting of large networks. Communications of the ACM, 5(11), 558-562.

[28] Kant, G., Laarman, A., Meijer, J., van de Pol, J., Blom, S., & van Dijk, T. (2015). LTSmin: High-Performance Language-Independent Model Checking. In TACAS.

[29] Kaplan, H., Shafrir, N., & Tarjan, R. E. (2002). Union-find with deletions. In Proceedings of the thirteenth annual ACM-SIAM symposium on Discrete algorithms (pp. 19-28).

[30] Kaveh, A. (2014). Computational structural analysis and finite element methods. Wien: Springer.

[31] Laarman, A. W. (2014). Scalable multi-core model checking. Ph.D. thesis, University of Twente.

[32] Laarman, A., & Faragó, D. (2013). Improved on-the-fly livelock detection. In NASA Formal Methods (pp. 32-47). Springer.

[33] Laarman, A., & Wijs, A. (2014). Partial-order reduction for multi-core LTL model checking. In Hardware and Software: Verification and Testing (pp. 267-283). Springer.

[34] Leskovec, J., Kleinberg, J., & Faloutsos, C. (2005). Graphs over Time: Densification Laws, Shrinking Diameters and Possible Explanations. ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD).

[35] Leskovec, J. SNAP: Stanford network analysis project. http://snap.stanford.edu/index.html, last accessed 9 Sep 2015.

[36] Lowe, G. (2015). Concurrent depth-first search algorithms based on Tarjan's Algorithm. International Journal on Software Tools for Technology Transfer, 1-19.

[37] Liu, Y., Sun, J., & Dong, J. S. (2009). Scalable multi-core model checking fairness enhanced systems. In Formal Methods and Software Engineering (pp. 426-445). Springer Berlin Heidelberg.

[38] Madduri, K., & Bader, D. A. GTgraph: A suite of synthetic graph generators. http://www.cse.psu.edu/ kxm85/software/GTgraph/, last accessed 9 Sep 2015.

[39] McLendon III, W., Hendrickson, B., Plimpton, S., & Rauchwerger, L. (2005). Finding strongly connected components in distributed graphs. Journal of Parallel and Distributed Computing, 65(8):901-910.

[40] Munro, I. (1971). Efficient determination of the transitive closure of a directed graph. Information Processing Letters, 1(2), 56-58.

[41] Nuutila, E., & Soisalon-Soininen, E. (1993). Efficient transitive closure computation. Technical Report TKO-B113, Helsinki University of Technology, Laboratory of Information Processing Science.

[42] Orzan, S. M. (2004). On distributed verification and verified distribution. Ph.D. dissertation, Vrije Universiteit.

[43] Pelánek, R. (2007). BEEM: Benchmarks for Explicit Model Checkers. In SPIN, volume 4595 of LNCS, pp. 263–267. Springer

[44] Purdom Jr, P. (1970). A transitive closure algorithm. BIT Numerical Mathematics, 10(1), 76-94.

[45] Reif, J. H. (1985). Depth-first search is inherently sequential. Information Processing Letters, 20(5), 229-234.

[46] Renault, E., Duret-Lutz, A., Kordon, F., & Poitrenaud, D. (2015). Parallel explicit model checking for generalized Büchi automata. In Tools and Algorithms for the Construction and Analysis of Systems (pp. 613-627). Springer Berlin Heidelberg.

[47] Savage, C. (1982). Depth-first search and the vertex cover problem. Information Processing Letters, 14(5),

[48] Schudy, W. (2008). Finding strongly connected components in parallel using $O(\log^2 n)$ reachability queries. In Parallelism in algorithms and architectures (pp. 146-151). ACM.

[49] Slota, G. M., Rajamanickam, S., & Madduri, K. (2014). BFS and coloring-based parallel algorithms for strongly connected components and related problems. In Parallel and Distributed Processing Symposium, 2014 IEEE 28th International (pp. 550-559). IEEE.

[50] Spencer, T. H. (1991). More time-work tradeoffs for parallel graph algorithms. In Proceedings of the third annual ACM symposium on Parallel algorithms and architectures (pp. 81-93). ACM.

[51] Takac, L., & Zábovský, M. Data Analysis in Public Social Networks, International Scientific Conference & International Workshop Present Day Trends of Innovations, May 2012 Lomza, Poland.

[52] Tarjan, R. E. (1972). Depth-first search and linear graph algorithms. SIAM. Comput. 1:146-60.

[53] Tarjan, R. E. (1976). Edge-disjoint spanning trees and depth-first search. Acta Informatica, 6(2), 171-185.

[54] Tarjan, R. E., & van Leeuwen, J. (1984). Worst-case analysis of set union algorithms. Journal of the ACM (JACM), 31(2), 245-281.

[55] de la Torre, P., & Kruskal, C. (1991). Fast and efficient parallel algorithms for single source lexicographic depth-first search, breadth-first search and topological-first search. In International Conference on Parallel Processing (Vol. 3, pp. 286-287).

[56] Träff, J. L. (2013). A Note on (Parallel) Depth-and Breadth-First Search by Arc Elimination. arXiv preprint arXiv:1305.1222.

[57] Vardi, M. Y., & Wolper, P. (1986). An automata-theoretic approach to automatic program verification. In 1st Symposium in Logic in Computer Science (LICS). IEEE Computer Society.

[58] Wyllie, J. C. (1979). The complexity of parallel computations. Technical Report TR79-387, Cornell University.

A. The Complete UFSCC Algorithm

The current appendix presents the full version of the UFSCC algorithm and the iterable union-find data structure it relies on. We do not provide detailed correctness arguments here (cf. [8]).

Algorithm 3 presents the UFSCC algorithm using the interface of the union-find structure: for iteration it calls PICKFROMLIST, and MAKECLAIM reports the successor status (NEW, FOUND, or DEAD). The exact interface definition of the union-find structure follows the conditions that can be found in Algorithm 2 (red lines). E.g., REMOVEFROMLIST marks a vertex DONE. MAKECLAIM adds the worker to the worker set of the supernode (with an atomic bitwise OR on a bitvector). PICKFROMLIST, however, combines iteration with adding SCCs to DEAD (Line 16 in Algorithm 2): It either returns NULL (marking the SCC DEAD instantly), or a vertex $v' \in S(v) \setminus$ DONE (though another worker may have marked v' DONE just before returning v', which we allow).

Algorithm 4 shows the implementation of the different union-find operations for creating, finding and uniting disjoint sets. The former two remain similar to typical union-find implementations and require no locking, allowing for high concurrency. That synchronization is not required, is a consequence of the particular requirements from the UFSCC algorithm, which guarantees us, that UNITE and PICKFROMLIST (marking supernodes dead) globally always occur after the last FIND on the supernode since at that point all its edges have been processed. The UNITE procedure

(1) selects a new root (Line 42),
(2) locks the other root vertices on both cycles (Line 44-46),
(3) swaps the list next pointers to create one large cycle (Line 49),
(4) updates the parent for the ‘non-root’ (Line 50),
(5) updates the worker set (Line 51-54), and
(6) finally releases the locks (Line 55-57).

This order of execution is crucial to maintain correctness as even interchanging steps (4) and (5) could lead to erroneous results. By only locking the smaller SCC, we allow concurrent updates to the larger SCC, which helped significantly to improve the performance.

Algorithm 5 provides the cyclic list implementation that allows UFSCC to treat disjoint sets as queues, *while they are being merged and searched.* PICKFROMLIST traverses over DONE vertices until a BUSY one is found at Line 3-4. At Line 13, the procedure waits (for UNITE) until locked vertices become BUSY or DONE. The list contracts vertices at Line 15 for two subsequent DONE vertices (see also Figure 5), which resembles path halving [54]. No synchronization is further required as path-halving is only performed on DONE vertices (whereas UNITE only merges BUSY list nodes). REMOVEFROMLIST only needs to mark the vertex DONE and it will be taken care of by the path halving later.

Algorithm 3 The implementation UFSCC algorithm

```
 1: ∀p ∈ [1 . . . p] : R_p := ∅ .................... [local stack for each worker]
 2: procedure UFSCC_MAIN(v_0, p)
 3:   | for each p ∈ [1 . . . p] do
 4:   |   | MAKECLAIM(v_0, p) ...................... [Set worker IDs for v_0]
 5:   | UFSCC_1(v_0) || . . . || UFSCC_p(v_0) .................. [run in parallel]
 6: procedure UFSCC_p(v)
 7:   | R_p.PUSH(v) ...................................... [start ‘new’ SCC]
 8:   | while v' := PICKFROMLIST(v) do
 9:   |   | for each w ∈ RANDOM(POST(v')) do
10:   |   |   | claim := MAKECLAIM(w, p)
11:   |   |   | if claim = CLAIM_NEW then
12:   |   |   |   | UFSCC_p(w) ....................... [recursively explore w]
13:   |   |   | else if claim = CLAIM_FOUND then
14:   |   |   |   | while ¬SAMESET(v, w) do
15:   |   |   |   |   | r := R_p.POP()
16:   |   |   |   |   | UNITE(r, R_p.TOP())
17:   |   | REMOVEFROMLIST(v') ................. [fully explored POST(v')]
18:   | if v = R_p.TOP() then R_p.POP() ............ [remove completed SCC]
```

Algorithm 4 The concurrent, iterable union-find data structure

```
 1: ∀v ∈ V : .................... [Shared union-find structure implementing S]
 2:   | UF[v].parent := v .............................. [union-find parent]
 3:   | UF[v].workers := ∅ .................................... [worker set]
 4:   | UF[v].next := v .................................. [list next pointer]
 5:   | UF[v].uf_status := LIVE .................. [∈ {LIVE, LOCK, DEAD}]
 6:   | UF[v].list_status := BUSY .............. [∈ {BUSY, LOCK, DONE}]
 7: procedure MAKECLAIM(a, p)
 8:   | a_Root := FIND(a)
 9:   | if UF[a_Root].uf_status = DEAD then
10:   |   | return CLAIM_DEAD .................................. [empty list]
11:   | if p ∈ UF[a_Root].workers then
12:   |   | return CLAIM_FOUND .................... [SCC contains worker ID]
13:   | while p ∉ UF[a_Root].workers do ..................... [add worker ID]
14:   |   | UF[a_Root].workers := UF[a_Root].workers ∪ {p}
15:   |   | a_Root := FIND(a_Root) .............. [ensure that worker is added]
16:   | return CLAIM_NEW
17: procedure FIND(a)
18:   | if UF[a].parent ≠ a then
19:   |   | UF[a].parent := FIND(UF[a].parent)
20:   | return UF[a].parent
21: procedure SAMESET(a, b)
22:   | a_Root := FIND(a)
23:   | b_Root := FIND(b)
24:   | if a_Root = b_Root then return TRUE
25:   | if UF[a_Root].parent = a_Root then return FALSE
26:   | return SAMESET(a_Root, b_Root)
27: procedure LOCKROOT(a)
28:   | if CAS(UF[a].uf_status, LIVE, LOCK) then
29:   |   | if UF[a].parent = a then return TRUE
30:   |   | UNLOCKROOT(a)
31:   | return FALSE
32: procedure LOCKLIST(a)
33:   | a_List := PICKFROMLIST(a)
34:   | if a_List = NULL then return NULL ...................... [DEAD SCC]
35:   | if CAS(UF[a_List].list_status, BUSY, LOCK) then
36:   |   | return a_List ............................ [successfully locked list]
37:   | return LOCKLIST(UF[a_List].next)  [a_List is locked by another worker]
38: procedure UNITE(a, b)
39:   | a_Root := FIND(a)
40:   | b_Root := FIND(b)
41:   | if a_Root = b_Root then return ........................ [already united]
42:   | r := MAX(a_Root, b_Root) ................. [Largest index is new root]
43:   | q := MIN(a_Root, b_Root)
44:   | if ¬LOCKROOT(q) then return UNITE(a_Root, b_Root)
45:   | a_List := LOCKLIST(a)
46:   | b_List := LOCKLIST(b)
47:   | if a_List = NULL ∨ b_List = NULL then
48:   |   | return ............................. [DEAD SCC ⇒ already united]
49:   | SWAP(UF[a_List].next, UF[b_List].next) ................ [non-atomic]
50:   | UF[q].parent := r ................................... [update parent]
51:   | do ............................................ [update worker set]
52:   |   | r := FIND(r)
53:   |   | UF[r].workers := UF[r].workers ∪ UF[q].workers
54:   | while UF[r].parent ≠ r ............... [ensure that we update the root]
55:   | UF[a_List].list_status := BUSY ............... [unlock list on a_List]
56:   | UF[b_List].list_status := BUSY ............... [unlock list on b_List]
57:   | UF[q].uf_status := LIVE ................................. [unlock q]
```

Algorithm 5 The list part of the iterable union-find

```
 1: procedure PICKFROMLIST(a)
 2:   | do
 3:   |   | if UF[a].list_status = BUSY then return a
 4:   | while UF[a].list_status = LOCK .............. [exit if status = DONE]
 5:   | b := UF[a].next
 6:   | if a = b then ................................ [Cycle becomes empty]
 7:   |   | a_Root := FIND(a)
 8:   |   | if CAS(UF[a_Root].uf_status, LIVE, DEAD) then
 9:   |   |   | report SCC a_Root
10:   |   | return NULL ........................ [empty cycle ⇒ finished SCC]
11:   | do
12:   |   | if UF[b].list_status = BUSY then return b
13:   | while UF[b].list_status = LOCK .............. [exit if status = DONE]
14:   | c := UF[b].next ..................................... [a → b → c]
15:   | UF[a].next := c ................................ [a and b are DONE]
16:   | return PICKFROMLIST(c)
17: procedure REMOVEFROMLIST(a)
18:   | while UF[a].list_status ≠ DONE do
19:   |   | CAS(UF[a].list_status, BUSY, DONE) . [only remove BUSY nodes]
```