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 and 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 union-find with a cyclic list).

We demonstrate scalability on a 64-core machine using 75 real-world graphs (from model checking), 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 [48]. Later it was used for planarity [24], spanning trees [49], topological sort [13], fair cycle detection [15] (a problem arising in model checking [53]), 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 algorithm more generally.

Traditional parallellization. Direct parallelization of DFS-based algorithms is a challenge. Lexicographical, or ordered DFS is P-complete [42], 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 tried to improve FB [19, 23, 39], but the expected worst-case complexity of $\mathcal{O}(n \cdot (n+m))$ remains [5]. Parallelizations of topological sort [6, 25], fair cycle detection [10, 11] and spanning trees [4] have similar limitations.

Another important 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 [28, 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 $I_G = (v_0, \text{POST}())$, where $v_0 \subseteq V$ is a small set of initial vertices and $\text{POST}(v) \stackrel{\text{def}}{=} \{v' \in V \mid (v, v') \in E\}$, and requires only $\mathcal{O}(n)$ 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]. It also benefits algorithms that rely on SCC detection but do not require explicit graph representations, e.g. [38].

Parallel Randomized DFS (PRDFS). A novel approach has shown that the DFS-based algorithms can be parallelized more directly, without sacrificing complexity and on-the-flyness [17, 18, 29, 30, 33, 34, 43]. The idea is simple: (1) start from naively running the sequential algorithm on $\mathcal{P}$ 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 [6]), *showing that the oft-cited theoretical result from Reif [42] does not even apply directly*. In fact, it is unknown whether the SCC decomposition problem is P-complete [46], nor whether the non-lexicographical DFS is [51].

For correctness, the pruning process in PRDFS should carefully limit the influence on the search order of other threads. Trivially, in the case of SCCs, a thread running Tarjan can remove an SCC from the graph as soon as it is completely detected [34], as long as done atomically [43]. This would result in limited scalability for

PPOPP '16, Month 12–16, 2016, Madrid, ST, Spain.

http://dx.doi.org/10.1145/nnnnnnn.nnnnnnn

Table 1: Complexities of fix-point (e.g. [45]) and PRDFS solutions (e.g. [43]) for problems taking linear time sequentially: $\mathcal{O}(n+m)$.

	Best-case ($\mathcal{O}$)		Worst-case ($\mathcal{O}$)	
	Runtime	Work	Runtime	Work
Traditional	$\frac{n+m}{\mathcal{P}}$	$n+m$	$n(n+m)$	$n(n+m)$
PRDFS	$\frac{(n+m)}{\mathcal{P}}$	$n+m$	$n+m$	$\mathcal{P}(n+m)$

graphs consisting of a single SCC [43]. But, more intricate ways of pruning already showed better results in other areas [17, 29, 30].

Time and work tradeoff play an import role in parallelizing many of the above algorithms [46]. 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 $\mathcal{O}(\mathcal{P} \times (n+m))$, i.e.: a factor $\mathcal{P}$ 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 an efficient PRDFS-based SCCs algorithm solves both parallel and on-the-fly SCC decomposition to large extent.*

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 suffers 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 [50]. In a sense therefore it is based on SCC algorithms before Tarjan's [37, 41]. 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 to a shutter form, as shown in Fig. 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.

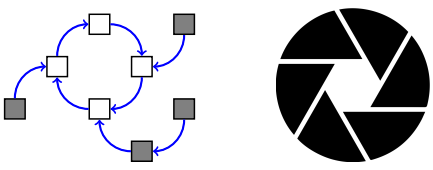

Figure 1: Schematic of the concurrent, iterable queue, which operation resembles a closing camera shutter (on the right). White nodes (invariably on a cycle) are still queued, whereas gray nodes have been dequeued (contracting the cycle), but can nonetheless be used to query queued nodes, as they invariably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

2. Preliminaries

Given a directed graph $G \stackrel{\text{def}}{=} (V, E)$, we denote an edge $(v, v') \in E$ as $v \to v'$. A *path* is a sequence of vertices $v_0 \dots v_k$, s.t. $\forall_{0 \le i \le k} : v_i \in V$ and $\forall_{0 \le i < k} : v_i \to v_{i+1}$. The transitive closure of E is denoted $v \to^* w$, i.e. there is some path $v \dots w \in V^*$. If a vertex reaches itself via a non-empty path, the path is called a *cycle*. Vertices v and w are *strongly connected* iff $v \to^* w \wedge w \to^* v$, 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 $n \stackrel{\text{def}}{=} |V|$ and $m \stackrel{\text{def}}{=} |E|$.

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: $G_I \stackrel{\text{def}}{=} (v_0, \text{POST}())$, where v_0 is an initial vertex and $\text{POST}(v)$ a *successors function*: $\text{POST}(v) \stackrel{\text{def}}{=} \{w \mid v \to w\}$. 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 [41], and later optimized by Munro [37]. 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 these sets 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 Dijksta [16]). 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 connected components are tracked in a collection of disjoint sets, which we represent using a map: $S \colon V \to 2^V$ with the invariant: $\forall v, w \in V \colon w \in S(v) \Rightarrow S(v) = S(w)$. 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 $S(v) = \{v\}$ and $S(w) = \{w, x\}$, then $\text{UNITE}(S, v, w)$ yields $S(v) = S(w) = S(x) = \{v, w, x\}$ keeping all other mappings the same.

Algorithm 1 Sequential set-based SCC algorithm

```
1:  ∀v ∈ V : S(v) := {v}
2:  DEAD := VISITED := ∅
3:  R := ∅
4:  SETBASED(v_0)
5:  procedure SETBASED(v)
6:      VISITED := VISITED ∪ {v}
7:      R.PUSH(v)
8:      for each w ∈ POST(v) do
9:          if w ∈ DEAD then continue ..... [already completed SCC]
10:         else if w ∉ VISITED then ..... [unvisited node w]
11:             SETBASED(w)
12:         else ..... [cycle found ⇒ unite all states on cycle]
13:             while S(v) ≠ S(w) do
14:                 r := R.POP()
15:                 UNITE(S, r, R.TOP())
16:     if v = R.TOP() then ..... [completely explored SCC]
17:         report SCC S(v)
18:         DEAD := DEAD ∪ S(v)
19:         R.POP()
```

The algorithm's base DFS can be seen on Line 6, Line 8, and Line 10-11 (ignoring the condition at Line 16). The recursive procedure is called for every unvisited successor vertex w from v. Because cycles are collapsed immediately at Line 13–15, the stack R is maintained independently of the program stack and invariantly contains a subset of the latter (in the same order).

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. $v \notin$ DEAD $\cup$ VISITED, 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 $\stackrel{\text{def}}{=}$ 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:

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

As a consequence, the LIVE vertices can be reconstructed using the map 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:

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

Or, for each LIVE vertex v, its mapped supernode $S(v)$ has exactly one $r \in R \cap S(v)$ s.t. $S(v) = S(r)$. 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 $\exists r \in R : S(r) = S(w)$. Such a LIVE vertex is handled by Line 13-15, where the top two vertices from R are united, until r is encountered, or rather until it holds that $S(r) = S(v) = S(w)$. Because r has a path to v (an inherited invariant of the program stack), all united components are indeed connected, i.e. lie on a cycle. Moreover, because of r's uniqueness (Eq. 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 $S(v)$ is reported and marked DEAD if v remains on top of the R stack at Line 16, 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* [50] 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 [22]. 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 S and the UNITE() operation can be implemented with a union-find structure as shown by [21]. 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.) A basic way of doing this would be by 'removing' completed SCCs from the graph once identified by one worker. 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. 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 Eq. 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 S, it is no longer useful to retain a local VISITED set. Eq. 1 showed that indeed the LIVE vertices can be deduced from 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. $v \in S(r)$, 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.

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 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.

First of all, instead of performing only a DFS, the algorithm also iterates over vertices from $S(v)$, at Line 7-16, until every $v' \in S(v)$ is marked DONE (principle 2 and 4). Therefore, once this while-loop is finished, we have that $S(v) \subseteq$ DONE, hence the SCC may be marked DEAD in Line 17. The if-condition at that line succeeds if worker p wins the race to add $S(v)$ 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 18).

For every $v' \in S(v) \setminus$ 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

Algorithm 2 The UFSCC algorithm (code for worker p)

```
1:  ∀v ∈ V : S(v) := {v} ..... [global S]
2:  DEAD := DONE := ∅ ..... [global DEAD and DONE]
3:  ∀p ∈ [1...P] : R_p := ∅ ..... [local R_p]
4:  UFSCC_1(v_0) || ... ||UFSCC_p(v_0)
5:  procedure UFSCC_p(v)
6:      R_p.PUSH(v)
7:      while v' ∈ S(v) \ DONE do
8:          for each w ∈ RANDOM(POST(v')) do
9:              if w ∈ DEAD then continue ..... [DEAD]
10:             else if ∄w' ∈ R_p : w ∈ S(w') then ..... [NEW]
11:                 UFSCC_p(w)
12:             else ..... [LIVE]
13:                 while S(v) ≠ S(w) do
14:                     r := R_p.POP()
15:                     UNITE(S, r, R_p.TOP())
16:         DONE := DONE ∪ {v'}
17:     if S(v) ⊈ DEAD then DEAD := DEAD∪S(v); report SCC S(v)
18:     if v = R_p.TOP() then R_p.POP()
```

the for-loop from Line 8-15, we infer that $\text{POST}(v') \subseteq (\text{DEAD} \cup S(v))$ and therefore v' may be marked DONE (principle 4).

Compared to the VISITED set from Algorithm 1, this algorithm uses 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. Sec. 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 $\forall a, b \in S(v) : a \leftrightarrow b$ holds, we have that $v \to v' \to w$ 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 Th. 1) by showing that vertices added to $S(v)$ are always strongly connected with v (or any other vertex in $S(v)$). Completeness (see Th. 2) follows from the requirements to mark an SCC DEAD (Lem. 2–4). Complete correctness follows from termination in Th. 3.

Lemma 1 (Stack)**.** $\forall r \in R_p : r \to^* R_p.\text{TOP}()$.

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:* $\forall v, w : w \in S(v) \Rightarrow v \to^* w \to^* v$.

Proof. Initially the hypothesis holds, since $\forall v : S(v) = \{v\}$. Assuming that the theorem holds for S before the executing a statement of the algorithm, we show it holds after, considering only the non-trivial case, i.e. Line 15. Before Line 13, we have $S(v) = S(R_p.\text{TOP}())$ (either $v = R_p.\text{TOP}()$ or a recursive procedure has united v with the current $R_p.\text{TOP}()$). The while-loop (Line 13-15) continuously pops the top vertex of R_p and unites it with the new $R_p.\text{TOP}()$. Since $\exists w' \in R_p : w \in S(w')$ (the condition at Line 10 did not hold), the loop ends iff $S(R.\text{TOP}()) = S(v) = S(v') = S(w) = S(w')$. Because $v' \to w$ and $w' \to^* R.\text{TOP}()$ (Lem. 1), we have that the cycle $v' \to^* w \to^* v'$ is merged, thus preserving the hypothesis in the updated S. □

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

Proof. This follows directly from Line 17 and the fact that v is never removed from $S(v)$. □

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

Proof. The only place where DONE is modified is in Line 16. Vertices are never removed from DEAD or $S(v)$ for any v. In Line 8-15, 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 Lem. 2, we obtain that $w \in$ DEAD upon return of UFSCC$_p(w)$.
3. there is some $w' \in R_p$ s.t. $S(w) = S(w')$. Supernodes of vertices on R are united until $S(v) = S(v') = S(w)$.

All three cases thus satisfy the conclusion of the hypothesis before v' is added to DONE at Line 16. □

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

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) $S(t)$ is a maximal SCC.*

Proof. (a. every reachable vertex t is DEAD). Since $v \in$ DEAD (Lem. 2), we have $\forall v' \in S(v) : v' \in$ DONE (Lem. 4). By Lem. 3, we deduce that every successor of v' is DEAD (as $v' \in S(v)$ implies $v' \in$ DEAD by Line 17). Therefore, by applying the above reasoning recursively on the successors of v' we obtain that every reachable vertex from v is DEAD.

(b. $S(t)$ is a maximal SCC). Assume by contradiction that for disjoint DEAD sets $S(v) \neq S(w)$ we have $v \to^* w \to^* v$. W.l.o.g., we can assume $v \to w$. Since $S(w) \neq S(v)$, we deduce that $w \in$ DEAD when this edge is encountered (Lem. 3). However, since $w \to^* v$ we must also have $v \in$ DEAD (Th. 2a), contradicting our assumption. Hence, every DEAD $S(t)$ 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 $S(v)$ 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 15. □

4. Implementation

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

- Data structures for R_p, S, DEAD and DONE.
- A mechanism to iterate over $v' \in S(v) \setminus$ DONE (Line 7).
- Means to check if a vertex v is part of $(S(v) \cap R_p)$ (Line 10).
- An implementation of the UNITE() procedure (Line 15).
- A technique to add vertices to DONE and DEAD (Line 16,17).

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

R_p can be implemented with a standard stack structure. 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(S, a, b) procedure, the new root of $S(a)$ and $S(b)$ is determined by selecting either the root of $S(a)$ or $S(b)$, 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 [22].

The DEAD set is implemented with a status field in the union-find structure. We have that $S(v)$ is DEAD if the status for the root of v is DEAD, thus marking an SCC DEAD, indeed implementing

Line 17 of Algorithm 2 atomically. This marking is achieved in quasi-constant time (a FIND() call with a status update).

Worker set. We show how $\nexists w' \in R_p : S(w') = S(w)$ (Line 10) is implemented. In the union-find structure, for each node we keep track of a bitset of size $\mathcal{P}$. 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 $S(w)$ 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 $S(w)$ using CAS and recursively calls UFSCC(w) (Line 10). In the UNITE(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.

Cyclic linked list. The most challenging aspect of the implementation is to keep track of the DONE vertices and iterate over $S(v) \setminus$ 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 LIVE or DONE.

The idea is that the cyclic list contains all LIVE 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 LIVE vertices is reachable from every vertex $v' \in S(v)$. If all vertices from $S(v)$ are marked DONE, the cycle becomes empty (we end up with a DONE vertex directing to itself) and $S(v)$ can be marked DEAD.

In the algorithm, Line 7 can be implemented by recursively traversing v.`next` to eventually encounter either a vertex $v' \in S(v) \setminus$ DONE or otherwise it detects that there is no such vertex and the while-loop ends. Line 16 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(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 Fig. 2. Here UNITE(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 LIVE 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 LIVE vertices a' and f'. The locks also protect the non-atomic pointer swap.

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 LIVE vertices, if two subsequent DONE vertices are

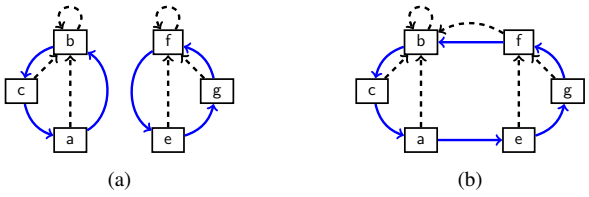

Figure 2: Cyclic list before (a) and after (b) a UNITE(S, a, f) call. Dashed edges depict parent pointers and solid edges `next` pointers.

Figure 3: Cyclic list before (a) and after (b) a list 'removal'. Solid edges depict `next` pointers and gray nodes are DONE.

encountered, the `next` pointer for the first one is updated (Fig. 3). Note that the 'removed' vertex remains pointing to the cyclic list.

We use a lock on the union-find structure (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 Fig. 4. 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[$\mathcal{P}$]	`worker_set`	..... [worker set]
bit[2]	`uf_status`	[$\in$ {LIVE, LOCK, DEAD}]
bit[2]	`list_status`	[$\in$ {LIVE, LOCK, DONE}]

Figure 4: The node structure in our iterable union-find.

Memory usage. The space complexity for Algorithm 2, using the implementation that we just discussed, consists of two aspects. First, each worker has a local stack R_p, which contains at most $\mathcal{D}$ items, where $\mathcal{D}$ is the diameter if the graph. 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 (Fig. 4). Assume that storing a vertex take H words (machine words). The pointers in the structure use at most H space and the worker set takes $\mathcal{P}$ space. Then, the algorithm uses at most $\mathcal{O}((\mathcal{P} \times \mathcal{D} \times H) + (|V| \times (3H + \mathcal{P}))) = \mathcal{O}(|V| \times \mathcal{P} \times H)$ space.

In practice, the graph diameter is small ($\mathcal{D} \ll |V|$), and $\mathcal{P}$ is a relatively small constant, such that a worker set takes at most H space. Therefore, practical memory usage is $|V| \times 4H$ space.

Runtime/work tradeoff. Our treatment of the complexity here is not as formal as in e.g. [51], 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 $\mathcal{O}(\mathcal{P})$ time and work (instructions). Assuming again that $\mathcal{P}$ 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 LIVE vertex in the cyclic list is bounded by $\mathcal{O}(n)$ (the maximal sequence of DONE vertices before reaching a LIVE vertex). However, we suspect that due to the lazy removal of DONE nodes, similar to path-compression [50], supernode iteration is amortized quasi constant under the condition that $\mathcal{P} \ll n$. Though if this not the case, we could always use a more complicated doubly linked list instead. 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 $\mathcal{O}(\alpha\mathcal{P}(n + m))$. As typically, the inverse Ackermann function is bounded by a

constant, we obtain $\mathcal{O}(\mathcal{P}(n+m))$. *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 follow when assuming that each vertex is processed by only one worker. [1]

5. Related Work

Parallel on-the-fly algorithms. Renault et al. [43] present a PRDFS algorithm that spawns multiple instances of Tarjan's algorithm and communicates completely explored SCCs via a shared union-find structure. They do not however explicitly describe their work as a parallel SCC algorithm, but focus on parallelizing model checking. Lowe [33] 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. Sec. 1 discussed several fix point based solutions. A notable improvement to FB [19] is the trimming procedure [36], which removes trivial SCC without fix point recursion. OBF [5] further improves this by subdividing the graph in a number of independent sub-graphs. The Coloring algorithm [39] propagates priorized colors dividing the graph in disconnected subgraphs whose backwards slices identify SCCs. Barnat et al. [7] provide CUDA implementations for FB, OBF, and Coloring.

Hong et al. [23] improve FB, 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 (with Coloring) and decomposed with FB. Slota et al. [45] propose orthogonal optimization heuristics and combine them with Hong's work.

Parallel DFS. Parallel DFS is remotely related. It has long been researched intensively, though there are two types of works [20]: 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, 51]. Recent work from Träff [52] proposes a strategy to processes incoming edges, allowing them to be handled in parallel. The resulting complexity is $\mathcal{O}(\frac{m}{\mathcal{P}}+n)$, providing speedup for dense graphs. Research into the characterization of search orders provides other new venues to analyze these algorithms [8, 14]

Union-find. The investigation of van Leeuwen and Tarjan settled the question which of many union-find implementations were superior [50]. Goel er al. [22] however later showed that a simpler randomized implementation can also be superior. As discussed, concurrent union-find structure exists [2]. There are also versions that support deletion [27]. To the best of our knowledge, we are the first to combine both iteration and removal in the union-find structure, allowing the disjoint sets to be processed as a queue and at the same time indexed and merged.

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.

[1] We acknowledge that for fair comparison with Table 1, the impact of the worker set (a factor $\mathcal{P}$) cannot be neglected, yielding $\mathcal{O}(\mathcal{P} \times \max(\alpha, \mathcal{P}) \times (n+m))$. However, we also think that the worker set can often be stored locally on the R stack trading more redundant re-explorations due to late updates for faster supernode operations. Initial experimentation with such an approach showed good results.

Table 2: Graphs characteristics for model checking, synthetic and explicit graphs.

graph	states	transitions	# sccs	max scc size
leader-filters.7	26,302,351	91,692,858	26,302,351	1
bakery.6	11,845,035	40,400,559	2,636,212	8,567,257
cambridge.6	3,354,295	9,483,191	8,413	3,345,883
lup.3	1,948,617	3,621,672	1	1,948,617
resistance.1	13,762,624	32,052,024	3	13,762,622
sorter.3	1,288,478	2,740,540	1,177,164	278
L1751L1751T1	9,198,003	24,528,008	3	3,066,001
L351L351T4	3,819,231	11,334,492	31	123,201
L5L5T16	3,276,775	9,830,300	131,071	25
Li10Lo200	4,000,000	15,200,000	100	40,000
Li50Lo40	4,000,000	15,840,000	2,500	1,600
Li200Lo10	4,000,000	15,960,000	40,000	100
livej	4,847,571	68,993,773	971,232	3,828,682
patent	3,774,768	16,518,948	3,774,768	1
pokec	1,632,804	30,622,564	325,893	1,304,537
random	10,000,001	100,000,000	877	9,999,125
rmat	9,999,994	100,000,000	2,247,072	7,752,923
ssca2	1,048,577	157,877,986	31	1,048,546
wiki-links	5,710,993	130,160,392	2,002,220	3,703,148

Experimental setup. All experiments were performed on a machine with 4 AMD Opteron™ 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[2] UFSCC in the LTSMIN model checker [26], which we use for decomposing SCCs on implicitly given graphs. LTSMIN's POST() implementation applies measures to permute the order for a state'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. [43] (see Sec. 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 [33], 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. [23]. This implementation is compared against benchmarks of Tarjan and Hong's concurrent algorithm (both provided by Hong et al.). While Slota et al. [45] 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.

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.

[2] All our implementations and benchmarks will be made available online.

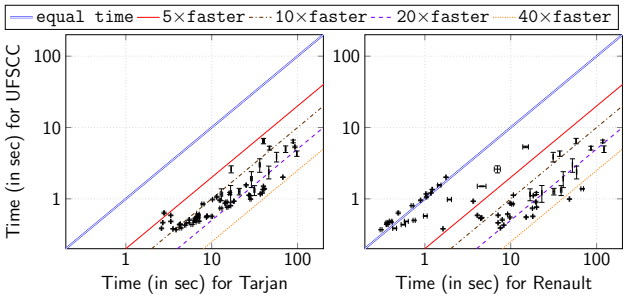

Figure 5: Comparison of times for 62 model checking graphs for concurrent UFSCC against Tarjan (left) and concurrent Renault (right), where UFSCC and Renault use 64 workers.

Experiments on model checking graphs. We use model checking problems from the BEEM database [40], 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'.

Fig. 5 (left) compares the runtimes for UFSCC on 64 workers against Tarjan's sequential algorithm in a scatter plot. The central diagonal represents equal runtimes and the dots below that line experiments that exhibit speedup. We observe that in most cases UFSCC performs at least 10× faster than Tarjan. The performance improvements for all model checking graphs range from 4.4 (`sorter.3`) to 33.6× faster (`leader-filters.7`) and the geometric mean performance increase for UFSCC over Tarjan is 13.87. Compared to the sequential runtime of UFSCC, its multi-core version exhibits a geometric mean speedup of 18.22. 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).

Fig. 5 (right) compares UFSCC with Renault on model checking graphs, both using 64 workers. For a number of graphs Renault performs slightly better (up to 1.4× 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 5.67.

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. 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 Fig. 6 and Table 3 quantifies the performance results for 64 workers. In Fig. 6, 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 (as reflected by the error bars), 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 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 `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).

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 (where multiple workers explore the same vertex). 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. [5]) to find out how particular aspects of a graph influence UFSCC's scalability. The first type of graph, called $\mathtt{L}m\mathtt{L}m\mathtt{T}n$, is the Cartesian product for two cycles of m states

Table 3: Comparison of sequential on-the-fly Tarjan with concurrent UFSCC and Renault on 64 cores.

graph	execution time (s)			UFSCC speedup vs	
	Tarjan	Renault	UFSCC	Tarjan	Renault
`leader-filters.7`	67.528	1.776	2.011	33.580	0.883
`bakery.6`	41.491	68.816	1.382	30.019	49.789
`cambridge.6`	15.310	20.169	0.764	20.044	26.406
`lup.3`	5.507	8.012	0.510	10.808	15.724
`resistance.1`	39.529	57.615	6.473	6.106	8.900
`sorter.3`	2.794	0.494	0.635	4.400	0.778
`L1751L1751T1`	26.925	24.416	2.271	11.856	10.751
`L351L351T4`	11.158	3.049	0.808	13.808	3.773
`L5L5T16`	7.543	0.593	0.823	9.163	0.720
`Li10Lo200`	14.439	4.581	1.242	11.622	3.687
`Li200Lo10`	12.350	6.062	2.856	4.325	2.123
`Li50Lo40`	12.764	5.976	1.723	7.410	3.469

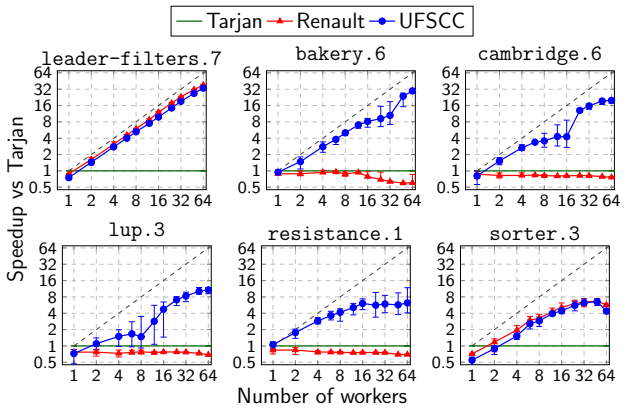

Figure 6: Scalability for the on-the-fly UFSCC and Renault implementations relative to Tarjan's algorithm on a set of model checking inputs. Error bars represent minimum and maximum speedups.

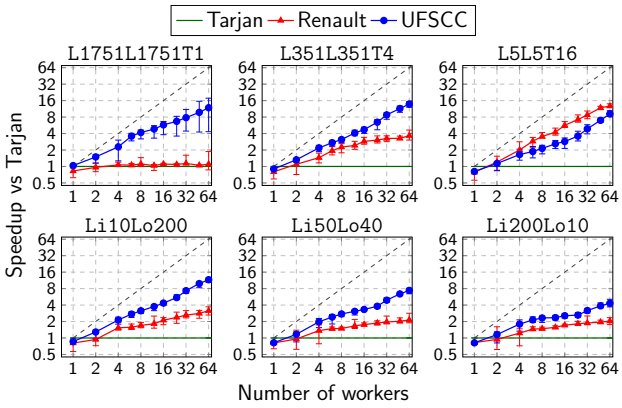

Figure 7: Scalability for selected synthetic graphs of UFSCC and Renault relative to Tarjan's algorithm. Error bars represent minimum and maximum speedups.

Table 4: Comparison of sequential offline Tarjan with concurrent UFSCC and Hong on 64 cores.

graph	execution time (s)			UFSCC speedup vs	
	Tarjan	Hong	UFSCC	Tarjan	Hong
leader-filters.7	2.760	1.629	0.506	5.455	3.219
bakery.6	1.727	5.223	0.646	2.672	8.081
cambridge.6	0.277	0.389	0.148	1.875	2.632
lup.3	0.417	0.105	0.144	2.900	0.731
resistance.1	2.035	5.673	2.964	0.687	1.914
sorter.3	0.091	3.776	0.055	1.659	68.886
L1751L1751T1	0.882	116.252	1.042	0.846	111.564
L351L351T4	0.252	29.592	0.270	0.935	109.693
L5L5T16	0.197	error	0.077	2.560	-
Li10Lo200	0.251	32.483	0.334	0.750	97.215
Li50Lo40	0.222	22.554	0.198	1.120	114.073
Li200Lo10	0.224	40.252	0.299	0.748	134.447
livej	2.934	0.298	0.775	3.784	0.385
patent	0.909	0.063	0.271	3.353	0.233
pokec	1.357	0.130	0.321	4.234	0.406
random	9.989	0.634	1.566	6.377	0.405
rmat	8.288	0.531	1.490	5.563	0.356
ssca2	1.651	0.250	0.338	4.885	0.739
wiki-links	4.230	1.691	0.967	4.375	1.749

with a binary tree of depth n (generated by taking the parallel composition of processes with said control flow graphs: $Loop(m) \| Loop(m) \| Tree(n)$). This graph has $2^{n+1} - 1$ components of size m^2.

The second type is called `Li`m`Lo`n and is a parallel composition of two sequences of m states with two cycles of n states $(Line(m) \| Line(m) \| Loop(n) \| Loop(n))$. This graph has m^2 components of size n^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 Fig. 7.

In Fig. 7, we notice that scalability of UFSCC isconsistent with the results in Fig. 6. This 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 (except for `L5L5T16`, which consists 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. [23]). 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 [32] database and represent the LiveJournal social network [3], the citation among US patents [31], and the Pokec social network [47]. 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 [35] 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).

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

- On real-world graphs, UFSCC shows increased performance compared to Tarjan, but Hong clearly outperforms UFSCC (aside from `wiki-links`).
- 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`).

The synthetic graphs 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.

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 (as discussed in footnote [1]), 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.

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, et al. (2006). Group Formation in Large Social Networks: Membership, Growth, and Evolution. KDD.

[4] Bader, D., & Cong, G. (2004). A fast, parallel spanning tree algorithm for symmetric multiprocessors. In Parallel and Distributed Processing Symposium, 2004. 18th International (p. 38). IEEE.

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

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

[6] 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.

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

[8] 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.

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

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

[15] 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.

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

[17] 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.

[18] 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.

[19] 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.

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

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

[22] 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.

[23] 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.

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

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

[26] Kant et al. (2015). LTSmin: High-Performance Language-Independent Model Checking. In TACAS.

[27] 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).

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

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

[30] 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 International Publishing.

[31] 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).

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

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

[34] 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.

[35] 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.

[36] 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.

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

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

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

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

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

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

[43] Renault, E. et al. (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.

[44] Savage, C. (1982). Depth-first search and the vertex cover problem. Information Processing Letters, 14(5), 233-235.

[45] 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.

[46] 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.

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

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

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

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

[51] 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).

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

[53] 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.

A. Complete UFSCC algorithm

The current appendix presents the full version of the UFSCC algorithm and the iterable union-find data structure it relies on. The description is intended as a detailed overview of our algorithm and we do not provide correctness arguments here.

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 17 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 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 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 36), (2) locks the other root vertices on both cycles (Line 38-40), (3) swaps the list next pointers to create one large cycle (Line 43), (4) updates the parent for the 'non-root' (Line 44), (5) updates the worker set (Line 45-48), (6) and finally releases the locks (Line 49-51). 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 LIVE one is found at Line 3-4. At Line 12, the procedure waits (for UNITE) until locked vertices become LIVE or DONE. The list contracts vertices at Line 14 for two subsequent DONE vertices (see also Fig. 3), which resembles path halving [50]. No synchronization is further required as path-halving is only performed on DONE vertices (whereas UNITE only merges LIVE list nodes). REMOVEFROMLIST only needs to mark the vertex DONE and it will be taken care of by the path halving later.

Algorithm 4 The concurrent, iterable union-find data structure

```
1:  ∀v ∈ V : ..... [Shared union-find structure, implements 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 := LIVE ..... [∈ {LIVE, 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 LOCKROOT(a)
22:     if CAS(UF[a].uf_status, LIVE, LOCK) then
23:         if UF[a].parent = a then return TRUE
24:         UNLOCKROOT(a)
25:     return FALSE
26: procedure LOCKLIST(a)
27:     a_List := PICKFROMLIST(a)
28:     if a_List = NULL then return NULL ..... [DEAD SCC]
29:     if CAS(UF[a_List].list_status, LIVE, LOCK) then
30:         return a_List ..... [successfully locked list]
31:     return LOCKLIST(UF[a_List].next) [a_List is locked by another worker]
32: procedure UNITE(a, b)
33:     a_Root := FIND(a)
34:     b_Root := FIND(b)
35:     if a_Root = b_Root then return ..... [already united]
36:     r := MAX(a_Root, b_Root) ..... [Largest index is new root]
37:     q := MIN(a_Root, b_Root)
38:     if ¬LOCKROOT(q) then return UNITE(a_Root, b_Root)
39:     a_List := LOCKLIST(a)
40:     b_List := LOCKLIST(b)
41:     if a_List = NULL ∨ b_List = NULL then
42:         return ..... [DEAD SCC ⇒ already united]
43:     SWAP(UF[a_List].next, UF[b_List].next) ..... [non-atomic]
44:     UF[q].parent := r ..... [update parent]
45:     do ..... [update worker set]
46:         r := FIND(r)
47:         UF[r].workers := UF[r].workers ∪ UF[q].workers
48:     while UF[r].parent ≠ r ..... [ensure that we update the root]
49:     UF[a_List].list_status := LIVE ..... [unlock list on a_List]
50:     UF[b_List].list_status := LIVE ..... [unlock list on b_List]
51:     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 = LIVE 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 report SCC a_Root
9:          return NULL ..... [empty cycle ⇒ finished SCC]
10:     do
11:         if UF[b].list_status = LIVE then return b
12:     while UF[b].list_status = LOCK ..... [exit if status = DONE]
13:     c := UF[b].next ..... [a → b → c]
14:     CAS(UF[a].next, b, c) ..... [a and b are DONE]
15:     return PICKFROMLIST(c)
16: procedure REMOVEFROMLIST(a)
17:     while UF[a].list_status ≠ DONE do
18:         CAS(UF[a].list_status, LIVE, DONE)
```

B. Artifact description

This artifact supports the PPoPP'16 paper: Multi-Core On-The-Fly SCC Decomposition.

B.1 Abstract

The artifact described here consists of experiments that evaluate our novel multi-core on-the-fly strongly connected component (SCC) algorithm, called UFSCC. With these experiments, we demonstrate scalability and show that our algorithm outperforms existing work on a 64-core machine on various types of graphs. The current artifact paper describes how to compile and run the necessary tools, and provides a description on how to reproduce all experiments described in the paper.

B.2 Description

The goal of the experiments in is to demonstrate the performance improvements with an increasing the number of processors, i.e. the scalability of the algorithm, and compare this performance against existing solutions. The artifact therefore implements the UFSCC algorithm in existing tool sets that perform graph analysis. The artifact comprises two parts:

- Experiments for an *on-the-fly* environment, in which an input graph is provided implicitly and computed on-the-fly. The model checker LTSMIN [26] is used for these experiments. Here, UFSCC is compared with Tarjan's sequential SCC algorithm [48] and Renault's parallel SCC algorithm [43].
- Experiments for an *offline* environment, in which an input graph is provided explicitly, i.e. where all vertices and edges of the graph are known beforehand. The tool provided by Hong et. al [23], which we refer to as HONG-UFSCC, is used for these experiments. Here, UFSCC is compared with Tarjan's sequential SCC algorithm [48] and Hong's parallel SCC algorithm [23].

The structure of the current paper is as follows: We first provide a check list (App. B.2.1) describing specific properties of the artifact. Then, in App. B.2.2, we show where all necessary files are located in the artifact. This is followed by the hardware and software dependencies (App. B.2.3, App. B.2.4). App. B.3 describes the installation procedure. Finally, App. B.4 and App. B.5 explain how to perform and analyze experiments.

B.2.1 Check-list (artifact meta information)

- **Algorithm:** graph, on-the-fly, strongly connected components, multi-core, scalability
- **Program:** LTSMIN model checker, HONG-UFSCC
- **Data set:** included data sets for:
  - LTSMIN: BEEM (BEnchmarks for Explicit Model checkers [40]) + synthetic graphs
  - HONG-UFSCC: real-world + synthetic + selected BEEM graphs
- **Run-time environment:** POSIX / Linux
- **Hardware:** 64bit x86 multi-core machine with 64 cores
- **Output:** CSV with performance info, option to compile PDF table/graph from CSV
- **Experiment workflow:** bash script executes experiments
- **Publicly available?:** open source via Github

B.2.2 How delivered

The source code for the used tools, including the UFSCC implementation, is available at Github:

`https://github.com/utwente-fmt/ppopp16`

This project contains (most of) the experiments combined with scripts to perform them and analyze their results. Structurally, the project contains the following:

- A bash script `install.sh`, this script provides commands to compile and install LTSMIN and HONG-UFSCC (we provide detailed installation instructions in App. B.3).
- A directory `experiments` that contains:
  - A bash script `benchmark.sh`, this script is used to run all experiments.
  - A Python script `parse-output.py`, this script is used to analyze the program output and store the results in a CSV file (or in case of a failure, copy the program output to a failure folder).
  - A Python script `csv2graph.py`, this script generates either a time or speedup graph for a specific graph.
  - A Python script `csv2table.py`, this script generates a table containing info regarding times and speedups for multiple graphs.
  - A Python script `csv2scatter.py`, this script generates a scatterplot containing info regarding times and speedups for multiple graphs.
  - A bash script `parse-csv.sh`, this script is intended as an example on how to extract graphs/tables from the generated CSV data (using the `csv2..` scripts).
  - A directory `results-paper` that contains the CSV outputs for our experiments that we present in the PPoPP paper.
  - A directory `graphs`, that contains:
    - A directory `beem`, this contains a set of input graphs obtained from the BEEM database.
    - A directory `beem-selected`, this contains the input graphs for six 'selected' input graphs from the beem directory.
    - A directory `synthetic`, this contains several synthetically generated input graphs.

Due to size restrictions, the input graphs for the 'offline' experiments are provided in a separate zip file, provided at:

`https://www.dropbox.com/sh/1n145b3szeqflv9/AACXfz88_xGeJ20FZy-dz1UXa?dl=0`

This zip file contains two non-empty directories: `real-offline` and `constructed-offline`. In order to perform the offline experiments, please copy these directories to the `graphs` directory.

The `real-offline` directory contains various real-world graphs. The `livej` [3], `patent` [31] and `pokec` [47] graphs were obtained from the SNAP [32] database. The graph `wiki-links`[4] 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 [35] suite using default parameters. Using the Green-Marl tool, the graphs were transformed to a binary format (to be used in the HONG-UFSCC tool).

B.2.3 Hardware dependencies

To replicate our experiments, a 64-core x86 machine with 64 bit addressing is required. All experiments in were performed on a machine with 4 AMD Opteron™ 6376 processors, each with 16 cores, forming a total of 64 cores and 128GB of main memory.

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

The experiments are specifically (hardcoded) set up to be performed with 64 cores. It is still possible to run the scripts with fewer cores available, although the results for benchmark runs with more threads than the available cores should be discounted. Benchmarking with more than 64 cores is not possible due to limitations of the tools.

B.2.4 Software dependencies

We recommend evaluating the artifact on POSIX / Linux. We performed all experiments on Ubuntu 14.04. To install LTSMIN and HONG-UFSCC and run the experiments, the following dependencies are required:

- `g++` (with builtin atomic functions and OpenMP support)
- `popt` (version $\geq$ 1.7, package `libpopt-dev`)
- `zlib` (package `zlibc zlib1g zlib1g-dev`)
- `Flex` (package `flex`)
- `Apache Ant` (package `ant`)
- `make`
- `automake`
- `autoconf`
- `libtool`
- `DiVinE` (patched, see App. B.3.1 for installation)
- `Python` 2.7.X with the modules `numpy` and `scipy` (packages `python-numpy` and `python-scipy`)
- A `Latex` installation with the `pgfplots` package installed (package `texlive-latex-extra`)

B.3 Installation

The current section describes how to install the LTSMIN tool in App. B.3.1 and how to install the HONG-UFSCC tool in App. B.3.2.

B.3.1 Installing LTSmin

Before installing LTSMIN, a patched version of DiVinE 2.4 is required. The source code for DiVinE is available at:

`https://github.com/utwente-fmt/divine2.git`

In the `divine2` directory, use the following commands to install DiVinE:

- `mkdir _build && cd _build`
- `cmake .. -DGUI=OFF -DRX_PATH= -DCMAKE_INSTALL_PREFIX=install -DMURPHI=OFF`
- `make`
- `make install`

(see `http://fmt.cs.utwente.nl/tools/ltsmin/#divine`) Here, `install` is a directory (included in the PATH) where to install DiVinE.

The source code for LTSMIN is available at:

`https://github.com/vbloemen/ltsmin/tree/vincent3`

In the `ltsmin` directory, use the following commands to install LTSMIN:

- `git submodule update --init`
- `./ltsminreconf`
- `./configure --prefix=install`
- `make && make install`

(see also `http://fmt.cs.utwente.nl/tools/ltsmin/`) Here, `install` is a directory (included in the PATH) where to install LTSMIN. To test whether LTSMIN has been installed correctly, in the `experiments/graphs` directory, use the command `dve2lts-mc --strategy=ufscc -s28 beem/adding.3.dve`. This runs the UFSCC algorithm on the `adding.3` graph. If successful, various performance statistics will be printed on the standard output (for instance `total scc count: 1894376`).

B.3.2 Installing Hong-UFSCC

The source code for the HONG-UFSCC tool is available at Github:

`https://github.com/vbloemen/hong-ufscc.git`

In the `hong-ufscc` directory, the code is compiled with the command: `make`. If successful, the current directory now contains an executable: `scc`. Calling `scc` with no parameters will print the help message, which describes the options. To make this executable usable for the experiments, please make sure that the directory containing the `scc` executable is included in the PATH.

B.4 Experiment workflow

The experiments can be executed with the provided bash script from the Github project (`bench.sh`).

The procedure to perform all experiments from the paper is as follows:

1. Install LTSMIN and HONG-UFSCC (see App. B.3).
2. Copy the offline input graphs from the zip file to the directories:
   - `experiments/graphs/real-offline`
   - `experiments/graphs/constructed-offline`
3. `cd experiments`
4. Run `./bench.sh` to perform all experiments,

The `bench.sh` script will iterate over all files in the subdirectories of `experiments/graphs` and perform a sequence of experiments for each of these graphs. For each experiment, the program is called (LTSMIN or HONG-UFSCC) with as arguments the number of processor instances, the algorithm to be used, and the input graph. The output of the program is stored in a temporary file and analyzed by `experiments/parse-output.py`. This Python script will then parse the output file and append the results in the corresponding CSV file in the `experiments/results` directory.

For the results in the paper, we ran all experiments 50 times. The `bench.sh` script is set up to run the experiments 10 times (which should be a sufficient number to obtain meaningful results).

Specific experiments: The `bench.sh` script was written so that it can be easily modified to perform an experiment on a single graph, using a algorithm, using a specific number of cores, or combinations of these aspects.

B.5 Evaluation and expected result

After performing the experiments, the CSV files in the directory `experiments/results` should contain performance information for each experiment (one row per experiment). We provide three Python scripts to translate the CSV data to Latex files, which can consecutively be compiled to PDF files. The `parse-csv.sh` script provides examples on how to use these Python scripts. We show how to use these scripts as follows:

- `python csv2graph.py INFILE MODEL`, this generates the Latex code to create a graph that depicts how the performance times for the algorithms change when increasing the number of cores, on the graph MODEL obtained from the CSV file INFILE.

- `python csv2graph.py INFILE MODEL ALG N` this generates the Latex code to create a graph that depicts the relative performance (speedup) of the algorithms compared to a baseline (algorithm ALG, using N cores), on the graph MODEL obtained from the CSV file INFILE.
- `python csv2table.py INFILE` this generates the Latex code to create a table that depicts the performance times for the algorithms (using 64 cores) and the relative performance of UFSCC compared to the others, on all graphs from the CSV file INFILE.
- `python csv2scatter.py INFILE ALG1 WORKERS1 ALG2 WORKERS2`, this generates the Latex code to create a scatterplot that compares the relative performance of algorithm ALG1 using WORKERS1 cores with ALG2 using WORKERS2 cores.

Expected results: The obtained results for all experiments should be similar to those that are provided in the directory `experiments/results-paper`. We expect similar *relative* performance of the algorithms, i.e. if we observe from the results `experiments/results-paper` that algorithm A is 5 times faster than algorithm B on graph X using Y cores, then this same 5-fold time-improvement should also be visible in the results from `experiments/results` (using the same configuration). We assume that the absolute performance times are too dependent on the specifics of the used hardware for valid comparison. The suggested method to make this comparison is by running `csv2table.py` and `csv2scatter.py` on each CSV file in the `experiments/results` and `experiments/results-paper` directories. We expect to observe that:

- The speedup graphs for every input graph should look similar to those from the `results-paper` directory.
- On `beem` graphs, UFSCC (using 64 cores) is typically 10 to 30 times faster compared Tarjan's algorithm
- On `beem` graphs, UFSCC (using 64 cores) is compared to Renault's algorithm (using 64 cores) at worst 0.7 times as fast and for half of the graph instances UFSCC is more than 10 times faster. We consider this result to be the main selling point for our algorithm.
- On the `constructed-offline` graphs that correspond the `beem-selected` ones, UFSCC is expected to perform slightly outperform Hong 's algorithm. and Tarjan's algorithm. Also, Tarjan's algorithm is expected to outperform UFSCC for most of the graphs.
- On the `real-offline` graphs, UFSCC outperforms Tarjan's algorithm, but Hong's algorithm outperforms UFSCC (except for `wiki-links`).
- We expect that some failed experiments, especially for experiments on `constructed-offline` graphs using Hong's algorithm. We have also observed a few failures for Renault's algorithm. We refer to the contents of the `experiments/failures` directory for the failed experiments.