University of Twente

Master Thesis

On-The-Fly Parallel Decomposition of Strongly Connected Components

Author:

Vincent Bloemen (s1004611)

Graduation Committee:

Prof. Dr. J.C. van de Pol
Dr. A.W. Laarman
Dr. S.C.C. Blom

May 27, 2015

Abstract

Several algorithms exist for decomposing *strongly connected components* (SCCs). To accommodate recent non-reversible trends in hardware, we focus on utilizing multi-core architectures. Specifically, we consider parallelizing SCC algorithms in the setting of an on-the-fly implementation (to be able to detect SCCs while constructing the graph - which is particularly useful for several verification techniques). We show that the current solutions are not capable of scaling efficiently and we propose a new algorithm that is able to do so.

Our parallel algorithm is (in contrast to the existing approaches) specifically designed to communicate partially discovered SCCs between workers. This is achieved by using a shared Union-Find structure. This structure has been extended to efficiently keep track of the search paths for each worker in combination with means to iterate and communicate fully explored vertices. We show that the designed algorithm is provably correct and performs in quasi-linear time. Experiments show that it outperforms existing techniques.

Contents

List of Algorithms

List of Figures

Chapter 1

Introduction

In Computer Science, a wide variety of problems can be represented with a (directed) graph structure. A graph describes abstract objects (*vertices* or *states*) and the relations that hold between these (*edges* or *transitions*). Consider for instance representing road systems, social networks, or practically any type of process. Graph algorithms are then applied for solving particular problems. One common technique is decomposing *strongly connected components* (SCCs); finding sets of vertices for which every vertex can reach each other vertex in the set. Figure 1.1 gives an example of a graph where the marked regions represent distinct SCCs. For instance, vertices b, c, and e are part of the same SCC because these can all reach each other. Applications for SCC decomposition include data- and (social) network analysis, as well as various verification techniques.

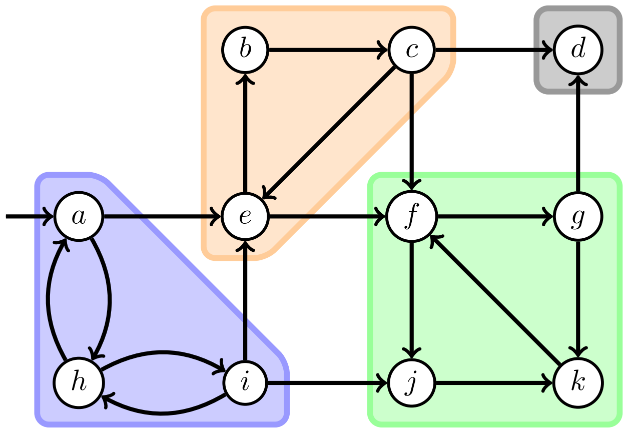


Figure 1.1: Example of a graph in which the marked regions represent the SCCs.

A graph can be *implicitly* represented. This means that instead of storing all edges in a data structure, a Next-State method is used to calculate the *successors* (reachable neighbor states, e.g. in Figure 1.1, the successors of f are g and j) from a given vertex. This representation is often preferred because it is more memory efficient (since the graph is not explicitly stored in memory) and because it may follow naturally from the problem context (consider for instance representing a chess game or the control flow of a program). However, this structure limits the freedom for a graph algorithm: it has no prior knowledge on the graph other than a given initial state and it can only expand this knowledge by requesting for successors of a known state. We refer to algorithms that are capable of performing on implicit graphs as *on-the-fly* (or *online*), other algorithms are referred to as *offline* (note that an on-the-fly algorithm is also offline).

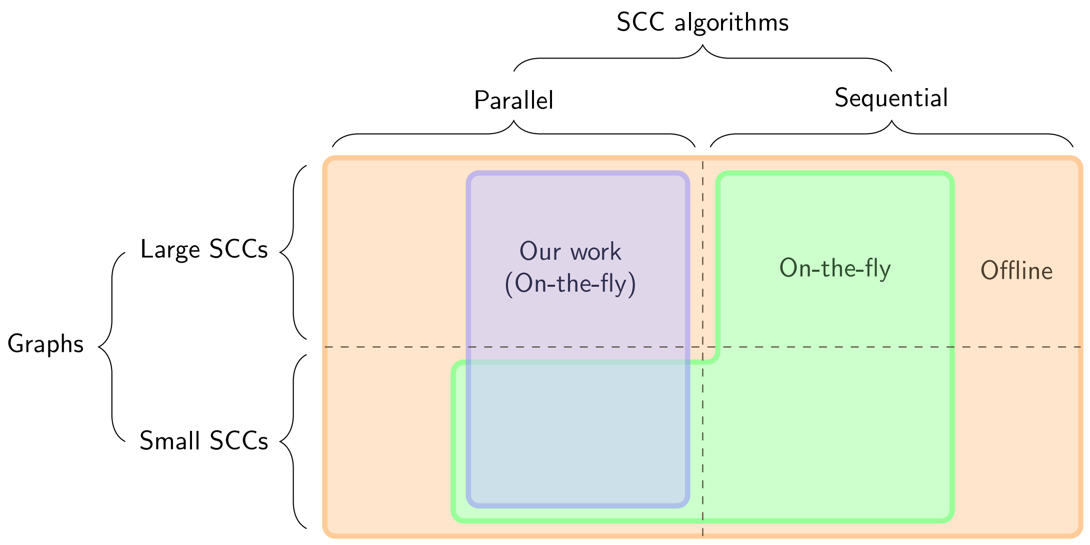


Figure 1.2: Visual representation of the topic for this project.

We focus on algorithms that decompose SCCs on-the-fly, these are particularly useful in several verification procedures. With the increasing scale of the models to be analyzed, a recurring problem is the *state-space explosion* [2]: the graph becomes too large for the system to handle. This is why many verification techniques are designed to be on-the-fly. A benefit is that an on-the-fly algorithm may finish as soon as an counter-example is found (without ever constructing the complete graph). On-the-fly SCC decomposition is applied in CTL and LTL verification [54], for instance when applying the EG operator or finding accepting cycles. It also finds applications in state-space reduction, for example in τ-compression [43].

With the increasing parallelization of processor architectures, there is a growing demand for concurrent algorithms. This report focuses on parallel, specifically multi-core, algorithms for the on-the-fly decomposition of SCCs. The *depth-first search* (DFS) algorithm from Tarjan [57] is well-known and regarded as the best sequential approach for finding SCCs with a linear time complexity. A number of concurrent algorithms have been designed [18, 44, 6, 46, 28, 56] that scale well on parallel architectures. However, these algorithms are not linear time (quadratic at best) and are designed offline as they require full knowledge of the graph.

The issue with multi-core on-the-fly algorithms is that the techniques are generally based on DFS traversal. Reif [51] showed that DFS is inherently sequential and P-complete, making efficient parallelization 'unlikely'. While this may be the case, several approaches exist [27, 16, 36, 15, 40, 53] that exhibit speedups on multi-core architectures compared to the best sequential methods. While parallel algorithms include distributed computing and GPU algorithms, we focus on multi-core implementations.

In this thesis we present a new technique for scalable parallel on-the-fly SCC decomposition. To the best of our knowledge, there are two existing approaches for this [53, 40]. While these approaches have been shown to scale, they both rely on holding a single worker responsible for discovering a complete SCC.[1] Therefore, if a graph contains a relatively large (to the number of vertices) SCC, the scalability is limited for these algorithms. With this in mind, we state the following research question:

Research question: *Is it possible to design a scalable, on-the-fly, concurrent SCC algorithm that efficiently communicates partially discovered SCCs?*

Figure 1.2 depicts the situation prior to this project and shows how our work extends on it. The marked

[1]Lowe's algorithm [40] actually does communicate intermediate results between workers, but this relies on an inefficient mechanism which causes a single worker to perform much extra work.

regions represent areas for which efficient algorithms are known. As can be seen, there are no on-the-fly algorithms efficiently capable of detecting large SCCs in parallel.

With the design of the algorithm, correctness is an important aspect. A parallel implementation introduces up to an exponential increase (with respect to the number of workers) in possible scenarios for an algorithm [34]. Therefore it is important to reason about its correctnes. This results in the following subquestion:

Subquestion 1: *Is the designed algorithm provably correct?*

Besides correctness, the algorithm's scalability needs to be examined. To do this, it is compared with existing techniques on several parameters, from the theoretical and empirical aspect. This provides us with information regarding the (relative) performance of the algorithm. This results in the following subquestion:

Subquestion 2: *In which cases does our algorithm outperform existing techniques?*

To define performance, the following aspects are considered as comparison measures:

- *Complexity and scalability*. A parallel algorithm should execute faster with an increasing number of workers. Moreover, we are interested in the speedups gained from doubling the number of processors. This is complemented by theoretical complexity analysis.
- *Memory usage*. A reason for using on-the-fly algorithms is to (attempt to) reduce the required memory. We therefore kept the memory usage in mind for the development of the algorithm.
- *Input structure*. The performance of an algorithm is influenced by the graph layout. Characteristics include the number of vertices and edges, the density of connectivity, the number of SCCs and the size of the SCCs (the largest SCC and average size) among others. The algorithms are therefore compared on a variety of graphs (both generated and existing). We mainly focus on graphs originating from the field of verification.

Contribution. We designed an on-the-fly multi-core SCC algorithm based on Union-Find techniques for contracting and communicating partially discovered SCCs. The design is built on the basis of global invariants in combination with an iteration mechanism for Union-Find sets. We were able to show that this technique is provably correct and is theoretically able to efficiently scale. An experimental study shows that this algorithm scales, and outperforms existing techniques in practice.[2] These results are of significant value since this work is the first successful approach to gain performance from communicating partially discovered SCCs by multiple workers in an on-the-fly fashion. Unlike related work, the proposed algorithm exhibits speedups for graphs containing large SCCs while it also performs on par with the state-of-the-art for graphs containing many small SCCs.

The report is structured as follows. The preliminaries are discussed in Chapter 2. The related work is described in Chapter 3. Chapter 4 presents a naive approach for the algorithm, an improved and final version is presented in Chapter 5. We show the results for the experiments in Chapter 6. Finally we provide the conclusions and directions for future work in Chapter 7.

[2]While the algorithm clearly outperforms existing techniques for a small number of workers, the communication overhead caused a performance drop when using more than eight workers (we expect to mitigate this problem in future work).

Chapter 2

Preliminaries

This chapter presents definitions for directed graphs and graph properties. Then we provide an overview of data structures. Finally, we show different graph traversal techniques.

2.1 Directed graphs

Definition 2.1 (*Directed graph*)**.** A directed graph $\mathcal{G}$ is a tuple $\langle \mathcal{V}, \mathcal{E} \rangle$, where $\mathcal{V}$ is a set of vertices (also referred to as nodes or states), and $\mathcal{E} \subseteq \mathcal{V} \times \mathcal{V}$ is a set of directed edges (or transitions). An edge between two vertices u and v will either be denoted as (u, v) or $u \rightarrow v$. Note that for a directed graph (as opposed to an undirected graph), $u \rightarrow v$ does not imply that $v \rightarrow u$.

Definition 2.2 (*Rooted graph*)**.** A *rooted graph* extends the directed graph structure with an initial state (the *root*). We have $\mathcal{G} = \langle \mathcal{V}, \mathcal{E}, v_0 \rangle$, where $\mathcal{V}$ and $\mathcal{E}$ are equally defined as in Definition 2.1. Here, $v_0 \in \mathcal{V}$ represents the *initial* state and denotes the starting point for graph traversal algorithms. An *unrooted* graph does not contain an explicit initial state. When we refer to a graph, we generally refer to a rooted graph unless explicitly stated as an unrooted graph. For the sake of clarity, we make the assumption that all vertices $|V|$ can be reached from v_0.

Definition 2.3 (*Transposed graph*)**.** A transposed graph $\mathcal{G}^T = \langle \mathcal{V}, \mathcal{E}^T \rangle$ is equivalent to the graph $\mathcal{G} = \langle \mathcal{V}, \mathcal{E} \rangle$ with all its edges reversed: $\mathcal{E}^T = \{(u, v) \mid (v, u) \in \mathcal{E}\}$.

Definition 2.4 (*Successor, predecessor*)**.** For $\mathcal{G} = \langle \mathcal{V}, \mathcal{E} \rangle$, if $(u, v) \in \mathcal{E}$, then v is called a *successor* of u and u is called the *predecessor* of v. We denote the set of all successors for a vertex u by $\textsc{post}(u) := \{v \mid (u, v) \in \mathcal{E}\}$. Similarly the set of all predecessors for a state u is denoted by $\textsc{pred}(u) := \{v' \mid (v', u) \in \mathcal{E}\}$. We denote two states $u, v \in \mathcal{V}$ as *neighbors* from each other if either $(u, v) \in \mathcal{E}$ or $(v, u) \in \mathcal{E}$ holds.

Definition 2.5 (*Path, cycle*)**.** Given $\mathcal{G} = \langle \mathcal{V}, \mathcal{E} \rangle$, a *path* is a sequence of vertices $s_0, \ldots, s_k$, s.t. $\forall_{0 \leq i \leq k} : s_i \in \mathcal{V}$ and $\forall_{0 \leq i < k} : (s_i, s_{i+1}) \in \mathcal{E}$. A *cycle* is a nonempty path in which the first and last vertex are the same.

Definition 2.6 (*Reachability*)**.** Given $\mathcal{G} = \langle \mathcal{V}, \mathcal{E} \rangle$ and $u, v \in \mathcal{V}$, we say that v is *reachable* from u (and u *reaches* v) iff a finite path exists from u to v. This path is denoted by $u \rightarrow^* v$. We define that every vertex is reachable by itself with a path of length 0.

Definition 2.7 (*Strong connectivity*)**.** Given $\mathcal{G} = \langle \mathcal{V}, \mathcal{E} \rangle$ and $u, v \in \mathcal{V}$, we say that v is *strongly connected* by u iff $v \rightarrow^* u \rightarrow^* v$; vertex u is reachable by v and v is reachable by u.

Definition 2.8 (*Strongly connected component*)**.** For $\mathcal{G} = \langle \mathcal{V}, \mathcal{E} \rangle$, a *strongly connected component* (SCC) is a maximal set of vertices $C \subseteq \mathcal{V}$ for which any two vertices $v, w \in C$ are strongly connected. An SCC is *maximal* in the sense that $\nexists C' : C \subsetneq C' \subseteq \mathcal{V}$, meaning that for every $t \in \mathcal{V} \backslash C$ and $v \in C : t \not\rightarrow^* v \vee v \not\rightarrow^* t$ (t cannot reach v and/or v cannot reach t).

An SCC consisting of a single vertex u is called *trivial* iff $(u, u) \notin \mathcal{E}$, other SSCs are called *non-trivial.*

Definition 2.9 (*Quotient graph*)**.** Let $\mathcal{V}_C$ be the set of all SCCs for graph $\mathcal{G} = \langle \mathcal{V}, \mathcal{E} \rangle$. The *quotient graph* of $\mathcal{G}$ is a directed graph $\mathcal{G}_C = \langle \mathcal{V}_C, \mathcal{E}_C \rangle$, where $\mathcal{E}_C = \{(C_1, C_2) \mid C_1, C_2 \in \mathcal{V}_C : C_1 \neq C_2 \wedge \exists u_1, u_2 \in \mathcal{V} : u_1 \in C_1 \wedge u_2 \in C_2 \wedge (u_1, u_2) \in \mathcal{E}\}$, i.e. there is an edge between SCCs C_1 and C_2 iff there is an edge between vertices from C_1 and C_2 in the original graph. Note that the quotient graph is acyclic.

Definition 2.10 (*Terminal SCC*)**.** A *terminal strongly connected component* is an SCC C for which all states in this SCC have no successors pointing to other SCCs: $\forall v \in C \wedge \forall w \in \textsc{post}(v) : w \in C$. Every graph must contain at least one terminal SCC, which we show by contradiction. Suppose that there is no terminal SCC is graph $\mathcal{G}$. In the quotient graph $\mathcal{G}_C = \langle \mathcal{V}_C, \mathcal{E}_C \rangle$ we have that every component $C \in \mathcal{V}_C$ contains a state that has a successor in some $C' \in \mathcal{V}_C \backslash C$. The only way of constructing such a quotient graph (with no terminal SCCs) is by creating a cycle for (a subset of) the components. This however contradicts with the definition of an SCC (and quotient graph), therefore a graph must contain at least one terminal SCC.

Definition 2.11 (*Fanout*)**.** Given $\mathcal{G} = \langle \mathcal{V}, \mathcal{E} \rangle$, the *fanout* for $\mathcal{G}$ is defined as the average number of successors for any vertex $v \in \mathcal{V}$. We have fanout $:= E[\ |\textsc{post}(v)|\ ] = \frac{|\mathcal{E}|}{|\mathcal{V}|}$.

Small-world phenomenon. The *small-world phenomenon* is a common property that holds in graphs, graphs that preserve this property are also referred to as *small-world* graphs [62]. Here, it may be the case that most vertices are not neighbors from each other, but it is very probable that a short path (compared to $|v|$) connects these vertices. As a side-effect, it is often the case that such graphs contain one large SCC and possibly many smaller sized components.

The small-world phenomenon is commonly found in Web graphs [9] and social networks [33], and it is also observed in models for formal verification [47]. The latter paper attempts to classify common graph layouts used by verification tools. For the purpose of SCC decomposition, a graph often (68% of the examined models from [47]) consists of one large SCC and many small components. Some state-of-the-art SCC algorithms were specifically designed with this observation in mind [56, 28, 40].

2.2 Parallelism

Parallelism [25] refers to performing multiple computations at the same time. This is achieved by for instance utilizing multiple processors (*workers*) from a multi-core architecture. It is important to understand that it is not trivial to translate sequential (or single-core) algorithms to use multiple processors. The programmer has to consider the behaviour of all threads at the same time, with all possible interleaving combinations (which grows exponentially).

Parallelism introduces *race-condition* errors. A race condition can occur when multiple workers operate on shared variables. As an example, assume we have a variable $x := 1$ and two workers increment x in parallel. First worker 1 could read $x = 1$, after which worker 2 reads $x = 1$. Then, worker 1 increments x and thus sets $x := 2$. Now, worker 2 still thinks that $x = 1$ and therefore also sets $x := 2$. We end up with an incorrect result due to a race condition. To prevent errors of this form, we could apply locking or use a lockless approach.

Locking. With locking, we ensure that only one worker can operate on a variable at a time (and therefore, operations are performed sequentially on this variable). To realize this, the variable is locked. A single instruction is used to set a lock and while this variable is locked, no other workers may read and/or write to the variable. When a worker has completed its operation, it releases the lock. Referring back to the example, if worker 1 locks x, worker 2 must wait until the lock has been released. Thus, worker 1 sets $x := 2$ and releases the lock. Now, worker 2 may lock x and it will subsequently correctly set $x := 3$.

Lockless. In some cases, it may be possible to perform operations lockless. This means that workers may simultaneously operate on the same variable by using atomic instructions. One of these is the *Compare&Swap* instruction. Here, by using its shorthand notation $\textsc{cas}(x, y, z)$ we only set $x := z$ if $x = y$ holds before the instruction. Furthermore, the instruction returns *True* if it was successful and *False* otherwise. In the example, we can utilize a lockless approach. Both workers could execute CAS$(x, x, x+1)$; increment x by one if it has not been changed. Internally, this may use multiple *local* instructions that do not affect the variable (read $x' := x$, store $x'' := x' + 1$). Assume that workers 1 and 2 both read $x = 1$ and worker 1 successfully applies CAS$(x, 1, 2)$. The *Compare&Swap* instruction for worker 2 will now fail (CAS$(x, 1, 2) \wedge x \neq 1$) and worker 2 has to try this instruction again (where it first reads $x = 2$) until it succeeds. A lockless approach is generally more efficient compared to one that uses locking [25].

We measure the performance gain for a parallel algorithm by analyzing the *speedup*. If the time used for a parallel algorithm with 8 workers is 4 times faster compared to a sequential version of the algorithm, we say that the speedup for 8 workers is 4.

2.3 Data structures

2.3.1 Graph data structures

There are several means for representing a graph $\mathcal{G} = \langle \mathcal{V}, \mathcal{E} \rangle$, each with advantages and disadvantages over others:

- *Adjacency Matrix*. An Adjacency Matrix AM is a 2D binary array of size $|\mathcal{V}| \times |\mathcal{V}|$. In this graph representation, $\forall_{0 \leq i,j < |\mathcal{V}|} : AM[i, j] = 1$ iff $(i, j) \in \mathcal{E}$ (the edge (i, j) exists in the graph). Otherwise, in case $AM[i, j] = 0$, we have that $(i, j) \notin \mathcal{E}$. This data structure uses $|\mathcal{V}|^2/8$ bytes of memory. An edge is updated and found in constant time, finding all successors takes $\mathcal{O}(|\mathcal{V}|)$ time.

- *Adjacency List*. An Adjacency List AL is an array of linked lists. The array is of size $|\mathcal{V}|$ and its indexes represent the *source* vertices for the edges. The linked list for each array entry represents the *destination* vertices for the source. An edge $(i, j) \in \mathcal{E}$ is represented by including list entry j in the list of $AL[i]$. This representation uses $8 \cdot |\mathcal{E}|$ bytes of memory (with a naïve implementation on a 32-bit computer). An edge is updated in constant time and finding an edge takes $\mathcal{O}(|\mathcal{V}|)$ time (on average it is bounded by the fanout: $\mathcal{O}\left(\frac{|\mathcal{E}|}{|\mathcal{V}|}\right)$), finding all successors also takes $\mathcal{O}(|\mathcal{V}|)$ time (similarly $\mathcal{O}\left(\frac{|\mathcal{E}|}{|\mathcal{V}|}\right)$ on average).

- *Implicit*. An implicit representation differs from from the previous two in the sense that edges are not explicitly stated. Here, we make use of a NEXT-STATE method that calculates the successors (or POST) for a given vertex. The advantage of this representation is that edges do not have to be stored in memory at all. However, it is not always feasible to represent edges by means of a NEXT-STATE method. Note that it may not be possible to calculate predecessors in this representation.

An graph is usually presented implicitly for *reactive systems*; here, the system (a graph in our case) reacts to external events (this reaction is implemented in the NEXT-STATE method).

An example for an implicitly stated graph (and a reactive system) is to represent a *Sokoban* puzzle. In this puzzle, the object for the player is to move boxes to specific locations. If we represent each state and edge explicitly, this would create a state space that grows exponentially based on the number of movable objects (considering that each box can be placed on any tile). For an implicit representation, the NEXT-STATE method only has to consider the four directions which the player can take in combination with a rule to detect if a box may be pushed.

2.3.2 Union-Find

A *Union-Find* data structure is used for keeping track of disjoint sets of objects. For this structure, there are three operations (as defined in [59]):

- MAKESET(x): Creates a new set containing the single object x.

- FIND(x): Return the 'representative' of the set in which object x is stored.
- UNION(x, y): Combine the two sets containing the elements x and y into a single set.

Sequential Union-Find. The UNION-FIND data structure is realized by representing sets by rooted trees. Each node x of the tree contains a pointer to its parent in the tree, and the root points to itself (and is called the *representative* of the set). This notion of *disjoint-set forests* was designed by Galler and Fischer [21]. The FIND(x) procedure returns the root of the tree, by recursively looking up the parent from x. A UNION(x, y) consists of finding the roots for both x and y and setting one root's parent to the other root. The trees can become linearly tall in the size of the set (in case of unfortunate parent updates). In terms of amortized complexity, a UNION-FIND algorithm's running time is expressed by creating n sets and by combining these to a single set (which uses m FIND operations). The time complexity for Galler and Fischer's algorithm [21] is then given as $\mathcal{O}(n + n \cdot m)$.

Improvements. Using the naïve algorithm as a basis, several improvements have been made (as discussed by Tarjan and van Leeuwen [59]). One improvement is with the notion of *weighted-union*. This is a method for reducing the height of the trees by making the root of the smaller tree point to the root of the larger one (in a UNION operation). This is realized by one of the following means:

- *Weighing by size* [21]: By using a field $size(x)$ to keep track of the size of the tree rooted at x. The tree with the smallest size is then found by comparing the *size* fields. A UNION(x, y) combines the *size* of the two roots from x and y.
- *Weighing by rank* [29]: By using a field $rank(x)$ to keep track of the height of the tree rooted at x. For a UNION(x, y) call, this rank is incremented by one in case the ranks of x and y are equal, otherwise the rank (for the taller tree) remains the same and is the new root.

Both means achieve the same effect of reducing the tree's height. The latter approach is preferred since it can be implemented using less space and requires fewer updates [59]. In complement to the *weighted-union*, the following techniques reduce the heights of the trees during the FIND operation:

- *Path compression* [30]: During the search for the root, the intermediately found nodes are updated to point directly to the root.
- *Path splitting* [61]: This technique updates the parent from every intermediate node to its grandparent, during a FIND search.
- *Path halving* [61]: Similar to splitting, with the adaption that the parent is updated for *every other node*.

When combining one of these techniques with *weighing by rank*, the rank may have a larger value than the actual tree height as a result of the modifications.

Hopcroft and Ullman's algorithm. Hopcroft and Ullman [29] combines the two improvements for the disjoint-set forests by applying *weighing by rank* and *path compression*. This improved UNION-FIND algorithm from Hopcroft and Ullman, can be found in Algorithm 1. We can observe that the rank always remains smaller than $\log_2(n)$, for n nodes. Thorough analysis [58, 59] of the algorithm (this also holds for any combination of the two improvements) results in a worst-case amortized (*quasi-linear*) running time of $\mathcal{O}(m \cdot \alpha(m, n))$, for $n - 1$ UNION and m FIND operations. Here, $\alpha(m, n)$ is the inverse Ackermann function, which is an extremely slow growing function and is generally regarded as a small constant for practical applications. This complexity is shown to be asymptotically optimal for UNION-FIND [59].

Algorithm 1 UNION-FIND structure [29]

```
1:  procedure MakeSet($x$)
2:      $x.parent := x$
3:      $x.rank := 0$

4:  procedure Find($x$)
5:      if $x.parent \neq x$ then
6:          $x.parent :=$ Find($x.parent$)
7:      return $x.parent$

8:  procedure Union($x, y$)
9:      $x_r :=$ Find($x$)
10:     $y_r :=$ Find($y$)
11:     if $x_r = y_r$ then return
12:     if $x_r.rank < y_r.rank$ then
13:         $x_r.parent := y_r$
14:     else if $x_r.rank > y_r.rank$ then
15:         $y_r.parent := x_r$
16:     else
17:         $y_r.parent := x_r$
18:         $x_r.rank := x_r.rank + 1$
```

Parallel Union-Find. Anderson and Woll [1] introduce an efficient data structure for UNION-FIND on a shared memory multiprocessor. This algorithm is *lockless* and thus uses atomic instructions to update the parent and rank. We present the basic observations concerning this structure.

For the FIND operation, *path halving* is applied. This heuristic is implemented using a *Compare&Swap* primitive. The UNION operation is implemented by *weighing by rank*. While identifying the roots of the the objects, the *rank* can only be updated by the first thread that updates the root. Furthermore, a consistent method for checking ranks is used; by comparing the node identifiers in case the *rank* is the same.

Besides the FIND and UNION operations, it also introduces the SAMESET operation which tests if two objects are contained in the same subset. This operation is important in the concurrency, because subsequent FIND calls may cause synchronization issues. The SAMESET operation is implemented by using two FIND operations for both elements to find their roots. Due to the concurrency it might be possible that a set has been updated. If this is the case (by checking the parent of the first root), the SAMESET operation is restarted. As a result, the algorithm can answer an amortized sequence of n Union-Find queries with $\mathcal{O}(n \cdot \mathcal{P})$ work [1].

2.4 Graph traversal

Depth-first search. *Depth-first search* (DFS) [11] is a graph traversal algorithm in which the algorithm starts *exploring* the graph from a given root. The algorithm continuously traverses to the *depth-most* unvisited vertices until this is no longer possible. At this point, the algorithm *backtracks* to a vertex that still has unvisited successors and continues traversing from there. This process is repeated until every reachable vertex from the root has been visited. We say that the root is now fully *explored*. Algorithm 2 depicts a standard version of a DFS algorithm.

During the search, the visited vertices can be ordered in several ways (by for instance using a stack S). The two most common methods are as follows:

- *pre-order*: This approach orders the vertices in the same way they are visited. So the root is on the bottom of the stack and the last explored vertex is on the top of the stack. In the algorithm, this means that the line S.PUSH(v) is inserted after line 3. See Figure 2.1 for an example of pre-order traversal.

- *post-order*: This approach orders the vertices in order they are explored. So the vertex at the first backtrack point is on the bottom of the stack and the root is on the top of the stack. In the algorithm, this means that the line S.PUSH(v) is inserted after line 7. See Figure 2.1 for an example of post-order traversal.

Algorithm 2 Depth-first search (recursive)

```
1: $\forall v \in \mathcal{V} : v.visited := v.explored := False$
2: procedure DFS($v$)
3:     $v.visited := True$
4:     for each $w \in$ POST($v$) do
5:         if $\neg w.visited$ then
6:             DFS($w$)
7:     $v.explored := true$
```

With a *back-edge* we say that we visit a vertex that is already part of the current search path. Concretely, Assume we have discovered the path $v_0, \ldots, v_i, \ldots, v_k$ and we encounter the edge $v_k \to v_i$. Then, this edge forms the cycle: $v_k \to v_i \to^* v_k$ and we refer to this edge as a back-edge.

Reif showed [51] that the lexicographical computation of depth-first search post-ordering of vertices is P-complete. Therefore it is claimed to be difficult to parallelize algorithms that are based on depth-first search. With the assumption that $NC \neq P$, no DFS-based algorithm can run in poly-logarithmic time with a polynomial number of processors.

Breadth-first search. *Breadth-first search* (BFS) [11] is a graph traversal algorithm in which the algorithm starts *exploring* the graph from a given root. BFS makes use of a *first-in-first-out* (FIFO) queue to store the successors and select which one to traverse next. This process continues until the queue is empty. Algorithm 3 depicts a standard version of a BFS algorithm. We refer to Figure 2.1 for an example representation of the vertex ordering. Note that in contrast to DFS, BFS is parallelizable.

Algorithm 3 Breadth-first search

```
1: $\forall v \in \mathcal{V} : v.visited := False$
2: procedure BFS($v$)
3:     $Q := \emptyset$
4:     $Q$.ENQUEUE($v$)
5:     $v.visited := True$
6:     while $Q \neq \emptyset$ do
7:         $w := Q$.DEQUEUE($v$)
8:         for each $u \in$ POST($w$) do
9:             if $\neg u.visited$ then
10:                $u.visited := True$
11:                $Q$.ENQUEUE($u$)
```

On-the-fly. According to the literature [54, 4, 16], an *on-the-fly* algorithm, is defined to: (1) have no previous knowledge about the graph other than a given root, (2) traverse a graph (and gain information about it) starting from this root, and (3) strictly make use of the POST or NEXT-STATE method to find edges of the graph. Note that it is not possible to directly find the set of predecessors for a vertex in such a graph. An on-the-fly algorithm is used on implicitly given graphs. We refer to an algorithm that does not demand these restrictions as *offline* (an on-the-fly algorithm is also offline).

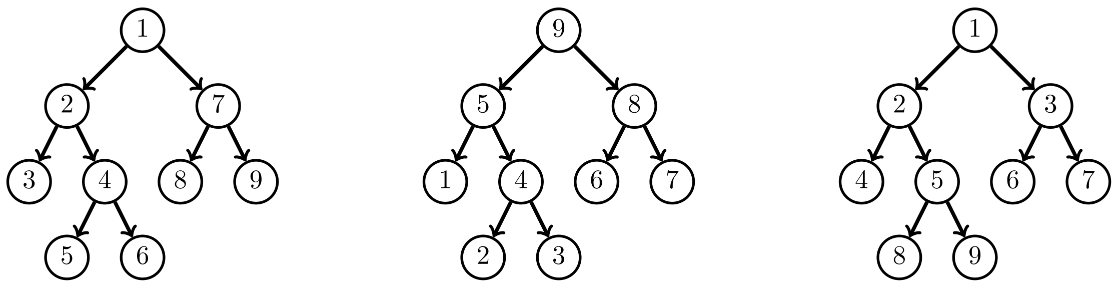


Figure 2.1: Graph traversal vertex ordering for pre-order DFS (left), post-order DFS (middle) and BFS (right).

2.5 Explicit-State LTL model checking

Model checking [10, 2] refers to the problem of determining whether a given system meets its specification. We consider an automata-theoretic approach, where the system (expressed as a graph) has finitely many states and the specification is expressed as a *Linear temporal logic* (LTL) formula. The task is to check for language containment, i.e. checking the language described by the LTL formula is contained in the system's language. However, this is an expensive procedure. The problem is therefore translated to language emptiness: The LTL formula is negated and translated to a Büchi automaton. This automaton is then synchronized with the system's state space. Finally, the combined Büchi automaton is checked for emptiness to verify if the system has met its specification.

Definition 2.12 (*Büchi automaton*)**.** A *Büchi automaton* is a directed (and rooted) graph with a number of additional properties. The automaton is given as a tuple $B = \langle Q, \Sigma, \delta, s_0, \mathcal{A}\rangle$. Here Q is a finite set of states (equivalent to vertices in a directed graph), Σ is the *alphabet* of the automaton; representing the actions that the system can take. The transitions (or edges) are given in δ, for which an edge has the form $(s, a, t) \in \delta$, with $s, t \in Q$ and $a \in \Sigma$. This implies that actions are taken at the traversal of edges. The initial state (or *root* is given by s_0 and $\mathcal{A} \subseteq Q$ represents the set of *accepting* states.

Checking for Büchi emptiness, can be solved by means of an *accepting cycle detection*. Here, we traverse the Büchi state space (rooted graph) to search for a path which contains an accepting state that lies on a cycle. If we succeed in finding an accepting cycle, we have found a counter-example.

We will not go in-depth on how the various components for LTL model checking. The key aspect related the topic of this thesis is the process of finding an accepting cycle. This can be achieved by searching for accepting states in the graph and, when found, performing another search to see if this state is part of a cycle. This approach is referred to as *Nested depth-first search* [12] (more on this in Section 3.3.1). Another approach is to decompose the graph in SCCs, if an accepting state is part of a non-trivial SCC (one that contains cycles), we have detected an accepting cycle. The main difference between these two techniques is that an NDFS algorithm is more memory efficient, while an SCC-based technique generally finds a counter-example faster [54, 20].

Chapter 3

Related Work

This chapter provides an overview of existing algorithms related to the subject of finding SCCs. A summary with a discussion is presented in Section 3.4

3.1 Sequential DFS-based algorithms

3.1.1 Tarjan's algorithm

Tarjan's algorithm [57] is perhaps the most well-known and arguably most efficient approach for finding SCCs sequentially. It performs a single depth-first search through the graph, in which each visited node is provided with two variables. The first variable is the *index*, this is a sequence counter that corresponds to the order in which the nodes are visited (the n^{th} node visited has $index = n$). The second variable is the *lowlink*, this variable represents the smallest index reachable from the current node. Each time a visited node is encountered, the *lowlink* is updated. Algorithm 4 depicts the standard implementation of TARJAN's algorithm. A stack S is used to keep track of the visited nodes.

Algorithm 4 Tarjan's algorithm [57]

```
1:  $\forall v \in \mathcal{V} : v.index := v.lowlink := 0$
2:  $counter := 0$
3:  $S := \emptyset$
4:  procedure TARJAN($v$)
5:      $counter := counter + 1$
6:      $v.lowlink := v.index := counter$
7:      $S$.PUSH($v$)
8:      for each $w \in$ POST($v$) do
9:          if $w.index = 0$ then [unvisited state]
10:             TARJAN($w$)
11:             $v.lowlink :=$ MIN($v.lowlink, w.lowlink$)
12:         else if $w \in S$ then [back-edge]
13:             $v.lowlink :=$ MIN($v.lowlink, w.index$)
14:     if $v.lowlink = v.index$ then [remove completed SCC]
15:         $w := S$.POP()
16:         while $w.index \geq v.index$ do
17:             $w := S$.POP()
```

Note that an SCC is decomposed at lines 14-17 since all members of the same SCC as v reside on top of v in the stack. Note also that this algorithm only finds the SCCs reachable by the initial node v, which is sufficient for on-the-fly decomposition. The original algorithm iteratively calls the TARJAN procedure for each node that remained unvisited (i.e. with $index = 0$). Because the algorithm only traverses the graph by using the POST call, it is well-suited for on-the-fly applications.

As noted by Schwoon and Esparza [54], several properties could be observed from the algorithm. The stack only contains the states from the current search path. Suppose that $w \in S$ lies on this path and the search finds an edge from the currently examined node, v, to w. Then we can conclude that a path exists from w to the *root* of the SCC (the vertex with the lowest *index* value). Also, a path from said root to v exists, so both v and w lie on the same SCC. Another observation is that the root r of an SCC is the first state for that SCC to be added to the stack. When r is fully explored, we can conclude that all SCCs reachable from r have been completely explored and removed from the stack.

Given a graph $\mathcal{G} = \langle \mathcal{V}, \mathcal{E} \rangle$, the time complexity for TARJAN's algorithm is $\mathcal{O}(|\mathcal{V}| + |\mathcal{E}|)$, and it uses $\mathcal{O}(|\mathcal{V}|)$ space. The time complexity is asymptotically optimal since any SCC algorithm must examine every vertex and edge (given a worst-case graph). In more detail, an SCC is examined after backtracking to the first visited node, w, from the SCC. This means that all reachable nodes from w are fully explored, while not necessarily all reachable nodes from the initial vertex are discovered yet.

For the purpose of finding accepting cycles, this algorithm has a significant drawback. It may be the case that an accepting cycle is quickly reached from the initial state. TARJAN's algorithm will detect this cycle *after* it finishes exploring all reachable states from the states on this cycle.

The Geldenhuys-Valmari algorithm Geldenhuys and Valmari [22] modified TARJAN's algorithm for the purpose of LTL model checking. The idea of the algorithm is that the last found accepting state of the current search path, is kept track of. Whenever a *back-edge* is found, which points to a previously visited state from the current search path (which updates the *lowlink* value), the algorithm terminates with an accepting cycle.

3.1.2 Dijkstra's algorithm

Dijkstra [14] proposed a different variation to TARJAN's algorithm. This algorithm is presented in Algorithm 5. Instead of keeping track of *lowlink* values, this algorithm maintains a stack of possible *root* candidates. On finding a back-edge (line 12), the algorithm pops vertices from the stack until the 'root' of the cycle is found. At backtracking, the *current* flag of reachable states are set to false so that these states do not interfere with a future search. This algorithm also runs in linear time, $\mathcal{O}(|\mathcal{V}| + |\mathcal{E}|)$ with $O(|\mathcal{V}|)$.

Couvreur's algorithm Couvreur [13] designed a variant on DIJKSTRA's algorithm for the purpose of finding accepting cycles. The main difference with a TARJAN-based accepting cycle algorithm is that here, information on partial SCCs is propagated when finding a back-edge. A comparative study reported [52] that these two approaches are comparable, though the Dijkstra-based algorithm reports counter-examples earlier.

Algorithm 5 Dijkstra's algorithm [14]

```
 1: $\forall v \in \mathcal{V} : v.index := 0, v.current := False$
 2: $Roots := \emptyset$
 3: $counter := 0$
 4: procedure DIJKSTRA($v$)
 5:     $counter := counter + 1$
 6:     $v.index := counter$
 7:     $Roots$.PUSH($v$)
 8:     $v.current := True$
 9:     for each $w \in$ POST($v$) do
10:         if $w.index = 0$ then [unvisited state]
11:             DIJKSTRA($w$)
12:         else if $w.current$ then [back-edge]
13:             $u := Roots$.TOP()
14:             while $u.index > w.index$ do
15:                 [Couvreur's variant: if $u \in \mathcal{A}$ then report cycle]
16:                 $u := Roots$.POP()
17:     if $Roots$.TOP() $= v$ then [remove completed SCC]
18:         $Roots$.POP()
```

3.1.3 Kosaraju-Sharir algorithm

Kosaraju and Sharir [55] developed an SCC algorithm by performing two depth-first searches through the graph. The algorithm, as shown in Algorithm 6, first performs a depth-first search to obtain the stack S of all nodes (in post-order). Then, until S is empty and using the transposed edges of the graph (or similarly by using PRED instead of POST) the SCC components are found. Note that this last procedure could also be done in a breadth-first search manner.

Algorithm 6 Kosaraju-Sharir algorithm [55]

```
 1: $\forall v \in \mathcal{V} : v.visited := False$
 2: $S := \emptyset$
 3: procedure KOSARAJU-SHARIR($G$)
 4:     for each $v \in \mathcal{V}$ do
 5:         if $v \notin S$ then
 6:             DFS-POST-ORDER($v$)
 7:     while $S \neq \emptyset$ do
 8:         $v := S$.POP()
 9:         if $\neg v.visited$ then DFS-REVERSE($v$)

10: procedure DFS-POST-ORDER($v$)
11:     $v.visited = True$
12:     for each $w \in$ POST($v$) do
13:         if $\neg w.visited$ then DFS-POST-ORDER($w$)
14:     $S$.PUSH($v$)

15: procedure DFS-REVERSE($v$)
16:     $v.visited = False$ [re-using the visited flag]
17:     for each $w \in$ PRED($v$) do
18:         if $w.visited$ then DFS-REVERSE($w$)
```

Even though the Kosaraju-Sharir algorithm also runs in linear time, $\mathcal{O}(|\mathcal{V}| + |\mathcal{E}|)$ with $O(|\mathcal{V}|)$ space, and it is often regarded as easier to understand, TARJAN's algorithm is often preferred as it traverses the graph only once. Besides this, the algorithm requires the ability to use the PRED call, i.e. any means to transpose edges in a graph. As this requirement is not coherent with the on-the-fly definition, it will not be possible to use this algorithm for on-the-fly SCC decomposition.

3.1.4 Set-based algorithms

Purdom's algorithm In 1970, Purdom [50] proposed an algorithm for computing the transitive closure of a graph (finding all reachable vertices from each vertex). The algorithm searches for SCCs (which is referred to as path equivalence) in the graph and replaces these by single nodes. This replacement procedure continues until the graph is acyclic. Afterwards, the transitive closure is calculated. The outline for the technique of finding SCCs is described in Algorithm 7. For a given vertex, the algorithm applies a DFS by keeping track of a stack with visited vertices. In case a vertex v is found that is already on the stack, a cycle has been found (that consists of all vertices from v to the top of the stack). All vertices from the top of the stack are removed until v is on top of the stack. With the removal of vertices, all incoming and outgoing edges are appended to the successors and predecessors of v. Moreover, the removed vertices are stored in a list for the *equivalence class* (which we represented with SET in the algorithm). As a result, the SCCs are computed and stored. Purdom's algorithm runs in $\mathcal{O}(|\mathcal{V}|^2)$ time.

Algorithm 7 Purdom's algorithm [50] for finding SCCs (based on descriptions from [19] and [50])

```
$\forall v \in \mathcal{V} : \textsc{set}(v) := \{v\}, v.visited := False$
$S := \{v_0\}$
procedure PURDOM($v$)
    $v.visited := True$
    for each $w \in \textsc{post}(v)$ do
        if $\neg w.visited$ then $S.\textsc{push}(w)$ [unvisited state]
        else [back-edge]
            while $S.\textsc{top}() \neq w$ do
                $t := S.\textsc{pop}()$
                $\textsc{set}(w) := \textsc{set}(w) \cup \textsc{set}(t)$
                $\textsc{post}(w) := \textsc{post}(w) \cup \textsc{post}(t)$ [merge successors]
        PURDOM($w$)
    if $S.\textsc{top}() = v$ then $S.\textsc{pop}()$
```

Munro's algorithm Munro [45] optimized Purdom's work in 1971 by using a more efficient data structure for merging the vertices. Instead of using a adjacency matrix (which was used by Purdom) for representing the edges, Munro notes the use of adjacency lists. This structure is combined with a similar algorithm as Purdom's. However, the modification of edges is done more efficiently by appending the successor lists to the '*supernode*' of the SCC. This reduced the complexity for a set-based SCC algorithm to $\mathcal{O}(|\mathcal{E}| + |\mathcal{V}| \log |\mathcal{V}|)$.

Gabow's algorithm The algorithms proposed by Purdom [50] and Munro [45] form the basis for the set-based algorithm. The general notion is to perform a DFS to traverse the graph, and by contracting cycles to single vertices when a back-edge is found (an edge to a previously visited vertex). The main point of interest is to construct an efficient manner for contracting cycles. Munro's technique applies the general notion of disjoint-set merging. Hopcroft and Ullman [29] designed an efficient UNION-FIND algorithm, for which merge (or UNION) operations could improve the algorithm to perform almost linearly (more on the underlying technique can be found in Section 2.3.2). GABOW's algorithm [19] is presented in Algorithm 8. An important difference between this algorithm compared to the ones from Purdom and Munro is that vertices are not strictly contracted anymore. All visited vertices are stored in the UNION-FIND structure, thus by checking existence in said structure vertices are prevented from being visited again (lines 7-8). This omits the requirement of removing edges or vertices from the graph. By using an extra DEAD state, the algorithm is able to distinguish partial SCCs from fully explored ones (which are referred to as DEAD SCCs).

Algorithm 8 GABOW's algorithm [19] (as presented in [52])

```
1: $\forall v \in \mathcal{V} : \text{UF}[v] := \text{NULL}$
2: $S := \emptyset$
3: procedure GABOW($v$)
4:     MAKESET($v$)
5:     $S$.PUSH($v$)
6:     for each $w \in \text{POST}(v)$ do
7:         $w' :=$ FIND($w$)
8:         if $w' =$ null then [unvisited state]
9:             GABOW($w$)
10:        else if $\neg$SAMESET($w'$, DEAD) then [back-edge]
11:            while $S$.TOP() $\neq w'$ do
12:                UNION($S$.POP(), $w'$)
13:    if $S$.TOP() $= v$ then [remove completed SCC]
14:        UNION($v$, DEAD)
15:        $S$.POP()
```

Note that this algorithm requires only one search through the graph, due to the underlying DFS nature using stack S. Since a vertex can be removed only once, the number of UNION calls (which merges vertices) is limited by the number of vertices. Also, the number of times the FIND operation is applied at line 7 is at most the number of edges in the graph. Because of this, the total run time for the algorithm, when applying an efficient UNION-FIND algorithm [29] (and considering its *amortized* complexity), is 'almost' linear time (the time used for an operation on the UNION-FIND structure is bounded by inverse Ackermann function). For all practical purposes, we assume that the complexity is $\mathcal{O}(|\mathcal{E}| + |\mathcal{V}|)$.

3.2 Parallel fixed-point algorithms

3.2.1 Forward-Backward algorithm

The first (and regarded as most basic) parallel algorithm for finding SCCs was discovered by Fleischer et al. [18]. This algorithm is known as either the *divide-and-conquer strong components* (DCSC) or *forward-backward* (FB) algorithm. The algorithm, as shown in Algorithm 9, starts off by selecting a pivot vertice from the graph. It will then compute the set of vertices that are reachable from the pivot (the forward slice, as denoted by FWD) and the set of vertices that can reach the pivot (the backward slice, or BWD). The intersection of the two slices form an SCC and the three remaining subsets of vertices are considered in future iterations (see lines 8-10). Because these subsets are strictly disjoint, they can be treated in parallel. The complexity for the FB algorithm is $\mathcal{O}(|\mathcal{V}| \cdot (|\mathcal{V}| + |\mathcal{E}|))$, while its expected running time is $\mathcal{O}(|\mathcal{E}| \cdot \log(|\mathcal{V}|))$ [18].

OWCTY algorithm A *leading trivial component* (LT) is a trivial component (an SCC consisting of a single vertex, with no self-loop) that has no incoming edges. Similarly, a *terminal trivial component* (TT) is a trivial component with no outgoing edges. A technique called *One-Way-Catch-Them-Young* (OWCTY) [17] is designed to remove such components from the graph. On the removal of these components, new LTs or TTs may arise. Therefore, the same method is applied recursively (until no trivial components can be removed anymore).

The combination of FB and OWCTY showed an overall improvement [44, 8, 32]. The idea of this variation is to apply OWCTY before the pivot selection, to remove a number of trivial components. Due to the recursive calls of the underlying FB algorithm, even more trivial components get removed. The complexity for this combination remains the same compared to FB.

Algorithm 9 Forward-Backward algorithm [18]

```
procedure FB($\mathcal{V}$)
    if $\mathcal{V} \neq \emptyset$ then
        $p := $ Pivot($\mathcal{V}$)
        $F := $ FWD($p, \mathcal{V}$)
        $B := $ BWD($p, \mathcal{V}$)
        [$F \cap B$ is an SCC]
        do in parallel
            FB($F \setminus B$)
            FB($B \setminus F$)
            FB($\mathcal{V} \setminus (F \cup B)$)
```

3.2.2 OBF algorithm

The *OWCTY-BWD-FWD* (OBF) algorithm [7, 5, 6] is based on the technique of subdividing the graph in a number of independent sub-graphs. As shown in Algorithm 10, it identifies and treats slices as follows:

O Remove leading trivial components (with OWCTY, line 4).

B Compute the backward slice from the vertices reached in the *O-step*, this defines a slice B (see line 6).

F The FB algorithm is applied on slice B in parallel. The successors of B are used as 'seeds' for the next iteration (see line 9).

Note that while the algorithm starts from an initial vertex, it is not considered to be *on-the-fly* since determining the backward slice (BWD) is not possible in an on-the-fly algorithm. The algorithm has been improved by also starting parallel procedures within the found chunks [5, 6]. The time complexity for this algorithm is $\mathcal{O}(|\mathcal{V}| \cdot (|\mathcal{V}| + |\mathcal{E}|))$; the same as for FB.

Algorithm 10 OBF algorithm [6]

```
1: procedure OBF($\mathcal{V}, v_0$)
2:     $Seeds := \{v_0\}$
3:     while $\mathcal{V} \neq \emptyset$ do
4:         $Eliminated, Reached := $ OWCTY($Seeds, \mathcal{V}$)
5:         $\mathcal{V} := \mathcal{V} \setminus Eliminated$
6:         $B := $ BWD($Reached, \mathcal{V}$)
7:         do in parallel
8:             FB($B$)
9:         $Seeds := $ FWD($B, \mathcal{V}$)
10:        $\mathcal{V} := \mathcal{V} \setminus B$
```

3.2.3 Other fixed-point algorithms

Colouring/Heads-off (CH) algorithm The *Colouring/Heads-off* (CH) algorithm [46] starts by assigning a unique colour (number) to each vertex. For each vertex, the colours are propagated to successors with a lower colour. This process continues until every successor has either the same, or a higher colour. Because every vertex of an SCC must have the same colour, edges between two different colours can be removed. The resulting disconnected subgraphs can be identified in parallel. In each subgraph the vertex which kept its initial colour is chosen as a pivot. The backward slice from this pivot then identifies an SCC. This SCC is removed and the algorithm is recursively applied on the remaining subgraphs. The time complexity for this algorithm is $\mathcal{O}(|\mathcal{V}| \cdot |\mathcal{E}|)$

We observed that a similar algorithm has been recent designed [41] for the Pregel [42] system. In this system, vertices are distributed over the workers and information is transfered via message-passing between vertices.

Hong's algorithm Hong et al. [28] adapted the FB algorithm due to its limited performance on *small-world* graph [62] instances. The algorithm uses a parallel breadth-first search (BFS) to find the forward and backward slice from the initial vertex. In small-world graphs, this means that this method will likely find a large SCC. This SCC is then removed from the graph and all subgraphs are identified by applying a weakly-connected component (WCC) algorithm. These subgraphs are then tested for one- and two-sized SCC components. The remaining components are decomposed using the standard FB algorithm. Its time complexity remains the same as for FB.

Multistep algorithm The Multistep algorithm [56] is based on observations from previous algorithms, aiming to combine the advantages and to minimize the drawbacks. It starts with a trimming procedure; one iteration of OWCTY. Then, it aims to find a large SCC by applying the parallel FB algorithm on the vertex with the most incoming and outgoing edges (similar to Hong's algorithm [28]). The found SCC is then removed and the CH algorithm is applied on the remaining sub-graphs. The algorithm uses Tarjan's algorithm for computing the remaining SCCs. Experimental evaluations (in particular on small-world graphs) suggest that this algorithm is arguably the best performing algorithm on a multi-core system. The time complexity is bounded to those of the used algorithms, therefore a quadratic worst-case complexity prevails.

GPU algorithms Algorithms for GPUs and many-core architectures are designed specifically with parallelization in mind. Efficient implementations [3, 38, 63] make use of the FB algorithm (while OBF and CH are also considered in [3]). In the forwards- and backwards search phases, GPU algorithms make use of parallel BFS to efficiently distribute the work. For these implementations, techniques designed specifically for GPUs should be adopted as the architecture is significantly different [38].

3.3 Parallel DFS-based algorithms

We observed in Section 2.4 that depth-first search has been proven to not scale well on multiple processors. However, by spawning multiple instances of a depth-first search, even if they do not share information, DFS-based algorithms could still benefit from parallelization [27, 15, 40].

3.3.1 Nested depth-first search

Nested depth-first search (NDFS) is an on-the-fly model checking algorithm for the purpose of finding accepting cycles [12]. It starts with a DFS to find accepting states. If an accepting state is found, a second, *nested*, DFS is started to find a cycle that includes the accepting state. The NDFS algorithm can be found in Algorithm 11. Here, DFSBLUE searches for accepting states (during backtracking) and the DFSRED procedure tries to find a cycle. Note that it is sufficient for DFSRED to find a vertex with a *cyan* colour (line 7), since every such state can reach the accepting state.

Note that the linear time complexity of the algorithm depends on the DFS property. It is important that the DFSBLUE sorts accepting states in DFS *post-order*. A nested search should not need to revisit states visited by a previous nested search because of this property (hence the check for $\neg w.red$ in line 8).

The parallelization of NDFS origins in SWARM verification [27]. This technique consists of running multiple, unsynchronized, instances of NDFS simultaneously. Here, successor vertices are chosen randomly to increase the likelihood of finding an accepting cycle quickly. Note that, because there is no information

Algorithm 11 Nested depth-first search (NDFS) [12], as presented in [15]

```
1:  $\forall v \in \mathcal{V} : v.cyan := v.blue := v.red := False$
2:  procedure NDFS()
3:      dfsBlue($v_0$)

4:  procedure dfsRed($v$)
5:      $v.red := True$
6:      for each $w \in$ post($v$) do
7:          if $w.cyan$ then report cycle
8:          else if $\neg w.red$ then dfsRed($w$)

9:  procedure dfsBlue($v$)
10:     $v.cyan := True$
11:     for each $w \in$ post($v$) do
12:         if $\neg w.blue$ then
13:             dfsBlue($w$)
14:     if $v \in \mathcal{A}$ then dfsRed($v$)
15:     $v.cyan := False$
16:     $v.blue := True$
```

shared between threads, this technique fails to improve NDFS in case the graph contains no accepting cycle (every thread will then explore the complete graph).

Two techniques were proposed to combine SWARM verification with some means of synchronization. First, the LNDFS algorithm [36] updates the colouring of *red* vertices globally (so each thread gains this information), the other colours are local for each worker. Unlike the original NDFS algorithm, the *red* colour is now updated in *post-order*, by using an extra *pink* colour similar to *cyan*. As a result, this technique prunes the search space. However, a synchronization step is applied to remain correct and the scalability might suffer on graphs with few accepting states.

Second, the ENDFS algorithm [16] shares both the *red* and *blue* colour globally. Here, DFSRED is also post-order by using an extra *pink* colour. This technique marks accepting vertices *dangerous* if these states possibly do not preserve the post-order nature. To 'repair' this, a sequential NDFS phase is used to double-check the vertice. Moreover, to remain correct, the vertices found by DFSRED are marked *red* after this search is complete, by maintaining thread-local sets of red states. While this algorithm provides better scalability from the start (due to sharing of multiple colours), the repair phase could hamper the process by possibly introducing duplicate work.

Experimental comparison between the two algorithms [37] led to believe that a both algorithms could complement each other. The CNDFS algorithm [15, 35] (as an improvement to an earlier attempt [37]) was designed to combine LNDFS and ENDFS. In this algorithm, the synchronization method from LNDFS is used (by waiting for instances of DFSRED to finish) to take away the need for a sequential repair procedure. Experiments [15] show that CNDFS is currently the fastest LTL model checking algorithm in practice. In terms of complexity, all of the NDFS-based algorithms perform in linear time.

3.3.2 Lowe's algorithm

Lowe [40] presents a variation to TARJAN's algorithm that utilizes multiple processors to achieve significant speed-ups compared to the sequential version. We present this algorithm in Algorithm 12 (Lowe presents an iterative version; we rewrote this to a recursive one). We refer to Algorithm 4 to compare it with TARJAN's algorithm. The algorithm is based on simultaneously running multiple synchronized instances of TARJAN's algorithm, each starting from a distinct vertex. Each search maintains its own stack. The *index* and *lowlink* of the vertices are shared globally over all workers, as well as the *Suspended* map. Moreover, vertices are globally marked as either UNSEEN, LIVE, or DEAD.

Initially, all vertices are marked UNSEEN. A search marks a vertex v with LIVE if it finds v, and $v.status =$ UNSEEN holds previously (Line 23). A vertex v is marked DEAD if the search has completed exploring the SCC containing v (Line 19). Whenever a search p encounters a vertex v that is marked with LIVE by another worker (hence $v \notin S_p$), it suspends the search until the vertex is marked DEAD (Line 14). This way, a vertex can only be in the stack of at most one search. A so-called *blocking cycle* can arise from this, in which each

Algorithm 12 Lowe's algorithm [40] (presented recursively)

```
1:  $\forall v \in \mathcal{V} : v.status := \textsc{unseen};\ v.index := v.lowlink := 0$
2:  $counter_p := 0$
3:  $S_p := Suspended := \emptyset$
4:  procedure $\textsc{Lowe}_p(v)$
5:      $\textsc{addNode}(v)$
6:      $S_p.\textsc{push}(v)$
7:      for each $w \in \textsc{post}(v)$ do
8:          if $w.index = 0$ then [unvisited state]
9:              $\textsc{Lowe}_p(w)$
10:             $v.lowlink := \textsc{min}(v.lowlink, w.lowlink)$
11:         else if $w \in S_p$ then [back-edge]
12:             $v.lowlink := \textsc{min}(v.lowlink, w.index)$
13:         else if $w.status \neq \textsc{dead}$ then
14:             $\textsc{Suspend}(v, w, p)$ [Wait until $w.status = \textsc{dead}$]
15:     if $v.lowlink = v.index$ then [remove completed SCC]
16:         $w := S_p.\textsc{pop}()$
17:         while $w.index \geq v.index$ do
18:             $w := S_p.\textsc{pop}()$
19:             $w.status := \textsc{dead}$ [unblocks all searches waiting for $w$]

20: procedure $\textsc{addNode}_p(v)$
21:     $counter := counter + 1$
22:     $v.lowlink := v.index := counter$
23:     $v.status := \textsc{live}$
24:     $S_p.\textsc{push}(v)$
```

worker waits on another to finish exploring a vertex. To overcome this problem, the relevant vertices of those searches are transferred to a single search as explained below.

The blocked searches are recorded in a *Suspended* map. The search trace, ending in a vertex v, is stored along with the successor vertex, w, that caused the block. A blocking cycle is detected by checking whether the suspended map contains a path from w to v. If this is the case, the vertices are transferred to a single search and the normal procedure can be resumed. An additional note is that no two concurrent attempts may take place to detect blocking cycles (hence the check for blocking cycles takes place in a synchronized environment). We refer to Algorithm 13 for an interpretation on this SUSPEND procedure. Note that if a search is suspended, the algorithm spawns a new search for the waiting worker to continue on. This way, a worker does not have to wait for others and remains able to 'contribute' to the SCC exploration.

Algorithm 13 Abstract interpretation for the Suspend procedure in Lowe's algorithm

```
procedure SUSPEND($v, w, p$)
    $block := w.search \in Suspended$ [check if we have a blocking cycle]
    if $block$ then
        search $S_q$ for every worker $q$ that is part of the blocking cycle [until we obtain the path $w \rightarrow^* v$]
        by recursively checking for which worker and state $q$ is waiting
        and push all these states on $S_p$ (with updated index and lowlink values)
        continue exploring the SCC
    else
        $Suspended := Suspended \cup \{p\}$ [store the search in the suspended map]
        while $w.status \neq$ DEAD do
            [Wait until another worker sets $w.status =$ DEAD]
        $Suspended := Suspended \setminus \{p\}$ [remove the search from the suspended map]
```

Experimental evaluation shows a three- to four-fold speedup on an eight-core machine, compared to the sequential TARJAN's algorithm. Note that these experiments were performed by using the algorithm in an *offline* configuration; it remains unknown whether these speedups hold when performed on-the-fly. Unfortunately we were not able to get the implementation for LOWE's algorithm working in our environment to test this for ourselves.

From the experiments that Lowe performed [40], we found that almost none of the examples contained large SCCs. Lowe provides an explanation for the result of one example containing a large SCC (for which the performance is worse compared to a sequential algorithm), which we cited as follows:

> *"This graph has a large SCC, accounting for over 70% of the states. The concurrent algorithms for SCCs and loops consider the nodes of this SCC sequentially and so (because the concurrent algorithms are inevitably more complex) are somewhat slower than the sequential algorithms."*

Experiments on randomly generated graphs show similar results (the algorithm's performance drops significantly when increasing the inter-connectivity of the graph). We therefore assume that the communication mechanism used in LOWE's algorithm to communicate partially discovered SCCs is inefficient.[1] The worst-case complexity of the algorithm is shown to be $\mathcal{O}(|\mathcal{V}|^2 + |\mathcal{E}|)$. The reason for this quadratic complexity (as opposed to the linear one from TARJAN's algorithm) is because of the cost of transferring vertices from one search to another [40].

3.3.3 Renault's algorithm

Renault has recently presented a new multi-core algorithm for detecting accepting cycles [53] by constructing SCCs. This algorithm is based on the SWARM principle [27] discussed in Section 3.3.1. The key aspect of

[1]We have not been able to implement LOWE's algorithm nor did we succeed in performing our own experiments.

RENAULT's algorithm is that it communicates information about completely explored SCCs over the workers.

RENAULT's algorithm (see Algorithm 14 for a recursive version) combines TARJAN's algorithm with a parallel UNION-FIND structure to store information about SCCs. For accepting cycle detection, an acceptance flag is combined in the UNION-FIND structure. Every time that a UNION operation takes place, this acceptance flag is propagated to the representative of the set. We refer to RENAULT as an SCC algorithm for the remainder of this thesis.

We observe in the algorithm that the communication takes effect in Line 10. This causes a worker to not re-explore an already completed SCC. Note that otherwise, the algorithm performs similar to TARJAN's algorithm. We observed that in case the graph consists of one large SCC of size $|\mathcal{V}|$, there is no communication possible for the SCC algorithm (note that the algorithm may still benefit from multiple workers when detecting accepting cycles - due to the propagation of the acceptance flag). The complexity of the algorithm is quasi-linearly bounded by $\mathcal{O}(\mathcal{P} \cdot (|V| + |E|))$, where each worker has the same complexity as for TARJAN's algorithm (and operations on the UNION-FIND structure take place in quasi-constant time).

Algorithm 14 Renault's algorithm [53] (presented recursively)

```
1:  $\forall v \in \mathcal{V} : v.index_p := v.lowlink_p := 0$
2:  $counter_p := 0$
3:  $S_p := \emptyset$
4:  procedure RENAULT$_p(v)$
5:      MAKESET$(v)$
6:      $counter := counter + 1$
7:      $v.lowlink_p := v.index_p := counter$
8:      $S_p$.PUSH$(v)$
9:      for each $w \in$ POST$(v)$ do
10:         if SAMESET$(w,$ DEAD$)$ then continue
11:         else if $w.index_p = 0$ then [unvisited state]
12:             RENAULT$(w)$
13:             $v.lowlink_p :=$ MIN$(v.lowlink_p, w.lowlink_p)$
14:             if $w.lowlink_p \leq v.index_p$ then
15:                 UNION$(v, w)$
16:                 [accepting cycle: if $v \in \mathcal{A}$ then report cycle]
17:         else if $w \in S_p$ then [back-edge]
18:             [accepting cycle: if $v \in \mathcal{A}$ then report cycle]
19:             $v.lowlink_p :=$ MIN$(v.lowlink_p, w.index_p)$
20:     if $v.lowlink_p = v.index_p$ then [remove completed SCC]
21:         UNION$(v,$ DEAD$)$ [globally mark the SCC DEAD]
22:         $w := S_p$.POP()
23:         while $w.index_p \geq v.index_p$ do
24:             $w := S_p$.POP()
```

Previous work has been observed that focused on the same principles [23, 39]. However, these algorithms performed the communication procedure inefficiently, thereby causing a quadratic complexity. Renault provided two algorithms: one based on DIJKSTRA's algorithm and one based on TARJAN's algorithm. For the detection of accepting cycles, a combination of the two is used (by dividing the workers over both algorithms). This allowed the algorithm to benefit from the advantages of both techniques.

3.4 Conclusion

To summarize the discussed material, we refer to Table 3.1. Each of the discussed SCC and accepting cycle algorithms are briefly presented in the table. A small description is given as well as the worst-case complexity for the algorithms.

Table 3.1: Comparison of SCC algorithms.

Algorithm	**Type**	**Parallel**	**On-the-fly**	**Complexity**	**Description**
Sequential DFS-based algorithms					
Tarjan [57]	SCC	×	✓	$\mathcal{O}(\lvert\mathcal{V}\rvert)$	Basic one-pass algorithm
Geldenhuys-Valmari [22]	SCC*	×	✓	$\mathcal{O}(\lvert\mathcal{V}\rvert)$	Early cycle termination
Dijkstra [14]	SCC	×	✓	$\mathcal{O}(\lvert\mathcal{V}\rvert)$	Uses stack of root candidates
Couvreur [13]	SCC*	×	✓	$\mathcal{O}(\lvert\mathcal{V}\rvert)$	Combines Dijkstra with LTL
Kosaraju-Sharir [55]	SCC	×	×	$\mathcal{O}(\lvert\mathcal{V}\rvert)$	Basic two-pass algorithm
Purdom [50]	SCC	×	✓	$\mathcal{O}(\lvert\mathcal{V}\rvert^2)$	Basic set-based algorithm
Munro [45]	SCC	×	✓	$\mathcal{O}(\lvert\mathcal{V}\rvert \log \lvert\mathcal{V}\rvert)$	Improved Purdom's work
Gabow [19]	SCC	×	✓	$\mathcal{O}(\lvert\mathcal{V}\rvert^+)$	Set-based with Union-Find
Parallel fixed-point algorithms					
FB [18]	SCC	✓	×	$\mathcal{O}(\lvert\mathcal{V}\rvert^2)$	Basic divide-and-conquer algorithm
FB+OWCTY [44]	SCC*	✓	×	$\mathcal{O}(\lvert\mathcal{V}\rvert^2)$	Early removal of trival components
OBF [6]	SCC	✓	×	$\mathcal{O}(\lvert\mathcal{V}\rvert^2)$	Partitions graph in sub-graph slices
CH [46]	SCC	✓	×	$\mathcal{O}(\lvert\mathcal{V}\rvert^2)$	Propagates colours to identify sub-graphs
Hong [28]	SCC	✓	×	$\mathcal{O}(\lvert\mathcal{V}\rvert^2)$	Removes large SCC, then CCs + FB
Multistep [56]	SCC	✓	×	$\mathcal{O}(\lvert\mathcal{V}\rvert^2)$	Removes large SCC, then CH + Tarjan
Parallel DFS algorithms					
LNdfs [36]	LTL	✓	✓	$\mathcal{O}(\mathcal{P} \cdot \lvert\mathcal{V}\rvert)$	Shares red states + synchronization
ENdfs [16]	LTL	✓	✓	$\mathcal{O}(\mathcal{P} \cdot \lvert\mathcal{V}\rvert)$	Shares red and blue states + repair phase
CNdfs [15]	LTL	✓	✓	$\mathcal{O}(\mathcal{P} \cdot \lvert\mathcal{V}\rvert)$	Combines LNdfs and ENdfs
Lowe [40]	SCC*	✓	✓	$\mathcal{O}(\lvert\mathcal{V}\rvert^2)$	Multiple Tarjan's + synchronization
Renault [53]	SCC*	✓	✓	$\mathcal{O}(\mathcal{P} \cdot \lvert\mathcal{V}\rvert^+)$	Multiple Tarjan's + dead communication

* These algorithms are based on SCC discovery, but also propose means to extract an accepting cycle.
$^+$ Due to the Union-Find structure, the complexity is multiplied with the (pracitcally constant) inverse Ackermann function.

An observation from recent SCC algorithms (Hong [28] and Multistep [56]) is that these all consider *small-world* graphs. The designs of the algorithms aim to efficiently deal with the (potentially) large SCC in these types of graphs. Furthermore, the approach from Renault suggests that a parallel Union-Find structure may remain efficient.

Another observation is that both of the parallel on-the-fly algorithms rely on holding a single worker responsible for discovering a complete SCC. Therefore, if a graph contains a relatively large (to the number of vertices) SCC, the scalability is limited for these algorithms.

By combining these two observations, we acquire the main motivation for desigining a new algorithm. Namely, that it remains unknown from the literature whether or not it is possible to construct an efficient scalable parallel algorithm which communicates partially discovered SCCs.

Chapter 4

Naive Approach

This chapter presents the design of a naive UNION-FIND based parallel SCC algorithm. First, we show how multiple workers can aid each other while discovering an SCC. We then consider global properties that should remain valid throughout the algorithm. Then, we propose a technique to combine the worker's search paths globally. We discuss the complications that arise for a strict DFS-based approach and propose means to mitigate these.

4.1 Communication of partially discovered SCCs

The aim for our design is to let multiple workers aid each other while exploring an SCC. The strategy for this is best explained by an example. Figure 4.1 shows a particular scenario in which two workers can explore an SCC more efficiently compared to a sequential version. Assume that a blue worker has searched the path $a \rightarrow b \rightarrow c \rightarrow d \rightarrow a$ and thus finds the cycle containing states a, b, c, d. Assume that a red worker has explored $a \rightarrow e \rightarrow f \rightarrow e$ and finds the cycle containing states e and f. This situation is depicted in the figure by the marked regions. Now, assume that the red worker continues searching from state f. It encounters the edge from f to c. Assuming that no information is communicated, the red worker will have to explore $f \rightarrow c \rightarrow d \rightarrow a$ to find the cycle a, e, f, c, d, a. However, the blue worker knows that c is part of the same SCC as a (hence $c \rightarrow^* a$). If communicated properly, the red worker can derive that $f \rightarrow c$ implies $f \rightarrow^* a$. Since a is part of the red worker's search path, the cycle $a \rightarrow e \rightarrow f \rightarrow^* a$ is formed. Finally, the red worker can add states e and f to the 'blue' SCC (a, b, c, d), resulting in the completed SCC: $\{a, b, c, d, e, f\}$.

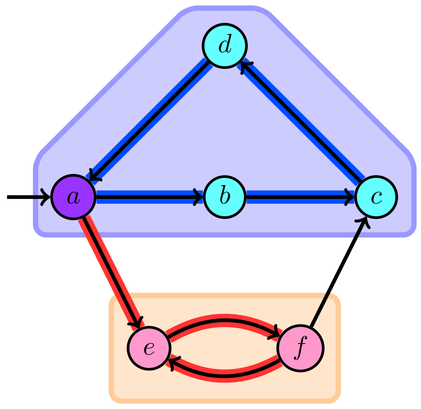


Figure 4.1: Example depicting the advantage of communicating partially discovered SCCs.

We base the design on set-based algorithms (see Section 3.1.4). The idea is to globally contract cycles to 'supernodes'. Algorithm 15 provides an abstract interpretation for the algorithm.

Algorithm 15 Parallelized (abstract) set-based algorithm

```
1:  $\forall v \in \mathcal{V} : \textsc{set}(v) := \{v\}$
2:  $S_p := \{v_0\}$
3:  procedure ParSetBased$_p(v)$
4:      for each $w \in \textsc{post}(v)$ do [all successors of $\textsc{set}(v)$]
5:          if $\forall w' \in \textsc{set}(w) : w' \notin S_p$ then $S_p.\textsc{push}(w)$ [unvisited state]
6:          else [back-edge]
7:              while $S_p.\textsc{top}() \neq \textsc{set}(w)$ do
8:                  $t := S_p.\textsc{pop}()$
9:                  $\textsc{set}(w) := \textsc{set}(w) \cup \textsc{set}(t)$ [merge states globally]
10:                 $\textsc{post}(w) := \textsc{post}(w) \cup \textsc{post}(t)$ [merge successors globally]
11:         ParSetBased$_p(w)$
12:     if $S_p.\textsc{top}() = v$ then $S_p.\textsc{pop}()$
```

Here, multiple workers perform a DFS over the states. If a back-edge is found, the complete cycle is globally merged to a single state (Lines 9-10). In the example of Figure 4.1 we would have two supernodes, with $\textsc{set}(a) = \textsc{set}(b) = \textsc{set}(c) = \textsc{set}(d) = \{a, b, c, d\}$ and $\textsc{set}(e) = \textsc{set}(f) = \{e, f\}$. The communication aspect is used in Lines 5 and 6. Here, we check if $\textsc{set}(w)$ contains any state that is part of S_p. If this is the case, a cycle is found. In the example of Figure 4.1 we have from $f \to c$, $a \in S_{red}$ and $a \in \textsc{set}(c)$ that $f \to c \to^* a \to^* f$ is a cycle.

4.2 Parallelizing a set-based approach

The basis for the algorithm is to share knowledge between workers about partially discovered SCCs. By focusing on a shared Union-Find structure for storing SCCs, we consider it possible to attain this without much overhead. The reason for this is because a Union-Find structure is able to combine two sets of arbitrary size in amortized quasi-constant time (see Section 2.3.2). The design for our initial algorithm is based on spawning multiple DFS instances and allow them to explore the same vertices. For the design of the algorithm, we use Invariants 4.1 and 4.2.

Considering the Union-Find structure from a notational aspect, we refer to a Union-Find set containing state v as $\textsc{uf}[v]$. If state w is also part of this set, we denote this as either $\textsc{SameSet}(v, w)$, $w \in \textsc{uf}[v]$, $v \in \textsc{uf}[w]$. Here, $\textsc{uf}[v] = \textsc{uf}[w] = \{v, w\}$. When we state a property from explicitly the $\textsc{uf}$ entry for v, we denote this as $\textsc{uf}[v].\mathit{property}$ (for instance, $\textsc{uf}[v].\mathit{parent} = w$).

Invariant 4.1. After the algorithm finishes, the $\textsc{uf}$ sets represent all reachable SCCs.

Corollary 4.1. *All states in the $\textsc{uf}$ sets must be strongly connected. For every state in the* Union-Find *structure, we must preserve the property of strong connectivity inside each* $\textsc{uf}$ *set at all times. Thus we have* $\forall v, w : v \in \textsc{uf}[w] \implies v \to^* w \to^* v$.

Proof. Assume by contradiction that for some v and w we have $v \in \textsc{uf}[w]$ and $v \not\to^* w \ \vee \ w \not\to^* v$. From the preliminaries on the Union-Find structure (Section 2.3.2), we observe that it is not possible to remove an element from a $\textsc{uf}$ set. Therefore, once the strong connectivity property fails for a particular $\textsc{uf}$ set, this will remain the case after the algorithm finishes and Invariant 4.1 fails as a result. Hence, if we use the $\textsc{uf}$ sets to store SCCs, they must obey the strong connectivity property at all times. □

In our design we adopt the observation from Renault regarding the communication on dead SCCs. We say that a state may be marked dead if has been fully explored (including its successors). We imply that a dead state only has dead successors, including the SCC containing the dead state. If a state has not been visited by any worker, the state is marked unseen. If otherwise, a state is neither dead nor unseen, we have that the state is marked live.

Invariant 4.2. A dead uf set implies a (maximal) SCC.

Corollary 4.2. dead uf *sets cannot be extended.*

Proof. If we assume that for some v, $\text{UF}[v]$ is dead and we unite $\text{UF}[v]$ with some $t \notin \text{UF}[v]$, then Invariant 4.2 fails as it contradicts with the maximality aspect of SCCs (see Definition 2.8). Therefore, dead uf sets cannot be extended. □

When traversing the graph, we consider possible cases for when worker p encounters the edge $v \to w$:

1. *w is globally* unseen*.* Here, we cannot gain knowledge from other workers as there is none.
2. *w is part of the worker's (local) search stack.* Here, we encounter a back-edge (and thus a cycle) to a state which worker p has previously visited; we can unite all states on this path and report this cycle globally.
3. *w is globally marked* dead*.* There is a worker that has completely explored the SCC containing w (assuming that we ensure this with the notion of dead states), therefore we can ignore every successor that is marked dead.
4. *w has been visited by some worker $q \neq p$.* Here, we can distinguish two possible sub-cases:
   (a) *$\exists t \in \text{UF}[w] : t$ is on the worker's local search stack.* Here, we observe an indirect cycle, namely: $v \to w \to^* t \to^* v$. Therefore, we can unite all states on this path and report this cycle globally. This particular case *forms the basis for our design* with regard to information sharing on partially discovered SCCs.
   (b) *$\forall t \in \text{UF}[w] : t$ is not on the worker's local search stack.* Here, it might or might not be possible that v and w lie on the same cycle; we cannot derive global properties from this.

4.3 Introducing Pset

Concerning the analysis from the previous section, we developed a method to answer the following question:

> *Given $v \to w$ for worker p, if w is neither part of a* dead *SCC nor part of the search stack for worker p, can we detect if there is some state $t \in \text{UF}[w]$ for which t* is *part of the search stack for worker p?*

We cannot lookup the representatives on local stacks in constant time. One could argue that it is possible if each worker keeps track of the representative for each uf set. If this is the case, the answer will then be provided by performing the check if $\text{Find}(w)$ is in the search stack.

We state that it is *not* possible to keep the representatives for each uf set on the local stacks for the workers. The reason being that any worker is able to update this uf set, with the implication that the representative might change. We show this with an example:

Assume that we have $\text{UF}[v] = \{v\}$ for a state v on the local stack for worker p. Assume that for worker $q \neq p$ we have $\text{UF}[w] = \{w\}$, where w is on the local stack for worker q. Suppose that any worker unites the states v and w, we obtain $\text{UF}[v] = \text{UF}[w] = \{v, w\}$. The representative for this set can either be v or w. Assuming w.l.o.g. that the representative for this set is w, we have that the local stack of p does not contain the representative for $\text{UF}[v]$ anymore.

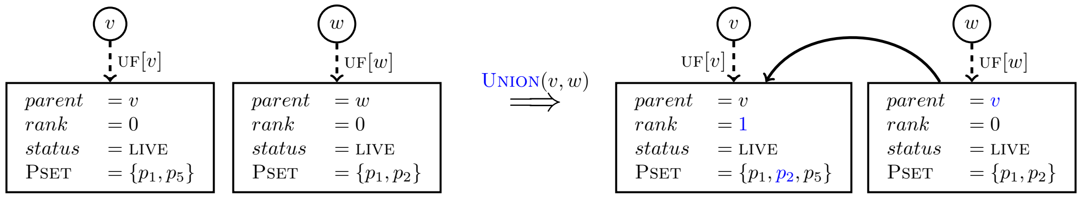


Figure 4.2: Example on how a state's PSET is updated after a UNION call.

Extending the Union-Find structure with a worker set. We propose to maintain a global set for each state in the UF set, in which we store all worker identifiers (IDs) that have visited this state. We refer to this set by PSET. Whenever a UNION procedure is executed for two states v and w, we take the set-theoretical union for their PSET's and store this as the PSET for the representative. Thus, for UNION(v, w) we have, w.l.o.g. that if v is the representative in the UF set: v.PSET $:= v$.PSET $\cup$ w.PSET. We refer to Figure 4.2 for an example. By propagating all worker IDs to the representative of the UF set, we ensure that all workers that have visited any state in this set have their worker ID in the PSET of the representative.

As a result of using this PSET, we can obtain which workers have visited any state in a UF set. With respect to the the stated question, we can detect if worker p has visited some $t \in$ UF$[w]$ with $p \in$ UF[FIND(w)].PSET. We state this in Invariant 4.3 (where S_p denotes the search path for worker p). As a result, the algorithm is based on Proposition 4.3. The idea behind this proposition is that some (other) worker q has detected the cycle $w \rightarrow^* u \rightarrow^* w$ where $w \notin S_p$ and $u \in S_p$. We state that worker q must assure that state w is fully explored. From a notational aspect, we also denote UF[FIND(w)].PSET with PSET(w).

Invariant 4.3. If some worker p discovers the edge $v \rightarrow w$, for $w \notin S_p$ and $\neg$ISDEAD(w), we have that $p \in$ PSET$(w) \implies w \rightarrow^* v$.

Proposition 4.3. *If worker p discovers the cycle $v \rightarrow w \rightarrow^* v$ (and $w \notin S_p$) from the implication of Invariant 4.3, worker p does not have to explicitly explore state w. We obtain from the invariant that some worker $q \neq p$ is currently exploring state w. Assuming that worker q completely explores this state, hence this should not be necessary for worker p. As a result, each worker may only need to explore a subset of all reachable states.*

Implementing the extended Union-Find structure Concerning the implementation of PSET, we can achieve this with a bitset of size $\mathcal{P}$ (the total number of workers). By applying a bitwise OR on two PSETs, we obtain the union of both sets. In contrast to RENAULT's and GABOW's algorithm, we do not include an artificial DEAD state for merging with completely explored SCCs. Instead, we keep track of a status field that is either UNSEEN, LIVE, or DEAD. This way, we can more easily derive the SCCs when the algorithm is finished. Algorithm 16 provides an overview on the design for the extended UNION-FIND structure.
We highlight some features of the extended UNION-FIND structure from Algorithm 16:

- All lines in the structure are assumed to be executed atomically. Note that this is not always possible in practice (e.g. Line 34), however we conjecture that race-conditions remain prevented.
- The methods in the UNION-FIND structure are designed to be lockless. As discussed in [1] and briefly touched in Section 2.3.2, we must consider the possibility that a UNION operation can take place at any time, thus the parent and PSET for a state may be updated after each line of the algorithm.
- In Lines 2 and 20, we make use of the *Compare&Swap* operation (denoted as CAS) to respectively update the status and parent for a state.
- We note that it is possible to combine Lines 20 and 21 to a single 128-bit CAS instruction.

- The FIND method is implemented using *path compression* and the UNION method applies *weighing by rank* (see Section 2.3.2 for details).
- The ADDWORKER(x, p) procedure ensures that we have that $p \in$ PSET(x), thus the worker ID p is stored in the PSET for the representable for UF$[x]$.
- The MARKDEAD(x) procedure sets the status for UF$[x]$ (for the complete set) to DEAD. Note that it is not possible to change this status afterwards.

We present our designed naive algorithm utilizing this extended UNION-FIND in the next section.

Algorithm 16 Extended lockless UNION-FIND structure for the naive approach (simplified)

```
procedure MakeSet($x$)
    CAS(UF$[x]$.$status$, UNSEEN, LIVE)

procedure Find($x$)
    if UF$[x]$.$parent \neq x$ then
        UF$[x]$.$parent$ := Find(UF$[x]$.$parent$)
    return UF$[x]$.$parent$

procedure SameSet($x, y$)
    $x_r$ := Find($x$)
    $y_r$ := Find($y$)
    if Find($x_r$) $\neq x_r \vee$ Find($y_r$) $\neq y_r$ then
        SameSet($x_r, y_r$)
    return $x_r = y_r$

procedure AddWorker($x, p$)
    while $p \notin$ UF[Find($x$)] do
        $x_r$ := Find($x$)
        UF$[x_r]$.PSet := UF$[x_r]$.PSet $\cup \{p\}$

procedure HasWorker($x, p$)
    return $p \in$ UF[Find($x$)].PSet

procedure UnionAux($x, y$)
    if CAS(UF$[y]$.$parent, y, x$) then
        UF$[x]$.PSet := UF$[x]$.PSet $\cup$ UF$[y]$.PSet
    if Find($x$) $\neq x \vee$ Find($y$) $\neq x$ then
        UnionAux(Find($x$), Find($y$))

procedure Union($x, y$)
    $x_r$ := Find($x$)
    $y_r$ := Find($y$)
    if UF$[x_r]$.$rank >$ UF$[y_r]$.$rank$ then
        UnionAux($x_r, y_r$)
    else
        UnionAux($y_r, x_r$)
        if UF$[x_r]$.$rank =$ UF$[y_r]$.$rank$ then
            UF$[y_r]$.$rank$ := UF$[y_r]$.$rank + 1$

procedure MarkDead($x$)
    UF[Find($x$)].$status$ := DEAD

procedure IsDead($x$)
    return UF[Find($x$)].$status$ = DEAD
```

4.4 The algorithm and its complications

We designed a parallel UNION-FIND based SCC algorithm based on Proposition 4.3. The result is presented in Algorithm 17. We explain the algorithm and discuss why the extra methods (AWAIT, WAITDEAD, FINDDEADLOCK) are required for maintaining correctness.

In PARSCC$_p(v)$, at Line 6 the worker ID is stored in PSET(v). We observe from Line 7 that any state from S_p contains p in its PSET. The successors for v are considered in Lines 8-12. We prohibit workers from re-exploring DEAD SCCs in Line 9. Proposition 4.3 takes effect in Lines 10 and 11. Here, we apply the notion Invariant 4.3 to check if the cycle $v \rightarrow w \rightarrow^* v$ has been globally observed (by any worker). If so, we pop and unite states from S_p (in MERGEPATH) until we encounter some state $t \in$ UF$[w]$: $t \in S_p$ as this is inferred by the invariant. Similar to DIJKSTRA's algorithm, we keep track of the *root* for the SCC. We call the PARSCC$_p$ recursively in Line 12. This line is executed for any globally UNSEEN state, and any state that is not DEAD, and has not been visited by worker p, and has not (yet) been observed to be part of a cycle for which some state is part of S_p.

Algorithm 17 A naive parallel SCC algorithm

```
 1: $\forall v \in \mathcal{V}$ : UF[$v$].$status$ := UNSEEN
 2: $\forall i \in [1 \ldots \mathcal{P}]$ : $Wait_i$ := NULL
 3: $S_p := \emptyset$
 4: procedure PARSCC$_p(v)$
 5:     MAKESET($v$)
 6:     ADDWORKER($v, p$)
 7:     $S_p$.PUSH($v$)
 8:     for $w \in$ POST($v$) do
 9:         if ISDEAD($w$) then continue
10:         if HASWORKER($w, p$) then
11:             MERGEPATH$_p(w)$
12:         else PARSCC$_p(w)$
13:     if $v = S_p$.TOP() then
14:         $S_p$.POP()
15:         if $S_p = \emptyset$ then return
16:         if ISDEAD($v$) then return
17:         else if $\neg$SAMESET($v, S_p$.TOP()) then
18:             $Wait[p] := v$
19:             AWAIT$_p(v)$
20:             $Wait[p]$ := NULL

21: precondition: $\exists w \in$ UF[$v$] : $w \in S_p$
22: procedure MERGEPATH$_p(v)$
23:     while $\neg$SAMESET($v, S_p$.TOP()) do
24:         UNION($v, S_p$.POP())

25: procedure AWAIT$_p(v)$
26:     do
27:         FINDDEADLOCK$_p(v)$
28:         WAITDEAD$_p(v)$
29:     while $\neg$ISDEAD($v$)$\wedge\neg$SAMESET($v, S_p$.TOP())

30: procedure WAITDEAD$_p(v)$
31:     for each $i \in [0 \ldots |S_p|]$ do [takes $\mathcal{O}(|\mathcal{V}|)$ time]
32:         if SAMESET($v, S_p[i]$) then
33:             MERGEPATH$_p(S_p[i])$
34:             return [SAMESET($v, S_p$.TOP())]
35:     for each $i \in [1 \ldots \mathcal{P}]$ do
36:         if HASWORKER($v, i$) then
37:             if $Wait[i] \notin$ UF[$v$] then return
38:     UF[FIND($v$)].PSET :\= $\{p\}$ [remove worker]
39:     while UF[FIND($v$)].PSET $\neq \emptyset$ ;
40:     MARKDEAD($v$)

41: procedure FINDDEADLOCK$_p(v)$
42:     $deadlock$ :=SEARCHCYCLE$_p(Wait, v)$
43:     for each $w \in deadlock$ do
44:         if HASWORKER($w, p$) then
45:             UNION($v, w$)
46:             MERGEPATH$_p(w)$
```

Waiting on other workers. When all successors are considered, the algorithm backtracks. We are only interested in possible root candidates for SCCs, hence Line 13. In case the algorithm is executed sequentially, this would be sufficient to mark the SCC as DEAD. However, due to Line 10, we may have omitted a part of the SCC. Therefore, another worker may still be busy exploring the component. Since it is only possible to discover completed SCCs *bottom-up* (that is, starting with terminal SCCs), we wait until all other workers have finished exploring this SCC.[1] We refer to Figure 4.3 for an example on why this otherwise could go wrong.

In this example, the blue worker has explored the cycle $c \rightarrow d \rightarrow c$ (thus UF[c] $= \{c, d\}$) and this worker has backtracked to c. We assume that the blue worker halts for the time being. The red worker has explored the path $a \rightarrow b \rightarrow d$. In state d, the red worker ignores successor $d \rightarrow c$ (observe in Line 10, with the notion that $d = S_{red}$.TOP). Now, the red worker observes no other successors from d and attempts to backtrack. If we allow this, the red worker backtracks to state b. Here, the red worker (again) sees no other successors, thus it backtracks again. Since PSET(b) $= \{b\}$, the red worker notices that it is the only worker in UF[b],

[1] Note that this may cause an uneven work spread amongst the workers. We hypothesize that it is an uncommon scenario.

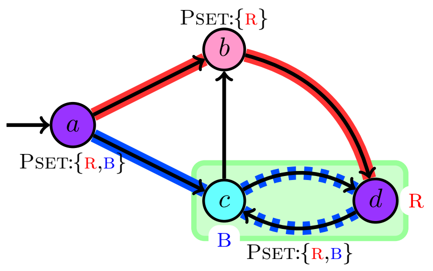


Figure 4.3: Example situation for explaining why a wait procedure is applied in Algorithm 17.

hence b gets marked DEAD. The problem is, that when the blue worker continues and explores the edge $c \to b$, it is unable to detect the cycle $c \to b \to d \to c$ as ISDEAD(b) holds. Therefore, we state that any worker must wait with backtracking from an SCC C until all other workers in PSET(C) want to do the same.

This waiting procedure is implemented using a $Wait$ array of size $\mathcal{P}$. The table entries denote the states from which each worker wants to backtrack. If we observe for an SCC C that $\forall i \in$ PSET$(C) : Wait[i] \in C$ (see Lines 35-40 in the algorithm), all workers in PSET(C) are waiting thus every one of these workers may safely backtrack.

Deadlock cycle of waiting workers. By applying the waiting procedure as discussed before, we observe another issue. It may be possible for two (or more) workers to wait on each other, while they backtrack from disjunct UF sets. Figure 4.4 exemplifies this problem.

Here, we have on the left picture that the red worker has detected the cycle $b \to c \to b$ and the blue worker has detected $d \to e \to d$. Thus, UF$[b] = \{b, c\}$, PSET$(b) = \{red\}$, UF$[d] = \{d, e\}$, and PSET$(d) = \{blue\}$. Suppose that the blue worker explores $d \to b$ (and the red worker explores $c \to e$) as shown in the right picture. From state b, the blue worker ignores the edge $b \to c$ because $c \in$ UF$[S_{blue}.\text{TOP}]$. Thus, the blue worker sets $Wait[blue] := b$ (Line 18) and waits on the red worker. The red worker, however, observes a similar situation and ends up with $Wait[red] := e$ while waiting for the blue worker. As UF$[b] \neq$ UF$[e]$ we have a so-called *deadlock cycle*.

The FINDDEADLOCK procedure (Line 42) is designed to detect such a *deadlock cycle*. Here, the SEARCHCYCLE$_p$ method recursively searches over the $Wait$ array in combination with their PSETs to detect a cycle. In the example it detects that the blue and red worker are waiting on each other. Lines 43-46 unite this cycle.

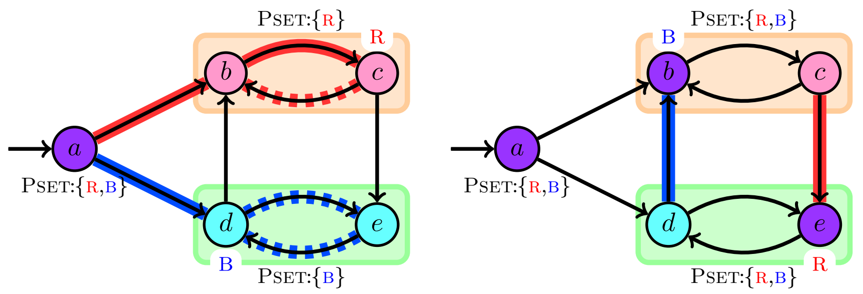


Figure 4.4: Example that shows how a deadlock cycle arises.

Reporting incomplete SCCs. A third problem in the algorithm is observed during the backtrack procedure. With the preventive measures taken from the previous issues, it remains possible for the algorithm to wrongly report partially discovered SCCs as complete ones. This is best explained with the example presented in Figure 4.5.

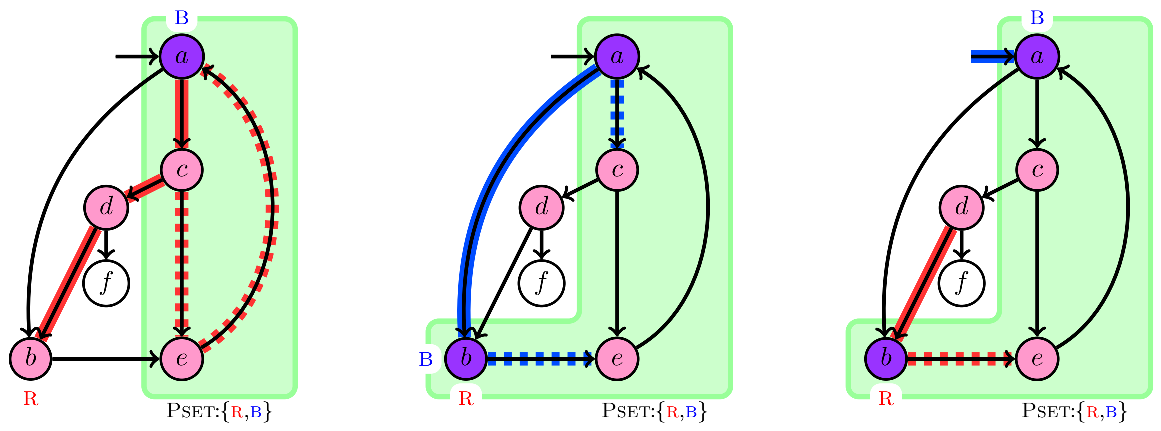


Figure 4.5: Example that shows how an incomplete SCC can be marked DEAD.

Here, in the left picture the red worker has first explored the cycle $a \to c \to e \to a$ after which it backtracked and continued exploring the path $c \to d \to b$. While the red worker halts for the time being, the blue worker ignores $a \to c$ (because $c \in \text{UF}[S_{blue}.\text{TOP} = a]$) explores $a \to b \to e$ as shown in the middle picture. Here, the blue worker correctly unites state b with $\text{UF}[a]$ as we have the cycle $a \to b \to e \to a$. The blue worker backtracks afterwards and waits in state a until the red worker has also finished exploring the SCC. In the right picture, the red worker continues from state b and observes the edge $b \to e$. Since $e \in \text{UF}[S_{red}.\text{TOP} = b]$, this edge is ignored. Now, we face the problem where the red worker has failed to include d in $\text{UF}[a]$. The red worker attempts to backtrack out of $\text{UF}[a]$ in which it succeeds since the blue worker is also waiting. As a result, the partial SCC $\{a, b, c, e\}$ is reported DEAD. As a side-effect, the red worker now has the DEAD state a on its stack.

Unfortunately, for this problem we did not find an efficient solution. The assumed best approach is to iterate over the complete stack to search if there is a *gap* in the SCC (and merge the states in this gap). This is shown in Lines 31-34. However, this does consists of performing an $\mathcal{O}(|\mathcal{V}|)$ operation on each backtrack.

4.5 Discussion

With all encountered problems being resolved, we conjecture that the algorithm is correct. Due to the preventive measure from Lines 31-34, the algorithm has a quadratic complexity. A preliminary experimental study shows that the algorithm is able to scale its performance for multiple workers. However, these results vary significantly for different models. By using four workers, for one instance we observed a two-fold speedup compared to TARJAN's algorithm, while on another it resulted in a speedup of 0.17 (meaning that TARJAN's sequential algorithm performed almost 6 times faster). Due to this inconsistency and quadratic complexity we investigated different means for parallel SCC decomposition. In Chapter 5 we discuss a new approach that we observed to outperform this technique.

Chapter 5

Improved Algorithm

This chapter presents the design of an improved UNION-FIND based parallel SCC algorithm (compared to the one presented in Chapter 4). We first state an observation which forms the basis for the algorithm, and how we apply this to our design. We present the resulting algorithm and provide a detailed design of the used sub-procedures. Finally, we discuss the algorithm along with a correctness proof and complexity analysis. For the final algorithm we refer to Algorithm 18 (the complete algorithm can be found in Appendix B).

5.1 Iterating over an SCC

In Chapter 4 we showed that we were not able to efficiently detect if an SCC has been fully explored. To overcome this problem we make it possible to iterate over the states in the SCC, by extending the UNION-FIND structure with a cyclic list (containing all states from the UF set). By removing fully explored states from the list we can detect a fully explored SCC by checking if the list is empty. Using this strategy, we propose a more efficient algorithm compared to the one presented in Chapter 4.

5.1.1 Necessary condition for reporting an SCC

We focus on the necessary condition for when an algorithm may report an SCC as fully explored (and prevent a partial SCC from being reported). We first state Observation 5.1 and derive Observation 5.2 from this notion. This observation is summarized in the GLOBALDEAD and GLOBALDONE predicates (Predicates 5.2 and 5.1). If GLOBALDEAD(v) is valid, then UF$[v]$ can be reported as a completed SCC. In Section 5.1.2, we improve the algorithm using these predicates.

Observation 5.1. Given an SCC C, for all states $v \in C$, we have that POST(v) solely consists of vertices that are contained in C or in SCCs which can be reached from C.

Proof. Assume that it is possible for some $v \in C$ to have a successor $w \in D$, where D is an SCC that is not reachable by C. This contradicts with the definition of reachability (Definition 2.6) since we have a finite path from v to w and thus also from C to D. Therefore D must be reachable from C (or we have $C = D$) and is the observation preserved. □

Observation 5.2. If SCCs are discovered and reported DEAD in a *bottom-up* approach, it is sufficient for an SCC C to be reported DEAD if for all states $v \in C$ we have that POST(v) solely consists of vertices that are contained in C or point to a DEAD SCC.

Proof. From Observation 5.1 we obtain that all successors for some state $v \in C$ point either to C, or to a reachable SCC from C. Since SCCs are discovered in a bottom-up fashion, all reachable SCCs must be completely explored before C. Therefore, all reachable SCCs from C are reported DEAD, which implies the observation. □

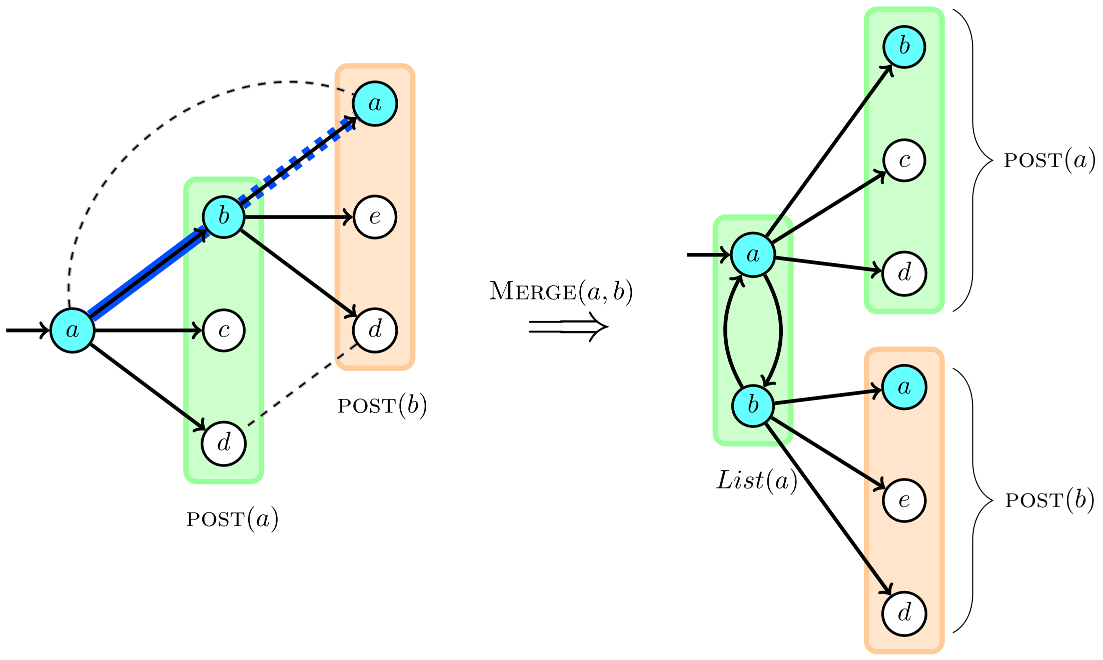


Figure 5.1: Illustrative representation for the list mechanism.

Predicate 5.1 (GLOBALDONE)**.** A state $v \in \mathcal{V}$ is called *GlobalDone* if $\forall w \in \text{POST}(v) : \text{SAMESET}(v, w) \vee \text{ISDEAD}(w)$.

Predicate 5.2 (GLOBALDEAD)**.** A state $v \in \mathcal{V}$ is called *GlobalDead* if $\forall v' \in \text{UF}[v] : \text{GLOBALDONE}(v')$ holds.

5.1.2 Introducing a cyclic-linked list structure

In order to verify if the GLOBALDEAD predicate is validated for some state v, we must assure the GLOBALDONE predicate for every state in UF$[v]$. To realize this, we provide means for iterating over the states in a UF set.

We extend the UNION-FIND structure from Algorithm 16 (which includes the PSET) with a cyclic singly-linked list. Figure 5.1 presents an overview on how we intend the list mechanism. This list is used to keep track of which states in the UF do not yet satisfy the GLOBALDONE predicate. Initially, we have $\forall v \in \mathcal{V} : \mathit{List}(v) := \text{UF}[v] := \{v\}$ since no states have been observed to be GLOBALDONE. When discovering a cycle, the UF sets for these states are united. This should also be the case for their lists, hence we introduce the MERGE procedure that combines the UNION procedure with a list merging technique. A list item is removed if we can assure that Predicate 5.1 holds for the corresponding state. In case all states are removed from the list, we can assure that the GLOBALDEAD predicate is valid, thus we may report the SCC as DEAD according to Observation 5.2. We redefine the notion for ISDEAD(v) to $\mathit{List}(v) = \emptyset$.

The main reasons for employing a list structure is that it is dynamically sized and that it can be implemented directly on the UNION-FIND structure using only a *list-next* reference. We implement this list cyclic. This origins from a practical point of view: if we want to verify if a list is empty, we must otherwise ensure that we have a reference to the list *head*. Since the representative for a UF set may constantly change, it is infeasible to keep track of the list head at a consistent place. We also considered keeping track of the combined set of successors for each state in a UF set. However, this design relies on dynamic allocation of these edges, which we observed to be inefficient for practical purposes. For the implementation we utilize observations from the design of a (non-cyclic) lockless parallel linked-list structure [24]. Concretely, we have:

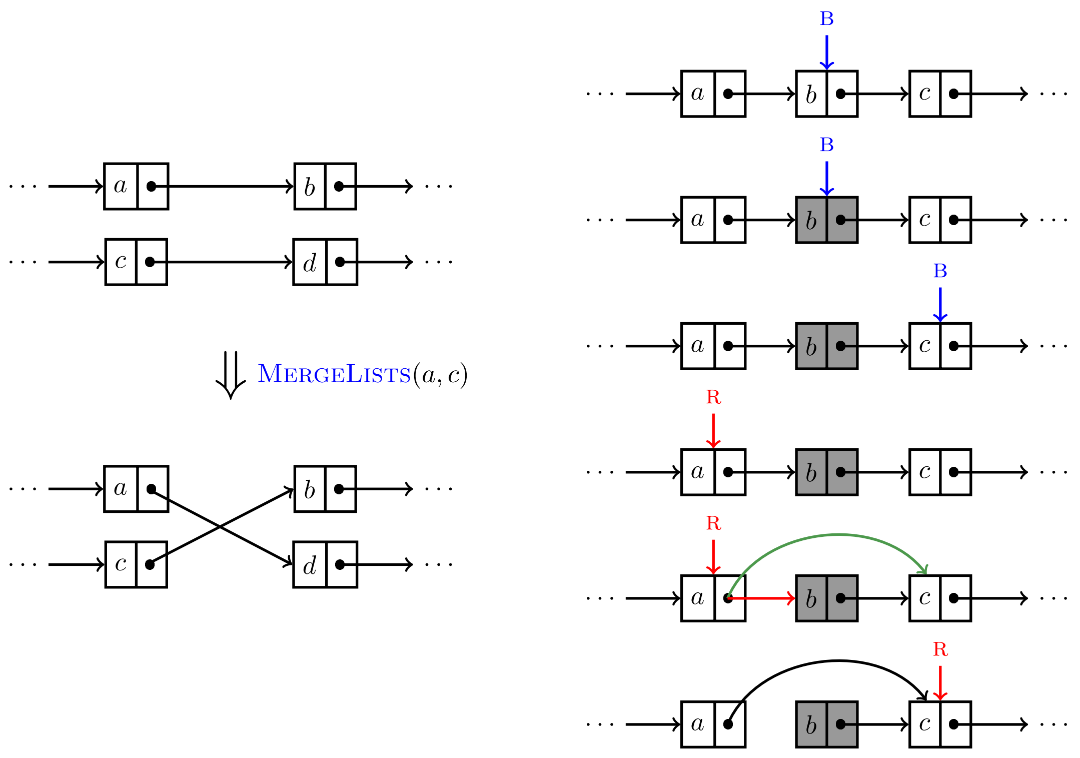


Figure 5.2: Illustrative representations of the internal list merge (left) and list removal (right) processes.

- Efficient list merging. We achieve this by swapping the *list-next* references of two previously disjunct lists. Figure 5.2 (left) provides an illustration on this procedure. We implemented this by locking the list entries (in the figure: a and c) to assure that the *list-next* pointer are unchanged by other workers during the SWAP process.
- Efficient list removal. We achieve this with a two-phase removal of a list item. A status variable is set to a TOMBSTONE value once called for removal. A second pass (by any worker, while iterating the list) updates the pointer referring to a TOMBSTONE entry to the next one. Figure 5.2 (right) gives an illustration on how this process is achieved. It is important that all removed list entries remain pointing towards the remainder of the list. Considering the example from Figure 5.2 (right), suppose that some worker p is currently exploring state b. Another worker q has just explored b (thus GlobalDone(b) is true) and removes it from the list. If the list entry for b does not point to the remainder of the list, worker p may be unable to search for remaining unexplored states in UF[b].
- Efficient list iteration and GlobalDead detection. We allow multiple workers to iterate over a list at the same time. By updating the entries pointing to TOMBSTONE states to their next *list-next* entries, as we discussed in the previous point, the length of the cycle reduces for every discovered TOMBSTONE. At some point, the cycle consists of a single TOMBSTONE entry which points to itself. If we encounter this situation, we can conclude that the GlobalDead predicate is validated.

5.2 The UF-SCC algorithm

The designed algorithm is presented in Algorithm 18 (the complete algorithm can be found in Appendix B). As a working title, we refer to it as the UF-SCC algorithm. We show how this algorithm operates.

Algorithm 18 The UF-SCC algorithm

```
1:  procedure UF-SCC-MAIN($v_0$, $\mathcal{P}$)
2:      for each $p \in [1 \dots \mathcal{P}]$ do
3:          MakeClaim($v_0$, $p$)
4:      UF-SCC$_1$($v_0$) $\parallel$ ... $\parallel$ UF-SCC$_\mathcal{P}$($v_0$)

5:  procedure UF-SCC$_p$($v$)
6:      $D_p$.push($v$) [start 'new' SCC]
7:      while $v' :=$ PickFromList($v$) do
8:          for each $w \in$ post($v'$) do
9:              $result :=$ MakeClaim($w$, $p$)
10:             if $result =$ claim_dead then continue
11:             else if $result =$ claim_success then
12:                 UF-SCC$_p$($w$) [recursively explore $w$]
13:             else [$result =$ claim_found]
14:                 while $\neg$SameSet($D_p$.top(), $w$) do
15:                     $w' := D_p$.pop()
16:                     Merge($w'$, $D_p$.top())
17:         RemoveFromList($v'$) [fully explored post($v'$)]
18:     if $v = D_p$.top() then $D_p$.pop() [remove completed SCC]
```

Lines 1 to 4 describe the initialization procedure for the algorithm. Here, we have that MakeClaim(v_0, p) is called for each worker $p \in [1 \dots \mathcal{P}]$. The MakeClaim procedure is the combination of the MakeSet, AddWorker, and HasWorker from Algorithm 16 (Section 5.3.1 discusses its implementation in detail). As a post-condition for MakeClaim(v_0, p) at Line 3 we obtain $p \in$ PSet(v_0). From this we assure that at Line 4, each UF-SCC$_p$ procedure is called for v_0 such that $p \in$ PSet(v_0) (PSet$(v_0) = \{1, \dots, \mathcal{P}\}$).

In the UF-SCC$_p(v)$ procedure, at Line 6 we push state v on the stack D_p. At Line 7, we pick a state from the list of v. By this we imply that $v' \in List(v) \subseteq$ uf$[v]$. In combination with Line 17, we ensure that we never consider the same vertex twice and reduce $|List(v)|$ in each step. From Corollary 4.1 we obtain that $v \rightarrow^* v'$ and $v' \rightarrow^* v$ as states in the uf set must preserve the strong connectivity property.

For Lines 8-16 we consider each successor w for v'. The return value of the MakeClaim(w, p) call is stored in the *result* variable, at Line 9. The specification for MakeClaim(w, p) ensures $\neg$IsDead$(w) \Longrightarrow p \in$ PSet(w) and we obtain the following for the *result* value:

$$result = \begin{cases} \text{claim\_dead} & \Longleftrightarrow \text{IsDead}(w) \\ \text{claim\_success} & \Longleftrightarrow \neg\text{IsDead}(w) \wedge p \notin old(\text{PSet}(w)) \\ \text{claim\_found} & \Longleftrightarrow \neg\text{IsDead}(w) \wedge p \in old(\text{PSet}(w)) \end{cases}$$

By interpreting this, we observe that at Line 10, we ignore w if it is part of a dead SCC. If we have $result =$ claim_success, we have that $\nexists t \in$ uf$[w] : w \in D_p$ and thus we recursively explore w by calling UF-SCC$_p(w)$ in Line 12. In case we have $result =$ claim_success, we acquired the knowledge that w (possibly indirectly) forms a cycle with v (see also Invariant 4.3). Then, at Lines 15 and 16 we unite the states on this cycle via the Merge procedure which we discussed before (it applies the Union procedure and also combines the list cycles).

We assume as a post-condition that, for any v, when UF-SCC$_p(v)$ finishes, we have $List(v) = \emptyset \Longrightarrow$ IsDead(v). This holds as a result from the failed PickFromList call in Line 7. If we take a close look at Lines 8 to 16, we observe that for every successor w of v' we have:

- IsDead(w) as a result of Line 10.
- IsDead(w) as a result of the post-condition for UF-SCC$_p$.
- SameSet(v', w) as a result of Lines 14-16.

Thus, by Predicate 5.1 we have that GlobalDone(v') holds at the start of Line 17. Therefore, we may correctly remove v' globally from $List(v)$. At Line 18 we take that $List(v) = \emptyset$, which implies GlobalDead(v). Line 18 ensures that all (DEAD) SCCs are removed from the stack.

We observe that if UF-SCC$_p(v)$ has finished, all reachable states from v have been reported DEAD as well. Hence, if UF-SCC$_p(v_0)$ finishes, we have found all SCCs reachable from v_0.

5.3 Detailed design

In this section we present a detailed design for the used methods in the UF-SCC algorithm.

5.3.1 The MakeClaim procedure

The MakeClaim procedure is described in Algorithm 19. At Line 6 we observe that only the first worker that called MakeClaim is able to initialize the UF node. During this initialization, we prevent that other workers may interfere. This is achieved by setting *uf-status* to a (temporary) INIT value. All other workers have to wait in Line 13 due to this INIT value. After the initialization is complete, the first worker sets the *uf-status* to LIVE and returns CLAIM_SUCCESS. Lines 14,15 and 16 refer to the procedures from Algorithm 16.

Algorithm 19 The MakeClaim procedure for UF-SCC

```
1: postcondition: $\neg$IsDead$(x) \implies p \in Pset(x)$
2: returns: CLAIM_DEAD $\iff$ IsDead$(x)$
3: returns: CLAIM_SUCCESS $\iff$ $\neg$IsDead$(x)$ $\wedge$ $p \notin old(Pset(x))$
4: returns: CLAIM_FOUND $\iff$ $\neg$IsDead$(x)$ $\wedge$ $p \in old(Pset(x))$
5: procedure MakeClaim$(x, p)$
6:     if CAS(UF$[x]$.$uf$-$status$, UNSEEN, INIT) then
7:         UF$[x]$.$parent := x$
8:         UF$[x]$.$list$-$next := x$
9:         UF$[x]$.$list$-$status :=$ LIVE
10:        UF$[x]$.PSET $:= \{p\}$
11:        UF$[x]$.$uf$-$status :=$ LIVE
12:        return CLAIM_SUCCESS
13:    while UF$[x]$.$uf$-$status =$ INIT ;
14:    if IsDead$(x)$ then return CLAIM_DEAD
15:    if HasWorker$(x, p)$ then return CLAIM_FOUND
16:    AddWorker$(x, p)$
17:    return CLAIM_SUCCESS
```

5.3.2 Picking and removing a state from the list

For a illustration on the list removal procedure, we refer to Figure 5.2(right).

The PickFromList procedure is described in Algorithm 20. Here, the approach is to continuously check if UF$[x]$.*list-status* = TOMBSTONE (Line 4). If so, we look one state ahead (n_1) and check if the same holds for UF$[n_1]$.*list-status*. At Line 8 we observe that UF$[x]$.*list-status* = UF$[n_1]$.*list-status* = TOMBSTONE. We 'remove' entry n_1 from the cycle in Line 10 and let UF$[x]$.*list-next* point to n_2, the subsequent state from n_1. We continue from n_2 and try again. Every time the algorithm iterates, Line 10 is executed, which reduces length of the cycle by one. If we observe that $List(x) = \emptyset$ (at either Line 6 or 9), NULL is returned. In other cases, we find and return some state $x' \in$ UF$[x]$ for which UF$[x']$.*list-status* $\neq$ TOMBSTONE $\iff x' \in List(x)$.

Note that we do not have to take extra caution with possible BUSY states. This is because we merely update the *list-next* pointers for states that have *list-status* = TOMBSTONE.

Algorithm 20 The PickFromList procedure for UF-SCC

```
1: returns: NULL $\iff List(x) = \emptyset$
2: returns: $x' \in List(x) \iff List(x) \neq \emptyset$
3: procedure PickFromList($x$)
4:     $n_1$ := UF[$x$].list-next
5:     while UF[$x$].list-status = TOMBSTONE do
6:         if $x = n_1$ then return NULL
7:         if UF[$n_1$].list-status = TOMBSTONE then
8:             $n_2$ := UF[$n_1$].list-next
9:             if $x = n_2$ then return NULL
10:            UF[$x$].list-next := $n_2$
11:            $x := n_2$
12:            $n_1$ := UF[$x$].list-next
13:        else return $n_1$
14:    return $x$
```

The RemoveFromList procedure is described in Algorithm 21. This procedure merely consists of setting the the list status for x to TOMBSTONE. It may be possible that a MergeLists procedure is taking place at the moment, therefore the while-loop is iterated until we (or another worker) is able to set the *list-status* for x from LIVE to TOMBSTONE. After the procedure finishes, we ensure that UF$[x]$.*list-status* = TOMBSTONE.

Algorithm 21 The RemoveFromList procedure for UF-SCC

```
1: postcondition: $v \notin List(v)$
2: procedure RemoveFromList($x$)
3:     while UF[$x$].list-status $\neq$ TOMBSTONE do
4:         CAS(UF[$x$].list-status, LIVE, TOMBSTONE)
```

5.3.3 The Merge procedure

The Merge procedure is described in Algorithm 22. This consists of locking states x and y, after which the MergeLists and Union procedures can take place. If a lock fails, we obtain that x and y already belong to the same UF set, implying that the post-condition is satisfied and the procedure returns. The post-condition results from these sub-procedures. We argue that the Merge process may be executed lockless. However, this would imply that other schemes have to be designed to prevent race-conditions from occurring. The locking mechanism is adopted for now, to be able to ensure correctness.

The locking mechanism from Algorithm 23 is implemented by locking x and y in a 'lexicographic' order. We compare the hash values for x and y and possibly swap x and y if $x > y$ (Line 6). Then, x is locked in Line 7, after which we check if we actually locked the representative for UF$[x]$. Then, we do the same for y in Lines 9 and 10. If this succeeds, *True* is returned. Otherwise the states are unlocked (set to LIVE) and we try again. In case x and y already belong to the same UF set, *False* is returned at Line 5. An Unlock consists of setting the *uf-status* for a given state back to LIVE.

Algorithm 22 The MERGE procedure for UF-SCC

```
1: postcondition: $\text{UF}[x] = \text{UF}[y] = old(\text{UF}[x]) \cup old(\text{UF}[y])$
2: postcondition: $List(x) = old(List(x)) \cup old(List(y))$
3: postcondition: $Pset(x) = old(Pset(x)) \cup old(Pset(y))$
4: procedure MERGE$(x, y)$
5:     if LOCK$(x, y)$ then
6:         MERGELISTS$(x, y)$
7:         UNION$(x, y)$
8:         UNLOCK$(x)$
9:         UNLOCK$(y)$
```

Algorithm 23 The locking mechanism for the MERGE procedure in UF-SCC

```
1: procedure LOCK$(x, y)$
2:     while $True$ do
3:         $x :=$ FIND$(x)$
4:         $y :=$ FIND$(y)$
5:         if $x = y$ then return $False$
6:         if $x > y$ then SWAP$(x, y)$
7:         if CAS(UF$[x]$.$uf\text{-}status$, LIVE, LOCKED) then
8:             if UF$[x]$.$parent = x$ then
9:                 if CAS(UF$[y]$.$uf\text{-}status$, LIVE, LOCKED) then
10:                    if UF$[y]$.$parent = y$ then
11:                        return $True$
12:                    UF$[y]$.$uf\text{-}status :=$ LIVE
13:            UF$[x]$.$uf\text{-}status :=$ LIVE

14: procedure UNLOCK$(x)$
15:     UF$[x]$.$uf\text{-}status :=$ LIVE
```

Algorithm 24 The MERGELISTS procedure for UF-SCC

```
1: procedure MERGELISTS$(x, y)$
2:     $x :=$ LOCKLIST$(x)$
3:     $y :=$ LOCKLIST$(y)$
4:     SWAP(UF$[x]$.$list\text{-}next$, UF$[y]$.$list\text{-}next$)
5:     UF$[x]$.$list\text{-}status :=$ LIVE
6:     UF$[y]$.$list\text{-}status :=$ LIVE

7: procedure LOCKLIST$(x)$
8:     while UF$[x]$.$list\text{-}status \neq$ BUSY do
9:         $x :=$ PICKFROMLIST$(x)$
10:        CAS($uf[x]$.$list\text{-}status$, LIVE, BUSY)
11:    return $x$
```

The MergeLists procedure from Algorithm 24 describes how two disjunct lists are merged with each other. This process is also illustrated in Figure 5.2 (left). Here, we must first search for two non-TOMBSTONE list entries as we would otherwise not combine the cycles for both lists. The LockList procedure ensures that we obtain such a state. Furthermore, we set the status of this entry to BUSY to prevent any other worker from removing it (by calling RemoveFromList) from the cycle during the MergeLists procedure.

Note that because we have locked the UF nodes as well, no other MergeLists or LockList procedure can take place on this set (hence there can be only one BUSY entry in each list). Also, the MergeLists procedure is executed by Merge *before* the Union procedure. This implies that UF$[x]$ and UF$[y]$ are strictly disjoint from each other. We can therefore conclude that $List(x)$ and $List(y)$ cannot be empty during the MergeLists procedure.

5.4 Discussion

The proposed algorithm is a novel approach to detect SCCs in parallel. We show that it is provably correct and analyze the complexity.

5.4.1 Outline of correctness

This section provides an outline for the correctness proof. For a full proof of correctness, we refer the reader to Appendix A.

The correctness for this algorithm mainly is implied from Observation 5.2. We first state that for every call to UF-SCC$_p(v)$, we have that $p \in$ Pset(v) and as a consequence, $\forall v \in D_p : p \in$ Pset(v). We also infer that after UF-SCC$_p(v)$ finishes, we have GlobalDead(v). Then, we can observe that the property $\forall v : p \in$ Pset$(v) \implies$ GlobalDead$(v) \vee \exists v' \in$ UF$[v] : v' \in D_p$ holds valid for many of the lines from the algorithm.

By induction we are able to show that $\forall v, w : v \in$ UF$[w] \implies v \to^* w \to^* v$ holds as an invariant. *Completeness* follows trivially from this invariant. For *soundness* we show that every reachable state is handled correctly by the algorithm. Thus, we show that for any $v_0 \to v$, all successors from v and all states from $List(v)$ are considered. We can also conclude that the algorithm terminates by showing that for every iteration on $v' \in List(v)$, we eventually remove v' from the list and reduce the set of states that are not GlobalDone.

5.4.2 Complexity

This section examines the time- and space complexity of the UF-SCC algorithm, presented in Algorithm 18. Concerning the time complexity, we assume that the locking procedure operates in constant time. Also, we say that the number of workers is fixed, meaning that $\mathcal{P}$ is regarded as a constant value.[1]

Time complexity. First, consider that the UF-SCC$_p$ procedure can be called at most $|\mathcal{V}|$ times for every worker p. This is implied from the MakeClaim procedure; it only returns CLAIM_SUCCESS if it successfully added p to the Pset. We analyze the time complexity for the sub-procedures as follows and conclude afterwards.

The PickFromList procedure (see Algorithm 20) can return a state a maximum of $|\mathcal{V}|$ times in total for each worker, independent of the number of recursive UF-SCC calls. This is because the state is globally removed from the list at the end of each iteration. The PickFromList procedure itself executes in at most $\mathcal{O}(|\mathcal{V}|)$ time (in case the list contains a large sequence of TOMBSTONE states). We show that the *amortized* complexity for this procedure is $\mathcal{O}(1)$.

First note that if a worker encounters two successive TOMBSTONE states, the *list-next* pointer is updated to 'remove' the latter state from the list. Once removed, the latter state will never be visited again by

[1]In theory, we may have a polynomial (with respect to $|\mathcal{V}|$) number of workers. However, for all practical instances we can assume that the number of workers remain limited.

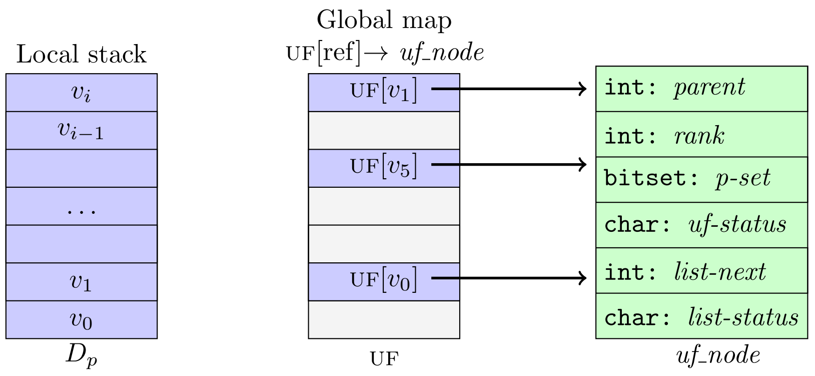


Figure 5.3: Representation of the used data structures for the UF-SCC algorithm.

the worker (nor by any other). Observe that at most we have $|\mathcal{V}| - 1$ of such removals in total. In other situations – where the current or the next state is LIVE – the state is returned in constant time. Therefore, in the complete UF-SCC algorithm, the PICKFROMLIST procedure takes a total of $\mathcal{O}(|\mathcal{V}|)$ time. Because this procedure is called at most $|\mathcal{V}|$ times for a single worker, it executes in amortized $\mathcal{O}(1)$ per call.

Every successor of each state from PICKFROMLIST is considered in the for-loop, thus we have $\mathcal{O}(|\mathcal{V}|+|\mathcal{E}|)$ iterations. The MAKECLAIM procedure (see Algorithm 19) executes in amortized quasi-constant time. Here, we have an amortized quasi-constant time for finding the representative of the UF set. Observe that the modifications to PSET by the HASWORKER and ADDWORKER procedures can also be implemented in quasi-constant time, since only the bit-value for the worker has to be addressed (and updated).

The while-loop for merging a cycle executes in amortized quasi-constant $\mathcal{O}(\mathcal{P})$ time. Since UF-SCC$_p$ is called at most $|\mathcal{V}|$ times, the stack D_p can only be popped $|\mathcal{V}|$ times. Thus, the while-loop may be iterated at most $|\mathcal{V}|$ times. The SAMESET operation is executed in amortized quasi-constant time. The MERGE procedure executes in quasi-constant $\mathcal{O}(\mathcal{P})$ time. This results from combining the two PSETs (note that a single *Fetch&Or* instruction is sufficient for up to 128 workers). The parent update and merging of two lists takes place in amortized quasi-constant time, following the PICKFROMLIST procedure. Note however that the MERGE procedure only unites two states at most $|\mathcal{V}| - 1$ times *globally*. Therefore, if we consider its impact on the complete UF-SCC$_p$ algorithm, we have an amortized total quasi-linear complexity of $\mathcal{O}(\mathcal{P} \cdot |\mathcal{V}|)$.

If we combine the time complexities for the sub-procedures (and use a fixed number of workers) we obtain a *quasi-linear*[2] time complexity of $\mathcal{O}(\mathcal{P} \cdot (|\mathcal{V}| + |\mathcal{E}|))$. This results from considering each state and successor by every worker in the for-loop.

Note that we expect to see scalability from the fact that states are removed *globally* from the list after being fully explored. In case these states are scheduled ideally over the workers (i.e. no two workers explore the same state) in combination with an equal distribution of MERGE calls, we conjecture that the best-case time complexity is bounded by $\mathcal{O}(\frac{|\mathcal{V}|+|\mathcal{E}|}{\mathcal{P}})$ and the diameter of the graph.

Space complexity. Concerning the space complexity, there are two aspects that play a role (see Figure 5.3). First, we have the local search stack (D_p) for each worker. In a worst-case scenario – with an unfortunate successor order causing one large path – the algorithm possibly cannot avoid recursively visiting each state (note that this is not bounded by the diameter of the graph). Thus the stack takes at most $|\mathcal{V}| \cdot H$ space, where H is size of the (pointer to a) state (for which we assume to take up 64 bits).

[2]The *quasi-* aspect origins from operations on the UNION-FIND structure, for which the amortized complexity is the inverse Ackermann function (see also Section 2.3.2).

The second aspect is the size of the UNION-FIND set. We assume that this is implemented by a map structure (as depicted in Figure 5.3). We must allocate at least $|\mathcal{V}|$ entries (for each state one). The size of a *uf-entry* is bounded by (respectively from top to bottom) $H + H + \mathcal{P} + 8 + H + 8 \leq \mathcal{P} + 4H$ bits (depending on the implementation).

The algorithm uses at most $\mathcal{P} \cdot |D_p| + |\text{UF}| = \mathcal{P} \cdot |\mathcal{V}| \cdot H + |\mathcal{V}| \cdot (\mathcal{P} + 4H) = |\mathcal{V}| \cdot (\mathcal{P} \cdot H + \mathcal{P} + 4H)$ space.

Chapter 6

Experiments

This chapter discusses experiments with the algorithm. First, we discuss the setup used for experimentation. Here we also focus on implementation aspects. Next, we test our implementation on BEEM [48] models and analyze the obtained results. Thereafter, we perform some additional experiments to test various properties of the UF-SCC algorithm.

6.1 Experimental setup

In this section, we discuss the experimental setup for evaluating the UF-SCC algorithm. We talk through the implementation for the algorithm, and its competitors. Section 6.1.2 covers the configuration used for the experimentation: the environment, how an experiment is performed, and which metrics are observed by the experiments.

6.1.1 Implementation

We have implemented the following algorithms in the multi-core library of the LTSmin toolset [31]:

- The UF-SCC algorithm, as presented in Algorithm 18. This includes the extended parallel Union-Find data structure specifically designed for the algorithm.
- Tarjan's algorithm [57], as presented in Algorithm 4.
- Renault's algorithm [53], as presented in Algorithm 14. This includes a Union-Find data structure, simplified from the one used in the UF-SCC algorithm.

We have chosen to re-implement Renault's algorithm instead of relying on the available one. By re-implementing Renault's algorithm in the LTSmin environment, we expect to mitigate inconsistencies caused by the programming language and possible optimizations.

Unfortunately, we were not able to implement Lowe's algorithm, nor were we capable of using his provided implementation. Because Lowe did not perform experiments in an on-the-fly context we also cannot reliably use his results. However, since an on-the-fly algorithm is more restricted we can assume that if UF-SCC performs better than Lowe, the same should remain valid for an on-the-fly implementation of Lowe's algorithm.

We highlight a couple of aspects concerning the implementation of the UF-SCC algorithm.

Next-State. LTSmin makes use of an implicit graph representation. This means that we can request successor states by calling a Next-State function (see Section 2.3.1). The LTSmin toolset already applies measures to randomize the order of successors. When multiple workers call this Next-State function, the order of the returned successors is permuted for each worker, which implies that these workers will visit the successors in a different order.

Stack usage. The algorithms are all implemented iteratively. This is due to the general assumption that iterative programs execute faster. Instead of querying successors one by one, we maintain a pointer to the next sibling in the stack. We query all successors all at once and store these in a stack frames.

Union-Find structure. The UNION-FIND structure is designed as an array (which we also refer to as UF), for which the index represents the hash of the state (see also Figure 5.3). Each entry consists of the following fields:

- *parent* : A reference to the parent state for the UNION-FIND set. If this state is the representative for the set, it references itself. When recursively looking up UF[*parent*], we eventually find the unique representative of the set.
- *rank* : The rank or height for the UNION-FIND tree. This is used in the UNION procedure to make sure that the 'smaller' tree is pointing to the representative of the 'larger' tree.
- *p-set* : A 64 bit value that contains the worker IDs for each worker (there is one bit reserved for each worker). By means of *Compare&Swap* operations, a worker bit is added to the set.
- *uf-status* : The status for a UF state. This can be any of the following: {UNSEEN, INIT, LIVE, LOCKED, DEAD}.
- *list-next* : A reference to the successive list entry, residing in the same set. As we explained in Section 5.1.2, recursive lookups for the *list-next* value ensures that we eventually return to this state (assuming that this entry's status is LIVE).
- *list-status* : The status for the UF list entry. This can be any of the following: {LIVE, BUSY, TOMBSTONE}.

Note that RENAULT's algorithm makes use of the standard UNION-FIND structure; only containing the *parent* and *rank* fields. RENAULT's algorithm originally uses a lockless UNION-FIND structure. We have also implemented the algorithm using the same locking approach as we did for UF-SCC (see Section 6.4.2 for a comparison of the two approaches).

6.1.2 Configuration

With our experiments, we want to assess the efficiency of the UF-SCC algorithm. With efficiency, we are particularly interested in comparing the total execution time used for a given number of workers to estimate scalability of the different algorithms.

Architecture. The 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.

Performing an experiment. The LTSMIN toolset contains a number of frontend-backend combinations by means of PINS2PINS modules [31]. Considering that the BEEM database provides DVE models, we make use of the `dve2lts-mc` module for testing our multi-core algorithms on the BEEM models. We perform a test as follows:

```
dve2lts-mc --strategy=algorithm --threads=workers -s28 model
```

Here, we state our algorithm (`ufscc/renault/tarjan`), the number of workers we want to use ($\in [1, \ldots, 64]$) and the model to be analyzed. We make use of a fixed hash table size of 2^{28} with the `-s28` option. We performed each test 5 times. The average time used and the 95% confidence interval is calculated from the results. Where suitable, we indicate this confidence interval by means of error bars. For random graphs, which we construct in Section 6.3.1, we directly use the PINS interface (we use `pins2lts-mc` instead of `dve2lts-mc`).

Measurements. Besides the measuring the execution times, we also analyze other properties. These consist of: the number of states explored (in total, and by each worker); the number of transitions encountered; the number of SCCs; the number of re-tried explorations and transitions; the number of union operations performed; and the total memory used. A number of these metrics were not observed to significantly vary throughout the experiments and were not reported.

For the remainder of the chapter, we use UF-SCC$_1$ to refer to the UF-SCC algorithm, performed on 1 thread (this holds similarly for TARJAN and RENAULT). With RENAULT$_p$ we refer to RENAULT's algorithm, performed on p threads (the same number of threads that the UF-SCC algorithm uses).

6.1.3 Validation

We evaluated the implementation's correctness using several techniques.

Manual verification. We constructed a proof of the UF-SCC algorithm (see Appendix A) to ensure that the algorithm itself is correct. Furthermore, we thoroughly examined each sub-procedure closely to make sure that predefined invariants and pre- and post-conditions were met by the implementation.

Verification in the implementation. Furthermore, we implemented assertions on various components. We rigorously checked if at all times the status (both the *uf-status* and *list-status*) of a state is valid for the to-be-executed code. This is accompanied by assertions on pre- and post-conditions. For instance, we check that SAMESET(a, b) holds when the UNION-FIND procedure finishes. We have to consider all possible interleavings when setting up the assertions. Therefore some checks cannot (easily) be performed. A number of debug methods were designed to provide useful information on for instance the information stored in UF sets.

Experimental verification. We also analyzed the output data. For every performed experiment, we checked the number of explored states, transitions and the number of SCCs (by automatically comparing these with a base). When examining errors in the implementation, we made use of the GNU Debugger (`gdb`) to analyze for instance where a deadlocks could occur (this tool is able to keep track of multiple threads). For experiments on small-sized input graphs we also produced graphs described in the DOT language. We depicted the search stacks, SCCs and the *uf-status* for states by using various colours and shapes. This visual representation was often useful for detecting problems.

6.2 Experiments on BEEM models

This section discusses the experiments carried out on BEEM [48] models.[1] We first discuss which models are used for the experiments. Hereafter the experiments performed on these models are analyzed. Finally, we summarize the observations from the experiments.

6.2.1 Models used

From the BEEM database, we made an initial selection of the models containing at least 100,000 and at most 10,000,000 states. Initial benchmarks suggested that models containing fewer than 100,000 states are generally *solved* (decomposing all reachable SCCs) in a fraction of a second. We chose to ignore these models for the following reasons:

- These models are not representative for practical applications of scalable SCC algorithms; if we are already able to solve a particular model in a fraction of a second, the demand for improving the performance is of less importance compared to more time-consuming models. Moreover, one of the main purposes for scalable algorithms is to be able to cope with larger input data.

[1]The BEEM models are available at http://spinroot.com/spin/beem.html.

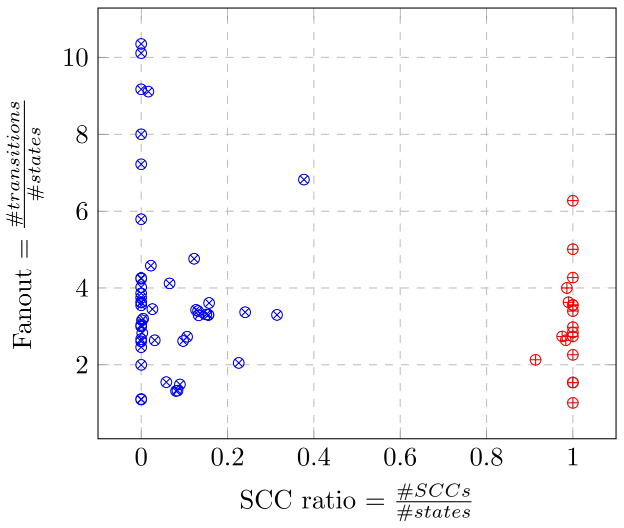


Figure 6.1: A scatter plot, illustrating statistics for the BEEM models.

- Because the absolute time used is of small scale, the experiments are more subject to external influences. The initial overhead introduced by the experimental setup is of relatively larger effect to the end-results compared to more time-consuming models.

The models containing more than 10,000,000 states were dismissed from the selection due to practical reasons. Initial benchmarks showed that experiments on these models generally exceeded our maximum time limit of 10 minutes. With this maximum time limit in place, we performed all experiments in just under three days.

Table 6.1 provides an overview of the models used in experimentation. Here, we analyzed the number of reachable states, the number of reachable transitions and number of reachable SCCs in each model. The remaining statistics are derived from combinations of these three mentioned classes. The highlighted cells and text respectively refer to models that were selected for in-depth analysis and observed distinguishing classes.

By analyzing the used models more closely, we observed that a large part of the models either contain a relatively large ratio of SCCs or a relatively small one, compared to the number of states. The scatter plot in Figure 6.1 expresses this observation illustratively. We observed a clear gap in the SCC ratio of the models. We indicated these two 'classes' with $\otimes$ and $\oplus$, we refer to these as blue and red respectively. Here, the red class contains a high SCC ratio. This infers that the number of SCCs is (almost) equal to the number of states. This means that on average, an SCC consists of only one state (as reflected in Table 6.1). As such SCCs are considered trivial, we refer to models of this class as *trivial models*. Conversely, the blue class contains fewer and therefore larger SCCs. We refer to models of this class as *non-trivial models*.

In Figure 6.1 we observed as well that the SCC ratio is not directly correlated to the fanout. A large fanout indicates a large amount of inter-connectivity, hence we expect a larger likelihood of 'back-edges' and thus larger and fewer SCCs on average. For models with a fanout of 7 or fewer, this notion did not seem to be applicable. However, we did observe that the models containing the largest fanout values (more than 7) also contain few SCCs.

6.2.2 Results

With the models being categorized in Section 6.2.1, we first verified whether there are differences in the algorithm's performance on both classes. Figure 6.2 presents the results for a speedup comparison between the UF-SCC algorithm with its sequential version, for both the trivial and non-trivial models.

Table 6.1: BEEM models used for experimentation.

model	#states	#transitions	fanout	#SCCs	avg. SCC size	#SCCs/#states
adding.2	836838	1289748	1.54	836838	1	1
adding.3	1894376	2921634	1.54	1894376	1	1
adding.4	3370680	5201282	1.54	3370680	1	1
adding.5	5271456	8135364	1.54	5271456	1	1
adding.6	7609684	11746148	1.54	7609684	1	1
anderson.1	352664	704302	2	20	17633.20	5.67E-05
at.3	1711620	6075360	3.55	1	1711620	5.84E-07
at.4	6597245	25470140	3.86	1	6597245	1.52E-07
bakery.4	157003	411843	2.62	15260	10.29	0.0972
bakery.5	7866401	27018304	3.43	1000072	7.87	0.127
blocks.3	695418	2094753	3.01	2	347709	2.88E-06
bopdp.3	1040953	2747408	2.64	1023793	1.02	0.984
bridge.3	838864	1896973	2.26	838864	1	1
brp.3	996627	2047490	2.05	225202	4.43	0.226
cambridge.5	698912	3199507	4.58	15689	44.55	0.0224
cambridge.6	3354295	9483191	2.83	8413	398.70	0.00251
elevator.3	416935	1025817	2.46	2	208467.50	4.80E-06
elevator.4	888053	2320984	2.61	6	148008.83	6.76E-06
elevator2.2	179200	1036800	5.79	1	179200	5.58E-06
elevator2.3	7667712	55377920	7.22	1	7667712	1.30E-07
extinction.3	751930	2669267	3.55	751930	1	1
extinction.4	2001372	7116790	3.56	2001372	1	1
firewire-link.7	399598	1096535	2.74	389698	1.02	0.975
fischer.3	2896705	12280586	4.24	1	2896705	3.45E-07
fischer.4	1272254	4609671	3.62	1	1272254	7.86E-07
fischer.6	8321728	33454192	4.02	1	8321728	1.20E-07
frogs.3	760789	766119	1.01	760789	1	1
hanoi.2	531441	1594320	3	1	531441	1.88E-06
iprotocol.3	1013456	3412754	3.37	244098	4.15	0.241
krebs.3	238876	1020147	4.27	238876	1	1
krebs.4	1047405	5246321	5.01	1047405	1	1
lamport-nonatomic.4	1257304	5360727	4.26	1	1257304	7.95E-07
lamport.2	110920	303058	2.73	11800	9.40	0.106
lamport.5	1066800	3630664	3.40	140496	7.59	0.132
lamport.6	8717688	31502176	3.61	1371572	6.36	0.157
lann.3	1832139	8725188	4.76	224688	8.15	0.123
lann.4	966855	3189852	3.30	304189	3.18	0.315
lann.5	993914	3604487	3.63	983962	1.01	0.990
leader-filters.5	1572886	4684565	2.98	1572886	1	1
loyd.2	362880	967681	2.67	2	181440	5.51E-06
mcs.3	571459	2077384	3.63	55	10390.16	9.62E-05
mcs.6	332544	1329920	4	327923	1.01	0.986
msmie.3	134844	200614	1.49	12099	11.15	0.0897
msmie.4	7125441	11056210	1.55	414699	17.18	0.0582
needham.3	206925	567099	2.74	206925	1	1
needham.4	6525019	22203080	3.40	6525019	1	1
peg-solitaire.4	873326	5473290	6.27	873326	1	1
peterson.2	124704	399138	3.20	574	217.25	0.00460
peterson.3	170156	538509	3.16	309	550.67	0.00182
peterson.4	1119560	3864896	3.45	29115	38.45	0.0260
phils.4	340789	3123558	9.17	1	340789	2.93E-06
phils.5	531440	4251516	8	2	265720	3.76E-06
production-cell.3	822612	2496342	3.04	1	822612	1.22E-06
production-cell.4	340685	968176	2.84	340685	1	1
protocols.4	439245	1454834	3.31	66214	6.63	0.151
protocols.5	996345	3272786	3.29	132706	7.51	0.133
public-subscribe.2	1846603	6087556	3.30	287436	6.42	0.156
public-subscribe.3	1846603	6087556	3.30	287436	6.42	0.156
public-subscribe.4	1846603	6087556	3.30	287436	6.42	0.156
reader-writer.3	604498	4125562	6.82	227893	2.65	0.377
rether.3	305334	334516	1.10	2	152667	6.55E-06
rether.4	1157052	1535386	1.33	97332	11.89	0.0841
rether.5	3017044	3302351	1.10	2	1508522	6.63E-07
rether.6	5919694	7822384	1.32	478204	12.38	0.0808
rether.7	4789409	5317199	1.11	2	2394704.50	4.18E-07
rushhour.3	156723	1583980	10.11	1	156723	6.38E-06
rushhour.4	327675	3390234	10.35	1	327675	3.05E-06
schedule-world.2	1570340	14308706	9.11	26018	60.36	0.0166
sokoban.2	761633	2012841	2.64	23912	31.85	0.0314
sorter.3	1288478	2740540	2.13	1177164	1.10	0.914
szymanski.3	1128424	4234041	3.75	8	141053	7.09E-06
telephony.3	765379	3155026	4.12	50329	15.21	0.0658

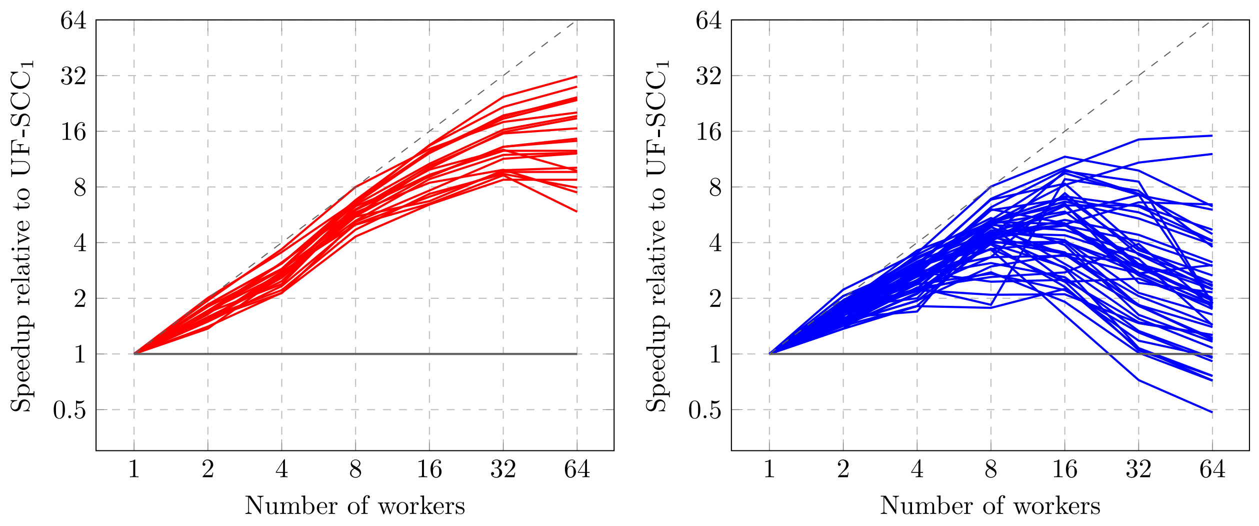


Figure 6.2: Speedups of the UF-SCC algorithm compared to its sequential version, for *trivial* (left) and *non-trivial* (right) BEEM models.

Trivial models. For trivial models, we generally observe a seemingly linear increase in the speedup when increasing the number of workers, more on this later. Closer analysis shows that the performance gains reduce when increasing from 32 to 64 workers. We provide two possible explanations for this phenomenon:

- We keep track of the worker IDs in the Union-Find structure. This set of worker IDs is implemented as a 64 bit value. Whenever a worker adds its worker ID to this set, its worker bit is set by a *Fetch&Or* operation to ensure that it is atomically set. If multiple workers try to add their worker IDs to the same state, a (slight) performance drop may be observed since multiple *Fetch&Or* operations may take place at the same time.

- It might also be the cause of a known issue in the used architecture – where internal memory gets shared between workers for more than 32 workers. This should however not explain why in some cases the algorithm performs worse with 64 workers compared to 32.

Non-trivial models. As can be observed in Figure 6.2, the results on non-trivial models are significantly different from those obtained on the trivial models. This suggests that we generally were just in classifying the models in two categories.

While the results are somewhat cluttered due to the large number of models, one could observe a general trend that the results follow. This being a linear increase towards 8 workers (for some this continues until 16 workers), after which it starts to decline in performance significantly – in some instances, the 64 worker version performs even worse than the sequential one. This peculiar result is analyzed in Section 6.3.

The two outliers (with a speedup larger than 10 for 64 workers) are `iprotocol.3` and `sokoban.2`, with respectively an SCC ratio of 0.241 and 0.0314 – the relatively high ratio possibly explains the results for `iprotocol.3`. We assume that these models follow properties that are similar to those of the identified trivial models; we did not perform an in-depth analysis on these models to validate the hypothesis.

Selected models. We take a closer look at specific models from both the trivial and non-trivial classes. For all trivial models from Table 6.1, we selected only those for which the sequential Tarjan algorithm takes more than 5 seconds. For the non-trivial models, we only selected those for which the sequential Tarjan algorithm takes more than 25 seconds. As a result of this selection, we have five representative trivial and

Table 6.2: Speedups for UF-SCC and Renault on BEEM models, relative to Tarjan's algorithm.

model	Tarjan (sec)	speedup Renault 1	8	64	speedup UF-SCC 1	8	64
`adding.5`	7.00	0.78	4.14	9.33	0.57	3.87	13.73
`adding.6`	10.59	0.79	5.12	11.03	0.60	4.34	18.26
`extinction.4`	7.25	0.80	5.25	18.13	0.58	3.82	15.10
`needham.4`	17.82	0.79	6.27	25.10	0.67	4.40	22.28
`peg-solitaire.4`	5.88	0.82	5.03	17.29	0.54	4.00	13.67
`at.4`	30.43	0.81	0.81	0.73	0.89	4.20	0.66
`bakery.5`	30.63	0.81	0.88	0.62	0.82	4.63	1.00
`elevator2.3`	50.90	0.86	0.83	0.59	0.84	4.69	0.80
`fischer.6`	38.61	0.82	0.81	0.73	0.94	3.21	0.42
`lamport.6`	29.56	0.78	2.24	2.67	0.73	4.09	3.35
avg. sel. trivial	8.92	0.80	5.12	15.19	0.59	4.08	16.31
avg. trivial	3.076	0.67	3.56	7.52	0.48	2.86	7.00
avg. sel. non-trivial	35.21	0.82	1.01	0.88	0.84	4.13	0.94
avg. non-trivial	4.85	0.71	1.03	0.96	0.60	2.55	1.25
avg. total	4.24	0.70	1.49	1.71	0.57	2.64	2.06

non-trivial models, that we hypothesize to provide the most consistent results (these models take the most time, hence external influences play a smaller role). Figure 6.3 illustrates comparative speedups with the sequential UF-SCC and Tarjan algorithms, as well as a comparison with Renault's SCC algorithm. With respect to Figure 6.1, we regard the results representative for their classes.

Table 6.2 shows how the speedups from UF-SCC and Renault relate to each other. This table also reflects the results back on the rest of the dataset. To prevent particular results from skewing the average, we calculate the average by taking the geometric mean (as opposed to the arithmetic average). This implies taking the n^{th} root for the product of the values.

We illustrate four particular cases in Figure 6.4. Here, the absolute times for the three algorithms are provided.

Scalability for trivial models. We refer to the three leftmost diagrams from Figure 6.3. In the comparison with UF-SCC$_1$ we observe that the speedups remain seemingly close to the dotted line. This line represents $y = x$, and suggests the ideal speedup attainable. Taking note that all of the examined models only contain trivial SCCs, the algorithm basically represents a reachability algorithm. Compared to its sequential version, with 8 workers on average a speedup of 6.92 is obtained ($\frac{4.08}{0.59}$). This practically linear scaling drops gradually. For respectively 16, 32 and 64 workers, we have found speedups of 13.28, 21.67 and 27.61.

When compared to Tarjan$_1$, the speedups are less impressive. this is due to the fact that the sequential Tarjan algorithm is 1.69 times faster on average ($\frac{0.59}{1}$). It is to be expected that the Tarjan algorithm performs better sequentially, because of the overhead in using the extended Union-Find structure and because of the synchronization measures taken to prevent data races. However, this still adds up to 6.20 seconds on average (and 3.20 seconds when considering all models).

Relative to Renault's algorithm, generally UF-SCC consistently performs worse. With 8 workers, Renault performs 1.25 times better than UF-SCC ($\frac{5.12}{4.08}$). We expected Renault's algorithm to perform better, simply because it does not (need to) communicate intermediate results from an SCC. Since all SCCs are trivial, there is no need for need for this communication overhead. Additionally, the UF-SCC algorithm has to store and check for worker IDs in the Union-Find structure. When there are more than 8 workers, for the `adding.5` and `adding.6` examples, the UF-SCC algorithm 'suddenly' gains performance relative to

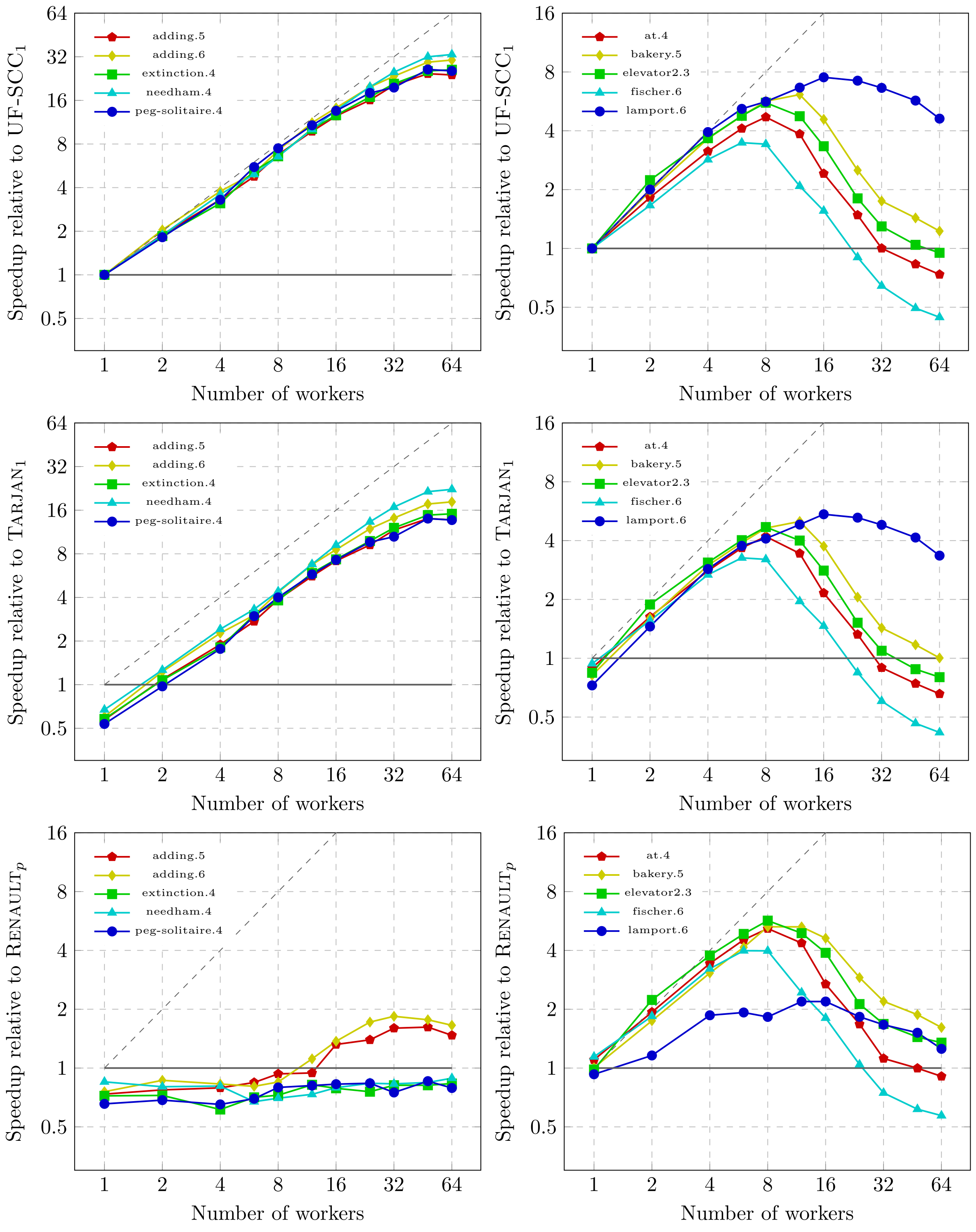


Figure 6.3: Speedups of the UF-SCC algorithm on selected models, relative to: (top) its sequential version; (middle) Tarjan's sequential algorithm; (bottom) Renault's SCC algorithm. Results from trivial models are presented on the left, and results from non-trivial models are presented on the right.

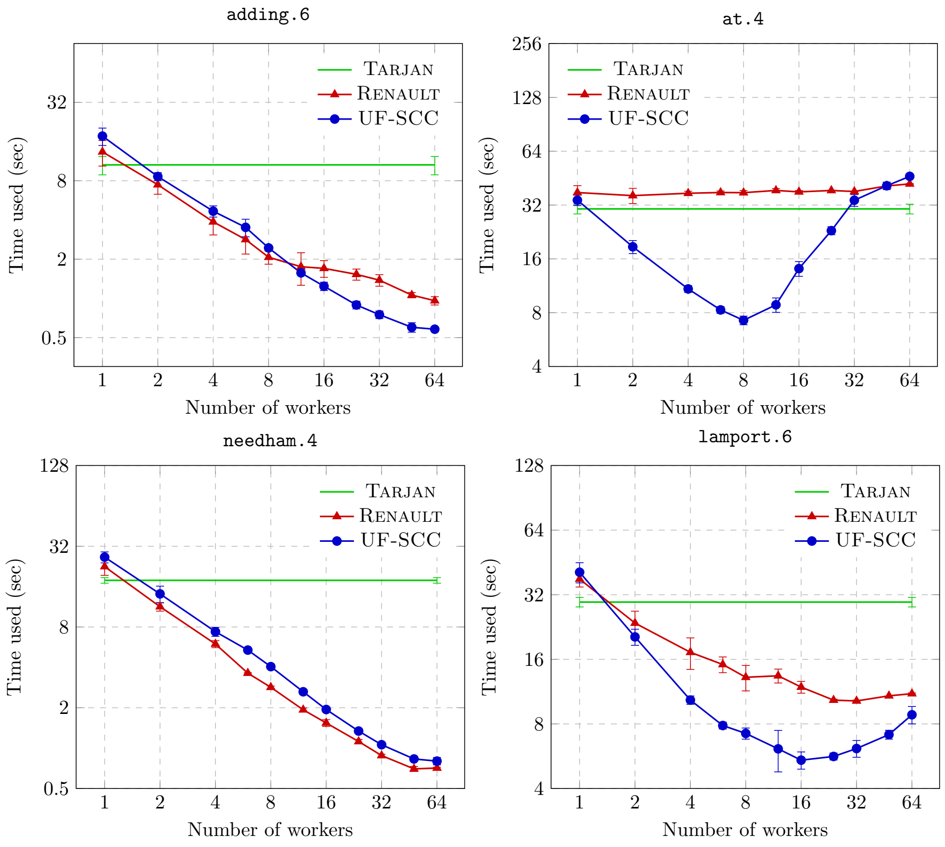


Figure 6.4: Absolute time usage for Tarjan, Renault and UF-SCC on four BEEM models.

Renault. In Figure 6.4 we observe that this is caused by Renault's algorithm, because its performance with 8 workers is comparable to that of 16 workers while it otherwise continuously improves. The most likely cause for this effect is that these specific models are constructed in a peculiar way, such that the scheduling algorithm causes multiple workers to take the same paths. In 'normal' circumstances, the results of UF-SCC and Renault tend to follow a similar curve as `needham.4`, which can be seen in Figure 6.4.

Scalability for non-trivial models. For the non-trivial models, we observe that the UF-SCC algorithm achieves peek performance between 8 or 16 workers and then gradually worsens. Until 8 workers, the results are somewhat comparable to those of the trivial models. On average, we obtain a speedup of 4.92 ($\frac{4.13}{0.84}$) for 8 workers relative to UF-SCC$_1$ or 4.13 relative to Tarjan.

From the results it is not clear where the sudden performance drop origins from. This is further investigated in Section 6.3. Interestingly, a similar performance drop can observed for Renault's algorithm in Figure 6.4. This indicates that it is likely the result of contention on the (basic) Union-Find structure.

Each of the five selected models also seem to peak at different speedups. This does not seem to be a result of the SCC ratios, since those of `bakery.5` and `fischer.6` are comparable (respectively 0.127 and 0.157), while their peeks differ significantly. A closer observation shows that the peek height correlates with the speedups for fewer workers. For example, `lamport.6` achieves a higher speedup compared to `fischer.6` for all number of workers, and this generally holds for the other models as well.

When analyzing the speedup in relation with Tarjan's algorithm, we observe a similar situation as we had for the trivial models. The sequential performance for Tarjan is on average 1.19 times faster compared to UF-SCC$_1$. Note however that is significantly less affecting than we observed in the trivial models (1.69), and this same trend holds for all tested BEEM models (on average 1.67 compared to 2.08). This means that, at least in the sequential case, the UF-SCC algorithm performs worse in terms of reachability (compared to Tarjan) than it does in terms of decomposing SCCs. Note that observation cannot be made for Renault's algorithm. This might imply that the UF-SCC algorithm is more efficient in dealing with non-trivial SCCs compared to Tarjan (and Renault).

For the comparison with Renault we observe similar results as we did in the comparison with Tarjan. The most striking difference is `lamport.6`. Figure 6.4 best explains what is going. We observe here for `lamport.6` that Renault's algorithm performs similar to UF-SCC; with more workers it performs better than Tarjan (while it does not perform as well as UF-SCC). The `at.4` model is representative for the other selected models and experiments on this model tell a different story. Here, UF-SCC is following the same pattern as we observed previously, but Renault behaves differently. It does not gain any performance when the number of workers increases. On the contrary, its performance remains practically the same. This can be explained by the observation that most of these models only contain one SCC, whereas `lamport.6` has an SCC ratio of 0.157. Since Renault only communicates dead SCCs, for the models containing one SCC this is the same as not communicating at all.

Failures. In some instances (for both trivial and non-trivial models) we observed failures. These events were rare and therefore difficult to reproduce. Moreover, these failures only seemed to occur for 64 workers, suggesting that the error is not obvious. With assertion statements we are able to catch these failures, but we have not yet been able to locate the error. See also Section 6.1.3 for a description of the validation techniques that were used.

6.2.3 Conclusions

From the experiments on BEEM models, we observed a number of interesting results for the UF-SCC algorithm, Renault's algorithm, and as well for the BEEM models. These are summarized as follows:

- A large part of the analyzed BEEM models contain either one or few (large) SCCs, or many (small-/trivial) SCCs. These models were categorized to be respectively non-trivial and trivial.

- For trivial models a performance drop can be observed when increasing the number of workers from 32 to 64.
- The UF-SCC algorithm scales almost linearly until 8 workers (5.96 speedup compared to UF-SCC$_1$), but this increase in speedup drops gradually to 14.58 for 64 workers.
- RENAULT's algorithm scales similarly and better for trivial models, compared to UF-SCC. Though, there are instances for which UF-SCC eventually gains the upper hand.
- Until 8 workers, the UF-SCC algorithm performs slightly worse on non-trivial models than on trivial ones (a speedup of 4.25 compared to 5.96 for trivial models, compared to UF-SCC$_1$).
- The maximum speedup for UF-SCC is generally achieved between 8 and 16 workers, for non-trivial models. After this point, the speedup drops on average until it is 2.08 for 64 workers. This is further analyzed in the remainder of this chapter.
- The UF-SCC algorithm (and RENAULT for trivial models) gained higher speedups for the models that took more time to explore.
- For all models, UF-SCC performs worse sequentially compared to TARJAN. However, this less applicable for non-trivial models.
- Perhaps the most important result, the UF-SCC algorithm is capable of gaining performance on non-trivial models (for eight workers it performs up to four times faster than the best sequential approach) whereas RENAULT does not.

6.3 Experiments on random models

This section discusses the experiments carried out on synthetic random models. We first discuss what aspects we considered for constructing models. Then, we analyze how the UF-SCC algorithm performs on the constructed models. Finally, we summarize the observation from the experiments.

6.3.1 Models used

Based on the observations from Section 6.2, the models that we want to construct should adhere to the following specification:

- The models should contain between 100,000 and 10,000,000 states. As also discussed in Section 6.2.1, smaller models are more subject to external influences and larger models are not practically feasable to analyze due to time restrictions.
- The models should contain (large) non-trivial SCCs, since we would like to analyze the performance of UF-SCC on handling partially discovered SCCs. This is especially important considering the observed significant performance drop in non-trivial models (see Figure 6.3).
- The models should be constructed such that the algorithm's performance is not affected by the ordering of successors and paths to follow. In Section 6.2.2 we observed a couple of models that behaved uncommon compared to what we regard as the 'standard', for example the `adding.6` model caused unexpected behavior from RENAULT's algorithm (see Figure 6.3).

We constructed a specification for randomly generated models. Here, the input is the *number of states* and the desired *fanout*, and also a *seed* value for consistent results. The graph is then implicitly formed by the implementation of a NEXT-STATE function. This NEXT-STATE function is presented in Algorithm 25. Here, Line 3 ensures that the 'random' results are consistent for every NEXT-STATE call. When we refer to a randomly generated model, with n states and a fanout of f, we do this with `rnd-n-f`.

Algorithm 25 NEXT-STATE function for randomly generated models.

```
returns: a set of uniformly distributed successor states
procedure NEXT-STATE($state, n\_states, fanout, seed$)
    $rnd$ := SET_SEED($state + (seed \cdot n\_states)$) [sets the seed unique for each $state$ and $seed$ value]
    $ret := \emptyset$
    $i := 0$
    while $i < fanout$ do
        $next$ := NEXT_RAND(0, $n\_states$-1) [uniformly distributed in domain $[0,\ n\_states - 1]$]
        if $next \notin ret$ then
            $ret := ret \cup \{next\}$
            $i := i + 1$
    return $ret$
```

For the experiments, we would like to analyze how the number of states and the interconnectivity affects the performance of the UF-SCC algorithm. We are also interested in investigating causes for the observed performance drop for non-trivial graphs.

6.3.2 Results

We perform our experiments on the constructed synthetic (random) model from Section 6.3.1. First we analyze how the number of states and fanout affects the scalability of the UF-SCC algorithm. Hereafter we investigate the performance drop as observed in Section 6.2.2.

Influence of number of states and fanout. We measured the performance for random models containing 100,000, 1000,000 and 10,000,000 states. This is combined with a fanout of 5, 10 and 15. The results for these experiments can be found in Figure 6.5 and Figure 6.6. For each combination of number of states and fanout, we ranged the *seed* value to create 8 different models.

Note that if we take a smaller fanout, the constructed graph is likely to not be connected anymore. As a result, we could for example finish exploring a model after visiting two nodes. The constructed model combinations (for a fanout of at least 5) were analyzed to at least explore 99% of the total number of provided states. Also, all of the models contained one large SCC.

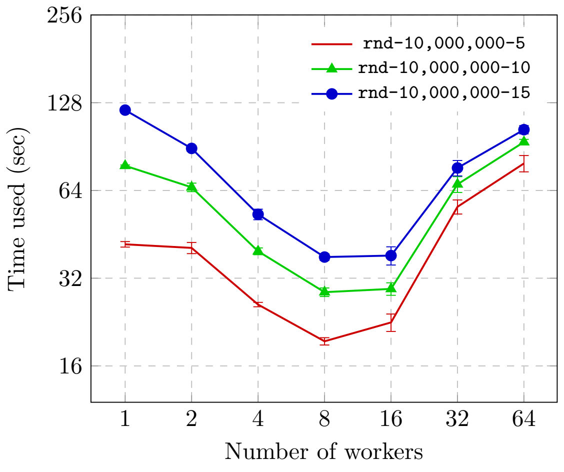


Figure 6.5: Absolute time usage for the UF-SCC algorithm on random models containing 10,000,000 states and a fanout of 5, 10, and 15.

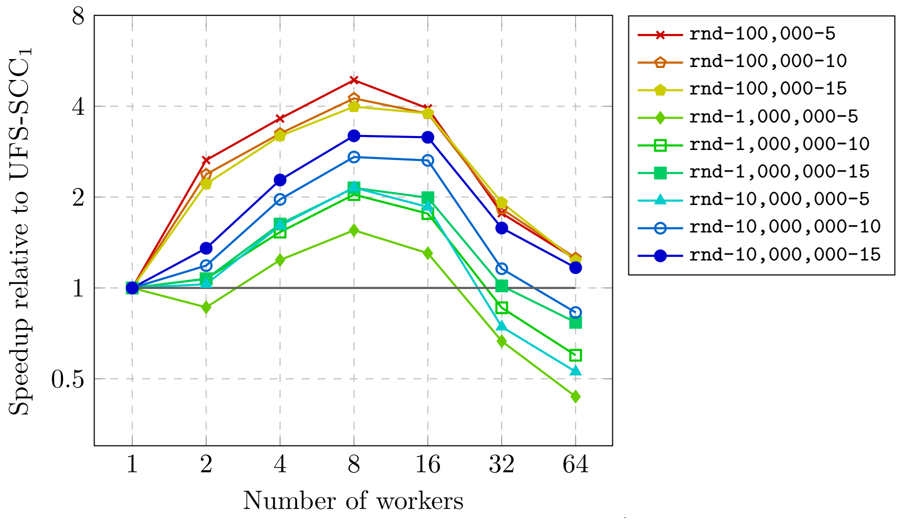


Figure 6.6: Speedups for the UF-SCC algorithm relative to its sequential performance, on specific configurations of random graphs.

Figure 6.5 provides an overview of how varying fanout values affect the performance of the algorithm. We ignore the performance drop that can be observed from eight workers and beyond since this is discussed before; we are concerned with the relative performance differences between the models. We observe that a lower fanout correlates to a better performance. This behavior is expected since a high fanout implies that more transitions have to be explored by the algorithm. For four workers and more, the time usage seems to increase by a constant value when raising the fanout. Although for fewer than four workers, the time usage is significantly more diverse between the three models. This causes an increase in the speedup for the models with a higher fanout. We argue that the increased speedup results from dividing the states over the workers: with a higher fanout, there is a higher probability for a worker to take a different successor and it is therefore more likely to explore a different part of the model.

In Figure 6.6 we measured the speedups relative to UF-SCC$_1$ on every random model. We first note that the speedups follow the same pattern from 4 workers and more. Each model peeks around 8 workers and reduces performance afterwards until it is again similar (or worse) to the sequential UF-SCC algorithm. A point of interest is how the number of states affects the speedup. We observe that the models with 100,000 states gain the most speedup, while the models with 1,000,000 states gain the least speedup. This leaves the models with the most states 'in the middle'.

We observed that for models with 100,000 states a lower fanout seems to cause a better speedup, contrasting to our earlier observation. Closer inspection shows that the 100,000 state models are explored in less than three seconds. We assume that because of this notion, the difference is caused by external influences.

6.3.3 Investigation of performance drop

We analyzed the UF-SCC algorithm on the `rnd-100,000-5` model with a profiler. We used the *callgrind* tool from *Valgrind*.[2] The reason for analyzing this particular model instead of a larger one is because applying the profiler significantly affects the performance. Also, this model was analyzed to have the most significant speedups, suggesting that the performance drop is more visible.

Table 6.3 shows the results for the profiling experiments. We analyzed which methods of the UF-SCC algorithm take the most time, for a varying number of workers. Figure 6.7 shows how the methods depend

[2]http://valgrind.org/.

Table 6.3: Time spent (percentage-wise) in particular methods, for a varying number workers when executing UF-SCC on `rnd-100,000-5`.

method	relative time spent for UF-SCC$_p$, for varying p						
	1	2	4	8	16	32	64
`ufscc-run`	99.95%	99.89%	99.7%	99.08%	97.85%	94.08%	79.87%
`successor`	86.27%	86.29%	85.96%	84.90%	84.81%	79.84%	71.30%
`backtrack`	5.74%	5.69%	5.97%	6.64%	6.21%	9.19%	5.37%
`explore-state`	57.94%	57.86%	56.88%	55.26%	50.17%	37.17%	23.59%
`uf-union`	5.40%	5.51%	5.72%	6.44%	8.79%	19.07%	32.72%
`uf-sameset`	7.29%	7.48%	8.41%	10.01%	15.28%	20.95%	14.06%
`uf-merge-list`	1.84%	1.77%	1.78%	1.81%	1.44%	< 1%	< 1%
`uf-find`	4.87%	4.93%	6.32%	7.60%	10.65%	18.17%	22.31%
`uf-make-claim`	3.39%	3.33%	3.31%	3.15%	2.83%	1.98%	1.02%
`uf-lock`	< 1%	< 1%	1.04%	1.42%	3.17%	14.93%	29.51%

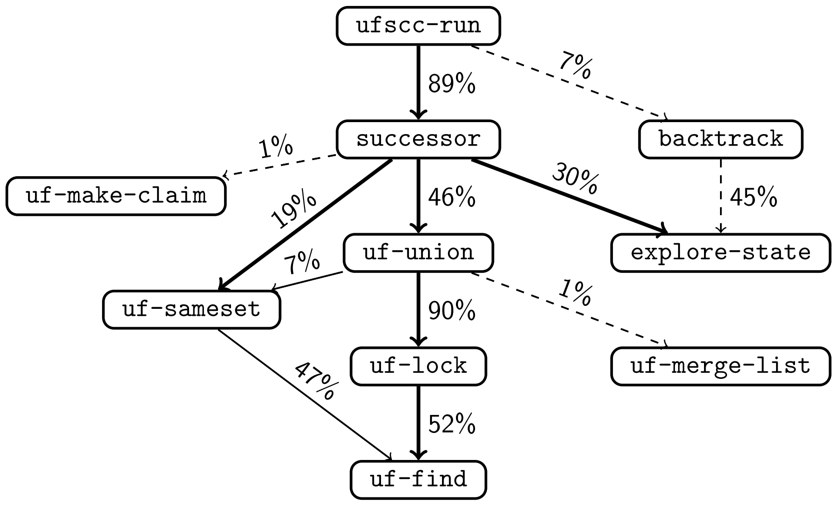


Figure 6.7: Illustrative representation of the relative dependence relation for methods in UF-SCC$_{64}$.

on each other percentage-wise. For instance, the `successor` method spends 46% of its time waiting on the `uf-union` method.

The most important observation is the prevalence of the `uf-lock` procedure for 64 workers. From the results we observe that the `uf-union` method spends 90% of its time waiting for `uf-lock`. In other words, 29.51% of the total work for UF-SCC$_{64}$ is spent in this locking procedure.

The locking procedure, as discussed in Section 5.3.3, attempts to lock two UF sets. It does so by first locking the lexicographically smallest representative (the lowest hash value). If this succeeds it tries to lock the other representative and reports that both UF sets are successfully locked. If one of the locks fails, it tries again until it is successful.

We have that the workers spend a large portion of their time in the `uf-lock` procedure. Therefore, we obtain that a large amount of lock attempts must have failed. We regard that a combination of the following speculations may be the cause for the issue:

- There could be a lot of contention on the status variable. Every time that one worker attempts a *Compare&Swap* operation to set a lock, other workers have to wait for this operation. This may also be affected by atomic reads of the status variable.

- There can be only one representative for a UF set, and we observe in this model (and this holds in the general sense as well) that there is one large SCC. We assume that the model has a high rate of inter-connectivity, implying that there are multiple 'back-edges' to the one SCC. Then we can infer that it should be very likely for a worker to perform a UNION operation, for which one of the states is the representative of the large SCC. As a result, multiple workers want to lock the same state and thus have to wait on each other.
- The locking mechanism is based on a homogeneous division for the workers. It may however be possible that worker p halts for a period of time after it successfully acquired a lock. All other workers that attempt a UNION on the same state must wait for worker p.

On a positive note, the method `uf-pick-from-list` is nowhere to be found in the profiler data. This means that picking states from the cyclic list structure appears to take a relatively insignificant amount of time. We take from this that our designed cyclic list structure may be capable of scaling efficiently over multiple workers. However, the results may be skewed resulting from the time spent in the `uf-lock` procedure.

Take in mind that the profiler data is not necessarily depicting the actual situation. However we do regard that the main observations hold in the general sense.

6.3.4 Conclusions

From the experiments on random models, we observed interesting results. These are summarized as follows:

- For random models, we observed that a larger fanout correlates to a better speedup. However, this phenomenon was not present in the smallest model - likely due to external influences.
- The number of states do not seem to affect the gained speedup; the highest speedups were attained on the smallest model, while the UF-SCC algorithm scaled the worst on the middle-sized model.
- All randomly generated models seem to follow the same pattern, where the performance peeks at 8 workers and drops afterwards.
- The major cause for the performance drop was observed to be the locking procedure. For 64 workers, a worker spends almost 30% of the time in this method.
- While the results may be skewed by the locking procedure, the profiler data suggests that the cyclic list structure is performing efficiently.

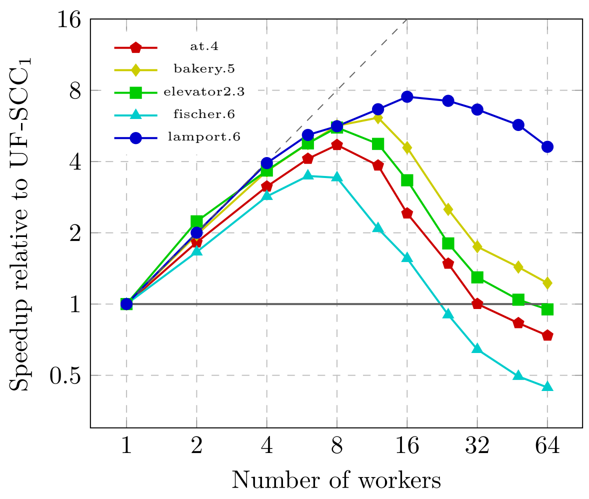
For future work in terms of improving the UF-SCC algorithm, we suggest taking a closer look at the locking mechanism. We identified this as the bottleneck for the algorithm's performance. Section 7.3 proposes some concrete ideas to mitigate this bottleneck.

6.4 Additional experiments

This section discusses experiments with the intent to better understand the observed performance drop for the UF-SCC algorithm.

6.4.1 Experimentation on hardware influence

We analyzed the influence of the hardware architecture on the performance of the UF-SCC algorithm. We used the following architectures:



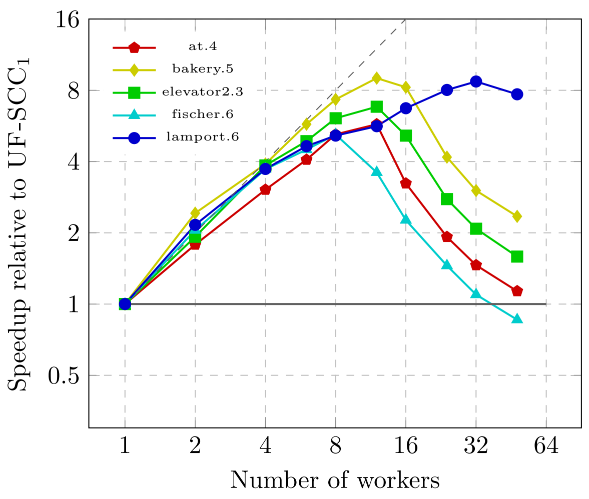


Figure 6.8: Relative speedups for the UF-SCC algorithm on non-trivial selected BEEM models, on the weleveld (left) and westervlier machine (right).

- *weleveld* : A machine with 4 AMD Opteron^{TM} 6376 processors, each with 16 cores, forming a total of 64 cores. There is a total of 512GB memory available. This machine was used for all other experiments (as indicated in Section 6.1.2).
- *westervlier* : A machine with 4 AMD Opteron^{TM} 6168 processors, each with 12 cores, forming a total of 48 cores. There is a total of 128GB memory available.

The selected non-trivial BEEM models were used for the analysis. We analyzed the speedup of the UF-SCC algorithm relative to its sequential performance on the same machine. Figure 6.8 shows the results for these experiments.

We observe a similar shape in both architectures for the performance gain and drop when increasing the number of workers. However, there is a significant difference in where the 'peek performance' lies. For the weleveld machine the algorithm peeks its performance generally at 8 workers, with a speedup of 4. The westervlier machine peeks on average at 12 workers, with a speedup of 6. We were unable to find strong evidence for explaining this effect. A hypothesis is that it may be related to the level 2 (or secondary) caches for the used processors. The processors used in the westervlier machine contain 12× 512KB sized caches,[3] possibly relating to the peek at 12 workers. The weleveld machine uses AMD Opteron^{TM} 6376 processors, these contain 16 cores but their level 2 caches are 8× 2MB, thus arguable not all cores are fully utilized which may cause the peek at 12 workers. Please note that this reasoning is highly speculative, further research could be to perform specific tests on the internals of the locking structure (e.g. analyzing the performance of multiple *Compare&Swap* operations).

Another observation is that the maximum speedup for the `lamport.6` model is less affected. This latter observation could be explained by the fact that `lamport.6` contains multiple smaller SCCs (instead of one large SCC) and thus more closely relates to trivial models, for which we did not encounter significant performance drops.

We observed that the choice of hardware does significantly affect the (relative) performance of the UF-SCC algorithm. Thus, the choice of hardware is an important factor to consider when analyzing the algorithm.

[3]Information obtained from http://www.cpu-world.com/.

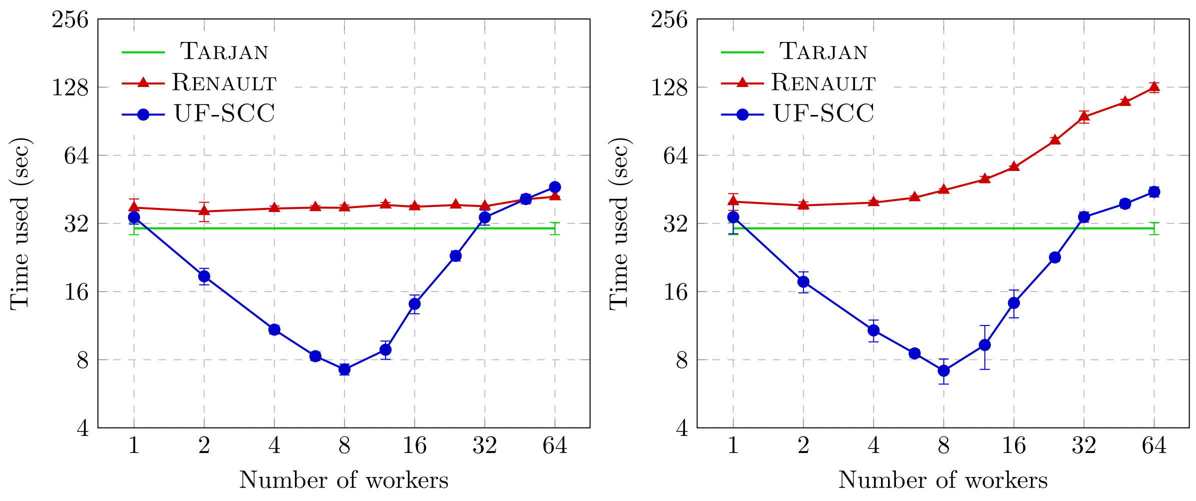


Figure 6.9: Absolute time usage on the `at.4` BEEM model for a lockless Renault implementation (left) and one where locking is used (right).

6.4.2 Experimentation on difference between locking and lockless

We analyzed the possible influence for the locking mechanism used in the UF-SCC algorithm. We were not (yet) able to design our approach in a lockless form. However, we argue that we can significantly increase the performance may we succeed in this.

To better understand the difference between a lockless and locking approach, we implemented a variant to Renault's algorithm that uses the same locking scheme as UF-SCC (note that Renault's standard implementation is lockless). Figure 6.9 shows how these two versions of Renault's algorithm differ from each other on the `at.4` BEEM model (this model consists of one SCC).

From the figure, it becomes clear that the approach that uses locking performs significantly worse as the number of workers increase. Note that since `at.4` contains a single SCC, Renault's algorithm is not capable of speeding up at all. The multiple workers do globally unite the UF sets. The performance decrease from the implementation with locks is caused by these Union calls. We observed similar results compared to the findings in Section 6.3.3.

Both versions of Renault's algorithm performed similar sequentially: 38 seconds and 40 seconds for respectively the lockless and locking approach. For 8 workers, the lockless version still performed in 38 seconds. Here, the locking approach performed in 44 seconds, which is still comparable. However, for 64 workers, the lockless version performed in 42 seconds but the locking version used 127 seconds to complete. Here, the locking approach is a factor of 3.0 slower than the lockless version.

From the experiment we observed that a lockless version (for Renault) is significantly outperforming the locking approach. For a lockless version of UF-SCC we therefore should expect similar results. We should also take in consideration that UF-SCC has a far more complicated Union procedure (it also has to update the Pset and *List*). This could imply that the performance decrease (for locking) is even more significant for UF-SCC. We therefore stress that improving the locking procedure for UF-SCC is arguably the most important aspect in future work.

Chapter 7

Conclusion and Future Work

In this chapter we compare our algorithm with related work, discuss the conclusions of our work and propose directions for future work.

7.1 Comparison with related work

In this section, we compare the UF-SCC algorithm with Renault [53] and Lowe [40]. We focus on both the theoretical and empirical aspects.

7.1.1 Renault's algorithm

Renault's algorithm is based on spawning multiple *independent* searches of Tarjan's algorithm. It tracks the SCCs in a Union-Find data structure and gains performance from communicating fully explored SCCs (see Section 3.3.3).

Theoretical evaluation. Renault's algorithm is comparable to UF-SCC in the sense that it also uses a Union-Find structure, though the underlying strategies are rather different. It performs in quasi-linear time, which is equivalent to UF-SCC. The space complexity is (due to the Union-Find structure) also similar, although the entries for the uf nodes are of larger size in the UF-SCC algorithm. The main difference is how Renault gains scalability. Both UF-SCC and Renault communicate information on completely explored SCCs between the workers. For a single SCC, Renault is not capable of gaining speedups, whereas UF-SCC is. Thus, the performance for Renault and UF-SCC should be similar on trivial graphs (models that contain a large number of SCCs relative to the number of states) and UF-SCC should clearly outperform Renault on non-trivial graphs (that contain larger SCCs).

Experimental evaluation. We mainly evaluated Renault's performance in Section 6.2. We found that for trivial graphs, Renault's algorithm generally performs slightly better than UF-SCC. While both algorithms scale almost linearly compared to Tarjan's algorithm, Renault executes about 25% faster than UF-SCC. For non-trivial graphs, Renault generally does not improve its performance at all. Because UF-SCC does scale on non-trivial graphs, we have that for eight workers, UF-SCC performs up to four times faster than Renault's algorithm.

7.1.2 Lowe's algorithm

Lowe's algorithm is based on spawning multiple *synchronized* searches of Tarjan's algorithm. It ensures that the workers perform strictly disjunct searches. A *suspend* procedure is used to combine multiple searches whenever the searches interfere with each other (see Section 3.3.2).

Theoretical evaluation. LOWE's algorithm operates quite different from UF-SCC. Intrinsically, it should also be able to communicate partially discovered SCCs. However, the 'repair' procedure caused by the *blocking cycle* is executed sequentially which causes a single worker to explore a complete SCC. This procedure also causes the algorithm to perform in quadratic time, though Lowe mentions that this can only occur in rare circumstances. We assume that the space complexity is similar to that of UF-SCC, resulting from similarities between the UNION-FIND structure and the *Suspended* map. In terms of scalability, we expect LOWE's algorithm to mainly gain performance on trivial graphs. From the algorithm it is unclear whether the algorithm scales for non-trivial graphs as well.

Experimental evaluation. We were not able to evaluate LOWE's algorithm ourselves. Therefore we rely on the results from Lowe's paper [40]. In Section 3.3.2, we presented results from LOWE's algorithm. Note that these experiments were performed in an *offline* setup, which should imply that the algorithm is less restricted and should perform equally or better compared to an on-the-fly version. Also, the benchmark environment is significantly different from the one we used: it is performed on a different architecture (it is examined for a maximum of eight cores) and it is implemented in the Scala language. For trivial graphs, LOWE's algorithm gains a three- to four-fold speedup (compared to TARJAN's algorithm) using eight workers, which is similar compared to the results for UF-SCC. Lowe also performed a few experiments on non-trivial graphs. Here, the performance for LOWE's algorithm was found to be *worse* compared to TARJAN's algorithm. Lowe also performed experiments on randomly generated graphs (similar to the ones we describe in Section 6.3). From these results we observe that for a fanout of larger than 1.5 (for 500,000 states) the performance is worse compared to TARJAN's algorithm. We therefore assume that LOWE's algorithm does not gain scalability for non-trivial graphs. Thus we conclude that UF-SCC performs also performs up to four times faster than LOWE's algorithm.

7.2 Conclusion

To summarize the results from this thesis:

- We showed that the state-of-the art parallel on-the-fly SCC algorithms do not scale on graphs containing large SCCs (compared to the number of states).
- We presented a new parallel on-the-fly SCC algorithm that is capable of scaling on graphs containing large SCC. We show that it performs up to four times faster (using eight threads) than the best known approaches. Furthermore, our algorithm performs on par with the best known techniques for graphs containing many small SCCs.

In Chapter 4 we presented a naive algorithm for on-the-fly multi-core SCC decomposition. This algorithm is based on a strict DFS search in combination with communication on partially discovered SCCs. This is realized by extending the Union-Find structure with a worker set. We observed complications with this algorithm and proposed solutions to resolved these. A theoretical examination shows that this algorithm has a quadratic time worst-case complexity.

In Chapter 5 we presented an improved algorithm for on-the-fly multi-core SCC decomposition. This algorithm also communicates partially discovered SCCs. The difference with the algorithm from Chapter 4 is that this version includes a parallel cyclic list in the Union-Find structure to iterate over the states. Workers pick states from this list and remove these globally once explored. This made it possible to let multiple workers simultaneously contribute to exploring an SCC. We proved that this algorithm is correct and that it runs in linear time.

In Chapter 6 we discussed experiments on the implementation for the improved algorithm. We found that the implementation was able to scale effectively (to a certain extent).

From the experiments we conclude that for trivial models our algorithm performs similar to that of RENAULT and LOWE, where RENAULT performs slightly better. For non-trivial models our algorithm is clearly able to outperform RENAULT and LOWE as it has observed to gain a speedup of up to four times for eight workers, while the other algorithms do not gain performance on non-trivial models. This indicates that our algorithm is able to scale efficiently by communicating partially discovered SCCs. We did observe a significant performance drop (on non-trivial models) for more than eight workers, which has been found to origin from the contention on the locking mechanism.

In conclusion, the designed algorithm has been proven to be correct and theoretically able to scale efficiently. We showed that in contrast to existing techniques, our algorithm is able to scale on large SCCs by communicating information on partially discovered SCCs. Experimentations showed that in its current state, the implementation outperforms existing techniques for up to eight workers. Future work will have to show if the used synchronization measures can be improved.

7.3 Future Work

With regard to future work, we consider the following directions:

1. Easing the synchronization measures for the Merge procedure in the algorithm. We currently use a locking structure which causes multiple workers to spend a significant portion of their time waiting for each other. One direction is to design a lockless mechanism that ensures correct behavior with respect to data races. Another approach is to improve upon the current locking structure, by proposing a new scheme to lock states and queue multiple Merge operations so that a worker does not explicitly have to wait on locked states.

2. Implementing Lowe's algorithm. While spending a fair effort on making Lowe's algorithm run on our system, we did not (yet) succeed in this. Since Lowe has implemented his algorithm in Scala (while we implemented our algorithm in C), we do not regard that it can be compared with our approach in a fair manner. We are furthermore interested in implementing Lowe's algorithm as we consider possible improvements can be made using the observations of this work.

3. Analyzing our algorithm's performance for applications of SCCs. We have shown that our algorithm performs well for detecting SCCs. We did not yet analyzed its performance in a more practical context. We regard that the algorithm is well-suited for handling variations; experiments will have to show this.

4. Extending the synthetic models to cover various types of graphs. We found that a randomly generated graph is not representable for a complete overview in terms of performance analysis. We expect to gain valuable information by for instance considering path and cycle length variations in the synthetic graphs.

5. Formalizing the proof of the algorithm by means of theorem proving. While we expect our 'hand-written' proof to be correct, we would like to show this in a more formalized setting. Moreover, by doing so we regard it probable to extract the algorithm's behavior in terms of necessary and superfluous measures. This also makes it possible to more easily consider variations of the algorithm. Promising results in this field have been shown by recent related work [60, 49].

Appendices

Appendix A

Correctness proof for UF-SCC

In this Appendix, we prove the parallel UF-SCC algorithm correct, as presented in Algorithm 26. Here, we modified Line 8 from [$w \in \textsc{post}(v')$] (see Algorithm 18) to [$w := \textsc{GetSuccessor}_p(v')$]; this allows us to reason more formally about the successors. We assume that each line of code is executed atomically (in practice, forms of locking and/or lock-free techniques are used to prevent data races). Concerning the external method calls, we assume that each of these methods adhere to the provided specification from Algorithm 27. Moreover, we assume that no other modifications are made by these methods other than stated from the specification. The main correctness results in Theorem A.16 with respect to soundness, Theorem A.17 with respect to completeness, and Theorem A.18 with respect to termination.

We use of the *Hoare triple* $\{P\}$ C $\{Q\}$ [26] style notation for stating pre- and post-conditions. Here, we assure that pre-condition Q holds before calling C, and that post-condition Q holds when C is completed.

Algorithm 26 The UF-SCC algorithm

```
 1: procedure UF-SCC-MAIN($v_0$, $\mathcal{P}$)
 2:     for each $p \in [1 \ldots \mathcal{P}]$ do
 3:         MakeClaim($v_0$, $p$)
 4:     UF-SCC$_1$($v_0$) ‖ … ‖ UF-SCC$_\mathcal{P}$($v_0$)

 5: procedure UF-SCC$_p$($v$)
 6:     $D_p$.push($v$) [start ‘new’ SCC]
 7:     while $v' :=$ PickFromList($v$) do
 8:         while $w :=$ GetSuccessor$_p$($v'$) do
 9:             $result :=$ MakeClaim($w$, $p$)
10:             if $result =$ claim_dead then continue
11:             else if $result =$ claim_success then
12:                 UF-SCC$_p$($w$) [recursively explore $w$]
13:             else [$result =$ claim_found]
14:                 while $\neg$SameSet($D_p$.top(), $w$) do
15:                     $w' := D_p$.pop()
16:                     Merge($w'$, $D_p$.top())
17:         RemoveFromList($v'$) [fully explored post($v'$)]
18:     if $v = D_p$.top() then $D_p$.pop() [remove completed SCC]
```

In the UF-SCC Algorithm 26, the representation for local methods are segregated from global methods by adding the subscript M_p, meaning that method M is locally executed for worker p. Concerning the local stack D_p, for the proof we extend this data type with $v \in D_p$ and $w = D_p[i]$, which respectively denotes that v is an element of (the set) D_p and w is the element at position i in the stack (where $D_p[0]$ is the bottom of the stack and $D_p[|D_p| - 1] = D_p.\textsc{top}()$).

Algorithm 27 Specification for the UF-SCC algorithm

```
1: definition $\textsc{uf}[v] \subseteq \mathcal{V}$
2: definition $List(v) \subseteq \textsc{uf}[v]$
3: definition $NextSCC(v) := \{w \mid v' \in \textsc{uf}[v] : (v' \to w) \in \mathcal{E}\}$
4: definition $Suc_p(v) \subseteq \textsc{post}(v)$
5: definition $\textsc{Pset}(v) \subseteq \{1, \ldots, \mathcal{P}\}$
6: predicate $\textsc{IsDead}(v) := List(v) = \emptyset$
7: predicate $GlobalDone(v) := \forall v' \in \textsc{post}(v) : \textsc{SameSet}(v, v') \ \vee\ \textsc{IsDead}(v')$
8: predicate $GlobalDead(v) := \forall v' \in NextSCC(v) : \textsc{SameSet}(v, v') \ \vee\ \textsc{IsDead}(v')$
9: invariant $\forall v, v' \in \textsc{uf}[v] : \textsc{Pset}(v') = \textsc{Pset}(v)$
10: invariant $\forall v, v' \in \textsc{uf}[v] : \textsc{uf}[v'] = \textsc{uf}[v]$
11: invariant $\forall v, v' \in \textsc{uf}[v] : List(v') = List(v)$
12: invariant $\forall v, v' : \textsc{SameSet}(v, v') \Rightarrow List(v) = List(v')$
13: invariant $\forall v, v' \in \textsc{uf}[v] \setminus List(v) : Next(v') \subseteq \textsc{uf}[v] \cup \textsc{dead}$

14: precondition: $\forall v : \textsc{uf}[v] = List(v) = \{v\}$
15: precondition: $\forall v : \textsc{Pset}(v) = \emptyset$
16: precondition: $\forall v, p \in [1 \ldots \mathcal{P}] : Suc_p(v) = Next(v)$
17: precondition: $\forall p \in [1 \ldots \mathcal{P}] : D_p = \emptyset$
18: procedure UF-SCC-MAIN$(v_0, \mathcal{P})$

19: procedure UF-SCC$_p(v)$

20: returns: $\textsc{null} \iff List(v) = \emptyset$
21: returns: $v' \in List(v) \iff List(v) \neq \emptyset$
22: procedure $\textsc{PickFromList}(v)$

23: postcondition: $old(Suc_p(v)) \neq \emptyset \implies \exists w \in old(Suc_p(v)) : Suc_p(v) = old(Suc_p(v)) \setminus \{w\}$
24: returns: $\textsc{null} \iff old(Suc_p(v)) = \emptyset$
25: returns: $w \iff \{w\} = old(Suc_p(v)) \setminus Suc_p(v)$
26: procedure $\textsc{GetSuccessor}_p(v)$ [The returned successor is removed from $Suc_p(v)$]

27: postcondition: $\neg\textsc{IsDead}(v) \implies p \in \textsc{Pset}(v)$
28: returns: $\textsc{claim\_dead} \iff \textsc{IsDead}(v)$
29: returns: $\textsc{claim\_success} \iff \neg\textsc{IsDead}(v) \ \wedge\ p \notin old(\textsc{Pset}(v))$
30: returns: $\textsc{claim\_found} \iff \neg\textsc{IsDead}(v) \ \wedge\ p \in old(\textsc{Pset}(v))$
31: procedure $\textsc{MakeClaim}(v, p)$

32: returns: $v \in \textsc{uf}[w]$
33: procedure $\textsc{SameSet}(v, w)$

34: postcondition: $\textsc{uf}[v] = \textsc{uf}[w] = old(\textsc{uf}[v]) \ \cup\ old(\textsc{uf}[w])$
35: postcondition: $List(v) = old(List(v)) \ \cup\ old(List(w))$
36: postcondition: $\textsc{Pset}(v) = old(\textsc{Pset}(v)) \ \cup\ old(\textsc{Pset}(w))$
37: procedure $\textsc{Merge}(v, w)$ [Global modification]

38: postcondition: $v \notin List(v)$
39: procedure $\textsc{RemoveFromList}(v)$ [Global modification]
```

Lemma A.1. *The* UF-SCC$_p$ *procedure from Algorithm 26 adheres to the following pre-condition:*
$\{p \in \textsc{Pset}(v)\}$ UF-SCC$_p(v)$ $\{\dots\}$

Proof. Only at Line 4 and Line 12, UF-SCC$_p$ can be called. At Line 4 it is called for v_0. We have as a result from Line 3 that $p \in \textsc{Pset}(v_0)$ holds, by the post-condition from $\textsc{MakeClaim}(v_0, p)$, if $\neg\textsc{IsDead}(v_0)$. By the assumptions preceding UF-SCC-MAIN, we have that $\forall v : \mathit{List}(v) = \{v\}$. Since no modifications to $\mathit{List}(v_0)$ take place in Lines 2-3, we can conclude that $\neg\textsc{IsDead}(v_0)$ holds by definition and therefore $p \in \textsc{Pset}(v_0)$.

At Line 12, for some w, the UF-SCC$_p(w)$ procedure is called, resulting from the fact that Line 11 evaluated to *True*. Therefore, result = CLAIM_SUCCESS, which is obtained from $\textsc{MakeClaim}(w, p)$ in Line 9. By its specification we obtain CLAIM_SUCCESS $\iff \neg\textsc{IsDead}(w)$. Therefore we have by the post-condition of $\textsc{MakeClaim}(w, p)$ that $p \in \textsc{Pset}(w)$. ☐

Corollary A.2. *All states on the D_p stack contain the worker ID p:*
$\forall p, v : v \in D_p \implies p \in \textsc{Pset}(v)$

Proof. By Lemma A.1 we have that $p \in \textsc{Pset}(v)$ at Line 6 of the UF-SCC$_p(v)$ procedure. Here, v gets pushed on D_p. Since Line 6 is the only place where states are pushed D_p and by observing that a worker ID is never removed from a state, we obtain that $\forall p, v : v \in D_p \implies p \in \textsc{Pset}(v)$. ☐

Corollary A.3. *All states in the same* UF *sets as the states on the D_p stack contain the worker ID p:*
$\forall p, v : v \in D_p \implies \forall v' \in \textsc{uf}[v] : p \in \textsc{Pset}(v')$

Proof. This follows directly from Corollary A.2 and the invariant over PSETS (Line 9 from Algorithm 27). ☐

Lemma A.4. *The* UF-SCC$_p$ *procedure from Algorithm 26 adheres to the following pre-condition:*
$\{v \notin D_p\}$ UF-SCC$_p(v)$ $\{\dots\}$

Proof. Only at Line 4 and Line 12, UF-SCC$_p$ can be called. At Line 4 it is called for v_0. The pre-condition trivially holds as no modifications to D_p have occurred yet.

In the UF-SCC$_p(v)$ procedure, UF-SCC$_p(w)$ is called in Line 12 for some state w. We proof the case for Line 12 by contradiction. Assume that $w \in D_p$ at the start of Line 12 in the UF-SCC$_p(v)$ procedure. Note that this invalidates the pre-condition of UF-SCC$_p$. We have that Line 11 evaluated to *True* and hence result = CLAIM_SUCCESS. This can only be the case if $\textsc{MakeClaim}(w, p)$ returned CLAIM_SUCCESS in Line 9. Since D_p is unmodified in Lines 9-11, $w \in D_p$ must hold at Line 9. By Corollary A.2 we obtain that $p \in \textsc{Pset}(w)$. This however contradicts with the specification of $\textsc{MakeClaim}(w, p)$, as $p \notin \mathit{old}(\textsc{Pset}(w))$. Therefore, $w \notin D_p$ at the start of Line 12 and thus the pre-condition holds. ☐

Lemma A.5. *The* UF-SCC$_p$ *procedure from Algorithm 26 adheres to the following post-condition:*
$\{\dots\}$ UF-SCC$_p(v)$ $\{v \notin D_p\}$

Proof. At the start of Line 6 in the UF-SCC$_p(v)$ procedure, we have by Lemma A.4 that $v \notin D_p$. After Line 6, $v = D_p.\textsc{top}()$. We assume (by strong induction on the number of recursive UF-SCC$_p$ calls) that the the post-condition holds true for the recursive UF-SCC$_p(w)$ procedure in Line 12. At the start of Line 18 we have that $v \notin D_p \vee v = D_p.\textsc{top}()$. This results from the fact that the only place where some state t can be pushed on D_p (on top of v) is in Line 6, in procedure UF-SCC$_p(t)$. As all recursive procedures have finished at Line 18, the post-condition ensures that none of these states reside on the D_p stack anymore, thus either $v = D.\textsc{top}()$ or v has already been popped by this or a recursive procedure. After executing Line 18 we conclude that $v \notin D_p$ must hold. ☐

Lemma A.6. *The* UF-SCC$_p$ *procedure from Algorithm 26 adheres to the following post-condition:*
$\{\dots\}$ UF-SCC$_p(v)$ $\{\textsc{IsDead}(v)\}$

Proof. This follows directly from the while condition in Line 7 from the UF-SCC$_p(v)$ procedure. By applying the specification from PickFromList(v), the while-loop ends if *List*$(v) = \text{NULL} \equiv \text{IsDead}(v)$. As Line 18 does not alter the situation, the post-condition is preserved. □

Lemma A.7. *In the* UF-SCC$_p$ *procedure of Algorithm 26, before executing Line 9 and after Line 12, we have that:*
(UF-SCC$_p(v)$ at Line 9 and after Line 12): $\forall t : p \in \text{Pset}(t) \implies \exists t' \in \text{uf}[t] : \text{UF-SCC}_p(t')$ *has been called.*

Proof. Note that MakeClaim(w, p) can only be called by UF-SCC$_p(v)$ in Line 9, hence no other workers can add worker ID p to a state with MakeClaim(w, q), for any $q \neq p$. The only other place where worker ID p can be added to a state is in the Merge(v, w) procedure, called at Line 16 (by any worker). This call merges the Psets from v and w. Moreover, after the Merge procedure we can assert that $v \in \text{uf}[w]$, and $p \in \text{Pset}(v)$ holds only if $p \in (\text{Pset}(v) \cup \text{Pset}(w))$ before the Merge(v, w) call. Thus, for any v such that $p \in \text{Pset}(v)$ we have that $\exists v' \in \text{uf}[v] : \text{MakeClaim}(v', p)$ is called, and $p \notin \text{Pset}(v')$ before this call. If $p \notin \text{Pset}(w)$ holds before Line 9 (and w.l.o.g. $\neg\text{IsDead}(w)$), the specification of MakeClaim ensures that CLAIM_SUCCESS is returned. By Lines 11 and 12 we can assert that the UF-SCC$_p(w)$ procedure will be called. Therefore, before Line 9 and after Line 12 we can assure that for all states t for which $p \in \text{Pset}(t)$ holds, the UF-SCC$_p(t')$ procedure has been called for some $t' \in \text{uf}[w]$. □

Lemma A.8. *After* UF-SCC$_p$ *from Algorithm 26 finishes executing, every state containing worker ID p is* DEAD *and/or 'part of' the D_p stack. The* UF-SCC$_p$ *procedure from Algorithm 26 adheres to the following post-condition:*
$\{\ldots\}$ UF-SCC$_p(v)$ $\{\forall t : p \in \text{Pset}(t) \implies \text{IsDead}(t) \ \vee \ \exists t' \in \text{uf}[t] : t' \in D_p\}$

Proof. By Lemma A.7 we obtain that after UF-SCC$_p(v)$ finishes, $p \in \text{Pset}(w)$ implies that the UF-SCC$_p(w')$ procedure has been called for some $w' \in \text{uf}[w]$. For any state v', if the UF-SCC$_p(v')$ procedure has finished we obtain from Lemma A.6 that IsDead(v') holds. We now show that for any t, $p \in \text{Pset}(t) \wedge \neg\text{IsDead}(t) \implies \exists t' \in \text{uf}[t] : t' \in D_p$ holds after UF-SCC$_p(v)$ finishes. Since $p \in \text{Pset}(t)$, we have that the UF-SCC$_p(t')$ procedure has been recursively called by UF-SCC$_p(v)$ for some $t' \in \text{uf}[t]$. Therefore, t' has been pushed on D_p as a result of Line 6. Consider the two cases where $D_p.\text{pop}()$ is called, at Line 15 and Line 18. At Line 18 in the UF-SCC$_p(t')$ procedure we have IsDead(t') (see Lemma A.6) and thus conforms with the post-condition. At Line 15 in the UF-SCC$_p(t')$ procedure, the popped state w' is immediately afterwards merged with the new $D_p.\text{top}() \in \text{uf}[t]$. Thus, after Line 16, by the specification of the Merge$(w', D_p.\text{top}())$ procedure, we again have that $w' \in \text{uf}[t]$. As a result we can conclude that $\forall t : p \in \text{Pset}(t) \implies \text{IsDead}(t) \ \vee \ \exists t' \in \text{uf}[t] : t' \in D_p$ holds after UF-SCC$_p(v)$ finishes. □

Corollary A.9. *The post-condition from Lemma A.8 also holds at the start of Line 9 in Algorithm 26:*
(UF-SCC$_p(v)$ at Line 9): $\forall t : p \in \text{Pset}(t) \implies \text{IsDead}(t) \ \vee \ \exists t' \in \text{uf}[t] : t' \in D_p$

Proof. First, assume w.l.o.g. we have some t such that $p \in \text{Pset}(t)$ holds at Line 9. By Lemma A.7 we have that UF-SCC$_p(t')$ has been called for some $t' \in \text{uf}[t]$. Note that at this point, UF-SCC$_p(t')$ must have also executed Line 6. From the (proof of) Lemma A.8 we obtain that before Line 15 and after Line 16, the post-condition is preserved. Therefore Corollary A.9 must also hold. □

Lemma A.10. *For any two states t and u that are in D_p and where the stack index of w is one more than the stack index of t, there is a state t' in* uf$[t]$ *such that $t' \to u$ is an edge in $\mathcal{G}$:*
$\forall p, i : 0 < i < |D_p| \implies \exists t \in \text{uf}[D_p[i-1]] : D_p[i] \in \text{post}(t)$

Proof. Assume that the invariant holds for all states currently on D_p, we show that it remains valid when a new state is pushed on the stack (note that the invariant is preserved in the base case where $|D_p| < 2$). In Line 7 of the UF-SCC$_p(v)$ procedure, some state $v' \in List(v) \subseteq \text{uf}[v]$ is selected by PickFromList(v). In Line 8 we obtain that state $w \in \text{post}(v')$. If UF-SCC$_p(w)$ is called in Line 12, we can assure that $v' \in \text{uf}[v] \wedge w \in \text{post}(v')$. Therefore, when UF-SCC$_p(w)$ pushes w on D_p at Line 6, the invariant remains valid. Since states can only be popped from the top of the stack, the ordering of D_p remains the same and thus Lemma A.10 holds. □

Theorem A.11. *All states in the same* UF *set are part of the same SCC:*
$\forall t, u : t \in \text{UF}[u] \implies t \to^* u \to^* t$

Proof. We show this by induction on the number of MERGE calls. MERGE is only called at Line 16 of Algorithm 26 and note that MERGE is the only process that adapts UF (by combining $\text{UF}[v]$ with $\text{UF}[w]$).

$n = 0$: Initially we have $\forall v : \text{UF}[v] = \{v\}$ by the specification of UF-SCC-MAIN. Since $\forall v : v \to^* v$ holds trivially for any state (with a path of length 0), Theorem A.11 holds for $n = 0$.

$n = k + 1$: We assume that Theorem A.11 holds for k MERGE calls. We show that it also holds for $k+1$ MERGE calls. At the start of Line 16 in Algorithm 26 we have that $\forall v, w : w \in \text{UF}[v] \implies v \to^* w \to^* v$ holds (induction hypothesis). By combining this induction hypothesis with Lemma A.10 we obtain that $\forall p, i : 0 < i < |D_p| \implies \exists v \in \text{UF}[D_p[i-1]] : D_p[i] \in \text{POST}(v) \implies D_p[i-1] \to^* D_p[i]$. By recursive application of this statement, we have $\forall v \in D_p : v \to^* D_p.\text{TOP}()$. At the start of Line 16 we can therefore assure that $D_p.\text{TOP}() \to^* w'$ (because w' was popped from the stack in Line 15).

We now show that $w' \to^* D_p.\text{TOP}()$ also holds. The body of the while procedure is being executed because $\neg\text{SAMESET}(D_p.\text{TOP}(), w)$ evaluated to *True* in Line 14, meaning that Lines 15 and 16 are executed iteratively until $D.\text{TOP} \in \text{UF}[w]$. This while procedure is being executed because the MAKECLAIM(w, p) call from Line 9 returned CLAIM_FOUND. From the specification of MAKECLAIM we obtain that at the start of Line 9 we have $\neg\text{ISDEAD}(w) \ \wedge \ p \in \text{PSET}(w)$. From Corollary A.9 we obtain that $\forall t : p \in \text{PSET}(t) \implies \text{ISDEAD}(t) \ \vee \ \exists t' \in \text{UF}[t] : t' \in D_p$, by combining these statements we obtain: $\exists t' \in \text{UF}[w] : t' \in D_p$. Since $\forall v \in D_p : v \to^* D_p.\text{TOP}()$ holds as a result of the induction hypothesis, we have that $\exists t' \in \text{UF}[w] \wedge t' \in D_p : t' \to^* D_p.\text{TOP}() \implies w \to^* D_p.\text{TOP}()$. Also, since $w \in \text{POST}(v')$ (Line 8) and $v' \in \text{UF}[v] = \text{UF}[D.\text{TOP}()]$ (Line 7 and by the proof of Lemma A.8) we have $D_p.\text{TOP}() \to^* w$ or equivalently $v \to^* w$.

Because $\neg\text{SAMESET}(D_p.\text{TOP}(), w)$ in Line 14 we assure that for every state w' previously popped from D_p in this while procedure at Line 15, $w' \notin \text{UF}[w]$. If this were not the case, Line 16 would ensure that $t \in \text{UF}[w]$ is *merged* with $D_p.\text{TOP}()$, and thereby falsifying the while condition. At the start of Line 15 we thus have that $D_p.\text{TOP}() \notin \text{UF}[w]$ and because $\exists t' \in \text{UF}[w] : t' \in D_p$, we can assure that $t' \neq D_p.\text{TOP}()$. Therefore, at the start of Line 16, we know that $t' \to^* D_p.\text{TOP}()$ (because either $t' = D_p.\text{TOP}()$ or t' resides lower on the stack). Since $t' \in \text{UF}[w]$ and $w' \in \text{UF}[v]$ and $v \to^* w$, we obtain $w' \to^* v \to^* w \to^* t' \to^* D_p.\text{TOP}() \implies w' \to^* D_p.\text{TOP}()$ (see Figure A.1 for an illustrative overview of this situation). Because both $D_p.\text{TOP}() \to^* w'$ and $w' \to^* D_p.\text{TOP}()$ hold at Line 16, we conclude that $\forall v, w : w \in \text{UF}[v] \implies v \to^* w \to^* v$ remains valid after the MERGE$(w', D_p.\text{TOP}())$ procedure, thereby proving the invariant. □

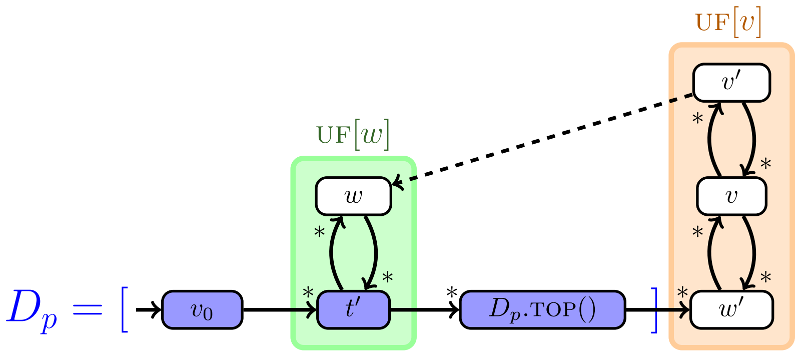


Figure A.1: Illustrative representation of Algorithm 26 at Line 16. Here, the highlighted states are part of D_p and the marked regions represent UF sets.

Corollary A.12. *For any two states v and w that are in D_p and the stack index of w is one more than the stack index of v, $\mathcal{G}$ contains the path $v \to^* w$:*
$\forall p, i : 0 < i < |D_p| \implies D_p[i-1] \to^* D_p[i]$

Proof. This follows directly from Theorem A.11 and Lemma A.10. □

Lemma A.13. *In Algorithm 26 at Line 17, all successors of v' are either* DEAD *or in the same set as v' and we say* GLOBALDONE(v')*:*
(UF-SCC$_p(v)$ *at Line 17*) : $\forall w \in \text{POST}(v') : \text{ISDEAD}(w) \vee \text{SAMESET}(v', w)$

Proof. First we show that GETSUCCESSOR$_p(v')$ iterates over every successor w of v'. At Line 8, GETSUCCESSOR$_p(v')$ is called. Initially we have $Suc_p(v') = \text{POST}(v')$ (Precondition for UF-SCC-MAIN). By the specification of GETSUCCESSOR$_p(v')$, one state w is removed (locally) from $Suc_p(v')$ and returned. Therefore, GETSUCCESSOR$_p(v')$ only returns NULL (after which we continue at Line 17) if worker p has considered every successor $w \in \text{POST}(v')$. We now show that the invariant holds. For every successor w of v' in Lines 9-16, there are three cases based on the result from MAKECLAIM(w, p) in Line 9:

$result$ = CLAIM_DEAD: Here, by the specification of MAKECLAIM we can assure that ISDEAD(w) holds, conforming with the invariant. Thus, the algorithm may continue with the next successor.

$result$ = CLAIM_SUCCESS: Here, we recursively call UF-SCC$_p(w)$. By Lemma A.6 we obtain that ISDEAD(w) holds after Line 12, conforming with the invariant. Thus, the algorithm may continue with the next successor.

$result$ = CLAIM_FOUND: Here, we find that $p \in \text{PSET}(w)$ and we observe from the (proof of) Theorem A.11 that $v' \in \text{UF}[D_p.\text{TOP}()]$ holds at Line 14. Therefore, when the while procedure ends, we have that SAMESET$(D_p.\text{TOP}(), w)$ and thus SAMESET(v', w). As this conforms with the invariant, the algorithm may continue with the next successor.

Because every case results in ISDEAD$(w) \vee$ SAMESET(v', w), we conclude that the invariant holds. □

Lemma A.14. *In Algorithm 26 at Line 18, all successors of* UF$[v]$ *are either* DEAD *or in the same set as v and we say* GLOBALDEAD(v)*:*
(UF-SCC$_p(v)$ *at Line 18*) : $\forall w \in \text{NextSCC}(v) : \text{ISDEAD}(w) \vee \text{SAMESET}(v, w)$

Proof. By Lemma A.13 we have that at Line 17, all successors of w' are either DEAD or in the same set as v' (and v). We show that PICKFROMLIST(v) at Line 7 only returns NULL if all states in UF$[v]$ are fully explored. PICKFROMLIST(v) returns some state $v' \in List(v) \subseteq \text{UF}[v]$. Initially, $\forall v : List(v) = \text{UF}[v] = \{v\}$ (pre-condition UF-SCC-MAIN). By Lemma A.13 we have that v' is fully explored (by worker p) at Line 17. Therefore, we can globally remove v' from $List(v)$ using REMOVEFROMLIST(v'). After consecutive iterations of the while procedure spanning Lines 7-17, eventually $List(v) = \emptyset \implies v' = \text{NULL}$. We show by contradiction that every successor of $NextSCC(v)$ is fully explored. Assume that at Line 18: $\exists t \in \text{UF}[v] : t \notin List(v) \wedge \exists t' \in \text{POST}(t) : \neg\text{ISDEAD}(t') \wedge \neg\text{SAMESET}(t, t')$. Because $t \notin List(v)$, we have that REMOVEFROMLIST(t) has been called at Line 18 by any worker q as this is the only way to remove a state from $List(v)$. However, this would imply that Lemma A.13 holds for this state and worker, contradicting the assumption. Therefore, all successors of UF-SCC$[v]$ are either DEAD or in the same set as v and Lemma A.14 holds. □

Corollary A.15. *All reachable states (from v_0) are fully explored when the* UF-SCC-MAIN *procedure from Algorithm 26 terminates:*
$\{\ldots\}$ UF-SCC-MAIN$(v_0, \mathcal{P})$ $\{\forall v : v_0 \to^* v \implies \text{ISDEAD}(v)\}$

Proof. By applying Lemma A.14 recursively combined with Lemma A.6, we obtain that after UF-SCC$_p(v_0)$ finishes (as called by Line 4), we have that v_0 and all reachable states by v_0 are DEAD. Therefore, every reachable state from v_0 is fully explored. □

Theorem A.16 (Soundness)**.** *All reachable states (from v_0) which are part of* **the same** *SCC, are part of the same set when the* UF-SCC-MAIN *procedure from Algorithm 26 terminates:*
$\forall v : v_0 \to^* v \implies \forall w : v \to^* w \to^* v \implies \text{SAMESET}(v, w)$

Proof. By Corollary A.15 we obtain that every reachable state is fully explored by UF-SCC-MAIN. Assume by contradiction that $\forall v : v_0 \to^* v \implies \exists w : v \to^* w \to^* v \implies \neg\text{SAMESET}(v, w)$. Then, at some point we must have fully explored w (because $v_0 \to^* w$). Assume w.l.o.g. that at Line 7 we somw worker picks state

v from $List(v)$. Now, when the algorithm proceeds until Line 17, we find by Lemma A.13 that all (direct) successors of v are either DEAD or in the same set as v. Because $v \rightarrow^* w$ and $\neg\textsc{SameSet}(v, w)$ we deduce (by recursively iterating over the successors of v) that $\textsc{IsDead}(w)$ must hold. However, since $\neg\textsc{IsDead}(v)$, $\neg\textsc{SameSet}(v, w)$ and $w \rightarrow^* v$ holds, state w cannot be DEAD as inferred by Lemma A.14. Since it is not possible to have both $\textsc{IsDead}(v)$ and $\textsc{IsDead}(w)$ while exploring either of these states (this can only be concluded *after* a worker finishes exploring the state), we must contradict the assumption and conclude that Theorem A.16 holds. □

Theorem A.17 (Completeness)**.** *All reachable states (from v_0) which are part of* **different** *SCCs, are* **not** *part of the same set when the* UF-SCC-MAIN *procedure from Algorithm 26 terminates:*
$\forall v : v_0 \rightarrow^* v \Longrightarrow \forall w : v \not\rightarrow^* w \vee w \not\rightarrow^* v \Longrightarrow \neg\textsc{SameSet}(v, w)$

Proof. By Theorem A.11 we have that for any two states v, w which are part of the same UF set, we assure that $v \rightarrow^* w \rightarrow^* v$. Therefore, if we assume that there is a state $t \in \textsc{uf}[v]$ and $t \not\rightarrow^* v$ or $v \not\rightarrow^* t$, we contradict Theorem A.11. Hence, the completeness property is met at all times. □

Theorem A.18 (Termination)**.** *The* UF-SCC-MAIN *procedure from Algorithm 26 always terminates.*

Proof. For termination of UF-SCC-MAIN, we must show that every while procedure and every recursion terminates in UF-SCC$_p$. We consider each case as follows:

Recursive UF-SCC$_p(v)$ call of Line 12: Since UF-SCC$_p(w)$ is only called if $result = \textsc{claim\_success}$ is obtained from $\textsc{MakeClaim}(w, p)$. From the specification of $\textsc{MakeClaim}(w, p)$ we observe from the post-condition that $p \notin old(\textsc{Pset}(w)) \wedge p \in \textsc{Pset}(w)$, thus CLAIM_SUCCESS can be obtained at most once for every state and worker. Therefore, the number of UF-SCC$_p(v)$ calls is bounded by the number of reachable states from v_0.

While procedure of Line 14-16: From the previous case we obtain that UF-SCC$_p(v)$ is called a finite number of times, bounded by the number of reachable states. Therefore, this same bound applies on the size of D_p. In this while procedure we reduce $|D_p|$ by one in each iteration. Thus the number of times that this procedure may be executed is also bounded by the number of reachable states.

While procedure of Line 8-16: As both previous cases terminate, the body of the while procedure is finished in a finite number of steps. Since the $\textsc{GetSuccessor}_p(v')$ method from Line 8 reduces $|Suc_p(v')|$ by one in each call, this method is called at most $|Suc_p(v')| + 1$ times. Because $Suc_p(v') \subseteq \textsc{post}(v)$ we have that $|Suc_p(v')|$ is bounded by the fanout of v', which is also bounded by the number of reachable states.

While procedure of Line 7-17: Since all previous cases terminate, the body of the while procedure is finished in a finite number of steps. When the worker executes Line 17, we can conclude that $v' \notin List(v')$ (post-condition of $\textsc{RemoveFromList}(v')$). Therefore, this state cannot be picked again in consecutive iterations. Since $List(v) \subseteq \textsc{uf}[v]$ and $|\textsc{uf}[v]|$ is by Theorem A.11 bounded by the number of reachable states from v, we have that this procedure is called finitely often.

Since every sub-procedure terminates, we conclude that the UF-SCC-MAIN procedure also terminates. □

Appendix B

Complete algorithm for UF-SCC

In this appendix, we present the complete UF-SCC algorithm in Algorithm 28.

Algorithm 28 The complete UF-SCC algorithm

```
 1: procedure UF-SCC-MAIN($v_0$, $\mathcal{P}$)
 2:     for each $p \in [1 \ldots \mathcal{P}]$ do
 3:         MakeClaim($v_0$, $p$)
 4:     UF-SCC$_1$($v_0$) $\|$ ... $\|$ UF-SCC$_{\mathcal{P}}$($v_0$)

 5: procedure UF-SCC$_p$($v$)
 6:     $D_p$.push($v$) [start 'new' SCC]
 7:     while $v' :=$ PickFromList($v$) do
 8:         for each $w \in$ post($v'$) do
 9:             $result :=$ MakeClaim($w$, $p$)
10:             if $result =$ claim_dead then continue
11:             else if $result =$ claim_success then
12:                 UF-SCC$_p$($w$) [recursively explore $w$]
13:             else [$result =$ claim_found]
14:                 while $\neg$SameSet($D_p$.top(), $w$) do
15:                     $w' := D_p$.pop()
16:                     Merge($w'$, $D_p$.top())
17:         RemoveFromList($v'$) [fully explored post($v'$)]
18:     if $v = D_p$.top() then $D_p$.pop() [remove completed SCC]

19: procedure PickFromList($x$)
20:     $n_1 :=$ uf[$x$].list-next
21:     while uf[$x$].list-status $=$ tombstone do
22:         if $x = n_1$ then return null
23:         if uf[$n_1$].list-status $=$ tombstone then
24:             $n_2 :=$ uf[$n_1$].list-next
25:             if $x = n_2$ then return null
26:             uf[$x$].list-next $:= n_2$
27:             $x := n_2$
28:             $n_1 :=$ uf[$x$].list-next
29:         else return $n_1$
30:     return $x$
```

```
procedure RemoveFromList($x$)
    while UF[$x$].$list\text{-}status \neq$ TOMBSTONE do
        CAS(UF[$x$].$list\text{-}status$, LIVE, TOMBSTONE)

procedure MakeClaim($x, p$)
    if CAS(UF[$x$].$uf\text{-}status$, UNSEEN, INIT) then
        UF[$x$].$parent := x$
        UF[$x$].$list\text{-}next := x$
        UF[$x$].$list\text{-}status :=$ LIVE
        UF[$x$].PSet $:= \{p\}$
        UF[$x$].$uf\text{-}status :=$ LIVE
        return CLAIM_SUCCESS
    while UF[$x$].$uf\text{-}status =$ INIT ;
    if IsDead($x$) then return CLAIM_DEAD
    if HasWorker($x, p$) then return CLAIM_FOUND
    AddWorker($x, p$)
    return CLAIM_SUCCESS

procedure Merge($x, y$)
    if Lock($x, y$) then
        MergeLists($x, y$)
        Union($x, y$)
        Unlock($x$)
        Unlock($y$)

procedure Lock($x, y$)
    while $True$ do
        $x :=$ Find($x$)
        $y :=$ Find($y$)
        if $x = y$ then return $False$
        if $x > y$ then SWAP($x, y$)
        if CAS(UF[$x$].$uf\text{-}status$, LIVE, LOCKED) then
            if UF[$x$].$parent = x$ then
                if CAS(UF[$y$].$uf\text{-}status$, LIVE, LOCKED) then
                    if UF[$y$].$parent = y$ then
                        return $True$
                    UF[$y$].$uf\text{-}status :=$ LIVE
            UF[$x$].$uf\text{-}status :=$ LIVE

procedure Unlock($x$)
    UF[$x$].$uf\text{-}status :=$ LIVE
```

```
procedure $\textsc{MergeLists}(x, y)$
    $x := \textsc{LockList}(x)$
    $y := \textsc{LockList}(y)$
    $\textsc{swap}(\textsc{uf}[x].\textit{list-next}, \textsc{uf}[y].\textit{list-next})$
    $\textsc{uf}[x].\textit{list-status} := \textsc{live}$
    $\textsc{uf}[y].\textit{list-status} := \textsc{live}$

procedure $\textsc{LockList}(x)$
    while $\textsc{uf}[x].\textit{list-status} \neq \textsc{busy}$ do
        $x := \textsc{PickFromList}(x)$
        $\textsc{cas}(uf[x].\textit{list-status}, \textsc{live}, \textsc{busy})$
    return $x$

procedure $\textsc{Find}(x)$
    if $\textsc{uf}[x].\textit{parent} \neq x$ then
        $\textsc{uf}[x].\textit{parent} := \textsc{Find}(\textsc{uf}[x].\textit{parent})$
    return $\textsc{uf}[x].\textit{parent}$

procedure $\textsc{SameSet}(x, y)$
    $x_r := \textsc{Find}(x)$
    $y_r := \textsc{Find}(y)$
    if $\textsc{Find}(x_r) \neq x_r \vee \textsc{Find}(y_r) \neq y_r$ then
        $\textsc{SameSet}(x_r, y_r)$
    return $x_r = y_r$

procedure $\textsc{AddWorker}(x, p)$
    while $p \notin \textsc{uf}[\textsc{Find}(x)]$ do
        $x_r := \textsc{Find}(x)$
        $\textsc{uf}[x_r].\textsc{Pset} := \textsc{uf}[x_r].\textsc{Pset} \cup \{p\}$

procedure $\textsc{HasWorker}(x, p)$
    return $p \in \textsc{uf}[\textsc{Find}(x)].\textsc{Pset}$
```

```
95:  procedure UnionAux($x, y$)
96:      if CAS(UF[$y$].$parent, y, x$) then
97:          UF[$x$].PSet := UF[$x$].PSet $\cup$ UF[$y$].PSet
98:      if Find($x$) $\neq x \vee$ Find($y$) $\neq x$ then
99:          UnionAux(Find($x$), Find($y$))

100: procedure Union($x, y$)
101:     $x_r$ := Find($x$)
102:     $y_r$ := Find($y$)
103:     if UF[$x_r$].$rank >$ UF[$y_r$].$rank$ then
104:         UnionAux($x_r, y_r$)
105:     else
106:         UnionAux($y_r, x_r$)
107:         if UF[$x_r$].$rank =$ UF[$y_r$].$rank$ then
108:             UF[$y_r$].$rank$ := UF[$y_r$].$rank + 1$

109: procedure IsDead($x$)
110:     return PickFromList($x$) = NULL
```

Bibliography

[1] Richard J. Anderson and Heather Woll. Wait-free Parallel Algorithms for the Union-find Problem. In *Proceedings of the Twenty-third Annual ACM Symposium on Theory of Computing*, STOC '91, pages 370–380, New York, NY, USA, 1991. ACM.

[2] Christel Baier, Joost-Pieter Katoen, et al. *Principles of model checking*, volume 26202649. MIT press Cambridge, 2008.

[3] Jiří Barnat, Petr Bauch, Lubos Brim, and Milan Ceska. Computing strongly connected components in parallel on CUDA. In *Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International*, pages 544–555. IEEE, 2011.

[4] Jiří Barnat, Luboš Brim, and Petr Ročkai. A Time-Optimal On-the-Fly parallel algorithm for model checking of weak LTL properties. In *Formal Methods and Software Engineering*, pages 407–425. Springer, 2009.

[5] Jiří Barnat, Jakub Chaloupka, and Jaco van de Pol. Improved Distributed Algorithms for SCC Decomposition. *Electronic Notes in Theoretical Computer Science*, 198(1):63–77, 2008.

[6] Jiří Barnat, Jakub Chaloupka, and Jaco van de Pol. Distributed Algorithms for SCC Decomposition. *Journal of Logic and Computation*, 21(1):23–44, 2011.

[7] Jiří Barnat and Pavel Moravec. Parallel Algorithms for Finding SCCs in Implicitly Given Graphs. In *Proceedings of the 11th International Workshop, FMICS 2006 and 5th International Workshop, PDMC Conference on Formal Methods: Applications and Technology*, FMICS'06/PDMC'06, pages 316–330, Berlin, Heidelberg, 2007. Springer-Verlag.

[8] Roderick Bloem, HaroldN. Gabow, and Fabio Somenzi. An Algorithm for Strongly Connected Component Analysis in n log n Symbolic Steps. In *Formal Methods in Computer-Aided Design*, volume 1954 of *Lecture Notes in Computer Science*, pages 56–73. Springer Berlin Heidelberg, 2000.

[9] Andrei Broder, Ravi Kumar, Farzin Maghoul, Prabhakar Raghavan, Sridhar Rajagopalan, Raymie Stata, Andrew Tomkins, and Janet Wiener. Graph structure in the web. *Computer Networks*, 33(1):309–320, 2000.

[10] Edmund M. Clarke, Jr., Orna Grumberg, and Doron A. Peled. *Model Checking.* MIT Press, Cambridge, MA, USA, 1999.

[11] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition.* The MIT Press, 3rd edition, 2009.

[12] C. Courcoubetis, M. Vardi, P. Wolper, and M. Yannakakis. Memory-Efficient Algorithms for the Verification of Temporal Properties. In *Computer-Aided Verification*, volume 531 of *Lecture Notes in Computer Science*, pages 233–242. Springer Berlin Heidelberg, 1991.

[13] Jean-Michel Couvreur. On-the-Fly Verification of Linear Temporal Logic. In *World Congress on Formal Methods*, volume 1708 of *Lecture Notes in Computer Science*, pages 253–271. Springer, 1999.

[14] EdsgerW. Dijkstra. Finding the Maximum Strong Components in a Directed Graph. In *Selected Writings on Computing: A personal Perspective*, Texts and Monographs in Computer Science, pages 22–30. Springer New York, 1982.

[15] Sami Evangelista, Alfons Laarman, Laure Petrucci, and Jaco van de Pol. Improved Multi-Core Nested Depth-First Search. In Supratik Chakraborty and Madhavan Mukund, editors, *ATVA*, volume 7561 of *Lecture Notes in Computer Science*, pages 269–283. Springer, 2012.

[16] Sami Evangelista, Laure Petrucci, and Samir Youcef. Parallel Nested Depth-First Searches for LTL Model Checking. In Tevfik Bultan and Pao-Ann Hsiung, editors, *ATVA*, volume 6996 of *Lecture Notes in Computer Science*, pages 381–396. Springer, 2011.

[17] Kathi Fisler, Ranan Fraer, Gila Kamhi, Moshe Y. Vardi, and Zijiang Yang. Is There a Best Symbolic Cycle-Detection Algorithm? In Tiziana Margaria and Wang Yi, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 2031 of *Lecture Notes in Computer Science*, pages 420–434. Springer, 2001.

[18] Lisa Fleischer, Bruce Hendrickson, and Ali Pinar. On Identifying Strongly Connected Components in Parallel. In José D. P. Rolim, editor, *IPDPS Workshops*, volume 1800 of *Lecture Notes in Computer Science*, pages 505–511. Springer, 2000.

[19] Harold N. Gabow. Path-Based Depth-first Search for Strong and Biconnected Components. *Information Processing Letters*, 74(3-4):107–114, 2000.

[20] Andreas Gaiser and Stefan Schwoon. Comparison of algorithms for checking emptiness on büchi automata. In Petr Hlinený, Václav Matyás, and Tomás Vojnar, editors, *MEMICS*, volume 13 of *OASICS*. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany, 2009.

[21] Bernard A. Galler and Michael J. Fisher. An Improved Equivalence Algorithm. *Communications of the ACM*, 7(5):301–303, May 1964.

[22] Jaco Geldenhuys and Antti Valmari. Tarjan's Algorithm Makes On-the-Fly LTL Verification More Efficient. In Kurt Jensen and Andreas Podelski, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 2988 of *Lecture Notes in Computer Science*, pages 205–219. Springer Berlin Heidelberg, 2004.

[23] Xuan-Linh Ha, Thanh-Tho Quan, Yang Liu, and Jun Sun. Multi-core Model Checking Algorithms for LTL Verification with Fairness Assumptions. In *Software Engineering Conference (APSEC, 2013 20th Asia-Pacific*, volume 1, pages 547–552, Dec 2013.

[24] Timothy L Harris. A pragmatic Implementation of Non-Blocking Linked-Lists. In *Distributed Computing*, pages 300–314. Springer, 2001.

[25] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008.

[26] C. A. R. Hoare. An Axiomatic Basis for Computer Programming. *Communications of the ACM*, 12(10):576–580, October 1969.

[27] Gerard J. Holzmann, Rajeev Joshi, and Alex Groce. Swarm Verification Techniques. *IEEE Transactions on Software Engineering*, 37(6):845–857, 2011.

[28] Sungpack Hong, Nicole C. Rodia, and Kunle Olukotun. On fast parallel detection of strongly connected components (SCC) in small-world graphs. In William Gropp and Satoshi Matsuoka, editors, *SC*, page 92. ACM, 2013.

[29] John E Hopcroft and Jeffrey D Ullman. A Linear List Merging Algorithm. 1971.

[30] John E. Hopcroft and Jeffrey D. Ullman. Set Merging Algorithms. *SIAM Journal on Computing*, 2(4):294–303, 1973.

[31] Gijs Kant, Alfons Laarman, Jeroen Meijer, Jaco van de Pol, Stefan Blom, and Tom van Dijk. LTSmin: High-Performance Language-Independent Model Checking. In Christel Baier and Cesare Tinelli, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 9035 of *Lecture Notes in Computer Science*, pages 692–707. Springer Berlin Heidelberg, 2015.

[32] Elzbieta Krepska, Thilo Kielmann, Wan Fokkink, and Henri Bal. An Efficient Distributed Algorithm for Finding Terminal Strongly Connected Components.

[33] Ravi Kumar, Jasmine Novak, and Andrew Tomkins. Structure and Evolution of Online Social Networks. In *Link mining: models, algorithms, and applications*, pages 337–357. Springer, 2010.

[34] Vipin Kumar, Ananth Grama, Anshul Gupta, and George Karypis. *Introduction to Parallel Computing.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 2002.

[35] Alfons Laarman. *Scalable Multi-Core Model Checking.* PhD thesis, Enschede, The Netherlands, 2014.

[36] Alfons Laarman, Rom Langerak, Jaco van de Pol, Michael Weber, and Anton Wijs. Multi-core Nested Depth-First Search. In Tevfik Bultan and Pao-Ann Hsiung, editors, *ATVA*, volume 6996 of *Lecture Notes in Computer Science*, pages 321–335. Springer, 2011.

[37] Alfons Laarman and Jaco van de Pol. Variations on Multi-Core Nested Depth-First Search. In Jiří Barnat and Keijo Heljanko, editors, *PDMC*, volume 72 of *EPTCS*, pages 13–28, 2011.

[38] Guohui Li, Zhe Zhu, Zhang Cong, and Fumin Yang. Efficient decomposition of strongly connected components on GPUs. *Journal of Systems Architecture*, 60(1):1–10, 2014.

[39] Yang Liu, Jun Sun, and JinSong Dong. Scalable Multi-core Model Checking Fairness Enhanced Systems. In Karin Breitman and Ana Cavalcanti, editors, *Formal Methods and Software Engineering*, volume 5885 of *Lecture Notes in Computer Science*, pages 426–445. Springer Berlin Heidelberg, 2009.

[40] Gavin Lowe. Concurrent depth-first search algorithms based on Tarjans Algorithm. *International Journal on Software Tools for Technology Transfer*, pages 1–19, 2015.

[41] Yi Lu. Distributed Graph Computing Systems: Design, Implementation and Applications. 2015.

[42] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: A System for Large-scale Graph Processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pages 135–146, New York, NY, USA, 2010. ACM.

[43] Radu Mateescu. On-the-fly State Space Reductions for Weak Equivalences. In *Proceedings of the 10th International Workshop on Formal Methods for Industrial Critical Systems*, FMICS '05, pages 80–89, New York, NY, USA, 2005. ACM.

[44] Will McLendon, Bruce Hendrickson, Steven J. Plimpton, and Lawrence Rauchwerger. Finding Strongly Connected Components in Distributed Graphs. *Journal of Parallel and Distributed Computing*, 65(8):901–910, 2005.

[45] Ian Munro. Efficient Determination of the Transitive Closure of a Directed Graph. *Information Processing Letters*, 1(2):56–58, 1971.

[46] Simona Orzan. *On Distributed Verification and Verified Distribution.* PhD thesis, Free University of Amsterdam, 2004.

[47] Radek Pelánek. Typical Structural Properties of State Spaces. In Susanne Graf and Laurent Mounier, editors, *Model Checking Software*, volume 2989 of *Lecture Notes in Computer Science*, pages 5–22. Springer Berlin Heidelberg, 2004.

[48] Radek Pelánek. BEEM: Benchmarks for Explicit Model Checkers. In Dragan Bonaki and Stefan Edelkamp, editors, *Model Checking Software*, volume 4595 of *Lecture Notes in Computer Science*, pages 263–267. Springer Berlin Heidelberg, 2007.

[49] Franois Pottier. Depth-First Search and Strong Connectivity in Coq. In *Journes Francophones des Langages Applicatifs (JFLA 2015)*, January 2015.

[50] Paul Purdom Jr. A transitive closure algorithm. *BIT Numerical Mathematics*, 10(1):76–94, 1970.

[51] John H. Reif. Depth-First Search is Inherently Sequential. *Information Processing Letters*, 20(5):229–234, 1985.

[52] Etienne Renault, Alexandre Duret-Lutz, Fabrice Kordon, and Denis Poitrenaud. Three SCC-based emptiness checks for generalized Büchi automata. In Ken McMillan, Aart Middeldorp, and Andrei Voronkov, editors, *Proceedings of the 19th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR'13)*, volume 8312 of *Lecture Notes in Computer Science*, pages 668–682. Springer, December 2013.

[53] Etienne Renault, Alexandre Duret-Lutz, Fabrice Kordon, and Denis Poitrenaud. Parallel Explicit Model Checking for Generalized Büchi Automata. In Christel Baier and Cesare Tinelli, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 9035 of *Lecture Notes in Computer Science*, pages 613–627. Springer Berlin Heidelberg, 2015.

[54] Stefan Schwoon and Javier Esparza. A Note on On-the-Fly Verification Algorithms. In Nicolas Halbwachs and Lenore D. Zuck, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 3440 of *Lecture Notes in Computer Science*, pages 174–190. Springer, 2005.

[55] M. Sharir. A Strong-connectivity Algorithm and its Applications in Data Flow analysis. *Computers & Mathematics with Applications*, 7(1):67 – 72, 1981.

[56] George M Slota, Sivasankaran Rajamanickam, and Kamesh Madduri. BFS and Coloring-based Parallel Algorithms for Strongly Connected Components and Related Problems. In *IEEE International Parallel and Distributed Processing Symposium*, 2014.

[57] Robert Endre Tarjan. Depth-First Search and Linear Graph Algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972.

[58] Robert Endre Tarjan. Efficiency of a Good But Not Linear Set Union Algorithm. *Journal of the ACM (JACM)*, 22(2):215–225, 1975.

[59] Robert Endre Tarjan and Jan van Leeuwen. Worst-case Analysis of Set Union Algorithms. *Journal of the ACM*, 31(2):245–281, 1984.

[60] Jaco van de Pol. Automated Verification of Nested DFS. In Manuel Nez and Matthias Gdemann, editors, *Formal Methods for Industrial Critical Systems*, volume 9128 of *Lecture Notes in Computer Science*, pages 181–197. Springer International Publishing, 2015.

[61] Jan van Leeuwen and Theodorus Petrus Weide. *Alternative Path Compression Techniques.* Department of Computer Science, 1977.

[62] Duncan J Watts and Steven H Strogatz. Collective dynamics of 'small-world' networks. *Nature*, 393(6684):440–442, 1998.

[63] Anton Wijs, Joost-Pieter Katoen, and Dragan Bošnački. GPU-Based Graph Decomposition into Strongly Connected and Maximal End Components. In *Computer Aided Verification*, pages 310–326. Springer, 2014.