Exploring a Parallel SCC Algorithm
Using TLA$^+$and the TLC Model Checker

Jaco van de Pol[1,2(✉)]

[1] Department of Computer Science, Aarhus University,
Åbogade 34, 8200 Aarhus, Denmark
jaco@cs.au.dk
[2] University of Twente, DSI, Formal Methods and Tools,
P.O.-Box 217, 7500 AE Enschede, The Netherlands

Abstract. We explore a parallel SCC-decomposition algorithm based on a concurrent Union-Find data structure. In order to increase confidence in the algorithm, it is modelled in TLA$^+$. The TLC model checker is used to demonstrate that it works correctly for all possible interleavings of two workers on a number of small input graphs.

To increase the understanding of the algorithm, we investigate some potential invariants. Some of these are refuted, revealing that the algorithm allows suboptimal (but still correct) executions. Finally, we investigate some modifications of the algorithm. It turns out that most modifications lead to an incorrect algorithm, as revealed by the TLC model checker.

We view this exploration as a first step to a full understanding and a rigorous correctness proof based on invariants or step-wise refinement.

Keywords: Parallel SCC algorithm · Concurrent Union-Find data structure · PlusCal/TLA$^+$ specification · TLC model checker

1 Introduction

This paper studies a parallel algorithm for the detection of Strongly Connected Components (SCCs), which proceeds by sharing and merging partial SCCs that are maintained in a concurrent Union-Find data structure [2,3]. Previous work provided an informal correctness proof of the algorithm, and demonstrated a good experimental parallel speedup, even for graphs with a few large SCCs.

In order to increase confidence in the correctness of the algorithm, and to facilitate a detailed understanding, this paper presents a TLA$^+$specification of the algorithm. This is analysed in the TLA$^+$-toolbox [7], mainly by means of the TLC model checker [17]. The specification allows to check that for a limited number of workers and a couple of small graphs, all fair executions terminate, and all possible interleavings yield the correct SCC-decomposition.

In addition, to increase detailed understanding of the algorithm, we postulated a number of additional assertions and invariants, several of which were refuted by the model checker. The counter examples revealed some weird and

T. Margaria and B. Steffen (Eds.): ISoLA 2022, LNCS 13701, pp. 535–555, 2022.
https://doi.org/10.1007/978-3-031-19849-6_30

suboptimal (but still correct) executions, in which work is being duplicated. Finally, efforts to improve the algorithm failed, because the model checker discovered that several subtle modifications to the algorithm lead to errors.

Context. SCC decomposition has numerous applications as a fundamental building block in graph algorithms. We are mostly interested in applications in formal methods, such as model checking LTL properties, preprocessing for weak bisimulation reduction, and analysis of Markov chains. Since these methods are used for the verification of safety-critical systems, they must themselves be guaranteed correct. At the same time, intricate parallel algorithms have been designed, to scale the methods to realistic systems. To maximize parallel speed up, locking mechanisms are avoided where possible. This makes the correctness argumentation quite hard. This has led to the formal verification of several model checking algorithms, like a full LTL model checker [6], a model checker for Timed Automata [16], a sequential [12] and parallel algorithm [11] for Nested Depth-First Search, and sequential SCC-decomposition algorithms [4,8]. We are not aware of a formal verification of a parallel SCC algorithm.

There are many SCC decomposition algorithms. Tarjan provided the first sequential linear-time algorithm [14]. The algorithm analysed in this paper is closer to Dijkstra's sequential SCC algorithm [5]. Since then, several distributed SCC algorithms have been developed [1,10]. These algorithms apply to graphs that are partitioned among several workers. This paper is concerned with parallel SCC algorithms (multi-core parallelism), where the graph is in global, shared memory so all workers have access to it. Typically, the workers share some information on explored SCCs. Examples are the algorithm by Gavin Lowe [9] and the algorithm by Étienne Renault [13]. These algorithms make different tradeoffs when two workers start working on the same SCC. For instance, one could suspend one of the workers, or one could redo the whole SCC with a single worker. This paper studies the algorithm from Vincent Bloemen, following the presentation of his thesis [2], first published in [3]. The special feature of this algorithm is that multiple workers can work on the same SCC, sharing partial SCCs with each other. This is relevant for graphs with a few large SCCs.

2 Preliminaries

The SCC algorithm takes as input a rooted directed graph $G = (V, E, v_0)$, where V is a finite set of nodes (states), $E \subseteq V \times V$ is the set of directed edges (transitions), and v_0 is the root (initial state). The algorithm works on-the-fly (useful for model checking). It starts at v_0 and discovers new E-successors through a function $next : V \to 2^V$. We write $v \to^* w$ if (v, w) is in the transitive-reflexive closure of E. A subset $S \subseteq V$ is strongly connected if for all $v, w \in S$, $v \to^* w$. We call such S a *partial* SCC. A *strongly connected component* (SCC) of G is a *maximal* subset that is strongly connected.

Partitions of a set can be maintained in the Union-Find data structure, which is a forest where every element has a pointer to its parent in the same equivalence class. The elements that point to themselves are the roots. One finds the

representative of an element by following the parent pointers to the root. Two elements can be united by assigning the parent pointer of the root of one to the root of the other. These operations run in amortized (near)-constant time [15].

2.1 A Parallel SCC Algorithm Based on Concurrent Union-Find

We first provide an informal explanation of the parallel SCC decomposition algorithm based on concurrent Union-Find, following the presentation in the PhD thesis of Vincent Bloemen [2], originally published in [3]. We present the algorithm in a top-down fashion. The full original algorithm is included in Appendix A for easy reference. We refer to the thesis for more detailed explanations.

Main Procedure. The main procedure UFSCC (Algorithm 1) runs an independent Depth-First-Search (DFS) for each worker p. A newly visited node v is pushed on a worker-local stack R_p (line 7), which will contain states to be merged in partial SCCs. There are two globally shared data-structures:

- A Union-Find forest, storing the current partitioning in partial SCCs
- A Cyclic List, enabling an enumeration over all nodes in a partial SCC

The DFS proceeds per partial SCC, rather than per node: From node v, UFSCC is recursively called (line 12) for each "new" successor w of each node v' that is in the same partial SCC as v. The nodes v' are picked from the Cyclic List (line 8, 18). The calls to successors w are performed in random order (line 10), to encourage workers to operate on different parts of the graph.

If a worker has already visited some node in the partial SCC of w (but it is not yet fully explored), a loop has been detected: In this case, v and w and all intermediate nodes on the local stack R_p definitely belong to the same SCC, so they can be united in the UF-forest (line 14–16). Note that this partial SCC is not necessarily complete.

To find out if the partial SCC of successor w is either completely "explored", or if it was already "found" by worker p, or else if it is entirely "new" for worker p, we maintain two more pieces of globally shared information. This information applies to partial SCCs, so it is stored at the roots of the Union-Find forest.

- A UF-status, indicating if this partial SCC is completely explored,
- A bit set UF-workers, storing all workers that have found this partial SCC.

Based on the UF-status and the Worker set, the function MakeClaim (Algorithm 2) can easily determine the status of a newly visited node. To establish the status of a node a, it must first find the root of a (line 2), to obtain the information on the partial SCC. MakeClaim has a side-effect: If a is newly visited, the worker set of the root is updated (line 8).[1] This is a bit complicated, since in the mean-time, other workers may have extended this partial SCC, so the UF-root of a may have advanced. This is solved by the while loop (line 7–9), after which p must be in the worker set of the UF-root of a.

[1] Although not stated explicitly, this assignment is taken to be atomic.

Concurrent Union-Find Forest. We proceed to the explanation of the implementation of the UF-forest (Algorithms 3 and 4). The Find procedure (Algorithm 3) simply follows the parent-pointers, until a self-loop is found, indicating that we reached the root of this equivalence class. The path is shortened for future calls.

To determine if a and b belong to the SameSet (Algorithm 3), we find their roots and check if they are equal; if so, they definitely belong to the same SCC (line 16). Since SameSet is a non-atomic operation, it can happen that the (common) root of a and b has advanced in between. If so, we repeat the whole procedure in the new situation (line 18). If not, we return False, since (at least at some point during SameSet) a and b were in a different partial SCC.

The concurrent Unite function (Algorithm 4) is the most complicated. Given the nodes a and b to unite, it will find their minimal root q and maximal root r (line 11–15). The unite function has three tasks:

- The parent of q shall be r (the direction is fixed to avoid cycles);
- The union of the worker sets of q and r shall be stored at r;
- The cyclic lists of q and r shall be combined into a single cyclic list.

The main complication is that concurrent unites may happen. We avoid intermediate updates to q by locking its root. The lock is implemented in the UF-status (with values "Live", "Lock" and "Explored"). LockRoot uses an atomic Compare-And-Swap (CAS) to ensure that only one worker can hold the lock on q. It also checks that the root of q has not been advanced in the meantime. If we cannot lock q, the whole Unite-procedure is restarted (line 16). After obtaining the lock on q, its parent pointer can be safely updated to r (line 20). Note that q can never become a root, so there is no need to ever unlock its UF-status.

The cyclic lists must be merged as well. This is explained in the next subsection. Finally, we must store the union of the worker sets of r and q at the new root. Another complication arises: In the meantime, the root of r may have advanced! This is solved by the loop on line 21–24: We keep copying and uniting the worker set of the most recent root of r and q, until r is its own root.

Cyclic List. We now explain the Cyclic List, which is used to enumerate all nodes in a partial SCC. The cyclic list is implemented by a simple next-pointer. Each node also has a list-status, which can be "Busy", "Lock", or "Done". The main operations are to *enumerate* the "Busy" elements of the list (PickFromList, Algorithm 5), to *remove* elements from the list (RemoveFromList, Algorithm 5), and to *merge* two cyclic lists (integrated in Unite, Algorithm 4, line 17–19). Two auxiliary operations are to *lock* (LockList, Algorithm 4), and to *unlock* (integrated in Unite, Algorithm 4, line 25–26) elements in the list.

Merging two cyclic lists (Algorithm 4) proceeds by locking a "Busy" element from each list (line 17, 18), then swapping their two next-pointers (line 19), and finally unlocking them (line 25–26). LockList traverses the cyclic list (line 6, 9) and tries to "Lock" elements by a CAS-operation. If this succeeds, we return the locked element. Otherwise, we simply give up and try the next element in the list. Unlocking (line 25, 26) proceeds by assigning the list-status to "Busy" again.

The RemoveFromList function (Algorithm 5) updates the list-status from "Busy" to "Done" in a CAS operation (line 19). Note that "Locked" elements cannot be removed, but are retried (line 18) as long they are locked. Removed ("Done") nodes are still part of the cyclic list. We keep them, since other workers might still point to them, and they should be able to pick the next "Busy" node.

Finally, PickFromList (Algorithm 5) picks the next "Busy" node from the cyclic list. Starting at node a, it waits until a is unlocked (busy waiting, line 2,4). If a is "Busy", it can be returned (line 3). If a is "Done", we proceed to the next node, b (line 5). If $a = b$, the cyclic list must be empty, so the partial SCC is completely explored. This is properly stored in the UF-status field of the UF-root of a (line 8). A CAS operation is used, to ensure that only one worker reports a newly discovered SCC (line 9). In this case, we return NULL, since this SCC doesn't contain a "Busy" node. Otherwise, we wait until b is unlocked (busy-wait loop line 11–13). If b is "Busy", we return it. If b is also "Done", we need to proceed with the next state in the cyclic list, c (line 14, 16). Finally, we shorten the path $a \to b \to c$ (skipping b, line 15) to avoid long chains of removed nodes in the next call to PickFromList.

2.2 TLA$^+$ and TLC

The underlying logic of TLA$^+$ is simple, but powerful. The formalism is first-order predicate logic, where (untyped) variables range over sets (in the sense of ZF). TLA$^+$ comes with a library of predefined sets and standard operators on them, including natural numbers, functions, sequences, and the definitions of (linear time) temporal logic.

A system specification consists of a collection of state variables, and the definition of an initial predicate (on state variables) and a next-state predicate (on state variables and their primed variants). A TLA$^+$ specification is completed by weak (or strong) fairness assumptions, to ensure that there is some progress.

The PlusCal language allows the specification of algorithms using a simple programming language with assignments, if-then-else statements, while-loops, recursive procedures, and (fair) parallel composition. Atomicity is specified by adding program labels that serve as interleaving points. As a result, the PlusCal specification of the UFSCC algorithm follows the original pseudocode rather closely. Specifications in PlusCal are automatically translated to TLA$^+$.

Given the specification, one creates a finite model by fixing the number of workers, and fixing a particular input graph. We used the TLC model checker to prove that all interleavings allowed by the algorithm terminate and lead to a correct SCC decomposition. The TLC model checker supports parallel computation, symmetry reduction, and stores visited states on disk. TLC produces a counter example trace when a property is violated. We used the Visual Studio Code plug-in for TLA, for its great support to filter and navigate huge counter examples. Also, managing multiple specifications and models is well-supported.

Proving correctness for all number of workers, or for each input graph, is beyond the scope of model checking. We have not yet used the TLA proof checker, which would require to specify inductive invariants for the UFSCC algorithm.

3 Modeling and Analysis Process

We constructed an initial specification (Sect. 3.1) of the UFSCC algorithm in PlusCal, following the pseudocode (Appendix A) closely. We extended the specification with properties to express its functional correctness, i.e. the detection of the correct SCCs (Sect. 3.3). We added additional assertions to test some intuitions about the algorithm. Since the state space was large, we started with random simulation runs to test these assertions for small graphs and a few workers. Later we realized that the state space was actually infinite, due to recursive calls in busy-waiting loops. We then made a second specification (Sect. 3.2), avoiding recursive busy waiting. This allowed us to investigate the full state space.

We detected that some of the conjectured invariants did not hold (Sect. 4). The counter-examples to these invariants show some suboptimal traces, that we had not anticipated. However, we have not found traces that violate the overall correctness of the algorithm. It is of course possible that errors would occur on larger graphs, or with more workers. We also analysed some modifications of the algorithm, most of which were incorrect, as detected by the model checker.

3.1 Initial Specification – Good for Simulation

The initial specification follows the pseudocode in a rather straightforward manner. All in all, building the first executable model that covered the whole algorithm only took around one full day. Since TLA$^+$ is a rich specification language, the global data structures UF (Union-Find forest) and CL (cyclic list) can be modelled directly as mathematical functions. These globally shared variables are declared and initialized as follows. Here *init* is the initial state of the graph, and *Workers* is the set of worker identities, both specified in a separate model.

```
variables
  UF = [ n\in Nodes |-> [
            parent      |-> n,
            workers     |-> IF n=init THEN Workers ELSE {},
            uf_status   |-> "live"                          ]];
  CL = [ n\in Nodes |-> [
            next        |-> n,
            list_status |-> "busy"                          ]];
```

All procedures in Appendix A could be easily formalised as PlusCal procedures. As an example, we provide the Find-operation (Algorithm 3, line 1–4) in PlusCal, below on the left. On the right, we show the procedure RemoveFromList (Algorithm 5, line 17–18) in PlusCal, using a busy-wait loop and the macro CAS. In PlusCal, a procedure cannot return a result. Instead, we declared thread-local variables for the return value of each procedure (like returnFind below). In an attempt to reduce the state space, we reset these global variables at the call site, as soon as we have read the result (not shown here). Note that the labels indicate interleaving points: All statements between two labels occur atomically.

```
procedure Find(a)
    variable p;
    {
f1: p := UF[a].p;
f2: if (p /= a)
    { call Find(p);
f3:   UF[a].p := returnFind };
    else
    { returnFind := p };
f4: return
   }
```

```
procedure RemoveFromList(a) {
r1: while
        (CL[a].stat/= "done")
  {
  CAS(CL[a].stat,"busy","done")
  };
return
}
```

Main Procedure and Parallel Processes. The main procedure is UFSCC. We use the "with"-construct from PlusCal to select an arbitrary successor w_1 non-deterministically from the successors of v_1, which is the element picked from the cyclic list of v. Here *next* refers to the transitions in the input graph, which is modeled in a separate model.

The whole system consists of a weakly-fair parallel composition over all *Workers* (a constant set defined in a separate model), where each process executes UFSCC from the *initial* state. Each worker maintains its Roots stack as a thread-local variable, initialized as the empty sequence. We use weak fairness to avoid that processes stutter for ever, violating termination of the algorithm.

```
procedure UFSCC(v)
    variables v1, w, succ, ...
{
m1: Roots := <<v>> \o Roots;
    call PickFromList(v);
m2: v1 := returnPick;
m3: while (v1 /= null) {
      succ := next[v1];
m4:   while (succ /= {}) {
        with (w1 \in succ) {
          w := w1;
          succ := succ \ {w1} };
... } ... } ... }
```

```
fair process (W \in Workers)
   variables
   Roots = << >>,
   returnFind, returnPick ;
{ main: call UFSCC(init); }
```

Atomicity. PlusCal uses program labels to specify atomicity: code between two program labels is executed atomically; interleaving (and branching) can only happen at program labels. We took the following modeling decision: To ensure that every "atomic" block performs at most one global memory access, we introduce a label (like f1) for each program statement with a global memory access. PlusCal also requires labels for loops, procedure calls and returns, etc.

There are two exceptions to the rule: The first exception is when the pseudocode insisted on atomic updates. For instance, line 23 in Algorithm 4 (Unite) states that

the worker set must be updated atomically. So, despite three global memory accesses, we model this line with only one label in the TLA$^+$ specification:

```
u11: UF[r].workers := UF[r].workers \union UF[q].workers;
```

In contrast, line 19 of the same algorithm tells that the pointer Swap happens non-atomically. So this we modelled using multiple labels (i.e., breaking it in two atomic pointer assignments), as shown on the left below.

The other exception is where the pseudocode uses CAS statements, which we modeled by a macro, shown below on the right. Note that macros cannot contain labels, so they are treated as atomic blocks by definition.

```
u8: tmp := CL[a].next;
    CL[a].next := CL[b].next;
u9: CL[b].next := tmp;
```

```
macro CAS(x,old,new) {
  returnCAS := (x=old);
  if (returnCAS) { x:=new }
}
```

Deviations from the Original Algorithm. We made the following deviations from the pseudocode when interpreting it in the specification:

- We slightly rearranged the code for esthetic reasons. For instance, we split UF and CL in two data-structures. We also inlined the LockRoot procedure, which was only called from one place in Unite.
- To simplify the main process, we did not model the initial assignment to the worker set (Algorithms 1, line 3, 4), but instead, we did this in the initialisation of the UF data structure (as shown in the code above).
- Although not indicated explicitly, we took the worker set update in MakeClaim (Algorithm 2) as an atomic update, similar to the update in Unite.
- Note that PickFromList returns NULL if the cyclic list is empty. This procedure is called: (i) From the main procedure UFSCC (Algorithm 1, line 8, 18); the while loop terminates when PickFromList returns NULL. (ii) From LockList, but now the case that NULL is returned is not handled. To make this an explicit assumption, we replaced line 7 in LockList (Algorithm 4) by an assertion checking for NULL. We have not detected a violation of this assertion.

3.2 Improved Specification – Good for Model Checking

The initial specification led to very large state spaces, even when run with only 2 workers on a graph with only 3 nodes. As a consequence, initially we could not apply complete model checking, but only run simulations.

Busy Waiting in Recursion. Later, we realized that the state space was actually infinite. The reason is that the pseudocode models busy-waiting loops with recursion. For instance, the procedure LockList (Algorithm 4, l. 5–9) was

initially modeled as shown below (left). Note that if the CAS fails, we retry locking the list by the recursive call at line 9. Recall that PlusCal is translated to TLA$^+$; the translation involves the introduction of a stack to model procedure calls and recursion. Although the LockList procedure terminates (after the other worker releases the lock), an unbounded number of executions of the loop can happen in between, leading to an unbounded stack.[2]

Our remedy was to replace tail-recursive calls by goto-statements. Note that the re-specification of LockList on the right below leads to a finite state space. We replaced all tail-recursion by goto-statements, to reduce the state space.

```
procedure LockList(a)
  variable s
  {
l1: call PickFromList(a);
l2: s := returnPick;
    assert s /= null;
    CAS(CL[s].stat,
           "busy", "lock");
l3: if (returnCAS) {
        returnLock := s;
        return
    } else {
        call LockList(s);
l4:     return
    }
} \* 1st model: tail recursion
```

```
procedure LockList(a)
  variable s
  {
l1: call PickFromList(a);
l2: s := returnPick;
    assert s /= null;
    CAS(CL[s].stat,
           "busy", "lock");
l3: if (returnCAS) {
      returnLock := s;
      return
    } else {
      a := s;
      goto l1;
    }
} \* 2nd model: goto-loop
```

Replace Busy-Wait by Await. In a final attempt to reduce the state space, we tried to avoid busy-wait loops at all, by using the *await* statement of PlusCal. As an example, we show the start of the procedure PickFromList (Algorithm 5, line 1–3). On the left we show the initial specification in PlusCal (we modelled the do-while by a goto statement). On the right, we show the improved version, where the busy-waiting loop with goto is replaced by the await-statement.

```
procedure PickFromList(a)
  variable status, b, c, root;
  {
pX: status := CL[a].stat;
     if (status = "lock")
       { goto pX } else
     if (status = "busy") {
         returnPick := a;
p1:      return
     };  ... }
```

```
procedure PickFromList(a)
  variable b, c, root ;
  {
pX: await CL[a_P].stat /= "lock";
    if (CL[a].stat = "busy") {
        returnPick := a;
p1:     return
     };
...}
```

[2] In this tail-recursive case, the translation could have avoided the use of a stack.

We have no formal justification for these modifications, but now at least the state space of the algorithm for a fixed graph and set of workers is finite. For 2 workers and graphs of 4 nodes (like those in Appendix B), the state space is around 2–15 Million nodes. For state spaces of this size, model checking is feasible on a consumer-laptop and runs within a couple of minutes. For 3 workers on a graph of only 3 nodes, the state space grew already to 67 Million nodes. This becomes painful for larger graphs, but one could still fall back on simulation.

3.3 Specifying the Correctness Property and Other Assertions

Next to the specification of the system, we need to specify the correctness criterion. The main claim is that UFSCC terminates, and upon termination the Union-Find forest contains the graph partitioning in the correct SCCs. We specified and checked the expected SCCs for each model instance separately.

Model Instances. A model instance is specified in a separate configuration file. The example below (left) shows a fragment of a model instance. It specifies a graph of 4 nodes and 5 edges. Note that the sequence of edges can be interpreted as a function from nodes to set of nodes. It can be easily checked that this graph has two SCCs. Since our Union-Find structure always takes the largest node as representative, we can specify the expected root for each node. These expected SCCs will be used to express the correctness claim.

The model instance also provides a value to the set of workers. In the example below (right), we introduce two workers w_1 and w_2 as distinct model constants.

```
Nodes == { 1,2,3,4 }
next == << {2}, {1,3}, {4}, {3} >>
init == 1
expected == << 2, 2, 4, 4 >>
```

```
CONSTANTS
    w1 = w1
    w2 = w2
    Workers = {w1, w2}
```

Main Correctness Claim. For the main correctness claim, we define an operator (logical function) that computes the root of a node in the UF-forest *in a single snapshot*. Note that this is quite different from the Find-procedure, which does not work atomically (the UF forest can be modified by concurrent workers) and has a side effect (path shortening). The ideal find operator is defined recursively. Note that TLA$^+$ allows that recursive operators are only partially specified. In this case, if the UF-structure contained loops, the value of "find" would not be defined everywhere.

```
RECURSIVE find(_)
   find(n) == IF UF[n].parent=n THEN n ELSE find(UF[n].parent)
Correct == (\A w\in Workers : pc[w] = "Done") =>
                          (\A x\in Nodes : find(x) = expected[x])
Termination == <>(\A w \in Workers: pc[w] = "Done")
```

Given the logical find-operator and the expected SCCs, partial correctness can be stated easily. In *Correct* above, we state that when all workers are done, the root of every node in the graph is as expected. Note that this is an invariant that trivially holds for all states, except those where all processes have finished. The LTL property *Termination* indicates that all fair runs lead to a state where all processes have returned from the initial UFSCC call. Together, *Correct* and *Termination* (and the fact that the program doesn't crash halfway due to a type/value error) specify total correctness under weak fairness.

4 Findings from Model Checking Experiments

The specification in the previous section has a finite state space, given a fixed input graph and a fixed number of workers. So in principle, the TLC model checker can generate the full state space and explore if the algorithm works correctly for all possible interleavings, and all possible graph traversals (recall that the next successor is selected non-deterministically). The *bad news* is that the state space could only be computed for rather small graphs and a few workers. The *good news* is that the algorithm was correct for all instances that we tried. We tried 2 workers on 10 graphs of 3–4 nodes, from various initial positions. See Appendix B for an impression of a few input graphs.

This is insufficient information to conclude that the algorithm is also correct for more workers on larger input graphs. The royal road to increase the confidence in the algorithm would be to identify and prove a number of inductive invariants that imply correctness. At the moment we don't know the proper invariants of the algorithm. The intermediate contribution of this work is to investigate a number of potential invariants. It appears that several conjectured invariants actually don't hold, as revealed by some weird (but not wrong) executions (Sects. 4.2, 4.3). This might diminish the confidence in the algorithm.

We also studied a number of modifications of the algorithm. These are partly inspired by efforts to "restore" some conjectured invariants and avoid weird executions, and partly by a wish to simplify or restructure the specification in clear layers. In particular, we would wish to separate code at the UF level from code at the CL level. We believe that this would facilitate a proof by step-wise refinement. Currently, it is mainly the Unite-procedure that mixes the two levels.

Our findings indicate that most modifications made the algorithm wrong (Sects. 4.1, 4.4). We don't know if this should increase or decrease the confidence in the algorithm: On the one hand, it shows that the correctness of the algorithm is rather fragile, and one could imagine that the original algorithm fails on a slightly different input graph. On the other hand, it shows that wrong algorithms can be caught by model checking, even with 2 workers running on the 10 small input graphs that we constructed. We will now discuss these findings in more detail.

4.1 Simplifying the Equivalence Check (SameSet)

Consider the procedure SameSet (Algorithm 3, l. 13–18). It finds the roots of a and b. If the roots are equal, a and b clearly belong to the same set. The reverse

is actually not true in a concurrent setting! First of all, SameSet could return False, even though a and b have been united in between by another worker. One could argue that this result is correct, since SameSet should have returned False if it had just been a bit faster. Still, at line 17, we only return False if the root of a has not changed in the meantime. Otherwise, we start all over (l. 18).

The problem with this approach is that it seems complicated, it looks asymmetric (we don't check if the root of b has changed), and it seems to relieve the symptom rather than the cause of the problem: what if the root of a is updated right after we check that it wasn't updated in line 17?

For these reasons, we tried a simplification, replacing line 17 and 18 by just returning "False". However, this simple change leads to serious consequences:

- On some graphs, the modified UFSCC gave wrong answers, by merging different SCCs into a single one.
- On some graphs, the modified UFSCC crashed, by attempting to pop an element from the empty Roots-stack.

The TLC model checker produces counter-examples, i.e. concrete runs of the modified algorithm that lead to the problematic behaviour. Although these traces get long, after some analysis they explained why these problems occur.

The disturbing situation in SameSet occurs when a and b are initially in the same partition, but the root of this partition is updated in between finding the root of a and b. In that case, the roots seems different, and a wrong result is reported. In this sense, returning False directly after line 16 would result in a SameSet procedure that is not even reflexive.

Now why is this problematic? The procedure SameSet is called by the main procedure UFSCC while popping roots from the stack *as long as the source and target of the current transition $v \to w$ are not in the same set!* (Algorithm 1, l. 14–16). The effect of the erroneous version of SameSet is that we keep popping and uniting states from the Roots stack. This means that an SCC is either merged with the previous SCC on the stack, or if there is no such SCC on the stack, we try to pop from the empty stack!

We conclude that the complication in SameSet is necessary. As already explained in [2], it guarantees to return True if and only if a and b are in the same set at some point during the execution of SameSet. We believe that this requirement can be formalized using the logical (ideal) find-operator as follows:

$$\big(\mathit{find}_{pre}(a) = \mathit{find}_{pre}(b)\big) \Rightarrow \big(\mathit{returnSame} = \mathit{True}\big) \Rightarrow \big(\mathit{find}_{post}(a) = \mathit{find}_{post}(b)\big)$$

4.2 Monotonicity of Worker Sets and Atomic Updates

The worker set is used to detect cycles in the graph: In UFSCC (Algorithm 1) and MakeClaim (Algorithm 2), if a worker p explores an edge (v, w) and p already occurs in the worker set of the root of w, then it ran into a cycle, and it can be concluded that v and w are in the same SCC. Recall that the worker sets are updated atomically in Unite (Algorithm 4, l. 23) and MakeClaim (Algorithm 2, l. 8).

A reasonable conjecture is that these atomic updates ensure that the worker sets are monotonically increasing. To challenge the specification, we check that this property holds for atomic worker updates, but is violated if we update the worker set in a non-atomic manner. The property is easily formalized as a TLA property on all reachable transitions, as *Monotonic1* below.

```
Monotonic1 == [][\A y\in Nodes :
                    UF[y].workers \subseteq UF[y].workers']_<<UF>>
Monotonic2 == [][\A y\in Nodes :
       UF[find(y)].workers \subseteq UF[find(y)].workers']_<<UF>>
```

Indeed, this property holds with atomic updates, but it is violated by a version of the specification with non-atomic updates. The model checker returns an execution which boils down to the following well-known scenario: Assume the worker set is X and it is extended concurrently by Y_1 and Y_2. With atomic updates, the end result is $(X \cup Y_1) \cup Y_2$ or $(X \cup Y_2) \cup Y_1$, which denotes the same set. With non-atomic updates, the workers can first both read X, and then one writes $X \cup Y_1$ and subsequently the other writes $X \cup Y_2$. The second update violates monotonicity, since the update from Y_1 is lost.

Note that property *Monotonic1* only looks at the worker set *per node*, while the relevant information would check the worker set *per partial SCC*. We challenged the specification further with property *Monotonic2*, which states that for every node, the worker set of its *root node* is increasing. Surprisingly, the model checker returned an execution where this property is failing, even in the model with atomic updates! After analysing the execution, the following scenario is possible due to the order of steps in Unite (Algorithm 4, l. 20–23).

Assume worker w_1 unites roots r and q, with worker set W_r and W_q. It first updates the parent pointer of q to r. In a second step, it adds W_q to W_r. But *in between these steps the invariant is violated*, since the worker set of q's root r is still W_r, suddenly missing elements from W_q. Assume that in between these two steps, another worker w_2 checks the worker set of a node v whose parent points to q (in MakeClaim). Even though w_2 might be in the worker set of q, it is *not yet* in the worker set of r, the new root of v! The situation will be restored soon by w_1, but it is too late: w_2 has already decided that its node v is "new", instead of recognizing that its partial SCC was already "found".

Is this bad? Apparently, the violation of *Monotonic2* doesn't lead to a wrong result. Still, it can lead to extra computations. If worker w_2 doesn't find itself in the worker set of the root of node v, it will continue the search, potentially revisiting a part of the graph unnecessarily. We tried to construct examples where this behaviour could lead to wrong answers, but we didn't succeed. Also, the computation always terminated (there are only finitely many unites). However, these examples led to discovery of violations of other expected properties, as reported in Subsect. 4.3. Our attempts to improve the algorithm so that property *Monotonic2* holds failed, as reported in Subsect. 4.4.

4.3 Duplication on the Stack

Related to non-monotonicity of the worker sets, we investigated another property that indicates duplication of work: this time, we check an assertion at the beginning of the UFSCC loop that the node v that is pushed on the Roots-stack R is not already on the stack. Indeed, this property was violated on the same graphs that violated the *Monotonicity2* property.

Two BSc students at AU, Jesper Steensgaard and Jonathan Starup, discovered another reason that some nodes occur multiple times on the Roots stack: UFSCC traverses all nodes v' in the same partial SCC, and recurses on all their successors w. This causes double work, since some of these successors belong to the same SCC, and will only be removed from the cyclic list when we backtrack from them. One could skip those successors w that belong to the current SCC.

4.4 Changing the Order of Updates During Unite

Finally, we tried some variants of the algorithm, changing the order of the steps in the Unite procedure. After initialisation, Unite performs the following 5 steps:

1. It locks two elements in the cyclic lists (l. 17, 18)
2. It merges the cyclic lists (l. 19)
3. It updates the parent pointer (l. 20)
4. It updates the worker set of the root in a loop (l. 21–24)
5. It releases the list locks (l. 25–26)

Note that, since these steps don't happen atomically, other workers might see inconsistent states, where for instance the cyclic lists of two nodes are already merged, but the two nodes are not yet the same according to the UF-forest. Similarly, they could be the same, but the worker set has not yet been updated. The "Locked" value of the UF-status and the CL-status are the only warning signs for other workers that something might be wrong.

We tried several modifications of the order of these 5 steps. TLC discovered problems for several reorderings, in particular it revealed concrete counter examples for (1, 2, 4, 3, 5), where the worker set is updated before the parent pointer. We had hoped that this order would restore the monotonicity property.

TLC also reported concrete counter-examples to (3, 4, 1, 2, 5) and (1, 2, 5, 3, 4), where updating the UF-parent happens outside the region between locking and unlocking the CL-nodes. We had hoped to minimize the locked region, to potentially increase performance, and also to separate the CL-related code from the UF-related code. Also, one would think that the CL-list locks only need to protect updates to the CL-list, but this is apparently not true.

On the other hand, we found no concrete counter-examples to the modification (1, 2, 3, 5, 4), so it seems that the update of the worker set can happen outside the locked region. Moving the loop of lines 21–24 outside the locked region could potentially give a performance speedup. Also, the order of updating the parent pointer and the cyclic list doesn't seem to matter: We found no counter-examples to (1, 3, 2, 4, 5) and (1, 3, 2, 5, 4). From our experiments, we conclude that:

– The parent pointer must be updated *before* the worker set is updated.
– The update of the UF parent-pointer must happen inside the CL lock region.

If one violates the first requirement, as in (1, 2, 4, 3, 5), the following can happen: Assume worker w_1 starts at some node n_1 and worker w_2 finds out that nodes n_1 and n_2 can be united. The new root will be n_2, so worker w_2 updates the worker set at n_2 to $\{w_1, w_2\}$ *before it updates the parent pointer.* In the meantime, worker w_1 now explores the edge from n_1 to n_2. It notices that it already found (the SCC of) node n_2 (since w_1 is now in n_2's worker set). However, n_1 and n_2 are not yet the same (since the parent pointer has not yet been updated by w_2). Hence, w_1 starts popping the Roots stack, possibly leading to spurious unites, or even to a crash due to popping from the empty stack (TLC found both scenarios).

If one violates the second requirement by updating the parent pointer *before* locking the list elements, as in (3, 4, 1, 2, 5), another worker might already remove the elements that still must be locked in the cyclic list. In this case, LockList (Algorithm 4) crashes due to the assertion triggered because PickFromList (Algorithm 5) returns NULL.

If one violates the second requirement by updating the parent pointer *after* unlocking the list elements, as in (1, 2, 5, 3, 4), it can happen that the *last node* returned by PickFromList (Algorithm 5) has already an unlocked CL-status, but its UF-status is still "Locked" instead of "Live". But then the CAS-operation on line 8 fails, so the UF-status of this node is never updated to "Explored". When backtracking in UFSCC and visiting the same node again through another path, it is not "Explored" so it will be considered "Found" and UFSCC erroneously starts popping from the Roots stack (Algorithm 1, l. 14–16).

This subtle error shows that the proper invariants of the algorithm should somehow relate the UF-status and the CL-status, to avoid that UF-locked states are returned by PickFromList.

5 Conclusion

In an attempt to understand a parallel SCC-detection algorithm based on a concurrent Union-Find forest, and to increase the trust in its correctness, we modeled the algorithm in TLA$^+$ and analysed it for a number of small graphs with the TLC model checker. The analysis revealed that the algorithm behaves correctly for 2 workers on a couple of small graphs. This does not prove its correctness, but it may increase confidence in its correctness.

We also showed that some natural invariants can be violated, leading to suboptimal runs of the algorithm with some work duplication. This may reduce the trust in the correctness, since such behaviour might violate the correctness on larger graphs or with more workers (beyond the horizon of the model checker). We did not find such examples, though. The original author was aware of possible work duplication. It should not affect correctness and it occurs only with a small probability. Avoiding it with extra locks could be even more costly. Experimental evaluation has demonstrated good speedups of the current strategy [2,3].

Finally, we tried to increase our understanding and confidence by slight modifications to the algorithm. The results are mixed. Some small modifications seem to be possible without violating correctness. One of the suggested modifications might even increase parallel speedup, by moving the loop to update the worker set outside the locked region. We tried some other small modifications to maintain interesting invariants that could avoid duplicate work and keep worker-set information monotonic. However, these modifications were erroneous and could be refuted by small graphs. Apparently, larger modifications would be required to maintain these invariants.

These experiments also reveal that the invariants needed for a full understanding and correctness proof of the algorithm will be quite complicated. We hope that the scenarios revealed by model checking and reported in this work will contribute to the discovery of the proper invariants and potentially to simplifications of the algorithm. Of course, these simplifications should not decrease the parallel performance of the algorithm. For instance, adding extra locks would ease correctness reasoning, but at the expense of decreased parallel performance.

Acknowledgements. The author acknowledges the investigation by Jesper Steensgaard and Jonathan Starup, who found some suboptimal behaviours in the cyclic list during their BSc talent-track project. The author is grateful to Stephan Merz for introducing him to TLA$^+$ during an extended research visit at Nancy (France). The author acknowledges useful discussions with Simon Thrane Hansen (AU) on simplifying the model and help with the TLC model checker. Finally, he is grateful to Stephan Merz, Vincent Bloemen, and the anonymous reviewers for useful feedback on a draft of the paper.

A The Original Concurrent Union-Find SCC Algorithm

Here we present the original algorithm UFSCC, taken from the thesis of Vincent Bloemen [2]. The pseudo-code is copied with consent of the author. We present the algorithm top-down, in Algorithm 1–6.

Algorithm 1 Implementation of the UFSCC algorithm ([2], Alg. 9)

```
1  ∀p ∈ [1...P] : R_p := ∅ // local stack for each worker
2  function UFSCC_main(v_0, P)
3      forall p ∈ [1...P] do
4          MakeClaim(v_0, p) // set worker IDs for state v_0
5      UFSCC_1(v_0) ‖ ... ‖ UFSCC_P(v_0) // run in parallel

6  function UFSCC_p(v)
7      R_p.Push(v)
8      v' := PickFromList(v) // pick a Busy state from S(v)
9      while v' ≠ NULL do
10         forall w ∈ Random(Succ(v')) do
11             claim := MakeClaim(w, p) // (ignore claim_explored)
12             if claim = claim_new then UFSCC_p(w) // (locally) new
13             else if claim = claim_found then // cycle detected
14                 while ¬SameSet(v, w) do
15                     r := R_p.Pop()
16                     Unite(r, R_p.Top()) // merge top two stack states
17         RemoveFromList(v') // fully handled all states from Succ(v')
18         v' := PickFromList(v') // pick next state
19     if v = R_p.Top() then R_p.Pop() // remove completed SCC
```

Algorithm 2 Determining the status of a state with the worker set ([2], Alg. 6)

```
1  function MakeClaim(a, p)
2      root_a := Find(a)
3      if UF[root_a].uf_status = Explored then
4          return claim_explored // completely explored SCC
5      if p ∈ UF[root_a].workers then
6          return claim_found // set already contains worker ID
7      while p ∉ UF[root_a].workers do
8          UF[root_a].workers := UF[root_a].workers ∪ {p} // add worker ID
9          root_a := Find(root_a) // ensure that the worker ID is added
10     return claim_new
```

Algorithm 3 Find and SameSet functions ([2], Alg. 5)

```
1  function Find(a)
2      if UF[a].parent ≠ a then
3          UF[a].parent := Find(UF[a].parent) // path compression
4      return UF[a].parent // return the root

13 function SameSet(a, b)
14     root_a := Find(a)
15     root_b := Find(b)
16     if root_a = root_b then return True
17     if UF[root_a].parent = root_a then return False
18     return SameSet(root_a, root_b)
```

(we removed the Unite-function from Alg. 3, since it will be refined by Alg. 4)

Algorithm 4 The Unite procedure of the iterable union-find ([2], Alg. 8)

```
1  function LockRoot(a)
2      if CAS(UF[a].uf_status, Live, Lock) then
3          if UF[a].parent = a then return True
4      return False // unable to lock root (unnecessary to unlock)

5  function LockList(a)
6      s := PickFromList(a) // pick a Busy state
7      if s = NULL then return NULL // cannot lock an explored SCC
8      if CAS(UF[s].list_status, Busy, Lock) then return s
9      return LockList(s) // locked by another worker, try again

10 function Unite(a, b)
11     root_a := Find(a)
12     root_b := Find(b)
13     if root_a = root_b then return // already united
14     r := max(root_a, root_b) // largest index is the new root
15     q := min(root_a, root_b)
16     if ¬LockRoot(q) then return Unite(root_a, root_b) // try again
17     a_List := LockList(a) // lock the list states
18     b_List := LockList(b)
19     Swap(UF[a_List].next, UF[b_List].next) // non-atomic instruction
20     UF[q].parent := r // update parent (to some state in S(r))
21     do // update the worker set for the root
22         r := Find(r)
23         UF[r].workers := UF[r].workers ∪ UF[q].workers // atomic
24     while UF[r].parent ≠ r // ensure that we update the root
25     UF[a_List].list_status := Busy // unlock the locked list states
26     UF[b_List].list_status := Busy
```

Algorithm 5 The cyclic-list part of the iterable union-find ([2], Alg. 7)

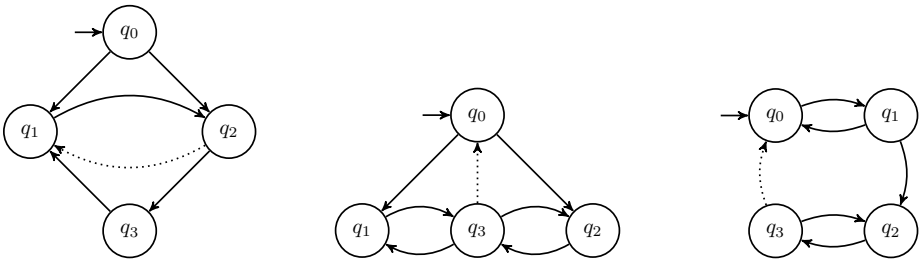
```
1  function PickFromList(a)
2      do // wait until a is not locked (another worker may be uniting)
3          if UF[a].list_status = Busy then return a
4      while UF[a].list_status ≠ Done
5      b := UF[a].next // a.list_status = Done (exit from do-while)
6      if a = b then // self-loop of a Done state ⇒ all states are Done
7          root_a := Find(a) // mark the SCC Explored, if not already done
8          if CAS(UF[root_a].uf_status, Live, Explored) then
9              report SCC root_a // optionally, report the SCC root
10         return NULL // no state to pick from an Explored SCC
11     do // also make sure that b is not locked, return b if it's Busy
12         if UF[b].list_status = Busy then return b
13     while UF[b].list_status ≠ Done // exit if status = Done
14     c := UF[b].next // we have a → b → c via next pointers
15     UF[a].next := c // a and b are Done ⇒ shrink the list: a → c
16     return PickFromList(c) // recursively traverse the cyclic list

17 function RemoveFromList(a)
18     while UF[a].list_status ≠ Done do // try again if a is locked
19         CAS(UF[a].list_status, Busy, Done) // only remove Busy state
```

Algorithm 6 Atomic Compare and Swap instruction ([2], Alg. 4)

```
1  function CAS(x, a, b)
2      if x = a then
3          x := b
4          return True
5      else return False
```

This algorithm is supposed to run in one atomic step.

B Example Input Graphs for SCC Algorithm



Above we show six small example input graphs on which we tested the TLA$^+$ specification of the pseudocode in Appendix A. (The dotted lines are optional).

References

1. Barnat, J., Chaloupka, J., van de Pol, J.: Distributed algorithms for SCC decomposition. J. Log. Comput. **21**(1), 23–44 (2011). https://doi.org/10.1093/logcom/exp003
2. Bloemen, V.: Strong connectivity and shortest paths for checking models. Ph.D. thesis, University of Twente, July 2019. https://doi.org/10.3990/1.9789036547864
3. Bloemen, V., Laarman, A., van de Pol, J.: Multi-core on-the-fly SCC decompositiona. In: PPOPP, pp. 8:1–8:12. ACM (2016). https://doi.org/10.1145/2851141.2851161
4. Chen, R., Cohen, C., Lévy, J., Merz, S., Théry, L.: Formal proofs of Tarjan's strongly connected components algorithm in Why3, Coq and Isabelle. In: ITP, LIPIcs, vol. 141, pp. 13:1–13:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2019). https://doi.org/10.4230/LIPIcs.ITP.201a9.13
5. Dijkstra, E.W.: Finding the maximum strong components in a directed graph. In: Selected Writings on Computing: A Personal Perspective. Texts and Monographs in Computer Science, Springer, New York (1982). https://doi.org/10.1007/978-1-4612-5695-3
6. Esparza, J., Lammich, P., Neumann, R., Nipkow, T., Schimpf, A., Smaus, J.-G.: A Fully verified executable LTL model checker. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 463–478. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39799-8_31
7. Kuppe, M.A., Lamport, L., Ricketts, D.: The TLA+ toolbox. In: F-IDE@FM. EPTCS, vol. 310, pp. 50–62 (2019). https://doi.org/10.4204/EPTCS.310.6
8. Lammich, P.: Verified efficient implementation of Gabow's strongly connected component algorithm. In: Klein, G., Gamboa, R. (eds.) Interactive Theorem Proving. ITP 2014, LNCS, vol. 8558. pp. 325–340. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08970-6_21
9. Lowe, G.: Concurrent depth-first search algorithms based on Tarjan's algorithm. Int. J. Softw. Tools Technol. Transf. **18**(2), 129–147 (2016). https://doi.org/10.1007/s10009-015-0382-1
10. McLendon-III, W., Hendrickson, B., Plimpton, S.J., Rauchwerger, L.: Finding strongly connected components in distributed graphs. J. Parallel Distrib. Comput. **65**(8), 901–910 (2005). https://doi.org/10.1016/j.jpdc.2005.03.007
11. Oortwijn, W., Huisman, M., Joosten, S.J.C., van de Pol, J.: Automated verification of parallel nested DFS. In: TACAS 2020. LNCS, vol. 12078, pp. 247–265. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-45190-5_14
12. Pol, J.C.: Automated verification of nested DFS. In: Núñez, M., Güdemann, M. (eds.) FMICS 2015. LNCS, vol. 9128, pp. 181–197. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-19458-5_12
13. Renault, E., Duret-Lutz, A., Kordon, F., Poitrenaud, D.: Variations on parallel explicit emptiness checks for generalized Büchi automata. Int. J. Softw. Tools Technol. Transf. **19**(6), 653–673 (2017). https://doi.org/10.1007/s10009-016-0422-5
14. Tarjan, R.E.: Depth-first search and linear graph algorithms. SIAM J. Comput. **1**(2), 146–160 (1972). https://doi.org/10.1137/0201010
15. Tarjan, R.E., van Leeuwen, J.: Worst-case analysis of set union algorithms. J. ACM **31**(2), 245–281 (1984). https://doi.org/10.1145/62.2160

16. Wimmer, S., Lammich, P.: Verified model checking of timed automata. In: Beyer, D., Huisman, M. (eds.) TACAS 2018. LNCS, vol. 10805, pp. 61–78. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-89960-2_4
17. Yu, Y., Manolios, P., Lamport, L.: Model checking TLA$^+$ specifications. In: Pierre, L., Kropf, T. (eds.) CHARME 1999. LNCS, vol. 1703, pp. 54–66. Springer, Heidelberg (1999). https://doi.org/10.1007/3-540-48153-2_6