Vrije Universiteit Amsterdam

Master Thesis

Parallel Algorithms for Detecting Strongly Connected Components

Author:
Vera Matei

Supervisor:
prof.dr. Wan Fokkink
Second Reader:
drs. Kees Verstoep

A thesis submitted in fulfilment of the requirements for the degree of Master of Science

in the

Faculty of Sciences
Department of Computer Science

October 31, 2016

Contents

Chapter 1

Introduction

Graph theory is the field of mathematics which studies graphs. A graph is a structure that models relations between objects. The objects are called *vertices* (or *nodes*), while the relations are represented by *edges*. For example, the users of a social network can be represented as vertices, while the friendship between two users is represented by an edge between the two. If the relationship can be asymmetric, then the edge will point into a direction and the graph will be called *directed*.

A fundamental concept in graph theory is a *strongly connected component* (SCC) [18]. An SCC in a directed graph G is a maximal subset of vertices U such that there is a path in G from every vertex in U to every other vertex in U. An example of a graph partitioned into SCCs can be found in Figure 1.1, where each marked region represents an SCC. For example, the vertices $1, 2, 3$ are in the same SCC, because they can all reach each other through a path of edges.

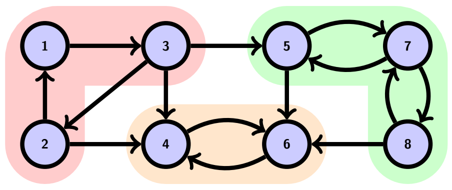


Figure 1.1: A graph partitioned into SCCs.

Algorithms for detecting SCCs can be applied to a wide variety of problems from different domains [43]. An example of this is 2-SAT, a special case of the boolean satisfiability problem in which one attempts to assign values to boolean variables in a formula such that it is satisfied [9]. Connected components also are an important concept in compiler construction, where strongly connected components within a call graph are identified to determine the order in which functions should be processed [49]. Yet another examples comes from the field of bio-informatics, for the problems in which multiple sequence alignment are performed in an effort to decipher the meaning of sequences of biological compounds [39].

Speed enhancements of single-core microprocessors are stagnating as the miniaturization of chips approaches its physical limits [48]. As a result, processors that run multiple cores in parallel have become increasingly relevant. These developments call for concurrent algorithms that can exploit this parallelism in an optimal way. Specifically, graph algorithms, which are used on increasingly large data sets, need to be adjusted to take advantage of parallelism. Given that the detection of SCC is a problem with a wide variety of applications, a lot of research has been dedicated to devising parallel algorithms for it. This thesis focuses on the following question: *What is the most efficient parallel algorithm for detecting strongly connected components?*

In order to answer this question, we start with an analysis of all the existing parallel algorithm for detecting SCCs. Many of the published papers report a small number of results and only compare theirs results to one or two other algorithms without updating the original implementation. Moreover, the authors of the papers could be biased when reporting the results, and thus some algorithms can be reported to perform better. As such, a fair comparison between these algorithms can only be made by reimplementing the most promising ones and testing them on a set of real-world graphs with various structures.

There was one algorithm proposed by Schudy [41] that did not report any data. Ebendal gave an implementation of this algorithm and proposed a couple of improvements such that showed very promising results, especially on sparse graphs [14]. Thus, our second research questions is: *Can we enhance Schudy's algorithm even further in order to make it perform better than the existing algorithms?*.

This thesis is structured as follows. Chapter 2 gives a high-level description of the existing algorithms and ends with a comparison between them. In Chapter 3, we discuss our implementations in detail and present some of the improvements we made for each algorithm. In Chapter 4, we show how well the selected algorithms perform against some instances of real-world graphs. Finally, in Chapter 5 we present our conclusions.

Chapter 2

Previous Work

Several algorithm have been developed that can detect SCCs in linear time [13] [15] [44]. In this chapter we will focus only on parallelizable algorithms. Our aim is to give a complete overview and an intuitive description of the different approaches developed for finding SCCs in parallel.

2.1 Divide-and-Conquer Algorithms

Most of the sequential algorithms rely on depth-first search. However, this technique has been proven to be difficult to parallelize [17] as it is $\mathcal{P}$-complete [38]. This is a strong motivation for a divide-and-conquer approach. In a divide-and-conquer approach, the graph is recursively divided in sub-graphs such that an SCC only spans one such sub-graph.

2.1.1 Forward-Backward Algorithm

One of the most popular parallel algorithms for finding SCCs is the so-called forward-backward (FB) algorithm, by Fleischer et al. [16]. This algorithm forms the basis of most other parallel algorithms for finding SCCs.

The algorithm uses the concept of forward and backward reachability. In a directed graph, a vertex u is *forward reachable* from a vertex v, if there is a path from v to u. Likewise, a vertex u is *backward reachable* from a vertex v, if there is a path from u to v.

The forward reachability set of a vertex v consists of all the vertices that are forward reachable from v. The backward reachability set is defined in the same fashion.

The following observations lie at the core of the algorithm:

Lemma 2.1.1. *In a graph $G = (V, E)$, let F and B be the forward and backward reachability sets respectively, starting from a vertex $v \in V$. The intersection of the two sets ($F \cap B$) is a unique SCC.*

Moreover, for all the other SCCs $S \subseteq G$ we have that either:

1. $S \subseteq F \setminus B$
2. $S \subseteq B \setminus F$

3. $S \subseteq V \setminus (F \cup B)$

The relation between these sets is visually represented in Figure 2.1.

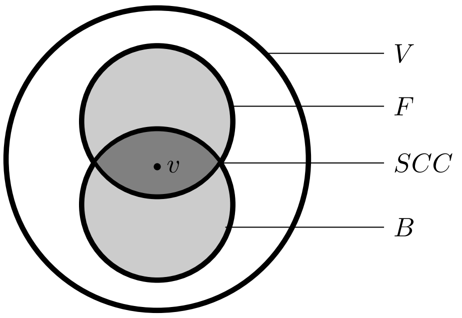


Figure 2.1: Visual representation of the vertex sets resulting from a forward and backward reachability search starting at vertex v.

The correctness of the first part follows immediately from the SCC definition. For any two vertices $w, w' \in S$, there is at least one forward and one backward path between the two, via vertex v, since both are forward and backward reachable from v. In addition, every vertex v' in the same SCC as v is in S, since there is a path from v to v' and vice versa.

According to Lemma 2.1.1, any SCC other than $F \cap B$ will be in disjoint sets, thus the search for the other SCCs can be performed in parallel on each of the three sets. Each of the three sets in turn creates three other subsets each time the algorithm is recursively applied. It is expected that the available processes-computing power is quickly distributed.

Trimming Step

The algorithm above was improved by McLendon et al., by adding one step to the design to trim away trivial SCCs [30]. Trivial SCCs are those that contain only one vertex. In the literature this step is also called OWCTY (One-Way-Catch-Them-Young) [2] [4] [10].

The justification of this step comes from the fact that the FB algorithm has a very high recursion depth on sparse graphs. For example, on graphs with no edges the recursion depth is $\Theta(|V|)$. In a sparse graph it is often the case that many of the resulting sets are empty, which will lead to a much slower distribution of the computation over the available processors than desired.

The optimisation is based on the observation that vertices which do not have any incoming or outgoing edges can only be in an SCC by themselves. If a vertex has no incoming edges, no other vertex can reach it, while if it has no outgoing edges it cannot reach any other vertex.

Thus in the trimming step, each vertex v that has no incoming or outgoing edges, is placed in an SCC of its own, and it is removed from the pool of vertices to be further inspected. If any vertex was removed, the procedure is repeated again, as removing one trivial SCC could create another.

An example of trimming is shown in Figure 2.2. In the first step of the trimming step, vertices c and d will be removed, since c has no incoming vertices

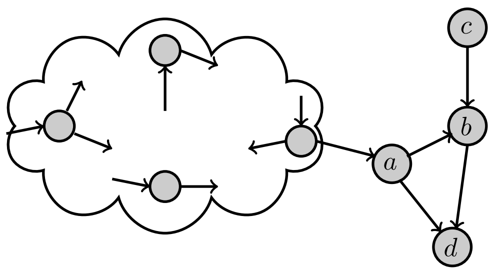


Figure 2.2: A trimming example.

and d has no outgoing ones. Then the procedure will be called again, where vertex b will be removed, and finally in the next call a will also get removed.

2.1.2 Schudy's Algorithm

Schudy addresses the lack of parallelism of the FB algorithm on sparse graphs as well [41]. He uses a slightly modified version of the FB algorithm as a starting point. The backward reachability search from pivot v is performed only on the already computed forward reachability set F. Thus, no fully backward reachable set is computed, so the second case of Lemma 2.1.1 is skipped. The visual representation of this can be found in Figure 2.3.

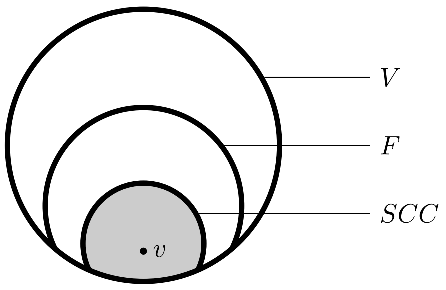


Figure 2.3: Visual representation of Schudy's adaptation of the FB algorithm.

The main idea of Schudy's improvement, is that there is not one single pivot selected, but a set of pivots. The pseudocode can be seen in Algorithm 1.

A random unique index from 1 to $|V|$ is assigned to all the vertices. Through a binary search we find the *smallest* index s, such that the pivots from 1 to s will together reach a subset of at least half of the vertices plus edges.

In Algorithm 1 the procedure `pivotSet` performs these computations, and the found pivots are stored in P. The `pivot` procedure retrieves the largest index s from P, after which s is removed from P.

Starting from s, the forward reachability set F is computed. Considering only the vertices in F, a backward reachability set C is computed. The set C is an SCC. These steps are exactly the same as in the adapted FB algorithm.

Next, a forward reachability set Q is computed from the set P of pivots. Since both F and Q are closed under forward reachability, they are used to

Algorithm 1 `schudy` (V)

```
1: if $V == \varnothing$ then return
2: $P =$ pivotSet$(V)$
3: $s =$ pivot$(P)$
4: $P = P \setminus \{s\}$
5: $F =$ forward$(\{s\}, V)$
6: $C =$ backward$(s\}, F)$
7: Output $C$
8: $Q =$ forward$(P, V)$
9: schudy $(V \setminus (F \cup Q))$
10: schudy $(F \setminus (C \cup Q))$
11: schudy $((F \cap Q) \setminus C)$
12: schudy $(Q \setminus F)$
```

further partition the vertices into four sets that have no overlapping SCCs:

1. $V \setminus (F \cup Q)$
2. $F \setminus (C \cup Q)$
3. $(F \cap Q) \setminus C$
4. $Q \setminus F$

A graphic representation of the resulting sets can be seen in Figure 2.4.

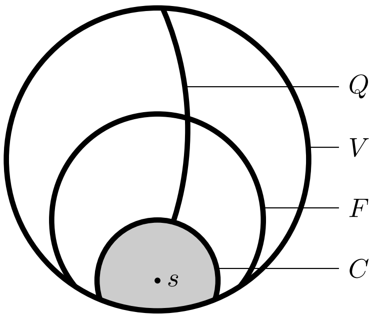


Figure 2.4: Visual representation of the vertices sets resulted from Schudy's division.

The main result of Schudy is that the FB multi-pivot optimisation improves, with high probability, the recursion depth from $\mathcal{O}(|V|)$ to $\mathcal{O}(\log |V|)$.

An important note is that from all the studies presented here, Schudy is the only one who only evaluated his work theoretically. He does not report on any experimental results.

2.1.3 Hong's Algorithm

Hong et al. present and explore a property of *real-world* graphs, namely the *small-world* property [22]. It has been shown that in graphs with the *small-world* property there exists a giant SCC of size $\mathcal{O}(|V|)$. Moreover, the distribution of the SCCs follow a power distribution: the number of SCCs grows exponentially as the size of the SCCs decreases, meaning there is a great number of smaller SCCs [19]. One such example can be seen in Figure 2.5.

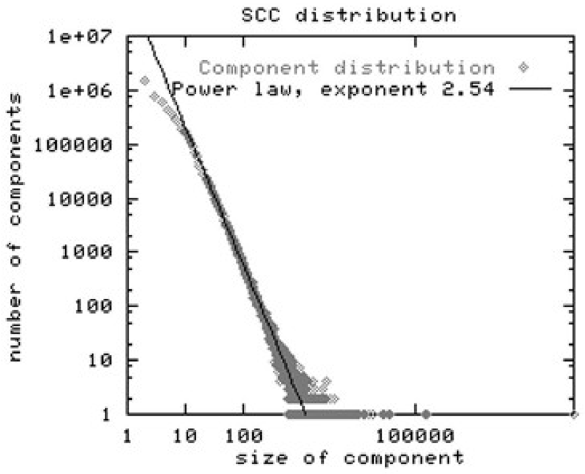


Figure 2.5: Distribution of SCCs on the Web. A vertex represents a web-page, and an edge a link between two web-pages [11].

The FB algorithm is using only task-parallelism, meaning that every thread is used to discover one SCC at a time. However, this can be a waste of resources as the discovery of the very large SCC takes $\mathcal{O}(|V|)$. It is also very likely that this large SCC is identified early on in the execution of the algorithm, since most other SCCs are connected to it. Thus, the other threads may run out of tasks and stay idle.

Hong's approach is to use all threads to first identify the big SCC, making use of data-parallelism. That is, use a forward and backward parallel reachability algorithm until a percentage of vertices are assigned to a SCC, then continue doing only task-parallelism with the regular FB algorithm. Their measurements indicate that after classifying more than 1% of the vertices, the huge SSC has also been identified.

After the huge SCC has been removed, there are a lot of smaller weakly connected components (WCCs) left. Two vertices are said to be in the same WCC if there would be a path between them if the edges would be undirected. As discussed previously, the FB algorithm then will not produce new tasks fast enough and so threads can stay idle. To solve this issue, Hong et al. apply a fast algorithm to identify each WCC and then FB can run independently on each WCC. Figure 2.6 depicts the decomposition of WCCs.

2.1.4 OWCTY-Backward-Forward (OBF)

The main idea behind the OBF algorithm is to divide the graph in slices, such that there is no SCC spanning two slices [5]. Because of this property, the FB algorithm can be applied on each slice in parallel.

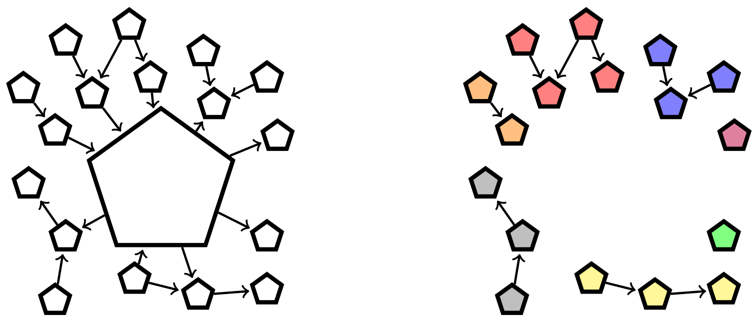


Figure 2.6: Every pentagon represents an SCC. The left image depicts the structure of SCCs on graphs with the small-world property. In the right image the very large SCC has been removed and every WCC is identified by a different color.

The algorithm assumes that the graph is rooted and the root is added to the *seeds* set, which will be the starting point. From there the following steps are repeated:

O Using OWCTY (see Section 2.1.1), eliminate the set of trivial components with no incoming edges. Also add all the successors of eliminated vertices to the set *reached*.

B Compute the backwards closure B of all the vertices in the set *reached*.

F Remove the slice B from the total pool of vertices, and let the FW-BW algorithm run on it. Recursively call the OBF procedure on the remaining vertices using the successors of the vertices in B as seeds.

The procedure terminates when all the vertices have been added to a slice. Figure 2.7 shows an example of two calls of the OBF algorithm on a small graph.

Recursive OBF

Instead of calling the FB algorithm on each slice, the recursive OBF algorithm will again call itself [3]. However, the OBF algorithm needs a rooted graph to work on. This is solved by dividing the slice in *rooted chunks*. A *rooted chunk* is created by picking a vertex at random from the slice and computing its forward closure within the slice. Then the OBF algorithm will be run in parallel on each *rooted chunk*.

Moreover there needs to be a termination criterion in case the whole chunk is one SCC. This is simply if the **B** step visits all the vertices from the chunk.

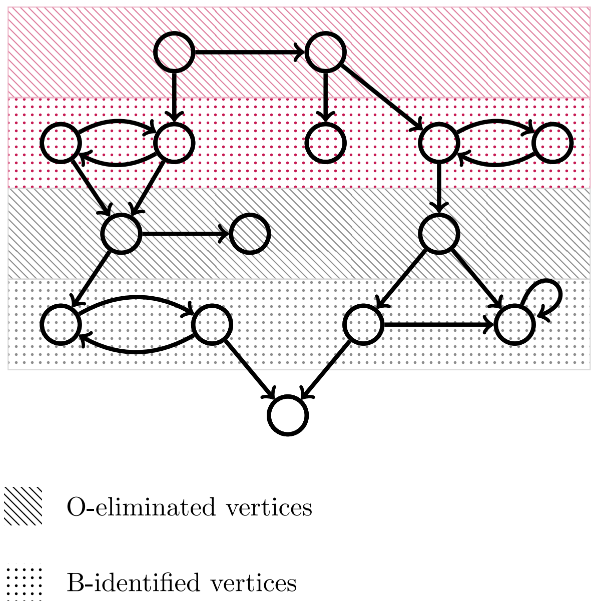


Figure 2.7: OBF slice division. The vertices on the red background are processed in the first call of the algorithm and the ones on the gray background in the second run.

2.1.5 Coloring/Heads Off (CH)

The CH algorithm uses the concept of graph coloring [37]. Every vertex has a unique number as identifier, which is called the color of the vertex.

The algorithm starts with a forward search which can start in parallel from a different number of vertices. When a vertex v has a higher color than a successor u, u will get the color of v. We say that v propagates its color *forward*. It is possible for a vertex to have its color changed several times. This phase of the algorithm stops when no vertex can change its color. At this point all vertices of an SCC have the same color, thus all edges between vertices of different colors can be removed.

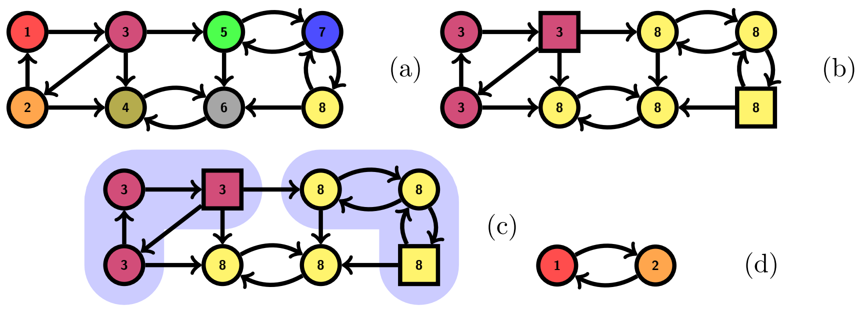


Figure 2.8: Example of a CH run. (a) Initial state. (b) Highest colors have been propagated. The square node represents the root. (c) All backward reachable vertices to the root form an SCC. (d) The algorithm will need to run again on the two vertices.

The vertices that kept their colors are considered the *root* of an SCC. Now from every vertex v a *backward* search is performed. If this search reaches the *root* of its color, then v is in the same SCC as the root. So we can say that all vertices that are *backward* reachable from the root and have the same color as the root form one SCC.

The discovered SCCs are removed and the algorithm continues by assigning new unique numbers to the remaining vertices. The algorithm terminates when all vertices have been assigned to an SCC.

2.1.6 Multistep

The Multistep algorithm is a combination of other algorithms [45]. The motivation of merging different algorithms comes from the observation that the performance of algorithms for detecting SCCs is heavily dependent of the graph's structure, i.e., the relative size and distribution of SCCs. When SCCs are detected and removed, the graph's structure changes, so it can make sense to also apply a different algorithm.

Multistep starts with the trimming step. Then, similar to Hong et al., it uses FB to find the huge SCC present in graphs with the small world property. Once the huge SCC has been removed, CH is applied until a partition reaches a threshold of nodes. To avoid parallelization overhead, remaining nodes are sorted using an efficient sequential algorithm, in this case Tarjan's algorithm [47].

Experiments have shown that for a graph with a few millions nodes, a good threshold for switching to the sequential algorithm is 100,000. However, the resulting performance is highly dependent on the graph's structure and also on the machine architecture.

2.2 Depth-First Search Based Algorithms

One important application for detecting SCCs is in the field of formal verification, namely the model checking problem [1, 12]. That is, the problem of exhaustively and automatically verifying certain properties on the state space of a system's model [6]. For this problem in particular, it is useful to be able to detect SCCs *on-the-fly*, that is, during the construction of the state space. This is because a counter-example for a property, might be found on the basis of an SCC without having to generate the whole state space [42]. The algorithms used for detecting SCCs on-the-fly use depth-first-search (DFS) and are based on Tarjan's algorithm.

2.2.1 Tarjan's Algorithm

Tarjan's algorithm is a sequential recursive algorithm. Even though it does not utilize parallelization, we describe it here as it will make the explanation of the following algorithms easier to follow. The pseudocode can be found in Algorithm 2.

Every vertex keeps track of two values *index* and *lowlink* that are initially set to zero. The *index* variable indicates in which order this vertices were visited, so once assigned it will never change. The *lowlink* is used to find edges to ancestors

Algorithm 2 TarjanInit($G(V, E)$, SCC, $color$)

```
1: $\forall v \in V : index[v], lowlink[v] \leftarrow 0$
2: $count \leftarrow 0$
3: $S \leftarrow \varnothing$
4: $\forall v \in V$ :DFSTarjan($v$)
5: Algorithm DFSTarjan($v$)
6:     $count \leftarrow count + 1$
7:     $lowlink, index \leftarrow count$
8:     for each $(v, u) \in E$
9:         if $index[u] = 0$ then
10:            DFSTarjan($u$)
11:            $lowlink[v] \leftarrow \mathtt{min}(lowlink[v], lowlink[u])$
12:        else if $u \in S$ then
13:            $lowlink[v] \leftarrow \mathtt{min}(lowlink[v], index[u])$
14:    if $lowlink[v] = index[v]$ then
15:        $T \leftarrow \varnothing$
16:        $u \leftarrow S.\mathtt{pop}()$
17:        $T \leftarrow T \cup u$
18:        while $index[u] \geq index[v]$
19:            $u \leftarrow S.\mathtt{pop}()$
20:            $T \leftarrow T \cup u$
21:        $SCC \leftarrow SSC \cup T$
```

and it can be changed during recursive DFS calls. Moreover the algorithm uses a *stack* to keep track of all the visited vertices.

The algorithm starts at an arbitrary vertex v. Its *index* and *lowlink* variables are set to the maximum index value of any other node plus one and the node gets added to the stack. A DFS is performed from v, and every visited node will have its fields changed in the same way and get pushed onto the *stack*.

At some point a vertex u is reached with no unvisited successors. If u has a successor w on the stack with a lower *lowlink* value, then it will copy w's *lowlink* value.

While tracking back from the DFS, if a vertex v's *lowlink* equals its *index* value, it means that none of the vertices v can reach can get to a vertex discovered before v, so all these vertices form an SCC. Thus all vertices up to v need to be popped from the stack.

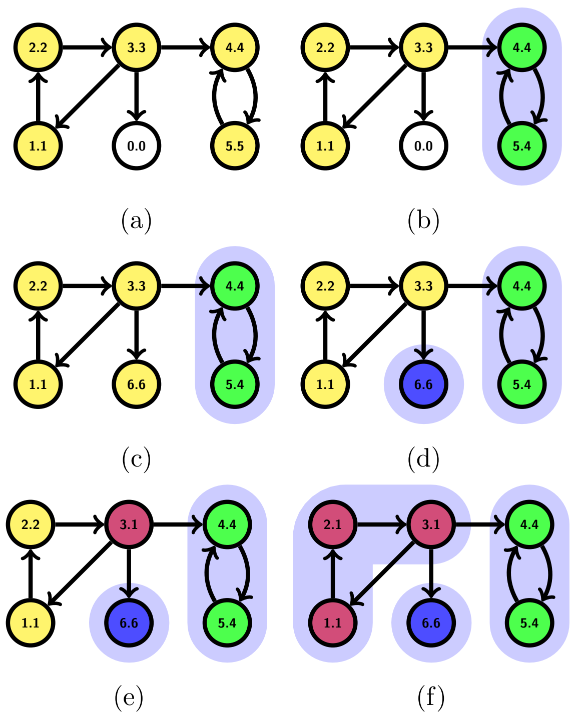


Figure 2.9: Example of a Tarjan's run, first number represents the *index* and second number the *lowlink*. (a) A DFS starting from the vertex in the left-low corner. (b) Backward search and SCC detection starting from index 5. (c) Continue DFS from index 3. (d) Backward search and SCC detection from index 6. (e) Backward search from index 3. (f) Backward search and SCC detection from index 2.

2.2.2 Lowe's Algorithm

Lowe's algorithm parallelizes Tarjan's algorithm by deploying multiple instances which run in parallel, starting from different vertices [29]. The algorithm explores vertices and marks SCCs in a similar fashion as Tarjan's algorithm, however it is modified in order to detect and resolve possible crashes between searches.

Every instance of the algorithm uses a local *stack*, but the fields of every vertex, *index* and *lowlink*, are shared data between the threads.

An important invariant of the algorithm is that a vertex can only belong to one stack. To make sure this is respected, every vertex has a *status* field that can be either `unseen`, `live` or `dead`. Initially the status of all vertices is `unseen`. A vertex can be visited only if its status is `unseen`, after which the status will change to `live`.

A search that encounters a vertex with the `live` status is suspended until the vertex changes its status to `dead`. A vertex changes its status to `dead` only when it is marked as belonging to an SCC. This can only be done by the thread that changed its status to `live`.

It is not hard to see that this procedure can lead to deadlocks. For example, a search from thread p changes the status of vertex v to `live` and next reaches vertex u which has the status `live` set by a search from thread q. Thread p suspends its search until thread q marks u `dead`. However, in its search q might encounter vertex v, which only p can change the status of, and so also thread q will suspend its search.

In order to recover from deadlocks, the algorithm keeps track of a *Suspended* map. This stores for each suspended search p, the vertex u at which the search has to stop and its predecessor, vertex v. When a path is found from u to v in the *Suspended* map, belonging to the suspended search q, the two searches are merged by having all the vertices belonging to the p stack pushed onto the q stack. The search is now continued by one thread.

The algorithm deploys as many instances as there are threads available. Whenever a thread is suspended, the algorithm deploys another instance in order to save resources.

2.2.3 Renault's Algorithm

Just like in Lowe's algorithm, in Renault's algorithm multiple instances of Tarjan's algorithms are deployed [40] at the same time. However, the essence of the algorithm is quite different. In Renault's algorithm threads only communicate the fully discovered SCCs. That is, there is only one shared union-find data structure [32], across all the threads containing the already detected vertices belonging to an SCC.

Unlike in Lowe's algorithm, in Renault's algorithm if two threads start the discovery of an SCC at the same time, they will both discover the SCC individually. Thus, when the graph contains only one SCC, each thread will work to discover the SCC by itself.

While Lowe's strategy may seem to be more efficient at the first glance, it does not always need to be so. Renault's algorithm has the advantage that it can visit a vertex without acquiring a lock. Moreover, it does not have the overhead of deadlocks.

2.3 Performance Comparison

When comparing performance of algorithms, one first looks at the time complexity of the algorithms. In this case it cannot really be a distinguishing factor, as all of the above parallel algorithm have the same running time of $\mathcal{O}(|V|^2)$ [7].

We will present a performance-based ordering of the above algorithms relying on the measurements resulting from the reported experiments. However, this ordering needs to be taken with a grain of salt, since it is done by comparing results reported in several different papers. This aggregation of results is needed, as every paper only reports on at most three algorithms. The upside is that all use either FB or Tarjan's as a comparison point, so we will consider speedups relative to these two. The experiments were run on real-world graphs.

It is important to keep in mind that the implementation and the architecture can have a dramatic impact on the obtained results.

Tarjan vs FB + OWCTY The only source in which these two are compared is Hong's paper [22]. The size distribution of SCCs matters a lot when comparing the two. If the size distribution is uniform, meaning most SCCs have roughly the same size, FB + OWCTY can have speedups of up to 7.2. However, when the size of the SCCs varies a lot, particularly when there are a few very big SCCs, FB becomes 10 times less efficient than Tarjan's algorithm. What is interesting to note in this last situation is that the number of threads seems to make little difference in efficiency. This suggests that FB is performing the discovery of SCCs sequentially, because it cannot generate enough tasks to take advantage of parallelism.

FB + OWCTY In all tested situations, the trimming step always seem to improve the performance of FB, giving it speedups of up to a factor 64 [30]. Hong et al. have parallelized the OWCTY step, and reported a linear improvement with the number of threads [22].

Hong In most instances, the performance of Hong's algorithm is superior to Tarjan and it can reach speedups of up to a factor 500 [22]. However, there was one instance in which Tarjan's outperformed Hong's algorithm by a factor 2. That seems to be the case only when the graph has a very large diameter and it has more than one huge SCC. What happens then is that, even after the removal of the very large SCC, the FB algorithm does not produce tasks fast enough, but also the algorithm for identifying WCC takes much longer.

Hong's algorithm in all cases performs better than FB + OWCTY, with speedups of up to a factor 30.

Multistep Compared to Hong's algorithm, Multistep always performs just as well or better, having in the best case a speedup of 3 [45]. Out of all the presented algorithms, this seems to be the most efficient one.

OBF Based on the results from [3], OBF is reported to have a speedup between 1.2 and 1.5 over FB, while recursive-OBF yields a speedup between 1.4 and 2. The experiments were run both on real-world graphs and also on synthetic graphs.

CH The author of the algorithm, Orzan, does not provide any direct comparison with other algorithms. However the authors of OBF run experiments using Orzan's implementation [3]. Compared with FB, CH performs worse, having speedups of up to 0.1. Given the important role that CH plays in the efficient Multistep algorithm, it stands to reason that there must be cases in which CH performs much better.

Lowe On experiments with small SCCs, Lowe's algorithm can have a speedup of up to 5 compared to Tarjan's algorithm. However, when running it on a graph with an SCC that contains 70% of the nodes, the performance was worse. This happens because the number of collisions between threads will increase dramatically, and the time taken to synchronize them outruns the performance gained from parallelism.

Renault On model-checking graphs, Renault's algorithm can have speedups of up to 32 [8]. However, there are also a few instances in which Renault's algorithm performs slightly worse than Tarjan's, but the difference seems to be very small. This situation arises in graphs with huge SCCs.

One thing to note is that the shape of the graph seems to influence the performance of the algorithm a great deal.

The published data shows that in order to obtain substantial improvements for graphs with large SCCs, the divide-and-conquer approach is most suited, with Multistep seemingly being the fastest algorithm developed. However, the structure of the graph also plays a big role in performance and can also make the basic FB algorithm perform well. In such cases an FB implementation could be preferred as it is more straightforward and less error prone.

Chapter 3

Implementation

In this chapter we will give a more detailed description of the implemented algorithms. Furthermore we will present some of the crucial design decisions and our arguments for selecting them.

3.1 Graph Representation

The most important data structure we had to make a decision about early on, was the one that will represent the actual graph. Since we are using Java, our first attempt used an object-oriented approach, meaning that we had a Vertex class containing all the information associated with a vertex. That is, a list containing its vertex neighbours, which SCC it belongs to and various other fields that a specific algorithm needs. For example, in the CH algorithm there is also a field called *color*, to store its current color.

An alternative design approach was to use arrays, as suggested in [22]. All arrays are of size $|V|$, and each index represents a vertex. The graph is represented using a so-called jagged array, which is a two-dimensional array, where the size in the second dimension can vary. The second dimension is used to store the neighbours of a vertex. Figure 3.1 shows an example of such a representation. The array representation was chosen eventually, as it was performing better in our experiments.

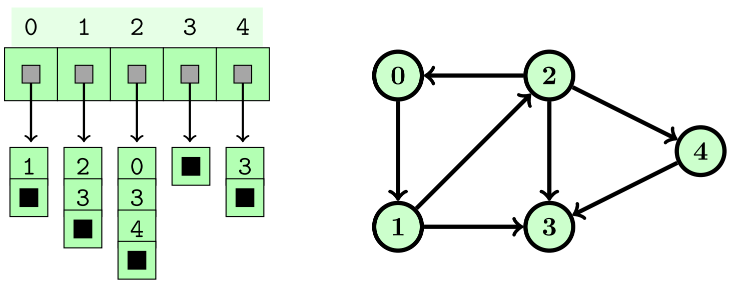


Figure 3.1: Example of a graph's jagged-array data structure representation.

This observation can be explained by considering either implementation's in-memory representation. The array-based implementation yields a limited number of small objects (one array for the first dimension and an additional array for each element in the second dimension). This leads to good cache locality and limits the work for the garbage collector. The object-oriented based approach, on the other hand, leads to more fragmented memory access patterns and more work for the garbage collector.

3.2 FB Algorithm

A straightforward implementation of the FB algorithm can be found in Algorithm 3. The algorithm takes as input a graph and an empty set that will eventually contain sets of vertices that form an SCC. The algorithm partitions the graph as depicted in Figure 2.1.

Algorithm 3 FB($G(V, E), SCC$)

```
1: if $V == \varnothing$ then return
2: $v$ = pivot($V$)
3: $F$ = forward($v, V$)
4: $B$ = backward($v, V$)
5: $SCC = F \cap B$
6: Output $SCC$
7: do in parallel:
8:     FB($F \setminus B$)
9:     FB($B \setminus F$)
10:    FB($V \setminus (F \cup B)$)
```

Hong el al. [22] present a more efficient implementation that does not use set operations. The same implementation was also used in the Multistep algorithm [45]. Our approach follows Hong's closely, but with a few tweaks.

To avoid the expensive operation of removing a vertex once it is classified, Hong's algorithm makes use of two auxiliary data structures: *mark* and *color*. *mark* is a boolean array that indicates whether a vertex has been assigned to an SCC. If the node n has been classified, *mark*(n) is set to `true` and the vertex will be ignored in further searches.

Similarly, *color* is an integer array that indicates to which partition a vertex belongs. All vertices that share the same color are in the same partition. When performing a search from a certain vertex v, we will only consider the vertices that have the same color as v, although there can be an edge between v and a vertex of a different color. However, there is one problem with this approach. Selecting the pivot can become expensive when a certain color does not occur too often. This is solved by maintaining for each partition also a set representation of its elements. This hybrid representation is more expensive in terms of memory usage, but it results in a better overall performance [22].

One of our observations is that it is not necessary to use the *mark* array at all, as its function can be also encoded in the *color* array, by marking every vertex with -1 once it is assigned to an SCC. This leads to a slight improvement in performance. We believe that is because *color* can stay longer in the cache when needed.

Algorithm 4 FBHong($G(V,E), SCC$)

```
    Local: $color$, atomic integer array of size $|V|$
1: $\forall n \in G : color(n) \leftarrow 0$
2: ParTrim($G(V,E), SCC$, $color$)
3: FB($G(V,E), SCC$, $color$, 0)
```

Algorithm 4 outlines Hong's version of the FB algorithm. After initialization, the algorithm performs the trimming step, and then it calls the recursive FB procedure.

Algorithm 5 ParTrim($G(V,E), SCC, color$)

```
    Local: $repeat$, boolean flag
1: do
2:     $repeat \leftarrow$ false
3:     for each $v \in V$, $color(v) \neq -1$ do in parallel
4:         if inDegree($v$, $color$) $= 0 \vee$ outDegree($v$, $color$) $= 0$ then
5:             $color(n) \leftarrow -1$
6:             $SCC \leftarrow SCC \cup \{\{v\}\}$
7:             $repeat \leftarrow$ true
8: while $repeat$
```

As explained in Section 2.1.1, the trimming step is applied recursively until no vertices can be trimmed anymore. A vertex is trimmed when it has no outgoing or incoming vertices. The procedure is straightforward and can be found in Algorithm 5.

Algorithm 6 FB($G(V,E), SCC, color, c_{cur}$)

```
1: $v \leftarrow$ pivot($V, c_{cur}$)
2: if $v = NIL$ then return
3: $c_{fw}, c_{bw}, c_{scc} \leftarrow$ get unique color
4: forwardSearch($G(V,E), SCC, color, c_{cur}, c_{fw}$)
5: backwardSearch($G(V,E), SCC, color, c_{cur}, c_{fw}, c_{bw}, c_{scc}$)
6: for $c \in \{c_{cur}, c_{fw}, c_{bw}\}$ do in parallel:
7:     FB($G(V,E), SCC$, $color$, $c$)
```

The recursive FB algorithm can be found in Algorithm 6. The procedure starts by selecting a vertex v for which the given color is c_{cur}. If no such vertex exists, then the algorithm terminates. Next, three unique colors need to be selected for the partitioning of the graph in the three SCC-disjoint sets. By running the procedure forwardSearch, detailed in Algorithm 7, all forward reachable vertices from v will get the color c_{fw}.

The backwardSearch procedure (Algorithm 8) will then perform a backward search to further partition the graph and identify the SCC. If a reached vertex has color c_{fw}, then we know it is both forward and backward reachable from v, so it will be assigned to the current SSC and marked accordingly. Else, if the color is the initial c_{cur}, then the vertex is marked with the c_{bw} color.

Finally, on each of the discovered partitions, the FB algorithm can be run again on individual threads.

Algorithm 7 FORWARDSEARCH$(G(V, E), SCC, color, c_{cur}, c_{fw})$

```
1: traverse $G$ from $v$ using outgoing edges
2:     for each visited vertex $u$ do
3:         if $color(u) = c_{cur}$ do
4:             $color(u) \leftarrow c_{fw}$
5:         else prune traversal beyond $u$
```

Algorithm 8 BACKWARDSEARCH$(G(V, E), SCC, color, c_{cur}, c_{fw}, c_{bw}, c_{scc})$

```
   Local: $S$, a set that will contain all the vertices of the current SCC
1: traverse $G$ from $v$ using incoming edges
2:     for each visited vertex $u$ do
3:         if $color(u) = c_{cur}$ do
4:             $color(u) \leftarrow c_{bw}$
5:         if $color(u) = c_{fw}$ do
6:             $color(u) \leftarrow c_{scc}$
7:             $S \leftarrow S \cup \{u\}$
8:         else prune traversal beyond $u$
9: $SCC \leftarrow SCC \cup \{S\}$
```

3.3 Hong's Algorithm

The improvement of Hong et al. over the original FB algorithm is twofold. First, it uses data-parallelism to discover the big SCC present in real-world graphs. This removal usually leads to many WCCs. The second improvement is to use an algorithm to identify these components and have the FB algorithm run on them in parallel.

What is meant by using the data-parallelism to discover an SCC, is that both the forward and backward search in the FB procedure are done using a parallel implementation of BFS. The way we implemented this procedure was to maintain a list of vertices that need to be explored. In the parallel implementation, every time the list grows over a certain threshold size, the list will be split in two and the current thread processes one half, and a new thread the other. Finding the right threshold value requires some experimentation. For our experiments we found that 128 worked best, but if it would be run on much larger graphs, we propose that the value should be changed to a larger value.

Hong et al. [22] recommend to use an extra trimming step between removing the large SCC and partitioning the graph in WCCs. This procedure is designed to trim away SCCs of size two. Figure 3.2 depicts a common patter for SCCs of size two. In the paper it is reported that identifying these SCCs in an extra step can improve performance by up to 50%. The reason why it can have such a strong impact is that it can reduce the number of iterations needed for the WCC algorithm. For example, in Figure 3.2 right, if v has a higher color than u it will give it to u, and in turn u will have to propagate it further.

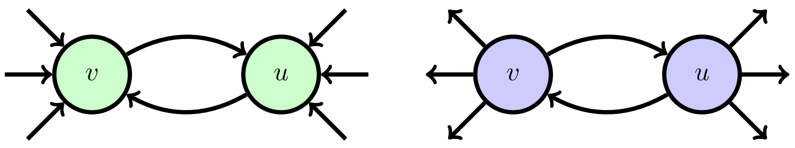


Figure 3.2: Type of SCC of size 2, which will be trimmed by the second trimming procedure.

The implementation of the trimming procedure is straightforward. For each vertex v with one incoming edge from u it checks if their color match and whether u has only one outgoing edge going to v. If so, they are marked to be in the same SCC.

Algorithm 9 WCC($G(V, E), SCC$)

```
    Local: $wcc$, atomic integer array of size $|V|$
 1: for each$v \in V$ s.t. $color(v) \neq -1$ do in parallel
 2:     $wcc(v) \leftarrow v$
 3: do
 4:     for each $v \in V$ s.t. $color(v) \neq -1$ do in parallel
 5:         foreach $u$ s.t. $(v, u) \in E \wedge color(u) = color(v)$ do
 6:             if $wcc(u) < wcc(v)$ then $wcc(v) \leftarrow wcc(u)$
 7:     for each $v \in V$ s.t. $color(v) \neq -1$ do in parallel
 8:         $c_v \leftarrow wcc(v)$
 9:         if $c_v \neq v \wedge c_v \neq wcc(c_v)$ then $wcc(v) \leftarrow wcc(w_c)$
10: while $wcc$ not changed
11: for each $v \in V$ s.t. $wcc(v) = v$ do in parallel
12:     $c \leftarrow$ new unique color
13:     for each $u \in V$ s.t. $wcc(u) = v$ do
14:         $color(u) \leftarrow c$
15:     run FB($G(V, E), SCC$, $color$, $c$) on a new thread
```

Algorithm 9 presents the procedure for finding WCCs. The procedure uses an extra integer array called *wcc*, in which initially each vertex is assigned its own index value. At every iteration, the vertices try to propagate their *wcc* value to their neighbours, and in turn the neighbours will accept it iff it is smaller than their current *wcc* value. This is a similar principle as in the Coloring procedure of the CH algorithm. To speed up the propagation at the end of an iteration, if a vertex v has value k and this value is different from its identification (v) and the *wcc* value of k has changed to m, then $wcc(v)$ will also change to m. Note that this change would have otherwise happened eventually, but it may require more iterations. Our experiments showed that this step is only useful when the graph is quite dense. Else it makes no difference or it can even slightly hinder performance. That is because in sparse graphs the propagation procedure needs to run a few times, and there are usually no cycles. Introducing the last step will therefore decrease the performance, since the vertices will need to be revisited without obtaining any gain.

At the last **for each** loop, every WCC has the same *wcc* value. Every WCC is assigned a fresh unique color, and then the FB procedure can be run in parallel. This FB procedure is using a non-parallel implementation of DFS for the graph exploration. However, our experiments showed that using the parallel implementation of BFS is more efficient, giving speedups of up to 50%. For this reason, in the experiments described in Section 4 we used this implementation.

3.4 Schudy's Algorithm

The straightforward implementation is presented in Algorithm 1. During his thesis research at VU University, Reggie Ebendal made two relevant improvements and showed they lead to a better performance [14].

Firstly, he considers half of the graph to be half of the vertices, instead of half of the 'vertices + edges'. The advantage of this is that it is much faster to compute and he argues that overall it offers just as good of a split.

The second improvement is that he first detects an SCC based on a pivot, and then separately splits the forward and remaining vertices in half. A picture of this idea is found in Figure 3.3. This differs from Schudy's approach in that Schudy would first partition the whole set of vertices in half, and only then use one vertex to compute an SCC. Performing computations on the four partitions requires one or two set operations.

Ebendal's approach has two advantages over Schudy's proposition. It leads to a more balanced partition of the work, because both the forward set and the remainder are split in half, while Schudy's method does not really guarantee this at every split. The even split is expected to happen because the vertices are chosen at random. However, the most important improvement is that there are fewer required set operations in order to partition the graph (seven vs. four).

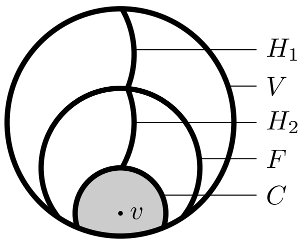


Figure 3.3: Visual representation of the vertices sets resulted from the Reggie's implementation of Schudy's algorithm.

Our implementation of the algorithm combines these ideas with Hong's approach. That is, we start with the parallel trimming step and use the same data structures and search method as Hong. When a partition has under a certain number of vertices, we will use Tarjan's algorithm, as splitting the load work further between threads creates too much overhead with little benefits.

The pseudocode can be found in Algorithm 10. The notation (V, c), denotes all the vertices that have color c. T is the value assigned to the threshold for running Tarjan's algorithm.

The forward and backward search is done just as in Hong's algorithm (Algorithms 7 and 8). The method HALFSET makes calls to FORWARDSEARCH until half the set is colored with c_2. More precisely, while $|(V, c)| < |(V, c_2)|$, a vertex

u is picked from (V, c) for which the method forwardSearch(G, u, c, c_2) is called.

Algorithm 10 Schudy$(G(V, E), SCC,$ *color*, $c_{cur})$

```
1: if $|(V, c_{cur})| < T$ then
2:     Tarjan$(G(V, E), SCC, c_{cur})$
3: if $|(V, c_{cur})| = 0$ then
4:     return
5: $v \leftarrow$ pivot$(V, c_{cur})$
6: $c_{fw}, c_{bw}, c_{scc} \leftarrow$ get unique color
7: forwardSearch$(G, v, c_{cur}, c_{fw})$
8: backwardSearch$(G, v, c_{cur}, c_{fw}, c_{bw}, c_{scc})$
9: for each $c \in \{c_{cur}, c_{fw}, c_{bw}\}$ do
10:     $c_2 \leftarrow$ new unique color
11:     halfSet$(V, c, c_2)$        ▷ half of the $c$ colored vertices will have color $c_2$
12:     do in parallel:
13:         Schudy$(G(V, E), SCC,$ color, $c)$
14:         Schudy$(G(V, E), SCC,$ color, $c_2)$
```

3.4.1 Choosing the Pivot

The correct choice of a pivot has a major impact on performance, particularly in the initial iteration of the algorithm. The pivot v should be selected such that it belongs to the very large SCC. If v is instead chosen outside of the very large SCC but such that a vertex of the very large SCC is forward reachable from v, then we can be in the situation that the edges are explored a second time when performing the halfSet procedure.

For this we use the same approach as the authors of Multistep. First, choose v such that the product of its in and out degree is maximized. This is just a heuristic, and it does not guarantee that v is part of the big SCC. However, this has always been the case for the tests we ran.

3.5 CH Algorithm

As described in Section 2.1.5, the CH algorithm assigns unique colors to all the vertices. The highest colors are propagated through outgoing edges until no vertex can change colors. All vertices that can reach the initial vertex of their color, will be assigned to an SCC.

The algorithm is using two extra arrays of size $|V|$: one integer array *color*, which maintains the color of every vertex, and one boolean array *visited*, which is only used in the coloring procedure. As in the previous algorithms, we do not modify the graph data structure. When a vertex v is assigned to an SCC, its color becomes $|V|$. Note that the colors range from 0 to $|V| - 1$.

An outline of the algorithm can be found in Algorithm 11. All vertices have as color the value of their index. That is, the ith vertex has color i. The *root* of a color c is the initial vertex that had the color c, thus in our case is the vertex at index c. Initially all vertices are assigned to a list L.

The COLORING procedure terminates when every vertex in L has the highest color that can arrive to it via incoming edges.

Every vertex that is backward reachable from the root and has the same color is placed in the same SCC, and their value in the *color* array will be set to $|V|$. Moreover, these vertices will also be removed from L, and a new iteration will start until L is empty.

Algorithm 11 CH($G(V, E), L, SCC$)

```
 1: for each $v \in V$ do in parallel
 2:     $L \leftarrow L \cup \{v\}$
 3:     $color(v) \leftarrow v$
 4:     $visited(v) \leftarrow$ false
 5: while $L \neq \varnothing$ do
 6:     COLORING($G, color, visited,$ COPYOF($L$))
 7:     for each $v \in L$ s.t. $v = color(v)$ do in parallel
 8:         $F_v \leftarrow \{u \in L : color(u) = v\}$
 9:         $SCC_v \leftarrow$ BACKWARD($G, L, color, v, u$)
10:         $SCC \leftarrow SCC \cup SCC_v$
11:         $L \leftarrow L \setminus SCC_v$
12:     for each $v \in L$ do
13:         $color(v) \leftarrow v$
```

The color propagation procedure is detailed in Algorithm 12. Its input consists of a graph, a list of vertices $L' \subseteq V$ that have not yet been assigned to an SCC, and two arrays of size $|V|$. The list L' is split equally across the available threads and the processing continues in parallel.

Every vertex $v \in L'$ is passing its color further to each outgoing neighbour u with a smaller color. If u has not been seen before, also not in another thread, it is added to a local list Q of nodes that will be processed in the next iteration. Once every thread has finished with propagating the colors, the threads merge their Q lists back into L'. The procedure terminates once no vertex changes colors anymore, and thus L' will be empty.

Algorithm 12 COLORING($G(V, E), L', color, visited$)

```
 1: while $L' \neq \varnothing$ do
 2:     for each $L_t \subseteq L'$ do in parallel
 3:         $Q = \varnothing$
 4:         for each $v \in L_t$ do
 5:             for each $u$ s.t. $(v, u) \in E$ do
 6:                 if $color(v) > color(u)$ then
 7:                     $color(u) \leftarrow color(v)$
 8:                     if $visited(u) =$ false then
 9:                         $visited(u) \leftarrow$ true
10:                         $Q = Q \cup \{u\}$
11:         for each $v \in Q$ do
12:             $visited(v) \leftarrow$ false
13:     Synchronization: $L' \leftarrow \cup_{\forall} Q$
```

Intuitively, it is easy to convince one of the correctness. There are two invariants that need to be true at the end of the procedure.

(1) $\forall v \neg\exists u$ *s.t.* $\{v,u\} \in L, \mathit{color}(v) < \mathit{color}(u)$, *and* v *is reachable from* u *using vertices from* L.

Take any path from an arbitrary u to v, via vertices $u = w_0, w_1, ...w_n = v$. We prove by induction that $\forall k \leq n \mathit{color}(w_k) \leq \mathit{color}(w_n)$.

Proof. We have the following lemma:

$$\forall i : \mathit{color}(w_i) \leq \mathit{color}(w_{i+1}). \tag{3.1}$$

The lemma holds, because $(w_i, w_{i+1}) \in E$ and thus the color of w_{i+1} would have been updated if otherwise (lines 5–7 of Algorithm 12).

Our **Induction Hypothesis** is: $\forall k < n, \mathit{color}(w_k) \leq \mathit{color}(w_n)$.

Base Case ($n = 1$): $\mathit{color}(w_0) \leq \mathit{color}(w_1)$ by the above lemma.

Induction Step ($n \to n + 1$):

We need to show that $\forall k \leq n + 1$, $\mathit{color}(w_k) \leq \mathit{color}(w_{n+1})$.

If $k \in [0, n)$ then this holds by the **IH**. Else, if $k = n$, then $\mathit{color}(w_n) \leq \mathit{color}(w_{n+1})$ by Lemma 3.1.

□

Since the vertices were chosen arbitrarily, we showed that for any vertex u that has a path to a vertex v, $\mathit{color}(u) \leq \mathit{color}(v)$.

(2) *Only vertices from* L *can be recolored and propagate colors.*

This invariant is necessary, because otherwise we might bring back a vertex that belongs to an SCC. If such a thing would be possible, the program could produce incorrect results.

Once a vertex s is assigned to an SCC, it is removed from L (line 12 of Algorithm 11). The only other way to reintroduce s back in L is by changing its color. However, in order for that to happen there must be a vertex v such that $\mathit{color}(v) > \mathit{color}(s)$. Such a vertex does not exist, since s has the maximum possible color. Hence, once a vertex is assigned to an SCC, it can never be recolored.

3.6 Multistep Algorithm

As stated in Section 2.1.6, the Multistep algorithm is actually a combination of previous algorithms, hence most of the methods used have been described previously. Just as in previous algorithms, we have a *color* array that is used to indicate to which partition a vertex belongs. When a vertex v is assigned to an SCC, $\mathit{color}(v) = |V|$.

The pseudocode of the algorithm is found in Algorithm 13. The algorithm starts with the trimming step (Algorithm 5). Then the parallel version of *FB* is used in order to remove the large SCC present in real-world graphs. To this end, the FB algorithm runs only once, where the pivot is chosen as described in Section 3.4.1. The original implementation runs CH directly afterwards, and when the number of vertices left are below a certain threshold, Tarjan is ran once. This approach is a bit wasteful, since the FB procedure partitions the graph in three disjoint subgraphs with respect to SCCs. In our adaptation of the algorithm we take advantage of this and spawn three threads after the large SCC is discovered. The CH procedure is similar to the one described in Algorithm 12, with the only difference that the **while** loop in line 4 has been removed.

Algorithm 13 MULTISTEP($G(V,E), SCC$, *color*, c_{cur})

```
1: ParTrim($G(V,E), SCC$, $color$)
2: $v \leftarrow$ pivot($V, c_{cur}$)
3: if $v = NIL$ then
4:     return
5: $c_{fw}, c_{bw}, c_{scc} \leftarrow$ get unique color
6: forwardSearch($G, v, c_{cur}, c_{fw}$)
7: backwardSearch($G, v, c_{cur}, c_{fw}, c_{bw}, c_{scc}$)
8: for each $c \in \{c_{cur}, c_{fw}, c_{bw}\}$ do in parallel:
9:     while $(V,c) > T$ do
10:        $\forall v \in (V,c) : visited(v) \leftarrow$ false
11:        $CH(G,(V,c),SCC,color,visited)$
12:    Tarjan($G,(V,c),SCC,color$)
```

3.7 Tarjan's Algorithm

As mentioned above, sometimes spawning threads can do more harm than good. In the situations where we have many small SCCs — as is often the case in real-world graphs (outside the big SSC) [19] — the overhead of division surpasses the benefits. Since Tarjan's algorithm is known to be the most efficient sequential algorithm for detecting SCCs, it makes sense to be used when there are only a few nodes left to assign to a partition.

The algorithm can be implemented iteratively or recursively, and it was shown that there is little difference between versions in terms of performance [31]. We implemented the recursive version as it appears to be more intuitive, although it has the disadvantage that it may need a larger stack allocation. The algorithm has been explained in detail in Section 2.2.1. The only change presented in our implementation is that now we also pass over the *color* array, in order to determine which vertices are still unexplored.

We performed a few experiments for both the Multistep and Schudy's algorithm in order to determine what is a good threshold value for starting the algorithm. The results depend strongly on the graph's structure, but even more so on the architecture. When running on a machine with fewer threads available, a higher threshold is more beneficial. Overall, we had the same result as the authors of Multistep: 100 000 vertices seems to work best for most cases.

3.8 Language Specific Considerations

All algorithms were implemented using version 8 of the Java programming language. This language was chosen because it strikes a convenient balance between expressiveness and performance. Java programs are compiled into bytecode and executed on the Java Virtual Machine (JVM).

The JVM optimizes a program's bytecode through just-in-time (JIT) compilation. This JIT compilation supports four so-called tiers, corresponding to progressive degrees of optimization. As each optimization comes at a cost, the JVM employs various advanced heuristics in deciding whether a given method is executed frequently enough to warrant further optimization. As a result "hot" methods may run just as fast (or even faster) on the JVM than they would if written in a lower-level language such as C.

3.8.1 Data Structures for Primitive Values

The Java collections framework (JCF), part of the standard Java libraries, provides a plethora of useful collection data structures, such as several list and set implementations. Its implementations are *generic*, meaning that they are parametric on the type of elements stored in the collection. This allows one to restrict the domain of elements stored in a collection to a particular subtype of `java.lang.Object`. This makes the JCF the preferred solution for most use cases.

The requirement that stored elements are subtypes of `java.lang.Object` means that JCF data structures cannot store primitive values (such as `int` and `double`). Storing such values requires that they are *boxed* when inserted and *unboxed* when retrieved. (Un)boxing is the process of converting between a primitive value and its object equivalent. In Java, this conversion does not come for free: the conversion costs time and boxed values require more memory to be stored. Moreover, when boxed values are no longer needed they must be garbage collected. When large numbers of boxed values are involved, this causes many small objects to be created. This puts a nontrivial amount of pressure on the garbage collector.

A solution to this issue is the use of custom collections in which values of a primitive type can be stored. There exist several third-party libraries that offer such collection implementations. For the algorithms presented here the `fastutil` library [28] developed at the Computer Science department of the University of Milan was selected. `fastutil` provides efficient implementations of the interfaces defined by the JCF for each primitive type. These implementations define a number of additional methods so that primitive values can be inserted and extracted without the overhead of (un)boxing.

Using the `fastutil` gave an overall improvement of more than 30% across all the parallel algorithms. The improvement for the Tarjan's algorithm was not as dramatic, since it only makes use of one data structure, namely a stack, which could have been replaced by the `fastutil` library.

3.8.2 Concurrency

The Java standard libraries provide a wide range of tools for concurrency control. Some of these are low-level primitives with direct equivalents in modern processor architectures. Others provide much higher-level functionality by which concurrent computations and data sharing across threads become a breeze.

For the algorithms implemented in the context of this thesis the following constructs found in the Java Development Kit (JDK) 8 were primarily relied upon:

- An `ExecutorService` with a fixed size thread pool (one thread for each processor, as reported by the underlying platform).
- A `Phaser`. This type of object acts as a flexible synchronization barrier. It can be used to synchronize the completion of a batch of work distributed across multiple threads, without a priori fixing the number of pieces in which the batch is split.
- Various `Atomic*` types, such as `AtomicIntegerArray` and `AtomicBoolean`. These allow concurrent read and write access to the data they wrap.

The `ExecutorService` and `Phaser` are used to define a custom type of concurrency control named `RecursiveExecutor`. It provides three operations:

- `perform(Consumer<RecursiveExecutor>)`. Creates a derived `RecursiveExecutor` and provides it to the given `Consumer`, then blocks until the `Consumer` returns. This operation is meant to be used in combination with the operation below.
- `performAsync(Runnable)`. Submits the given `Runnable` (effectively a piece of executable code) for concurrent asynchronous execution, and then immediately returns. If invoked on a `RecursiveExecutor` provided by the aforementioned `perform` operation, then said operation will not complete until all asynchronous operations spawned in its context have completed.
- `perform(BiConsumer<Integer, Integer>)`. Given n threads it invokes the provided `BiConsumer` n times, with arguments $(1, n), (2, n), \ldots, (n, n)$, respectively. These executions run concurrently, and the method does not return until all tasks it spawned completed. Observe that this operation can be (and is) defined in terms of the first two.

Chapter 4

Evaluation

In this chapter, we present the evaluation of the algorithms on several data sets. First, we describe the experimental set-up together with an overview of the graph instances. Next, the timings of the algorithms are presented. Finally, we discuss the results.

4.1 Experimental Set-up

The experiments were run on the DAS-5 cluster of VU University Amsterdam [46]. Its nodes are equipped with dual 8-core Intel Xeon E5-2630 CPUs running at a clockspeed of 2.4 GHz and 64 GB of internal memory. The machine runs the CentOS Linux (release 7.2) operating system. The project was built using jdk version `1.8.0_91`, and invocated with the `-Xmx2G` flag in order to increase the Java stack size.

The data sets have been selected from two public repositories [25, 35]. We chose some of the graph instances that have also been used in the original papers of Hong's and the Multistep algorithm [22, 45], in order to confirm that our own implementation of these algorithms is at least similar in performance to theirs.

The measurement of the tests was done after the graph was loaded into memory. Each data point is an average of 10 hot runs, of which the minimum and maximum times have been discarded.

Database	#Nodes $\times 10^6$	#Edges $\times 10^6$	Size Largest SCC $\times 10^6$	Size 2nd Largest SCC	#Trivial SCCs $\times 10^6$	Diameter
Baidu	2.1	17.8	0.6	510	1.1	5
Flickr	2.3	33.1	1.6	410	0.8	7
Livej	4.8	68.9	3.8	140	1	18
Twitter	41.6	1,468	33.5	280	7.8	23
Wikipedia	15.2	131.1	4.7	330	10	6
Patents	3.8	16.5	10^{-6}	1	3.8	22
Orkut	3.1	11.7	2.9	37	0.2	8
CA-Roads	2.0	5.5	1.2	350k	0.3	842

Table 4.1: The data sets used for experiments.

An overview of the graph instances is given in Table 4.1. The data sets were selected such that they have diverse structures. These includes graphs with varying diameter, some with a small diameter (Baidu, Flickr) some with a medium size diameter (Livej, Twitter) and one with a very big diameter (CA-Roads). Also, we included one example of acyclic graph. Moreover, the graphs also have very different density. For example, Flickr is almost twice as dense compared to Orkut. The first six graphs are real-world graphs. The Baidu [36] and Wikipedia [50] data sets consist of hyperlinks (edges) between pages (vertices) of a Chinese and English online encyclopedia respectively. Flickr [33], LiveJournal [27], and Twitter [24] are online social networks. These data sets consist of connections (edges) between users (vertices). The Patent graph data set [20] is based on the citation database of patents registered with the United States Patent and Trademark Office. Each vertex represents a patent; each edge is a citation by a patent of another patent.

The last two data sets are of undirected graphs. The graphs are transformed into directed graphs by randomly assigning a direction to each edge. The same transformation was also used in [22]. The Orkut data set also originates from an online social network, where the vertices represents the users and edges a friendship between two users [34]. CA-Roads is a road network in California. Vertices represent intersections and endpoints; edges are the roads between them [26].

4.2 Results

The plots in Figure 4.1 show the execution time of each algorithm for each data set. The time is shown on the y-axis, while the x-axis displays the number of threads grouped by algorithm. For a more comprehensive view of the performance, Figure 4.2 shows for each graph instance the speed up against Tarjan's sequential algorithm. The number of threads are on the x-axis and the speedup on the y-axis.

4.2.1 Discussion

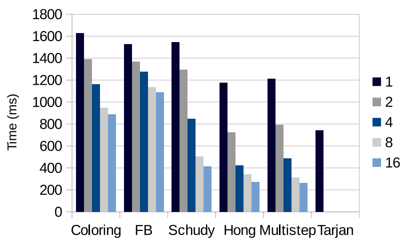
One of the main goals of this thesis is to see how well an enhanced implementation of Schudy's algorithm performs. With the exception of the CA-Roads graph, the algorithm always performs better than CH, FB, and the sequential Tarjan algorithm. However, it is still outperformed by the Hong and Multistep algorithms. To understand why this is the case, we take a closer look at the performance of all algorithms on the evaluation data sets. We also cover the special structure of the CA-Roads and Patents data sets.

Overall, it seems that the FB algorithm does not scale as well as the other algorithms, even when there are more threads assigned to it. Profiling shows that most threads stay idle. That is because most of the time is dedicated to finding the big SCC, which is done by a single thread.

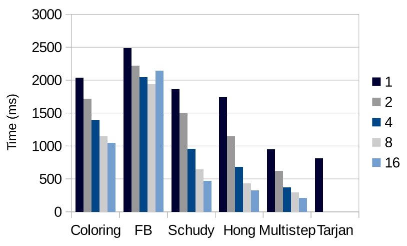
The CH algorithm performance seems to vary greatly across the different type of graphs. In some instances, it performs worse even in comparison to Tarjan's algorithm. Analyzing the time with the profiler shows that most time is spent in the colors propagation procedure (Algorithm 12). This is particularly expensive when the graph has a high diameter. This is because vertices can only propagate their colors to their immediate successors in a single iteration. Thus, we can see that in the graph instances with a high diameter (LiveJournal, Twitter, Pattern), the algorithm performs worse than FB and Tarjan. However, in the graphs with relatively small diameter (Baidu, Flickr, Wikipedia, Orkut), CH performs similar to or slightly better than FB and Tarjan.

The Multistep and Hong's algorithms also use a procedure for propagating colors. However, in both cases, this is done after the big SCC has been discovered. As we can see in the Table 4.1, the second largest SCC left in the graph has a very small amount of vertices (only a couple of hundred). Of course, this does not automatically imply that the diameter of the graph after the SCC removal is smaller, but there is a high chance of this being so.

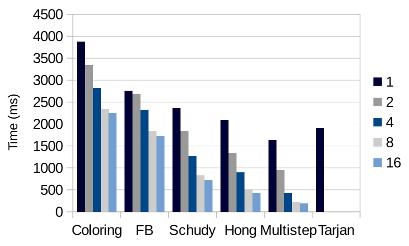
Until the big SCC is discovered, Schudy, Hong, and Multistep perform the same step of actions. What happens next is thus what gives the critical difference in performance between them. The profiler showed that percentage wise, the HalfSet procedure from the Schudy (Algorithm 10) is comparable in terms of performance to the Coloring procedure in Multistep and WCC in Hong. However, the profiler shows that both the Multistep and Hong's algorithm distribute the work more evenly across the threads. We postulate that this is



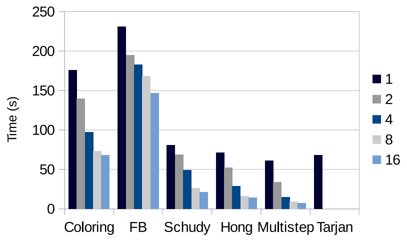
(a) Baidu



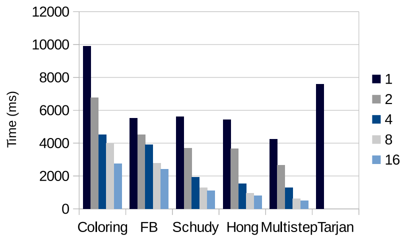
(b) Flickr



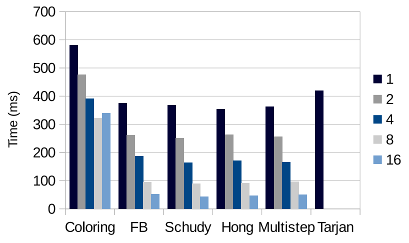
(c) LiveJournal



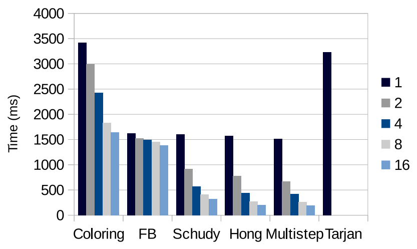
(d) Twitter



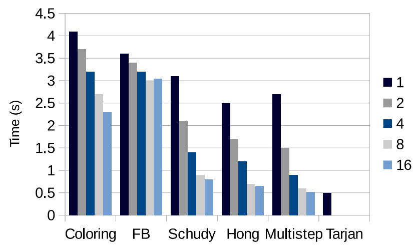
(e) Wikipedia



(f) Patents



(g) Orkut



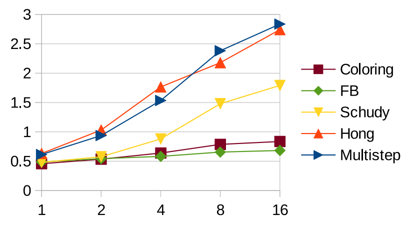
(h) CA-Roads

Figure 4.1: Running times for each data set.

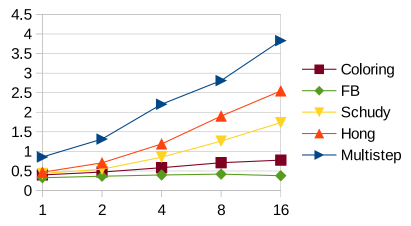
because the coloring propagation technique partitions the graph to a greater degree.

Both the Patents and CA-Roads graph instances are quite different in terms of layout. The CA-Roads is a type of graph that does not have the $s$mall-world property. It has more larger SCCs than the graphs with the $s$mall-world property. The larger SCC has a very large diameter (850) and the second largest SCC is 1000 times larger, when compared to other SCCs from graphs possessing the $s$mall-world of same size (e.g., Baidu and Flickr). The algorithms have worse results because as a result of the very large diameter, the coloring propagation takes a very long time and the parallelization of BFS is not as efficient on a graph with a very large diameter [21].

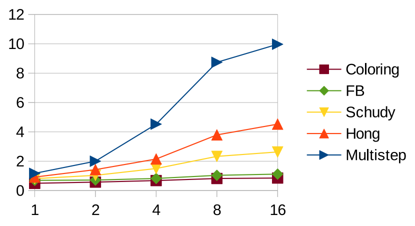
The Patents citation data set has no cycles. This is because a patent can only cite patents that were released before it. What this means in practice is that all SCCs are trivial elements of size 1. This means that all the SCCs are discovered during the trimming procedure. That is why the four algorithms (FB, Hong, Schudy, and Multistep) all exhibit a similar performance.



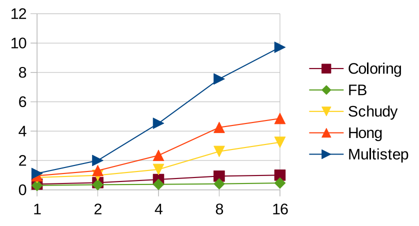
(a) Baidu



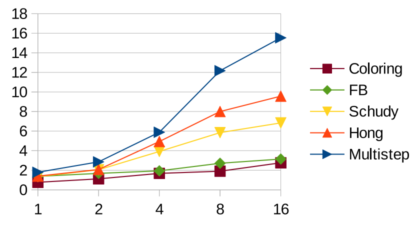
(b) Flickr



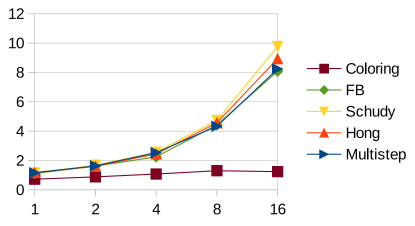
(c) LiveJournal



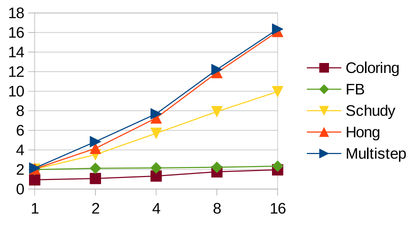
(d) Twitter



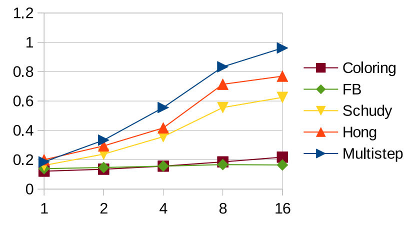
(e) Wikipedia



(f) Patents



(g) Orkut



(h) CA-Roads

Figure 4.2: Speed-up times compared to Tarjan for each data set.

Chapter 5

Conclusions

This thesis focuses on the existing parallel algorithms for detecting Strongly Connected Components (SCCs). A primary goal is to evaluate the effectiveness of these algorithms on real-world graphs with certain structural properties. Moreover, we aim to improve the performance of these algorithms where possible, by means of enhancements in the algorithms design, combining different existing techniques, and improvements in their implementation.

For this project, we have tested the most promising parallel algorithm for detecting SCCs on large and realistic data sets and proposed implementation improvements on the said algorithms. Furthermore, we combined different methods to further improve on Schudy's algorithm.

In Chapter 2, we present parallel algorithm for detecting SCCs. We compare two different approaches for this task. The first approach is based on divide-and-conquer, which uses forward and backward searches to find SCCs and divides the graph into disjoint subsets with respect to SCCs. The second approach is based on DFS. A characteristic of these algorithms is that they require threads to exchange information while running. Although the algorithms in this category are outperformed by the algorithms with a divide-and-conquer approach in general, these algorithms have the advantage of being able to be run "on-the-fly", i.e., while the graph is being constructed.

Furthermore, we presented a performance comparison between these algorithms based on experiments that were provided in the literature. The validity of this comparison is threatened by the fact that most papers presented running times for a few graphs (usually less than five), and the authors may be biased in choosing the graphs as well as in reporting the running times. Moreover, the comparison was done using the original implementation of the algorithm. This means that any optimizations that were not part of the original implementation are not taken into account in this comparison. For example, Multistep used a different BFS implementation that could possibly make Hong's algorithm be more efficient too.

In Chapter 3, we discussed our implementation of the selected algorithms, as well as some improvements to speed up the implementations. These include:

- Improvements on Hong's efficient representation of the graph instance, resulting in a further reduction of memory usage.
- A more flexible implementation of parallel BFS, which can be used by

Hong's algorithm after the detection of the large SCC. This lead to a performance improvement of up to 50% in graphs such as CA-Roads.

- The usage of efficient Java libraries to further improve the performance.

The evaluation of the algorithms on real-world graph instances is presented in Chapter 4. Overall, the Multistep algorithm exhibits the best performance, while also Hong's algorithm and the improved version of Schudy's algorithm obtain good results on most graphs. So the answer to our first research question is that the Multistep algorithm is the state-of-the-art algorithm for the parallel decomposition of SCCs. We note however, that the Multistep algorithm does not contain any new algorithmic ideas, but rather constitutes a small combination of existing techniques.

It is important to remark that Multistep does not actually propose any novel approach to parallel decomposition of SCCs. A novel approach may improve on the performance of Multistep.

Our experiments show that improvements to the implementation of the algorithms we used can have a major impact on performance. This is a promising direction for future work. One area of improvement is a more optimized implementation of basic graph operations, such as BFS [23, 51]. Since all algorithms spend a lot of time performing these operations, even small improvements to their implementation could be very worthwhile. Another fruitful approach for improving the performance of parallel algorithms for detecting SCCs could be to further integrate techniques that are employed within the different algorithms discussed in this thesis.

Bibliography

[1] Rajeev Alur, Costas Courcoubetis, and David Dill. Model-checking for probabilistic real-time systems. In *International Colloquium on Automata, Languages, and Programming*, pages 115–126. Springer, 1991.

[2] Jiří Barnat, Luboš Brim, and Milan Češka. DiVinE-CUDA-a tool for GPU accelerated LTL model checking. *arXiv preprint arXiv:0912.2555*, 2009.

[3] 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.

[4] Jiri Barnat, Jan Havlíček, and Petr Ročkai. Distributed LTL model checking with hash compaction. *Electronic Notes in Theoretical Computer Science*, 296:79–93, 2013.

[5] Jiří Barnat and Pavel Moravec. Parallel algorithms for finding SCCs in implicitly given graphs. In *International Workshop on Formal Methods for Industrial Critical Systems*, pages 316–330. Springer, 2006.

[6] Béatrice Bérard, Michel Bidoit, Alain Finkel, François Laroussinie, Antoine Petit, Laure Petrucci, and Philippe Schnoebelen. *Systems and software verification: model-checking techniques and tools.* Springer Science & Business Media, 2013.

[7] Vincent Bloemen. On-the-fly parallel decomposition of strongly connected components. Master's thesis, University of Twente, 2015.

[8] Vincent Bloemen, Alfons Laarman, and Jaco van de Pol. Multi-core on-the-fly SCC decomposition. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming.* ACM, 2016.

[9] Béla Bollobás, Christian Borgs, Jennifer T. Chayes, Jeong Han Kim, and David B. Wilson. The scaling window of the 2-SAT transition. *Random Structures & Algorithms*, 18(3):201–256, 2001.

[10] Lubos Brim, Jirí Barnat, et al. Platform dependent verification: On engineering verification tools for 21st century. In *PDMC*, pages 1–12, 2011.

[11] 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.

[12] Allan Cheng, Søren Christensen, and Kjeld Høyer Mortensen. Model checking coloured petri nets-exploiting strongly connected components. *DAIMI report series*, 26(519), 1997.

[13] Edsger W Dijkstra. Finding the maximum strong components in a directed graph. In *Selected Writings on Computing: A Personal Perspective*, pages 22–30. Springer, 1982.

[14] Reggie Ebendal. Divide-and-conquer algorithm for parallel computation of terminal strongly connected components. In *Bachelor's thesis.* VU University, Amsterdam, 2015.

[15] Sami Evangelista, Alfons Laarman, Laure Petrucci, and Jaco Van De Pol. Improved multi-core nested depth-first search. In *International Symposium on Automated Technology for Verification and Analysis*, pages 269–283. Springer, 2012.

[16] Lisa K Fleischer, Bruce Hendrickson, and Ali Pınar. On identifying strongly connected components in parallel. In *International Parallel and Distributed Processing Symposium*, pages 505–511. Springer, 2000.

[17] Bernard Gendron and Teodor Gabriel Crainic. Parallel branch-and-branch algorithms: Survey and synthesis. *Operations Research*, 42(6):1042–1066, 1994.

[18] Alan Gibbons. *Algorithmic Graph Theory.* Cambridge University Press, Cambridge, 1985.

[19] Michelle Girvan and Mark EJ Newman. Community structure in social and biological networks. *Proceedings of the national academy of sciences*, 99(12):7821–7826, 2002.

[20] Bronwyn H. Hall, Adam B. Jaffe, and Manuel Trajtenberg. The NBER patent citations data file: Lessons, insights and methodological tools. In *NBER Working Papers 8498, National Bureau of Economic Research, Inc*, 2001.

[21] Sungpack Hong, Tayo Oguntebi, and Kunle Olukotun. Efficient parallel graph exploration on multi-core cpu and gpu. In *Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on*, pages 78–88. IEEE, 2011.

[22] Sungpack Hong, Nicole C Rodia, and Kunle Olukotun. On fast parallel detection of strongly connected components (SCC) in small-world graphs. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, page 92. ACM, 2013.

[23] Richard E Korf and Peter Schultze. Large-scale parallel breadth-first search. In *AAAI*, volume 5, pages 1380–1385, 2005.

[24] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. What is Twitter, a social network or a news media? In *Proc. Int. World Wide Web Conf.*, pages 591–600, 2010.

[25] Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection. `http://snap.stanford.edu/data`, June 2014.

[26] Jure Leskovec, Kevin J. Lang, Anirban Dasgupta, and Michael W. Mahoney. Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters. *CoRR*, abs/0810.1355, 2008.

[27] Jure Leskovec, Kevin J. Lang, Anirban Dasgupta, and Michael W. Mahoney. Statistical properties of community structure in large social and information networks. In *Proc. Int. World Wide Web Conf.*, pages 695–704, 2008.

[28] Fast Utils Library. `http://fastutil.di.unimi.it/`, September 2016.

[29] Gavin Lowe. Concurrent depth-first search algorithms. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 202–216. Springer, 2014.

[30] William McLendon III, 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.

[31] Kurt Mehlhorn, Stefan Näher, and Peter Sanders. Engineering dfs-based graph algorithms. *Partially supported by DFG grant SA*, 933:3–1, 2007.

[32] Kurt Mehlhorn and Peter Sanders. *Algorithms and Data Structures: The Basic Toolbox.* Springer Publishing Company, Incorporated, 1st edition, 2008.

[33] Alan Mislove, Hema Swetha Koppula, Krishna P. Gummadi, Peter Druschel, and Bobby Bhattacharjee. Growth of the Flickr social network. In *Proc. Workshop on Online Social Networks*, pages 25–30, 2008.

[34] Alan Mislove, Massimiliano Marcon, Krishna P. Gummadi, Peter Druschel, and Bobby Bhattacharjee. Measurement and analysis of online social networks. In *Proc. Internet Measurement Conf.*, 2007.

[35] Koblenz network collection. `http://konect.uni-koblenz.de./`, September 2016.

[36] Xing Niu, Xinruo Sun, Haofen Wang, Shu Rong, Guilin Qi, and Yong Yu. Zhishi.me – weaving Chinese linking open data. In *Proc. Int. Semantic Web Conf.*, pages 205–220, 2011.

[37] S. Orzan. *On distributed verification and verified distribution.* PhD thesis, VU University, Amsterdam, 2004.

[38] John H Reif. Depth-first search is inherently sequential. *Information Processing Letters*, 20(5):229–234, 1985.

[39] Elisabeth Remy, Brigitte Mossé, Claudine Chaouiya, and Denis Thieffry. A description of dynamical graphs associated to elementary regulatory circuits. *Bioinformatics*, 19(suppl 2):172–178, 2003.

[40] Xavier Renault, Fabrice Kordon, and Jérôme Hugues. From AADL architectural models to petri nets: Checking model viability. In *2009 IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing*, pages 313–320. IEEE, 2009.

[41] Warren Schudy. Finding strongly connected components in parallel using O(log n) reachability queries. In *Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures*, pages 146–151. ACM, 2008.

[42] 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, 11th International Conference*, volume 3440 of *Lecture Notes in Computer Science*, pages 174–190. Springer, 2005.

[43] Robert Sedgewick and Kevin Wayne. *Algorithms (Fourth edition deluxe)*. Addison-Wesley, 2016.

[44] Micha Sharir. A strong-connectivity algorithm and its applications in data flow analysis. *Computers & Mathematics with Applications*, 7(1):67–72, 1981.

[45] George M Slota, Sivasankaran Rajamanickam, and Kamesh Madduri. BFS and coloring-based parallel algorithms for strongly connected components and related problems. In *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*, pages 550–559. IEEE, 2014.

[46] DAS-4: Distributed ASCI Supercomputer. `http://www.cs.vu.nl/das5/home.shtml`, September 2016.

[47] Robert Tarjan. Depth-first search and linear graph algorithms. *SIAM journal on computing*, 1(2):146–160, 1972.

[48] Thomas N. Theis and Paul M. Solomon. It's time to reinvent the transistor! *Science*, 327(5973):1600–1601, 2010.

[49] Zhicheng Wang and Peter M. Maurer. Lecsim: A levelized event driven compiled logic simulation. In *Proceedings of the 27th ACM/IEEE Design Automation Conference*, DAC '90, pages 491–496, ACM, 1990.

[50] Wikimedia Foundation. Wikimedia downloads. `http://dumps.wikimedia.org/`, January 2010.

[51] Yuichiro Yasui and Katsuki Fujisawa. Fast and scalable numa-based thread parallel breadth-first search. In *High Performance Computing & Simulation (HPCS), 2015 International Conference on*, pages 377–385. IEEE, 2015.