Distributed Algorithms for SCC Decomposition

JIŘÍ BARNAT and JAKUB CHALOUPKA, Department of Computer Science, Faculty of Informatics, Masaryk University Brno, Czech Republic. E-mail: barnat@fi.muni.cz; xchalou1@fi.muni.cz

JACO VAN DE POL, Formal Methods and Tools, Department of EEMCS, University of Twente, The Netherlands. E-mail: j.c.vandepol@ewi.utwente.nl

Abstract
We study existing parallel algorithms for the decomposition of a partitioned graph into its strongly connected components (SCCs). In particular, we identify several individual procedures that the algorithms are assembled from and show how to assemble a new and more efficient algorithm, called Recursive OBF (OBFR), to solve the decomposition problem. We also report on a thorough experimental study to evaluate the new algorithm. It shows that it is possible to perform SCC decomposition in parallel efficiently and that OBFR, if properly implemented, is the best choice in most cases.

Keywords: parallel algorithms, strongly connected components

1 Introduction
The problem of finding strongly connected components (SCCs), known also as SCC decomposition, is one of the basic graph problems that finds its applications in many research fields, even beyond the scope of computer science. An efficient algorithmic solution to this problem is due to Tarjan [25], who showed that, given a graph with \( n \) vertices and \( m \) edges, it is possible to identify and list all SCCs of the graph in \( O(n + m) \) time and \( O(n) \) space.

Among many applications, the algorithm may be used also for the analysis of computer systems. In particular, algorithms for SCC decomposition find their application in distributed formal verification tools such as CADP [18], DiVinE [2], DUPPAAL [5], LiQuor [12], µCRL [6], etc. Namely, they allow the tools to verify quantitative properties of probabilistic systems, compute \( \tau \)-confluence [8], form a pre-processing step for branching bisimulation reduction, or verify systems with fairness constraints or properties given by extensions of Büchi automata.

Unfortunately, graphs modelling complex computer systems tend to be very large, which makes it hard to handle them on a single machine. One way to tackle this problem is to distribute the graph across a cluster of workstations and employ a distributed algorithm to decompose the partitioned graph. However, Tarjan’s algorithm (and all other linear algorithms for SCC decomposition, e.g. Kosaraju’s algorithm also known as Double DFS [15]) strongly rely on the depth-first search post-ordering of vertices, whose computation is known to be \( P \)-complete [23], and thus, difficult to be computed in parallel. Therefore, different approaches have been used to design parallel algorithms for solving the problem.
2 Distributed Algorithms for SCC Decomposition

A parallel algorithm based on matrix multiplication was described in [19] and further improved in [1, 14]. The algorithm works in \( O(\log^2 n) \) time in the worst case. However, to achieve this low time complexity it requires \( O(n^{2.376}) \) parallel processors. As typical graphs that we are interested in contain millions of vertices the algorithm is practically unusable and is only interesting from a theoretical point of view. Another parallel algorithm for finding SCCs was given in [17]. It exploits the fact that it is possible to efficiently compute the set of vertices reachable from a certain vertex or set of vertices in parallel. The general idea of the algorithm is to repeatedly pick a vertex of the graph and identify the component to which it belongs, by using a forward and a backward parallel reachability procedure. The algorithm proved to be efficient enough in practice, which resulted in several theoretical improvements of it [20, 22]. The worst-time complexity of the algorithm is \( O(n \cdot (n + m)) \). Nevertheless, the algorithm exhibits \( O(m \cdot \log n) \) expected time [17]. Another algorithm was introduced in [22]. That algorithm is more involved, but still, its basic building block is a simple parallel value iteration technique.

In this article, which can be viewed as a full version of [3, 4], we summarize a number of known procedures used for distributed SCC decomposition. Moreover, we present a new algorithm based on re-arranging these procedures, and extensively compare its implementation with existing algorithms. The rest of the article is organized as follows. We recapitulate basic terms and definitions in Section 2, describe known techniques and algorithms for solving SCC decomposition in Section 3. The new algorithm based on recursive application of OBF [3, 4] is described in Section 4. Compared with [3], we added full proofs for the correctness and the complexity claims. Results of experiments are in Section 5. In particular, we compare our new algorithm with the algorithms from [17, 22], and we measure the effect of decomposing sub-graphs one by one, or in parallel. Contributions of the article are summarized and future work is outlined in Section 6.

2 Preliminaries

2.1 Directed graphs

A (directed) graph \( G \) is a pair \((V, E)\), where \( V \) is a set of vertices, and \( E \subseteq V \times V \) is a set of (directed) edges. If \((u, v) \in E\), then \( v \) is called (immediate) successor of \( u \) and \( u \) is called (immediate) predecessor of \( v \). The indegree of a vertex \( v \) is the number of immediate predecessors of \( v \). The transposed graph of \( G \) is \( G^T \), is the graph \( G \) with all edges reversed, i.e. \( E^T = \{(u, v) \mid (v, u) \in E\} \).

Let \( G = (V, E) \) be a directed graph. Let \( E^* \) be a transitive and reflexive closure of \( E \) and \( s, t \in V \) two vertices. We say that vertex \( t \) is reachable from vertex \( s \) if \((s, t) \in E^*\). If \( s_k \) is reachable from \( s_0 \), then there is a sequence of vertices \( s_0, \ldots, s_k \), s.t. \((s_i, s_{i+1}) \in E \) for all \( 0 \leq i < k \). We call this sequence a path. A path is simple if it contains no duplicated vertices. The length of the path is \( k \), i.e. the number of edges. A graph is rooted if there is an initial vertex \( s_0 \in V \) such that all vertices in \( V \) are reachable from \( s_0 \). Given a graph \( G \), we use \( n, m \) and \( l \), to denote the number of vertices and edges, and the length of the longest simple path between any two vertices in \( G \), respectively.

A set of vertices \( C \subseteq V \) is strongly connected, if for any vertices \( u, v \in C \), we have that \( v \) is reachable from \( u \). A strongly connected component (SCC) is a maximal strongly connected \( C \subseteq V \), i.e. such that no \( C' \) with \( C \subseteq C' \subseteq V \) is strongly connected. A maximal SCC \( C \) is trivial if \( C \) is made of a single vertex \( c \) and \((c, c) \notin E \), and is non-trivial otherwise. Henceforward, an SCC is also referred to simply as a component.

Let \( W_G \) be the set of all SCCs of graph \( G \). The quotient graph of graph \( G \) is a directed graph \( SCC_G = (W_G, H_G) \), where \( H_G = \{(w_1, w_2) \mid \exists (u_1, u_2) \in E \} \), i.e. there is an edge between SCCs if and only if there is an edge between some members of the SCCs in the
original graph. Note that the quotient graph of any directed graph is acyclic. Given a graph $G$, we denote by $N$, $M$ and $L$, the number of vertices and edges, and the length of the longest (simple) path in the quotient graph of $G$, respectively. An SCC is leading if it has no predecessors in the quotient graph. A set $S \subseteq V$ is SCC-closed if each SCC in the graph is either completely inside the set or completely outside the set; such $S$ is also referred to as an independent sub-graph.

For $v \in W \subseteq V$, the forward closure of $v$ in $W$ is the set of reachable states from $v$ in the graph $(V, E_W)$, where $E_W = \{(x, y) | (x, y) \in E \land x, y \in W\}$. If $W$ is not specified, the whole graph is meant. The forward closure of $S \subseteq W$ in $W$ is the union of forward closures of all vertices from $S$ in $W$. Finally, the backward closure of $v$ (or $S$) in $W$ is the forward closure of $v$ (or $S$) in $W$ in the graph $G^T$.

2.2 Graph representation

A directed graph can be given in many ways. We restrict ourselves to explicit vertex representations, excluding symbolic representations, e.g. based on binary decision diagrams.

Beside the standard representations by adjacency lists or an adjacency matrix we also mention graphs that are given implicitly (do not confuse with symbolic representation, this is still an explicit vertex representation). A rooted graph is given implicitly if it is defined by its initial vertex and a function returning immediate successors of an arbitrary vertex. Within the context of implicitly given graphs there are some restrictions that algorithms have to follow. If an algorithm requires any piece of information that cannot be concluded from the implicit definition of the graph, it has to compute the information first. For example, there is no way to directly identify immediate predecessors of a given vertex from the implicit definition of the graph. If the algorithm needs to enumerate immediate predecessors, then the predecessors must be stored, while enumerating the whole graph first. Similarly, in order to number the vertices of an implicitly given graph, one must enumerate all its vertices first. For numbering the vertices of implicitly given graphs a parallel procedure was introduced in [18]. Note that all vertices of an implicitly given graph are reachable from the initial vertex by definition.

The reason for dealing with implicitly given graphs comes from practice. In many cases, the description of rules according to which the graph can be generated is more space efficient than the enumeration of all vertices and edges. The difference might be quite significant. For example, in the context of model checking [13], the implicit definition of the graph is up to exponentially more succinct compared with the explicit one. This is commonly referred to as the state explosion problem [13]. However, it turns out that, in the situation where the graph has to be traversed more than once, which is the case for all parallel SCC decomposition algorithms, it is advantageous to first generate the whole graph and store it in an explicit form. All subsequent computations are then performed using the explicit representation. We save the time for repeated generation of successors and since the graphs we are interested in are mainly sparse, the needed memory is proportional to the number of vertices only.

3 Known algorithms

Before describing individual parallel algorithms, we describe the basic techniques that the later algorithms will use. This allows us to describe the algorithms and analyse their behaviour in a more compact and clearer way.

All parallel algorithms presented in this article build on the same basic principle. The graph to be decomposed is divided into independent (SCC-closed) sub-graphs. These are further divided into smaller independent sub-graphs until they become SCCs. All the algorithms take advantage of the fact that computation on separate independent sub-graphs can be done in parallel.
4 Distributed Algorithms for SCC Decomposition

3.1 Reachability relation

Computation of the reachability relation is the core procedure used in all the algorithms. The task of the procedure is to identify all vertices that are reachable from a given vertex, i.e. to compute its forward closure. The standard breadth-first or depth-first traversals of the graph can be employed to do so using $O(n)$ space and $O(n+m)$ time.

The reachability procedure is the first place where parallelism comes into play in the algorithms. The parallelization of a reachability procedure has by now become a standard technique [10, 11, 21, 24]. A so called partition function is used to assign vertices to processors. Each processor is responsible for the exploration of the vertices assigned to it by the partition function. Each processor maintains its own set of already visited vertices and its own list of vertices to be explored. If a vertex has been visited previously (it is in the set of visited vertices), then its re-exploration is omitted. Otherwise, its immediate successors are generated and distributed into lists of vertices, to be explored according to the partition function.

The algorithms described in the next section use the notion of backward reachability, in addition to the notion of forward reachability. The task of a backward reachability procedure is to identify all vertices that a given vertex can be reached from. The procedure for backward reachability mimics the behaviour of the procedure for the forward reachability except it uses immediate predecessors instead of immediate successors during graph traversal.

Note that in many cases, the forward and backward reachability procedure are restricted to a particular independent sub-graph of the original graph. This can be achieved by an additional marking of that sub-graph, or simply by deleting edges that leave that sub-graph.

3.2 Pivot selection

In several algorithms, there is a point at which a certain vertex (called pivot) must be selected from the current independent sub-graph to start the decomposition of that sub-graph. Pivot selection plays a significant role in the complexity of the algorithms. Imagine, we always pick a pivot belonging to a component that has no descendant components in the component graph of the sub-graph being decomposed. Due to the acyclicity of the component graph such a component always exists. Having such a pivot, all vertices belonging to the corresponding component can be identified using only a single forward reachability initiated at the pivot. Decomposing the graph to SCCs in this manner results in a linear-time procedure. Unfortunately, to pick pivots so that the condition above is satisfied means to pick pivots in the depth-first search post-ordering, which is, as stated in the Section 1, difficult to be done in parallel. Since the optimal pivot selection is difficult, pivots are typically selected randomly.

3.3 Trivial SCCs

This sub-section presents an efficient technique for the elimination of leading and terminal trivial (LT and TT, respectively) components from any independent sub-graph. Use of this technique can significantly speed up all the SCC decomposition algorithms, since they are not that efficient on detecting trivial components.

Every vertex that has zero predecessors must be a trivial component and as such it can be immediately removed (along with all incident edges) from the graph. Removing such a vertex may, however, produce new vertices without predecessors that can be removed in the same way. We refer to this recursive elimination technique as One-Way-Catch-Them-Young (OWCTY) elimination [16]. The technique can be applied in an analogous way to vertices without successors (Reversed OWCTY)
Distributed Algorithms for SCC Decomposition

3.4 FB

The FB algorithm [17] is the basic algorithm, outlined in Section 1. We illustrate the basic principle of this algorithm. Figure 1 shows the basic step of the algorithm. First, a vertex (called pivot) is selected at random from an independent sub-graph (the whole graph in this situation) that is not known to be a single SCC yet. Second, the forward and the backward closure of the pivot are computed; these are depicted by shaded regions. This procedure divides the graph into four independent sub-graphs. The vertices that are both in the forward and the backward closure form the SCC of pivot and need not be further processed. The other three sub-graphs are: vertices in the forward closure but not in the backward closure, vertices in the backward closure but not in the forward closure and vertices that are neither in the forward nor in the backward closure. These three sub-graphs have to be further decomposed. They can be decomposed independently and hence in parallel. Recursive application of the basic step is used to do it.

The pseudocode of the algorithm is in Figure 2. A pivot is selected using procedure PIVOT and its forward and backward closures are computed using parallel reachability procedures FWD and BWD. Both reachability procedures have two parameters. Besides the vertex or vertices to start from, each reachability procedure is also given a set of vertices that its exploration is limited to. This ensures that given a sub-graph, the procedure will explore only immediate successors or predecessor of vertices within the sub-graph. The sets of vertices as computed by forward and backward reachability procedures are referred to as $F$ and $B$, respectively. Having computed both sets $F$ and $B$, a new component is identified as the intersection of $F$ and $B$, and recursive calls for three new subgraphs are made. As stated in Section 1, the time complexity of the algorithm is $O(n \cdot (n+m))$. 

3.5 Colouring/heads-off

The colouring algorithm was introduced in [22]. It uses a totally ordered set of colours. Initially, each vertex has its own colour. The colours are repeatedly propagated to successors with a smaller colour, until all edges are non-decreasing. A forward reachability procedure augmented to propagate maximal visited colours can be used for this task. Note that a vertex can be re-coloured several times, which results in time complexity of $O(n \cdot m)$ [9]. The final colour of a vertex is the colour of its maximal predecessor, i.e. predecessor with maximal colour. Here a predecessor does not necessarily mean an immediate predecessor (as in the rest of this article), but here it means any vertex in the backward closure. After colouring, all vertices in a single SCC have the same colour. This is because all vertices in a single SCC share the same set of predecessors. So all edges between vertices of different colours can be removed. This technique is able to divide the graph into more than four parts, as opposed to the technique presented in Sub-section 3.4. Unfortunately, we do not know how to do this in linear time. A graph division obtained after colouring is depicted in Figure 3.

In the second step, one takes as roots those vertices that kept their initial colour. The SCC of each root consists of those vertices that are backward reachable (within the same colour) from it. These SCCs are removed (heads-off) and the algorithm proceeds with the remaining sub-graph and with the original colour assignment.

The pseudocode of the algorithm is in Figure 4. Computation of maximal predecessors is done by the procedure FWD-MAXPRED, which returns the list of roots as $PredList$. It also computes for each $k \in PredList$ the set $V_k$ of vertices with maximal predecessor $k$. The SCCs of the roots are identified by the standard procedure BWD, which performs backward reachability. The removal of these SCCs
Distributed Algorithms for SCC Decomposition

Figure 4. Colouring/heads-off (CH) algorithm

```latex
\begin{alg}
\begin{algorithmic}[1]
\Procedure{CH}{V}
\If{V \neq \emptyset}
\State PredList, (V_k)_{k \in \text{PredList}} := \text{FWD-MAXPRED}(V)
\Comment{line 8 was referred to as heads-off in the previous paragraph. Edges are not removed there. Instead, separate recursive calls of the main procedure restricted to the appropriate sub-graphs are used.}
\Comment{The time complexity of the algorithm is } \mathcal{O}(L + 1) \cdot n \cdot m\Comment{, where } \mathcal{O}(n \cdot m) \Comment{comes from the complexity of the FWD-MAXPRED procedure. The total complexity follows from the fact that every time a recursive call is invoked, it is on a graph with strictly shorter longest path in the quotient graph.}
\EndIf
\EndProcedure
\end{algorithmic}
\end{alg}
```

Figure 5. OBF algorithm

```latex
\begin{alg}
\begin{algorithmic}[1]
\Procedure{OBF}{V, v}
\State Seeds := \{v\}
\Comment{v is initial vertex}
\While{V \neq \emptyset}
\State Eliminated, Reached := OWCTY(Seeds, V)
\State V := V \setminus \text{Eliminated}
\Comment{[All elements of Eliminated are trivial SCCs]}
\State B := BWD(Reached, V)
\Comment{in parallel do}
\State FB(B)
\Comment{od}
\State Seeds := FWD-SEEDS(B, V)
\State V := V \setminus B
\Comment{od}
\EndWhile
\EndProcedure
\end{algorithmic}
\end{alg}
```

on line 8 was referred to as heads-off in the previous paragraph. Edges are not removed there. Instead, separate recursive calls of the main procedure restricted to the appropriate sub-graphs are used.

The time complexity of the algorithm is $O(L + 1) \cdot n \cdot m$, where $O(n \cdot m)$ comes from the complexity of the FWD-MAXPRED procedure. The total complexity follows from the fact that every time a recursive call is invoked, it is on a graph with strictly shorter longest path in the quotient graph.

### 3.6 OBF

This algorithm is based on a recent technique OWCTY-BWD-FWD (OBF) [3, 4] which gave name to the whole algorithm. It identifies a number of independent sub-graphs (called OBF slices) in $O(n + m)$ time. The slices are then decomposed using the FB algorithm. This algorithm assumes the input graph to be rooted, i.e. we have an initial vertex from which all other vertices are reachable.

The OBF technique repeatedly employs OWCTY elimination, succeeded with backward and forward reachability. Each iteration identifies one OBF slice. The pseudocode of the algorithm is in Figure 5. A graph and two steps of the technique performed on the graph are depicted in Figure 6.
We simultaneously describe the figure and the pseudocode. We start with the initial vertex (the vertex with no predecessors in the figure, the vertex \( v \) in the pseudocode). The OWCTY elimination procedure (line 5 in pseudocode) eliminates all LT components (the set \( \text{Eliminated} \) in the pseudocode) and visits some vertices of all components immediately reachable from the eliminated trivial ones. Visited but not eliminated vertices are shown as vertices with a little cross (the set \( \text{Reached} \)). A backward reachability (\( \text{BWD()} \)) performed from vertices with the little cross identifies the first OBF slice (the set \( \text{B} \)). Note that the slice contains exactly all SCC immediately reachable from the eliminated trivial components. The decomposition of the slice is initiated as an independent parallel procedure (line 10). Then a forward reachability procedure that stops on immediate successors of vertices in the slice is executed (\( \text{FWD-SEEDS()} \)). These successors (vertices with the little circle in the figure, \( \text{Seeds} \) on line 12 in the pseudocode) are used to start the next iteration of OBF. The time complexity of the algorithm is \( O(n \cdot (n+m)) \), the same as for the FB algorithm.

4 Recursive OBF

As shown in [4], OBF performs better than FB in a number of experiments. Note that in OBF the graph is split into slices in linear time. On each slice, algorithm FB is applied. But, as OBF is better than FB, we now propose to \textit{recursively apply} OBF to the slices.

However, the slice may not be rooted, so we must:

- repeatedly pick a vertex from the slice and compute its forward closure within the slice; we call this a ‘rooted chunk’. Subsequently run OBF on each rooted chunk within the slice;
- add a termination criterion in case the whole slice is one SCC.

Adding a termination criterion is easy. No special work has to be done. We simply count the vertices visited during the first backward search in the first rooted chunk (The ‘B’ part of OBF). If the slice consists of exactly one SCC there will be only one rooted chunk in it; O will not eliminate any vertex,
and so B will be started from the root and explores the whole slice. Conversely, if B starting from
the root of the first chunk explores the whole slice, the slice is one SCC, for it is both the forward
and the backward closure of the root. We now describe Recursive OBF (OBFR) in more detail.

The pseudocode of OBFR is in Figure 7. The suffix ‘-P’ in the name of the procedure means that
it runs in parallel on independent subgraphs. The term OBFR without any suffixes is used to refer to
Recursive OBF as such, without specifying the degree of parallelism (see Sub-section 4.1).

We start with the whole graph. Vertices in recognized SCCs are removed from the ‘working’ set
V until we end up with an empty set at which point all SCCs have been identified.

Initially, we assume that we do not have a vertex from which all other vertices are reachable (initial
vertex). To start OBF we need such a vertex, so we pick one vertex (line 3) and compute its forward
closure Range in V using procedure FWD() (line 4). OBF is then applied on Range. Vertices from
V \ Range will be processed in the next iterations of the main while-loop (lines 2–24).

Before OBF is started on Range, Range is saved into OriginalRange, this will enable us to determine
if a slice found by OBF is an SCC. Of course, in the actual implementation we only store the size of
OriginalRange. On line 9, there is an invariant ‘(The forward closure of Seeds in Range = Range)’. 
In the first iteration of the while-loop on lines 8–23 the invariant holds trivially, because Seeds
contains just one vertex and Range was computed as a forward closure of that vertex. Procedure
OWCTY() eliminates LT components by repeatedly removing indegree 0 vertices reachable from
Seeds. Eliminated vertices are returned as the set Eliminated, and subsequently removed from Range.

If (B = OriginalRange) then B is SCC
else
  in parallel do
    OBFR-P(B)
  od
Seed := FWD-SEEDS(B, Range)
fi
Range := Range \ B
od end

Figure 7. OBFR
non-eliminated vertex has to contain some vertex from Reached. All elements from Eliminated are trivial SCCs. Now a backward search is started from vertices in Reached. This search is implemented by procedure BWD(). Backward closure of Reached in Range is returned as the set B. This is the first SCC-closed slice found by OBF. If the set B equals the set OriginalRange, it means that all vertices in the SCC-closed set OriginalRange are reachable from the same single vertex (note that B = OriginalRange is only possible in the first iteration of the while-loop 8–23) and so B is indeed an SCC. Consequently, Range \ B is the empty set and the while-loop finishes.

If B \ OriginalRange we run OBFR-P() on B recursively. Moreover, note that the nested procedure can be run in parallel, which increases parallelism. Seeds for the next iteration of the while-loop 8–23 are computed by the procedure FWD-SEEDS, which simply returns all vertices from Range that are immediate successors of vertices in B but not in B. Since all paths that reach vertices in Range \ B from B must contain some vertex from Seeds, after we subtract B from Range, the invariant of line 9 is satisfied. When Range = \emptyset, the while-loop 8–23 finishes and we handle the remaining vertices in V.

We now formally prove the correctness of the algorithm. The key point is the invariant on line 9. It ensures that the whole graph is eventually processed. As argued earlier, it trivially holds in the first iteration of the while-loop on lines 8–23. Thus, it remains to show that, if the invariant holds in iteration i, then it holds also in iteration i + 1. Together with the fact than Range gets smaller in every iteration, it implies that the whole rooted chunk computed on line 4 is processed on lines 8–23. Another important point is that the set B computed on line 13 is an independent (SCC-closed) sub-graph. This implies partial correctness. Since line 18 is executed only if B is smaller than OriginalRange, finite depth of recursion and thus termination of the algorithm is ensured. All the statements in this paragraph are proved below.

We sometimes use a set of vertices to refer to the graph induced by that set. To prove the invariant, we need to strengthen it a bit. In addition to the fact that the forward closure of Seeds in Range is equal to Range, we argue that Range is an independent sub-graph of OriginalRange. Since initially Range = OriginalRange, the strengthened invariant holds in the first iteration of the while-loop.

The following lemmata analyse one iteration of the while-loop on lines 8–23. In the whole iteration Range is used to refer to the set Range on line 9, i.e. at the very beginning of the iteration. The same goes for Seeds. The set Range computed on line 11 is referred to as Range'. The set Range computed on line 22 is referred to as Range''. The set Seeds computed on line 20 is referred to as Seeds'.

**Lemma 1**

Vertices eliminated by OWCTY() (the set Eliminated on line 10) are trivial SCCs of OriginalRange.

**Proof.** Let us suppose, for the sake of contradiction, that OWCTY() eliminates a vertex v such that there is a vertex v' such that there is a path in OriginalRange from v to v' and vice versa. Range is an independent sub-graph of OriginalRange. It follows, that Range contains a cycle c = (v_0, v_1, ..., v_k) with v_0 = v_v = v. At the moment when v was eliminated it must have had indegree 0, which means that vertex v_{k-1} must have been eliminated earlier, since there is an edge from v_{k-1} to v. By repeating this argument, we get that all vertices v_{k-2}, v_{k-3}, ..., v_0 were eliminated before v and since v_0 = v, it means that v was eliminated before v. An obvious contradiction. 

**Lemma 2**

Let Reached be the set of vertices at which OWCTY() stops (cf. line 10; these are the non-eliminated vertices from Seeds and the non-eliminated successors of the eliminated vertices). Then the forward closure of Reached in Range' is equal to Range'.

**Proof.** Since the forward closure of Seeds in Range is equal to Range, for each v \ Range' there is w \ Seeds such that there is a path p = (v_0, v_1, ..., v_k), where v_0 = w, v_k = v and k \geq 0. Since OWCTY()
eliminates only indegree 0 vertices, there is $j \geq 0$ such that vertices $v_0, \ldots, v_{j-1}$ were eliminated and vertices $v_j, \ldots, v_k$ were not, and vertex $v_j$ is in the set $\text{Reached}$. It follows that $v$ is reachable from $v_j$ in $\text{Range}'$. Therefore, the forward closure of $\text{Reached}$ in $\text{Range}'$ is $\text{Range}'$. ■

**Lemma 3**

The set $B$ computed on line 13 (The backward closure of $\text{Reached}$ in $\text{Range}'$) is an independent sub-graph of $\text{Range}'$. (No SCC has vertices both in $B$ and $\text{Range}' \setminus B$).

**Proof.** It is sufficient to show that there is no edge from $\text{Range}' \setminus B$ to $B$. However, that is obvious for the existence of such edge $(w, v)$ would imply that $w \in B$, which is impossible since, according to the assumption, $w \in \text{Range}' \setminus B$. ■

**Lemma 4**

Let $\text{Seeds}'$ be the successors of the vertices in $B$ which are in $\text{Range}'' = \text{Range}' \setminus B$. Then the forward closure of $\text{Seeds}'$ in $\text{Range}''$ is $\text{Range}''$.

**Proof.** Since $\text{Reached} \subseteq B$, the forward closure of $B$ in $\text{Range}'$ is $\text{Range}'$ by Lemma 2. Therefore, for each vertex $v \in \text{Range}''$ there is $w \in B$ such that there is a path $p = (v_0, v_1, \ldots, v_k)$, where $v_0 = w$, $v_k = v$ and $k \geq 1$. Let $j$ be the greatest index with the property that $v_j \in B$, then $v_j+1 \in \text{Seeds}'$ and the path $p' = (v_{j+1}, \ldots, v_k)$ is a path in $\text{Range}''$. Thus $v$ is reachable from $v_j+1$ in $\text{Range}''$. It follows that the forward closure of $\text{Seeds}'$ in $\text{Range}''$ is equal to $\text{Range}''$. Together with the fact that $\text{Range}''$ is $\text{Range}$ without some independent sub-graphs (Lemmas 1 and 3) it implies that if $\text{Range}'' \neq \emptyset$, then the strengthened invariant is satisfied in the next iteration.

So far, we proved the strengthened invariant of line 9 by analysing one iteration of the while-loop on lines 8–23. It follows that the whole set $\text{OriginalRange}$ computed on line 4 is eventually processed and divided into independent subgraphs by the while-loop. To prove the correctness of the algorithm, we still need to show that it correctly identifies an SCC when it sees it and that it never creates a sub-graph that is not independent, part of which was already shown.

**Lemma 5**

If the set $\text{OriginalRange}$ on line 7 is an independent sub-graph of the whole input graph then $\text{OriginalRange}$ is an SCC of the whole input graph if and only if, for an arbitrary vertex $v \in \text{OriginalRange}$, the forward closure of $v$ in $\text{OriginalRange}$ is equal to $\text{OriginalRange}$, OWCTY$(v), \text{OriginalRange}$ does not eliminate any vertex, and the backward closure of $v$ in $\text{OriginalRange}$ is equal to $\text{OriginalRange}$.

**Proof.** Forward implication. If $\text{OriginalRange}$ is an SCC, then for each pair of vertices $z, w \in \text{OriginalRange}$ there is a path from $z$ to $w$ in $\text{OriginalRange}$. The statements for the forward and the backward closures follow directly. There is a vertex $w \in \text{OriginalRange}$ such that there is a path from $w$ to $v$ in $\text{OriginalRange}$, so indegree$(v) > 0$, and so OWCTY() started from $v$ cannot eliminate any vertex.

Backward implication. For each pair of vertices $z, w \in \text{OriginalRange}$ there is a path from $z$ to $w$ in $\text{OriginalRange}$, which follows from the assumption about the forward and the backward closures. (There is a path from $z$ to $v$ and a path from $v$ to $w$). Since $\text{OriginalRange}$ is an independent sub-graph of the whole input graph (Lemma 3), $\text{OriginalRange}$ is an SCC of the whole input graph. ■

**Lemma 6**

Let $G = (V, E)$ be an arbitrary graph. For arbitrary vertex $v \in V$, the forward closure of $v$ in $V$, denoted by $A$, is an independent sub-graph of $G$.

**Proof.** Similar to the proof of Lemma 3. (There is no edge from $A$ to $V \setminus A$.) ■
Distributed Algorithms for SCC Decomposition

**Theorem 1**

The algorithm in Figure 7 correctly identifies all SCCs in the input graph.

**Proof.** In the while-loop on lines 2–24 the graph is correctly divided into independent sub-graphs by repeated application of lines 3 and 4 (Lemma 6). The sub-graphs that are SCCs are correctly identified by Lemma 5. The sub-graphs that are not SCCs are divided into smaller independent sub-graphs by Lemmas 1–4. To these smaller sub-graphs, the procedure is applied recursively. The only case when the recursive application is not executed is the case when \( B = \text{OriginalRange} \), which can happen only in the first iteration of the while-loop on lines 8–23. This is exactly the case when \( \text{OriginalRange} \) is one SCC, again by Lemma 5. The rest follows from the fact that the relation ‘being an independent subgraph of’ is transitive.

**Lemma 7**

The overall time complexity of Recursive OBF is \( \mathcal{O}(r+1) \cdot (m+n) \), where \( r \) is the maximal depth of recursion (\( r = 0 \) if no recursive calls are executed).

**Proof.** Two distinct OBFR procedures on the same depth of recursion operate on disjoint parts of the graph, so at most \( \mathcal{O}(m+n) \) work is done for each recursion depth. Thus the overall complexity is \( \mathcal{O}((r+1) \cdot (m+n)) \).

**Theorem 2**

The depth of recursion of Recursive OBF is at most \( L \) (the length of the longest path in the quotient graph of the whole graph).

**Proof.** The proof proceeds by induction on \( L \).

Induction basis. If \( L = 0 \), then the whole graph is one SCC. This is detected on the recursion level zero, so the maximal depth of recursion is 0.

Induction step. It is sufficient to show that application of the procedure in Figure 7 (not counting recursive calls) to a graph with \( L=k > 0 \) divides it into sub-graphs with \( L \) at most \( k-1 \). There are two possible cases.

**Case 1**

The SCC of the vertex \( v \) selected on line 3 is not the first vertex of any of the longest paths in the quotient graph. Then, obviously, the forward closure of \( v \) is an independent sub-graph of the quotient graph of which does not contain paths longer than \( k-1 \). The same goes for all independent sub-graphs into which it might be further divided in the while-loop on lines 8–23.

**Case 2**

The SCC of the vertex \( v \) selected on line 3 is the first vertex of one of the longest paths in the quotient graph. Then at least one of the longest paths is in the quotient graph of the forward closure of \( v \). The important point is that all longest paths in the quotient graph of the forward closure must have the SCC of \( v \) as their first vertex. (The path not containing the SCC of \( v \) can be extended, because the SCC of \( v \) is a leading SCC). If the SCC of \( v \) is trivial, it is eliminated by OWCTY. If it is non-trivial, it is equal to the first OBF slice. In both cases, the SCC of \( v \) is removed in the first iteration of the while-loop on lines 8–23. Which leaves us with a graph with \( L \) less than \( k \). The rest follows easily.

**Corollary 1**

The overall time complexity of Recursive OBF is \( \mathcal{O}((L+1) \cdot (m+n)) \).
The upper bound cannot be tightened as shown by the following example. Define $G_k = (V_k, E_k)$ as follows. Let

$$
V'_0 = \{0\}
$$

$$
V'_{i+1} = V'_i \cup \{2i+1, 2i+2\}
$$

$$
E'_0 = \{(0, 0)\}
$$

$$
E'_{i+1} = E'_i \cup \{(2i+1, 2i+1), (2i+2, 2i+2), (\max(2i-1, 0), 2i+1), (2i+2, 2i), (2i+2, 2i+1)\}
$$

for $i \in \{0, \ldots, k\}$. Now

$$
V_k = V'_k \cup \{2k+1\}
$$

$$
E_k = E'_k \cup \{(2k+1, 2k+1), (\max(2k-1, 0), 2k+1)\}
$$

Figure 8 shows $G_2$. Note that $G_k$ has $2k+2$ vertices and $5k+3$ edges. One possible behaviour of OBFR on $G_k$ is as follows. Suppose OBFR picks the vertex $2k$ first. All vertices of $G_k$ are reachable from $2k$ so the first rooted chunk is the whole graph. OBFR is then run on this rooted chunk. No vertex is eliminated by OWCTY(), for $2k$ has a predecessor (itself). The first OBF slice is then $\{2k\}$ which is identified as an SCC by subsequent recursive call to OBFR. The first OBF then continues on successors of $\{2k\}$, these are $2k-2$ and $2k-1$. Again, OWCTY() does not eliminate anything. Then a backward reachability is started from $\{2k-2, 2k-1\}$ and explores the whole remaining graph except for the vertex $2k+1$. So, the second OBF slice is equal to the graph $G_{k-1}$ and OBFR is called recursively to process it.

We have shown that maximal recursion depth of OBFR on $G_k$ is $k+1$. At recursion depth $i$, a graph with at least $2(2k-i)+2$ vertices and at least $5(2k-i)+3$ edges is explored at least once. So by Corollary 1, the overall time complexity of OBFR on $G_k$ is $\Omega(n \cdot (n+m))$.

4.1 Increasing the degree of parallelism

In [4], it was noticed that OBF has a better worst-case running time than CH, mainly due to possible re-colouring. Still, our initial experiments (cf. Figure 11) showed that CH performs better on graphs with many small SCCs. We attribute this to the higher degree of parallelism in CH, which outweighs the extra costs due to re-colouring in this case.

There is room to increase parallelism in OBFR-P() too. The pseudocode of this ‘more parallel’ version is in Figure 9. It exploits the fact that, after we pick a vertex in $V$ and identify its forward closure $Range$ in $V$, we can run OBF on $Range$ in parallel and without waiting for its completion we can pick another vertex from $V$ and start computing its closure.

So we essentially have three versions of OBFR varying in the ‘degree of parallelism’. This is illustrated in Figure 10. Each diagram starts with a bold vertical axis, where the downward direction represents the progression of time. The numbered columns represent independent parallel procedures.
Distributed Algorithms for SCC Decomposition

```plaintext
proc OBFR-MP(V)
  while (V ≠ ∅) do
    Pick a vertex v ∈ V
    Range := FWD(v, V)
    Seeds := {v}
    V := V \ Range
    in parallel do
      OBFR-MPX(Seeds, Range)
    od
  end

proc OBFR-MPX(Seeds, Range)
  OriginalRange := Range
  while Range ≠ ∅ do
    Eliminated, Reached, Range := OWCTY(Seeds, Range)
    All elements of Eliminated are trivial SCCs
    B := BWD(Reached, Range)
    if (B = OriginalRange) then
      B is SCC
    else
      in parallel do
        OBFR-PMX(B)
      od
      Seeds := FWD-SEEDS(B, Range)
    fi
    Range := Range \ B
  od
end
```

Figure 9. OBFR with increased parallelism

An arrow from column $i$ to column $j$ indicates that procedure $i$ starts procedure $j$. For simplicity, the figure does not show recursive calls of OBFR.

Assume we have a graph whose vertices are partitioned into the following disjoint sets according to how OBFR works on the graph:

$V = B_{11} ∪ B_{12} ∪ B_{13} ∪ B_{21} ∪ B_{31} ∪ B_{32}$. $B_{1(1−3)} = B_{11} ∪ B_{12} ∪ B_{13}$ is the closure (Range) of the first picked vertex (first rooted chunk) and the individual sets are the slices identified by OBF in the closure. Similarly $B_{2(1)} = B_{21}$ is the closure of the second picked vertex (second rooted chunk) and $B_{3(1−2)} = B_{31} ∪ B_{32}$ is the closure of the third picked vertex (third rooted chunk). For simplicity, we assume there are no trivial components eliminated by OWCTY.

The leftmost diagram in Figure 10 illustrates the operation of the basic OBFR when no parallel procedures are executed. SCCs are processed one by one (delete lines 17 and 19 from Figure 7).

The middle diagram in Figure 10 illustrates the operation of OBFR in Figure 7. Each time a new slice is identified by OBF, a new parallel procedure is started to process the slice. The algorithm first picks a vertex, identifies the set $B_{1(1−3)}$, then the slices $B_{11}$, $B_{12}$ and $B_{13}$. Only then it can pick another vertex from the unexplored part of the graph, identify $B_{2(1)}$, ...

The rightmost diagram in Figure 10 illustrates the operation of the ‘more parallel’ OBFR in Figure 9. It does slicing of $B_{1(1−3)}$, $B_{2(1)}$ and $B_{3(1−2)}$ in separate parallel procedures. This allows it to get to $B_{2(1)}$ and $B_{3(1−2)}$ much faster.
5 Experimental evaluation

The experiments were carried out on a cluster of eight workstations interconnected with 1 Gbps Ethernet. Each workstation was equipped with AMD Athlon\textsuperscript{TM} 64 3500+ Processor and 1 GB RAM. We used the LAM/MPI library for message passing. Our implementation is a distributed memory one. The graph is partitioned into a number (in our case 8) of disjoint parts. Each workstation owns one part. Each workstation runs the same code and communicates with other workstations via the message passing library only. The computation at each workstation proceeds sequentially (the execution of independent parallel procedures is serialized) meaning that no additional threads are executed. This is achieved by maintaining an appropriate piece of information about each procedure in an ‘array of procedures’ and iterating over its elements repeatedly to let each procedure perform some work. Note that a single procedure runs in parallel over different partitions of the graph.

We observed that OBFR suffers from the amount of synchronization points among individual procedures. However, the amount of synchronization points may be significantly reduced if independent procedures are started as soon as all data they depend on are ready. Starting independent procedures can be viewed as an implementation detail, however, it has proven to have significant impact on the performance. The three different versions presented in the previous section are recapitulated in the following.

- **OBFR-S**: No procedures are executed in parallel. When OBFR identifies a slice it waits for the complete computation on the slice to finish before continuing.
- **OBFR-P**: OBFR identifies the slices, and starts a parallel procedure on each slice as soon as the slice is identified.
- **OBFR-MP**: Does the same as the previous one, but additionally, within a slice, it starts a parallel procedure as soon as a new forward chunk (forward closure of a picked vertex in a possibly not-rooted slice) within a slice is found.

Our experiments show that indeed the total running time of the algorithm decreases by adding more parallelism, despite the extra overhead (e.g. running various termination detection procedures in parallel), and despite the fact that a single reachability computation is already parallel.
We compare OBFR with three other algorithms. Namely FB [17], OBF + FB [4] and CH (colouring [22]). Like OBFR, FB and OBF + FB can be implemented with different degrees of parallelism. For the comparisons we implemented only the most parallel versions of these algorithms, which give the best results. These implementations are denoted by FB-P and OBF-FB-P. CH processes SCCs inherently in parallel; we reused the code from [22] and all experiments are carried out in the same software/hardware environment.

5.1 Measurements

For the evaluation, we used synthetic graphs with a regular structure and fixed size SCCs. The aim was to find out how the algorithms work as the SCC size changes. We used two types of graphs. The first type of graph, called \( LmLmTn \) was of the form \( \text{Loop}(m) \parallel \text{Loop}(m) \parallel \text{Tree}(n) \), where \( \text{Loop}(m) \) is a cycle with \( m \) states, \( \text{Tree}(n) \) is the binary tree of depth \( n \) and \( \parallel \) denotes the Cartesian product of graphs. This graph has \( 2^n + 1 \) components of size \( (m + 1)^2 \). Its quotient graph is a binary tree.

The second type of graph, called \( \text{LimLon} \), uses \( \text{Line}(m) \), being a sequence of \( m \) states. It is of the form \( \text{Line}(m) \parallel \text{Line}(m) \parallel \text{Loop}(n) \parallel \text{Loop}(n) \) and consequently has \( m^2 \) components of size \( n^2 \). The quotient graph of the second type is a square mesh with edges oriented right and down. In the second type, there are many paths of the same length to the same vertex.

We also experimented with graphs that arise as state spaces in real model checking applications. The names of these graphs are prefixed with ‘cwi’, ‘vasy’ and ‘swp’. The former two are taken from the VLTS Benchmark Suite [7]. The swp-graph, called \( \text{swp}_{dnwq}p \), models the behaviour of a sliding window protocol with \( m \) distinct data elements, window size \( 2n \), and queue size \( p \). The complete list is in Tables 1 and 2.

The size of the graphs is relatively small and in principle they could be decomposed on a single machine, but they are large enough for experiments with distributed algorithms to provide insight.

The results for synthetic graphs are in Table 3. The results for real graphs are in Table 4. All run-times are in seconds, ‘n/a’ means that the run-time exceeded 36 000 s (10 h). Graphs of dependency of run-time on SCC size are in Figure 11 and 12. We measured this dependency for synthetic graphs only. Figure 11 does not contain results for all graphs of type 1 since numbers of vertices of some of these graphs differ too much. Only graphs with \( \sim 3000000 \) vertices were chosen. The graphs of type 2 have all approximately \( \sim 4000000 \) vertices, so Figure 12 contains results for all of them.

5.2 Evaluation

There is one important issue concerning space complexity. To implement a reachability analysis in linear time, we need a way to determine whether a vertex has been already visited or not in constant time. This is usually accomplished by allocating an array of booleans with \( n \) elements, one for each vertex. Algorithms that perform many reachabilities in parallel must have such an array for each of them. Our implementations that fall into this category are FB-P, OBF-FB-P, OBFR-P, OBFR-MP. There is no problem with reachabilities in the same depth of recursion. Since they operate on disjoint parts of the graph, one array of size \( n \) is enough. But for procedures in different depths we need separate arrays. And so the space complexity is \( O(m + n \cdot (\text{maximum depth of recursion})) \).

Although the maximum depth of recursion can be as high as \( n \), in our experiments the algorithm we are mainly interested in, OBFR, reached maximum depth of 15. This makes us believe that

---

1Note that we consider the graph of all transitions, while [22] considered only (invisible) \( \tau \)-transitions.
space complexity is not a problem of OBFR. However, the FB algorithm exceeded depth 200 in our experiments. It did not prevent the algorithm from successful computation of SCCs, because our graphs are relatively small. Nevertheless, this high-recursion depth kills the benefit of having accumulated memory of a cluster of workstations. If we add that FB is much slower if independent sub-graphs are not processed in parallel, we can conclude that FB is not a very good distributed algorithm. On the other hand, OBF + FB reached maximum recursion depth of 17. It seems that the
uppermost OBF is so successful in slicing the whole graph, that the amount of work left for FB that processes the slices is relatively small.

And now for some comments on the measured run-times. First, for the synthetic graphs. As one can see from Table 3, OBFR-MP and OBF-FB-P together are clear winners. Their run-times are practically the same because most of the decomposition was done by the first OBF which is the same
for both algorithms. The slices identified by the OBFR were then processed in parallel. It did not matter if OBF or FB was used for them because of the structure of the slices.

FB, OBFR-S and OBFR-P worked quite well on graphs with large SCCs, but they require a long time to decompose a graph with many small components. OBFR-P was the best of them, but its performance on graphs with many small components is still poor. The reason for the big difference between OBFR-P and OBFR-MP is that some slices identified by the first OBF contained many parts with no edges between them and waiting for OBF to finish on one part before moving to next part affects the performance considerably.
Distributed Algorithms for SCC Decomposition

Interestingly enough, for the synthetic graphs of type 1, unlike most of the other algorithms, especially OBFR-S, the CH algorithm worked better on graphs with many small components (Figure 13). We attribute this to the high degree parallelism in CH which outweighs the extra costs due to re-colouring in this case. However, it was not confirmed on type 2 graphs (Figure 12), on which CH has extremely poor performance. This is explained by many paths of the same length leading to the same vertex, which causes frequent re-colouring.

The experiments on real graphs (Table 4) have only one winner, OBFR-MP. Yet, its victory was not as clear as the victory for synthetic graphs. In particular, CH turned out to be successful. We included total run-times for all real graphs to allow for better comparison.

The structure of the graphs was not regular, so OBFR had to go deeper to decompose the graph. Since the decomposition was not done by the first OBF, the FB algorithm had much more work in OBF + FB than for synthetic graphs, which resulted in poor behaviour for some graphs, especially vasy_12323_27667 and vasy_4220_13944.

6 Conclusion

In this article, we listed and compared known distributed algorithms for the decomposition of directed graphs into their SCCs. We also proposed a new algorithm, called OBFR, based on recursive application of the OBF technique introduced in [4]. The correctness of the new algorithm was proven formally. We also report on an extensive experimental study we did to evaluate the new algorithm. OBFR outperformed all the other algorithms in most cases.

Our experiments show that the way the algorithm is implemented influences its performance a great deal. In particular, the best implementation turned out to be the one with the highest degree of parallelism, that is the one which starts another parallel procedure every time a part of the graph that can be processed independently has been identified.

There is one type of graphs where the CH algorithm [22] may be the best choice. These are graphs consisting of many unconnected islands. Such graphs arise for instance when considering...
only (invisible) \( \tau \)-transitions as a pre-processing step to branching bisimulation reduction. CH starts working on all islands simultaneously, but all the other algorithms process them one by one unless they contain indegree 0 vertices. If these islands are small enough, re-colouring is not a problem and CH is very fast. This suggests an aim for future work: to improve OBFR to work better on graphs with many unconnected islands.

OBFR is also suitable for multi-core shared-memory architectures that are going to be the standard in the near future. Implementing and evaluating OBFR on such architectures is another aim for future work.

Acknowledgement

We like to thank Simona Orzan for discussions on the CH-algorithm and sharing her implementation. We are grateful to the Centrum voor Wiskunde en Informatica (Amsterdam) for hosting a considerable part of the research reported here.

Funding

Czech Science Foundation (grant No. 201/09/1389); Academy of Sciences (grant No. IET408050503); EU FP6 NEST pathfinder project EC-MOAN (043235).

References

Distributed Algorithms for SCC Decomposition


Received 29 February 2008