Improved Distributed Algorithms
for SCC Decomposition

Jiří Barnat

Faculty of Informatics, Masaryk University,
Brno, Czech Republic
barnat@fi.muni.cz

Jakub Chaloupka

Faculty of Informatics, Masaryk University,
Brno, Czech Republic
xchalou@fi.muni.cz

Jaco van de Pol

Centrum voor Wiskunde en Informatica,
Amsterdam, The Netherlands
Jaco.van.de.Pol@cwi.nl

Abstract
We study and improve the OBF technique [1], which was used in distributed algorithms for the decomposition of a partitioned graph into its strongly connected components. In particular, we introduce a recursive variant of OBF and experimentally evaluate several different implementations of it that vary in the degree of parallelism. For the evaluation we used synthetic graphs with a few large components and graphs with many small components. We also experimented with graphs that arise as state spaces in real model checking applications. The experimental results are compared with that of other successful SCC decomposition techniques [6,5].

1 Introduction

Decomposing a directed graph into its strongly connected components is one of the basic graph problems. It has many applications, among others in analysis of computer systems. It can be solved in linear time. Standard algorithms

---

1 This work has been partially supported by the Grant Agency of Czech Republic grant No. 201/06/1338 and the Academy of Sciences grant No. 1ET14080503.

This is a preliminary version. The final version will be published in Electronic Notes in Theoretical Computer Science URL: www.elsevier.nl/locate/entcs
for SCC decomposition are Tarjan’s algorithm [7] and Kosaraju’s algorithm, also known as Double DFS [3]. However, graphs modelling complex computer systems tend to be very big 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. Unfortunately, all known linear sequential algorithms are based on depth first search (DFS) and no efficient parallel implementation of DFS is known, which renders the sequential algorithms unusable in a distributed setting. Different approaches must be used to design a good distributed algorithm.

Several distributed algorithms for SCC decomposition have already been proposed. They all exploit the fact that we can efficiently compute the set of vertices reachable from a certain vertex or set of vertices. The first distributed algorithm [5], known as FB, is based on the simple observation that the SCC to which a given vertex \( v \) belongs, is the intersection of the set of vertices reachable from \( v \) and the set of vertices reachable from \( v \) in the transposed graph.

Other algorithms [6,1] are more involved but their basic building block is still reachability analysis in the original or the transposed graph.

We focus on the OBF technique introduced in [1]. OBF is essentially a procedure that divides a rooted graph into independent sub-graphs, possibly eliminating some trivial SCCs in the process. No SCC crosses a boundary of an independent sub-graph so we can use whichever algorithm we like to decompose the sub-graphs. The original paper used the FB algorithm for the sub-graphs. We improve the OBF technique so that it can be applied recursively.

Having a number of independent sub-graphs we can run SCC decomposition on them in parallel, thus increasing the degree of parallelism. Note that in a distributed environment, a single reachability analysis itself runs in parallel already. It is not clear a priori whether the gain of decomposing all sub-graphs in parallel outweighs the overhead and complexity, compared to decomposing the sub-graphs one by one, as was done in [1].

The rest of the paper is organised as follows. Necessary definitions from graph theory and existing distributed algorithms are presented in Section 2. The new algorithm based on recursive application of OBF is described in Section 3. Results of experiments are in Section 4. In particular, we compare our new algorithm with the algorithms from [6,5,1], and we measure the effect of decomposing sub-graphs one by one, or in parallel. Contributions of the paper are summarised and future work is outlined in Section 5.

Acknowledgement.

We like to thank Simona Orzan for discussions on the CH-algorithm and sharing its implementation.
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 edges. If $uEv$, 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 edges having $v$ as endpoint, i.e., the number of elements in the set \{ $u \mid (u, v) \in E$\}. $G^T = (V, E^T)$, the transposed graph of $G = (V, E)$, is the graph $G$ with all edges reversed, i.e., $E^T = \{(u, v) \mid (v, u) \in E\}$.

A path is a sequence of vertices $s_0, \ldots, s_k$, s.t. $s_iEs_{i+1}$ for all $0 \leq i < k$; a simple path is one that contains no duplicated vertices. The length of this path is $k$, the number of edges. We write $sE^t$ if there is a path starting in $s$ and ending in $t$. A graph is rooted if there is an initial vertex $s_0 \in V$ such that $s_0E^t$ for all $t \in V$. Given a graph $G$, we use $n, m$ and $l$, to denote the number of vertices, edges, and the longest simple path between any two vertices in $G$, respectively.

A sub-graph $W \subseteq V$ is strongly connected if $sE^t$ and $tE^s$ in $W$ for all $s, t \in W$. A strongly connected component (SCC) is a maximal strongly connected sub-graph. The quotient graph of $G = (V, E)$ has the SCCs of $G$ as vertices. It has an edge between $X$ and $Y$, iff for some $x \in X$ and $y \in Y, xEy$. Note that by definition of SCCs, the quotient graph cannot contain cycles. An SCC is non-trivial if it contains at least one edge. 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 (also named independent sub-graph). Given a graph $G$, we denote by $N, M$ and $L$, the number of vertices, edges and the length of the longest (simple) path in the quotient graph of $G$, respectively.

For $v \in W \subseteq V$, the forward closure of $v$ in $W$ is the set $S = \{ s \in W \mid vE^*_ws \}$, where $E^*_W = \{(x, y) \mid (x, y) \in E \wedge x, y \in W\}$. 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 in $S$ 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 $G^T$.

Reachability analysis is a procedure that computes a (forward/backward) closure of a vertex or set. We often use the word search instead of reachability analysis. So, if we say that forward search restricted to $W \subseteq V$ is started from vertex $v$ or set $S$, we mean that forward closure of $v$ or $S$ in $W$ is about to be computed.

2.2 Existing techniques for distributed SCC decomposition

Next we briefly describe existing distributed algorithms for SCC decomposition from [5,1,6].

All algorithms below have an optimisation routine in common, which removes all initial trivial SCCs (called elim-atomic in [6], OWCTY [4] in [1]).
Fig. 1. OBF slice identification.

It repeatedly removes vertices whose indegree is (or becomes) zero, because these must be trivial components. Note that this procedure cannot detect trivial components that lie in between strongly connected components. This procedure is implemented as a distributed forward reachability on a part of the graph.

2.2.1 FB
This is the basic algorithm from [5]. It picks a pivot vertex $p$, computes its forward and backward closures $F$ and $B$ using distributed reachability. Note that $F \cap B$ is the SCC of $p$. Note that all other SCCs lie completely within one of the following subsets: $F \setminus B$, $B \setminus F$ and $V \setminus (B \cup F)$. Hence, the algorithm continues by recursively applying FB to these three independent subgraphs.

2.2.2 OBF (OWCTY-BWD-FWD) [1]
This algorithm identifies a number of slices in linear time, in such a way that each non-trivial component lies completely within one of the slices. On each slice, the algorithm FB is applied. Identifying the slices is done by repeating the following steps, starting from the initial vertex:

- **O** Remove vertices without predecessors using OWCTY [4] (these are trivial SCCs)
- **B** compute the backward closure $B$ on vertices that are reached in step 1, but not eliminated; this defines a slice $B$.
- **F** Remove slice $B$, to be processed separately, and start the next iteration from the successor vertices of $B$.

The procedure is illustrated in Figure 1.
2.2.3 Colouring/Heads-off (CH) [6]
This algorithm 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 eventually all edges are increasing. Note that a vertex can be recoloured several times. Also note that after colouring, all vertices in a single SCC have the same colour. So all edges between vertices of different colours can be removed.

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

3 Recursive OBF

As shown in [1], OBF performs better than FB in a number of experiments. Note that in OBF the graph is split in slices in linear time. On each slice, algorithm FB is applied. But, as OBF is better than FB, we now propose to 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. This is described in detail in the Pseudocode subsection.

3.1 Pseudocode of Recursive OBF

The pseudocode of Recursive OBF is in Figure 2. We start with the whole graph. Vertices in recognised 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 don’t 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 $V \setminus Range$ will be processed in the next iterations of the main while-loop (lines 2–23).
proc OBFR-P(V)
while (V ≠ ∅) do
  Pick a vertex v ∈ V
  Range := FWD(v, V)
  Seeds := {v}
  V := V \ Range
  OriginalRange := Range
  while Range ≠ ∅ do
    [Invariant: Forward closure of Seeds in Range = Range]
    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-P(B)
      od
    Seeds := FWD-SEEDS(B, Range)
  od
  Range := Range \ B
od end

Fig. 2. Recursive OBF

Before OBF is started on Range, Range is saved into OriginalRange, it will enable us to determine if a slice found by OBF is an SCC. On line 9 there is an invariant “(Forward closure of Seeds in Range) = Range”. In the first iteration of while-loop on lines 8–22 the invariant holds trivially, because Seeds contains just one vertex and Range was computed as a forward closure of that vertex. Procedure OWCTY() eliminates leading trivial components by repeatedly removing indegree 0 vertices reachable from Seeds. Eliminated vertices are returned as the set Eliminated, OWCTY() also removes eliminated vertices from Range. Vertices at which OWCTY() stops (they have positive indegree) are returned as the set Reached. The forward closure of Reached in Range = Range, since any path that leads from Seeds to a 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 8–22) and so \( B \) is indeed an SCC. Consequently, \( \text{Range} \setminus B \) is empty set and the while-loop finishes.

If \( B \neq \text{OriginalRange} \) we run OBFR-P() on \( B \) recursively. Moreover, note that the nested procedure can be run in parallel, which increases parallelism. \textit{Seeds} for the next iteration of the while-loop 8–22 are computed by the procedure FWD-SEEDS, which simply returns all vertices from \( \text{Range} \) that are immediate successors of vertices in \( B \) but not in \( B \). Since all paths that reach vertices in \( \text{Range} \setminus B \) from \( B \) must contain some vertex from \textit{Seeds}, after we subtract \( B \) from \( \text{Range} \), the invariant of line 9 is satisfied. When \( \text{Range} = \emptyset \), the while-loop 8–22 finishes and we handle the remaining vertices in \( V \).

**Theorem 3.1** The overall time complexity of Recursive OBF is \( \mathcal{O}(L(m+n)) \).

**Proof.** Recursive OBF (OBFR for short) first partitions the graph in a set of rooted chunks. Paths in the quotient graphs of the chunks can be no longer than \( L \) (the length of the longest path in the quotient graph of the whole graph). Since the chunks are rooted, they contain only one leading SCC, the SCC the root belongs to. The longest path in the quotient graph must contain the root (otherwise it could be extended). If the leading SCC is trivial, it is eliminated by OWCTY. If it is non-trivial, it is equal to the first OBF slice. In both cases, recursive calls to OBFR are invoked on graphs with strictly smaller \( L \). So the depth of recursion is at most \( L \). Since two distinct OBFR procedures on the same depth of recursion operate on disjoint parts of the graph, at most \( \mathcal{O}(m+n) \) work is done for each recursion depth. Thus the overall complexity is \( \mathcal{O}(L(m+n)) \). \( \blacksquare \)

3.2 Increasing the degree of parallelism

In [1] it was noticed that OBF has a better worst-case running time than CH, mainly due to possible recolouring. Still, our initial experiments (cf. Figure 5) 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 recolouring in this case.

There is room to increase parallelism in OBFR-P() too. The pseudocode of this “more parallel” version is in Figure 4. It exploits the fact that, after we pick a vertex in \( V \) and identify its forward closure \( \text{Range} \) in \( V \), we can run OBF on \( \text{Range} \) 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 Recursive OBF varying in the “degree of parallelism”. This is illustrated in Figure 3. Each diagram starts with a bold vertical axis, where the downward direction represents the progression of time. The numbered columns represent independent parallel procedures. 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 OBF.
Assume we have a graph whose vertices are partitioned into the following disjoint sets according to how Recursive OBF works on the graph: $V = B_{11} \cup B_{12} \cup B_{13} \cup B_{21} \cup B_{31} \cup B_{32}$. $B_{1(1-3)} = B_{11} \cup B_{12} \cup B_{13}$ 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 $B_{3(1-2)} = B_{31} \cup B_{32}$ 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 3 illustrates operation of the basic Recursive OBF when no parallel procedures are executed. SCCs are processed one by one (delete lines 16 and 18 from Figure 2).

The middle diagram in Figure 3 illustrates operation of Recursive OBF in Figure 2. 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 3 illustrates operation of the “more parallel” Recursive OBF in Figure 4. 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.

4 Experimental Evaluation

The experiments were carried out on a cluster of 8 workstations interconnected with 1 Gbps Ethernet. Each workstation was equipped with AMD Athlon™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 workstations 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 paral-
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
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-MP(B)
            od
            Seeds := FWD-SEEDS(B, Range)
        fi
        Range := Range \ B
    od
end

Fig. 4. Recursive OBFR with increased parallelism

Recursive OBFR with increased parallelism (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 Recursive 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 Recursive OBFR with three other algorithms. Namely FB [5], OBFR + FB [1] and CH (Colouring [6]). Like Recursive OBFR, FB and OBFR + 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 OBFR-FB-P. CH processes SCCs inherently in parallel; we reused the code from [6] and all experiments are carried out in the same software/hardware environment.

4.1 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 \textit{LmLMn} was of the form \textit{Loop(m) \textbar Loop(m) \textbar Tree(n)}, where \textit{Loop(m)} is a cycle with \textit{m} states, \textit{Tree(n)} is the binary tree of depth \textit{n}, and \textbar\textbar denotes the Cartesian product of graphs. This graph has \(2^{n+1} - 1\) components of size \((m + 1)^2\). Its quotient graph is a binary tree.

The second type of graph, called \textit{LmLM1n}, uses \textit{Line(m)}, being a sequence of \textit{m} states. It is of the form \textit{Line(m) \textbar Line(m) \textbar Loop(n) \textbar 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 VLT Benchmark Suite [2].\footnote{Note that we consider the graph of all transitions, while [6] considered only (invisible) \(\tau\)-transitions.} The \textit{swp}-graph, called \textit{swp-LmSWPn}, models the behaviour of a sliding window protocol with \textit{m} distinct data elements, window size \(2n\), and queue size \textit{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 runtimes are in seconds, "n/a" means that the runtime exceeded 36000 seconds (10 hours). Graphs of dependency of runtime on SCC size are in Figure 6 and in Figure 7. We measured this dependency for synthetic graphs only. Figure 6 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 approximately 3 000 000 vertices were chosen. The graphs of type 2 have all approximately 4 000 000 vertices, so Figure 7 contains results for all of them.

4.2 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, OB-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, Recursive OBFR, reached maximum depth of 15. This makes us believe that space complexity is not a
<table>
<thead>
<tr>
<th>State space</th>
<th>FB-P</th>
<th>OBFR-S</th>
<th>OBFR-P</th>
<th>OBFR-MP</th>
<th>OBF-FB-P</th>
<th>CH</th>
</tr>
</thead>
<tbody>
<tr>
<td>L10L10T10</td>
<td>10</td>
<td>128</td>
<td>25</td>
<td>8</td>
<td>8</td>
<td>75</td>
</tr>
<tr>
<td>L100L100T4</td>
<td>13</td>
<td>19</td>
<td>13</td>
<td>11</td>
<td>5</td>
<td>145</td>
</tr>
<tr>
<td>L15L15T10</td>
<td>16</td>
<td>118</td>
<td>56</td>
<td>16</td>
<td>17</td>
<td>142</td>
</tr>
<tr>
<td>L4L4T16</td>
<td>2743</td>
<td>6603</td>
<td>671</td>
<td>309</td>
<td>297</td>
<td>325</td>
</tr>
<tr>
<td>L20L20T12</td>
<td>224</td>
<td>575</td>
<td>287</td>
<td>74</td>
<td>71</td>
<td>456</td>
</tr>
<tr>
<td>L80L80T8</td>
<td>94</td>
<td>107</td>
<td>110</td>
<td>34</td>
<td>45</td>
<td>795</td>
</tr>
<tr>
<td>L350L350T4</td>
<td>83</td>
<td>91</td>
<td>88</td>
<td>38</td>
<td>45</td>
<td>1583</td>
</tr>
<tr>
<td>L750L750T0</td>
<td>34</td>
<td>31</td>
<td>43</td>
<td>17</td>
<td>16</td>
<td>1021</td>
</tr>
<tr>
<td>L750L750T1</td>
<td>148</td>
<td>138</td>
<td>166</td>
<td>87</td>
<td>82</td>
<td>6533</td>
</tr>
<tr>
<td>L200L010</td>
<td>1982</td>
<td>1964</td>
<td>1131</td>
<td>76</td>
<td>58</td>
<td>9317</td>
</tr>
<tr>
<td>L125L016</td>
<td>1105</td>
<td>975</td>
<td>740</td>
<td>61</td>
<td>52</td>
<td>5827</td>
</tr>
<tr>
<td>L100L020</td>
<td>754</td>
<td>588</td>
<td>520</td>
<td>65</td>
<td>51</td>
<td>4513</td>
</tr>
<tr>
<td>L80L025</td>
<td>548</td>
<td>465</td>
<td>454</td>
<td>57</td>
<td>77</td>
<td>3560</td>
</tr>
<tr>
<td>L67L030</td>
<td>510</td>
<td>356</td>
<td>484</td>
<td>58</td>
<td>44</td>
<td>3080</td>
</tr>
<tr>
<td>L50L040</td>
<td>357</td>
<td>236</td>
<td>163</td>
<td>48</td>
<td>48</td>
<td>3350</td>
</tr>
<tr>
<td>L40L050</td>
<td>286</td>
<td>175</td>
<td>126</td>
<td>50</td>
<td>43</td>
<td>2628</td>
</tr>
<tr>
<td>L30L067</td>
<td>174</td>
<td>127</td>
<td>110</td>
<td>43</td>
<td>44</td>
<td>2364</td>
</tr>
<tr>
<td>L25L080</td>
<td>140</td>
<td>102</td>
<td>103</td>
<td>46</td>
<td>46</td>
<td>2972</td>
</tr>
<tr>
<td>L20L0100</td>
<td>176</td>
<td>88</td>
<td>80</td>
<td>43</td>
<td>40</td>
<td>2782</td>
</tr>
<tr>
<td>L16L0125</td>
<td>106</td>
<td>77</td>
<td>115</td>
<td>71</td>
<td>38</td>
<td>2148</td>
</tr>
<tr>
<td>L10L0200</td>
<td>81</td>
<td>58</td>
<td>90</td>
<td>62</td>
<td>45</td>
<td>1895</td>
</tr>
</tbody>
</table>

Table 3
Runtimes for synthetic graphs (in seconds)

![Graph showing dependency of runtime on SCC size](image_url)

Fig. 6. Dependency of runtime on SCC size, type 1 synthetic graphs (logarithmic scale)
problem of Recursive OBF. 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 subgraphs

---

<table>
<thead>
<tr>
<th>State space</th>
<th>FB-P</th>
<th>OBFR-S</th>
<th>OBFR-P</th>
<th>OBFR-MP</th>
<th>OBF-FB-P</th>
<th>CH</th>
</tr>
</thead>
<tbody>
<tr>
<td>cwi_2165_8723</td>
<td>21</td>
<td>43</td>
<td>30</td>
<td>29</td>
<td>22</td>
<td>49</td>
</tr>
<tr>
<td>cwi_2416_17605</td>
<td>76</td>
<td>8791</td>
<td>942</td>
<td>51</td>
<td>56</td>
<td>126</td>
</tr>
<tr>
<td>cwi_7838_59101</td>
<td>65</td>
<td>58</td>
<td>107</td>
<td>102</td>
<td>72</td>
<td>227</td>
</tr>
<tr>
<td>vasy_11026_24660</td>
<td>3387</td>
<td>n/a</td>
<td>3391</td>
<td>416</td>
<td>827</td>
<td>471</td>
</tr>
<tr>
<td>vasy_1112_5290</td>
<td>168</td>
<td>5611</td>
<td>399</td>
<td>73</td>
<td>73</td>
<td>365</td>
</tr>
<tr>
<td>vasy_12323_27667</td>
<td>4483</td>
<td>n/a</td>
<td>3942</td>
<td>500</td>
<td>1016</td>
<td>509</td>
</tr>
<tr>
<td>vasy_2581_11442</td>
<td>169</td>
<td>6182</td>
<td>2084</td>
<td>64</td>
<td>109</td>
<td>276</td>
</tr>
<tr>
<td>vasy_4220_13944</td>
<td>531</td>
<td>8348</td>
<td>976</td>
<td>347</td>
<td>1987</td>
<td>151</td>
</tr>
<tr>
<td>vasy_4338_15666</td>
<td>209</td>
<td>14352</td>
<td>4445</td>
<td>107</td>
<td>110</td>
<td>310</td>
</tr>
<tr>
<td>vasy_6020_19353</td>
<td>60</td>
<td>147</td>
<td>93</td>
<td>51</td>
<td>34</td>
<td>130</td>
</tr>
<tr>
<td>vasy_6120_11031</td>
<td>888</td>
<td>26611</td>
<td>1483</td>
<td>282</td>
<td>299</td>
<td>592</td>
</tr>
<tr>
<td>vasy_8082_42933</td>
<td>162</td>
<td>440</td>
<td>640</td>
<td>455</td>
<td>407</td>
<td>280</td>
</tr>
<tr>
<td>swp_d2w2q2.s</td>
<td>12</td>
<td>9</td>
<td>12</td>
<td>16</td>
<td>6</td>
<td>44</td>
</tr>
<tr>
<td>swp_d2w2q3.s</td>
<td>55</td>
<td>13</td>
<td>28</td>
<td>55</td>
<td>18</td>
<td>102</td>
</tr>
<tr>
<td>swp_d3w2q2.s</td>
<td>38</td>
<td>16</td>
<td>42</td>
<td>35</td>
<td>15</td>
<td>70</td>
</tr>
<tr>
<td>Total runtime</td>
<td>10324</td>
<td>&gt;142621</td>
<td>18572</td>
<td>2583</td>
<td>5051</td>
<td>3702</td>
</tr>
</tbody>
</table>

Table 4
Runtimes for real graphs (in seconds)

Fig. 7. Dependency of runtime on SCC size, type 2 synthetic graphs (logarithmic scale)
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 runtimes. First for the synthetic graphs. As one can see from Table 3 OBFR-MP and OBFR-FB-P together are clear winners. Their runtimes 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 OBF 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.

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 5). We were unable to explain this behaviour. Moreover, it was not confirmed on type 2 graphs (Figure 7). Another interesting point is the extremely poor behaviour of CH on type 2 graphs. 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 runtimes for all real graphs to allow for better comparison.

The structure of the graphs was not regular, so recursive OBF 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.

5 Conclusion

We proposed a new algorithm for decomposition of directed graphs into their strongly connected components. We adopted the OBF technique introduced in [1] and improved it so that it can be applied recursively which resulted in an algorithm (Recursive OBF) that outperformed all the other algorithms in our experiments.

Our experiments show that the way the algorithm is implemented influ-
ences 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 has been identified that can be processed independently.

There is one type of graphs where Colouring [6] 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 preprocessing step to branching bisimulation reduction. Colouring 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 Colouring will be very fast.

The previous paragraph suggests aims for future work: To improve Recursive OBF to work better on graphs with many unconnected islands. We have reasons to believe that such an improvement is possible. More thorough experiments should also be carried out to confirm our appealing results of Recursive OBF.

References


