Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 198 (2008) 63–77
www.elsevier.com/locate/entcs

Improved Distributed Algorithms for SCC Decomposition[1]

Jiří Barnat

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

Jakub Chaloupka

Faculty of Informatics, Masaryk University, Brno, Czech Republic
xchalou1@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].

Keywords: SCC decomposition, parallel, OBF technique

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 for SCC decomposition are Tarjan's algorithm [7] and Kosaraju's algorithm, also known as

[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. 1ET408050503.

doi:10.1016/j.entcs.2008.02.001

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. $s_i E s_{i+1}$ 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 $s_0 E^* t$ 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 $S = \{s \in W \mid vE^*_W s\}$, 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]). 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

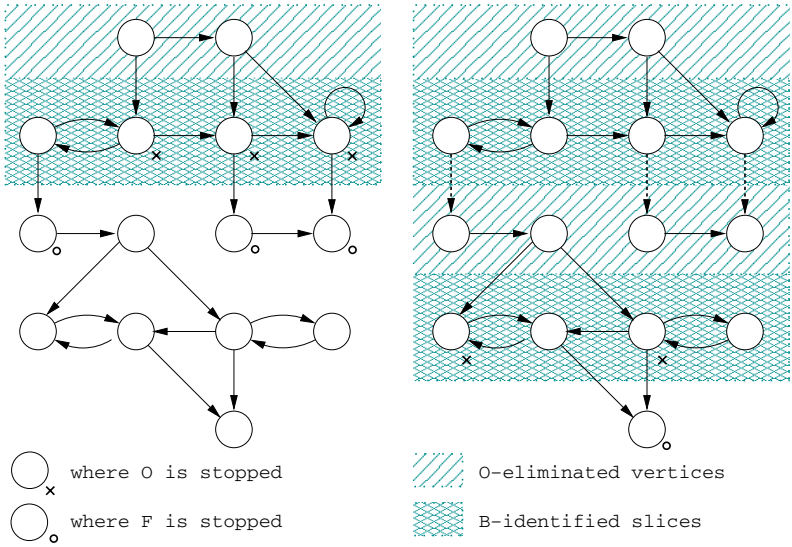

Fig. 1. OBF slice identification.

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).

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

```
1  proc OBFR-P(V)
2    while (V ≠ ∅) do
3      Pick a vertex v ∈ V
4      Range := FWD(v, V)
5      Seeds := {v}
6      V := V \ Range
7      OriginalRange := Range
8      while Range ≠ ∅ do
9        [Invariant: Forward closure of Seeds in Range = Range]
10       Eliminated, Reached, Range := OWCTY(Seeds, Range)
11       [All elements of Eliminated are trivial SCCs]
12       B := BWD(Reached, Range)
13       if (B = OriginalRange) then
14         B is SCC
15       else
16         in parallel do
17           OBFR-P(B)
18         od
19         Seeds := FWD-SEEDS(B, Range)
20       fi
21       Range := Range \ B
22     od
23   od
24 end
```

Fig. 2. Recursive OBF

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, $Range \setminus B$ is empty set and the while-loop finishes.

If $B \neq OriginalRange$ 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 8–22 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 $Range \setminus B$ 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 8–22 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 $\mathcal{O}(L.(m+n))$. □

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 *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 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

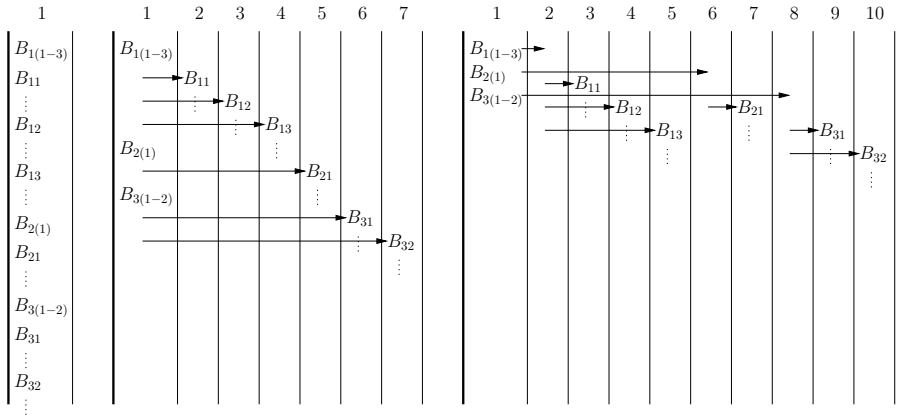

Fig. 3. Three versions of Recursive OBF different in degree of parallelism

(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^{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 Recursive OBF 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.

```
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 OBF with increased parallelism

OBFR-S	No procedures are executed in parallel. When OBF identifies a slice it waits for the complete computation on the slice to finish before continuing.
OBFR-P	OBF 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 OBF with three other algorithms. Namely FB [5], OBF

+ FB [1] and CH (Colouring [6]). Like Recursive OBF, 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 [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 LmLmTn was of the form $Loop(m)$ || $Loop(m)$ || $Tree(n)$, where $Loop(m)$ is a cycle with m states, $Tree(n)$ is the binary tree of depth n, and || 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 LimLon, uses $Line(m)$, being a sequence of m states. It is of the form $Line(m)$ || $Line(m)$ || $Loop(n)$ || $Loop(n)$ 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 [2] [2] The swp-graph, called swp_dmwnqp, 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 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

[2] Note that we consider the graph of *all* transitions, while [6] considered only (*invisible*) τ-transitions.

State space	N. of SCCs	Size of one SCC	States	Transitions
L10L10T10	2 047	121	247 687	742 940
L100L100T4	31	10 201	316 231	938 492
L15L15T10	2 047	256	524 032	1 571 840
L4L4T16	131 071	25	3 276 775	9 830 300
L20L20T12	8 191	441	3 612 231	10 836 252
L80L80T8	511	6 561	3 352 671	10 051 452
L350L350T4	31	123 201	3 819 231	11 334 492
L1750L1750T0	1	3 066 001	3 066 001	6 132 002
L1750L1750T1	3	3 066 001	9 198 003	24 528 008
Li200Lo10	40 000	100	4 000 000	15 960 000
Li125Lo16	15 625	256	4 000 000	15 936 000
Li100Lo20	10 000	400	4 000 000	15 920 000
Li80Lo25	6 400	625	4 000 000	15 900 000
Li67Lo30	4 489	900	4 040 100	16 039 800
Li50Lo40	2 500	1 600	4 000 000	15 840 000
Li40Lo50	1 600	2 500	4 000 000	15 800 000
Li30Lo67	900	4 489	4 040 100	15 891 060
Li25Lo80	625	6 400	4 000 000	15 680 000
Li20Lo100	400	10 000	4 000 000	15 600 000
Li16Lo125	256	15 625	4 000 000	15 500 000
Li10Lo200	100	40 000	4 000 000	15 200 000

Table 1
Synthetic graphs used in experiments

State space	N. of SCCs	Max. SCC size	States	Transitions
cwi_2165_8723	47 926	423 505	2 165 446	8 723 465
cwi_2416_17605	2 150 392	6	2 416 632	17 605 592
cwi_7838_59101	1	7 838 608	7 838 608	59 101 007
vasy_11026_24660	10 074 720	910	11 026 932	24 660 513
vasy_1112_5290	160 061	71 968	1 112 490	5 290 860
vasy_12323_27667	11 214 774	910	12 323 703	27 667 803
vasy_2581_11442	274 690	26 796	2 581 374	11 442 382
vasy_4220_13944	2 398 982	49 151	4 220 790	13 944 372
vasy_4338_15666	828 412	26 796	4 338 672	15 666 588
vasy_6020_19353	2 041	6 013 920	6 020 550	19 353 474
vasy_6120_11031	4 638 059	1 902	6 120 718	11 031 292
vasy_8082_42933	323 629	7 054 752	8 082 905	42 933 110
swp_d2w2q2.s	1	1 429 676	1 429 676	6 704 544
swp_d2w2q3.s	1	5 323 836	5 323 836	25 236 056
swp_d3w2q2.s	1	5 168 596	5 168 596	24 615 576

Table 2
Real graphs used in experiments

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 $O(m + n \cdot (maximum\ depth\ of\ recursion))$.

Although the maximum depth of recursion can be as high as n, in our experiments the algorithm we are mainly interested in, Recursive OBF, reached maximum depth of 15. This makes us believe that space complexity is not a 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 are not processed in parallel, we can con-

State space	FB-P	OBFR-S	OBFR-P	OBFR-MP	OBF-FB-P	CH
L10L10T10	10	128	25	8	8	75
L100L100T4	13	19	13	11	5	145
L15L15T10	16	118	56	16	17	142
L4L4T16	2743	6603	671	309	297	325
L20L20T12	224	575	287	74	71	456
L80L80T8	94	107	110	34	45	795
L350L350T4	83	91	88	38	45	1583
L1750L1750T0	34	31	43	17	16	1021
L1750L1750T1	148	138	166	87	82	6533
Li200Lo10	1982	1964	1131	76	58	9317
Li125Lo16	1105	975	740	61	52	5827
Li100Lo20	754	588	520	65	51	4513
Li80Lo25	548	465	454	57	77	3560
Li67Lo30	510	356	484	58	44	3080
Li50Lo40	357	236	163	48	48	3350
Li40Lo50	286	175	126	50	43	2628
Li30Lo67	174	127	110	43	44	2364
Li25Lo80	140	102	103	46	46	2972
Li20Lo100	176	88	80	43	40	2782
Li16Lo125	106	77	115	71	38	2148
Li10Lo200	81	58	90	62	45	1895

Table 3
Runtimes for synthetic graphs (in seconds)

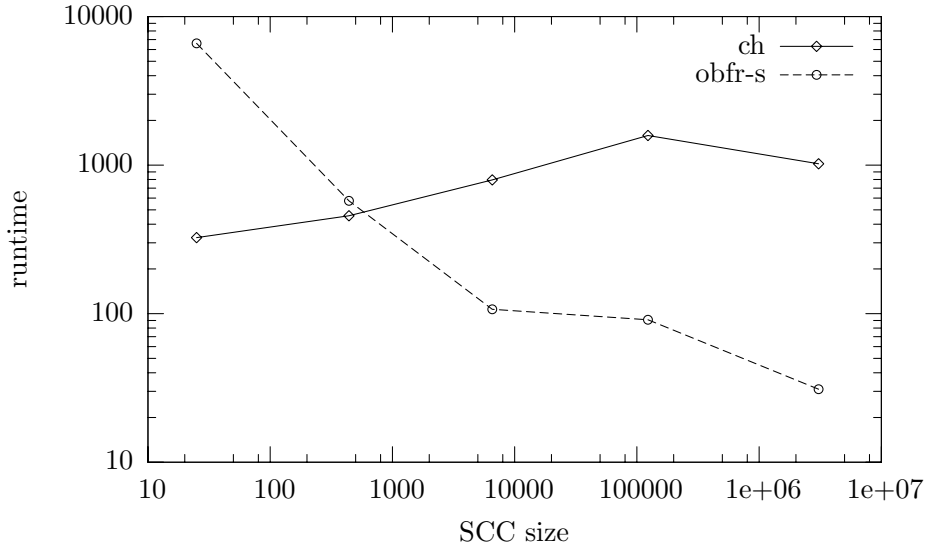

Fig. 5. Dependency of runtime on SCC size, comparison of OBFR-S and CH, type 1 synthetic graphs (logarithmic scale)

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

Table 4
Runtimes for real graphs (in seconds)

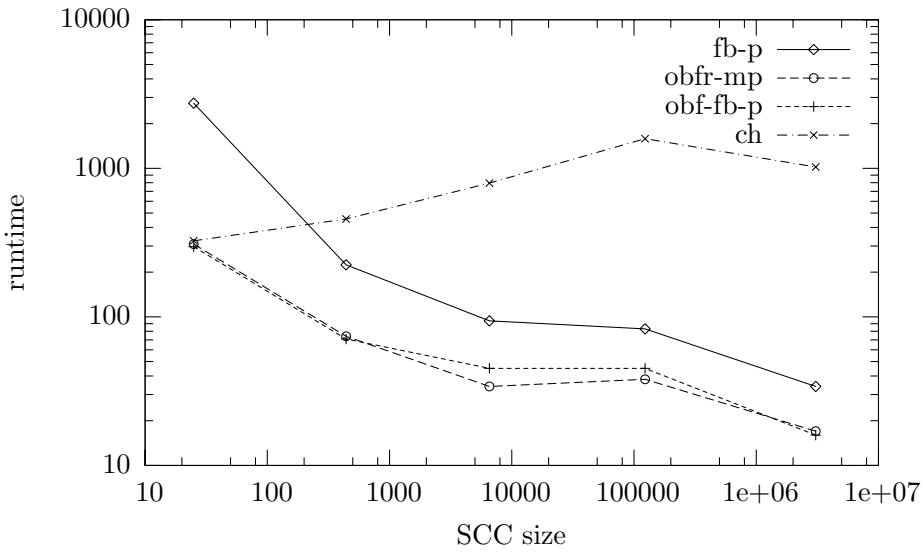

Fig. 6. Dependency of runtime on SCC size, type 1 synthetic graphs (logarithmic scale)

clude 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 OBF-FB-P together are clear

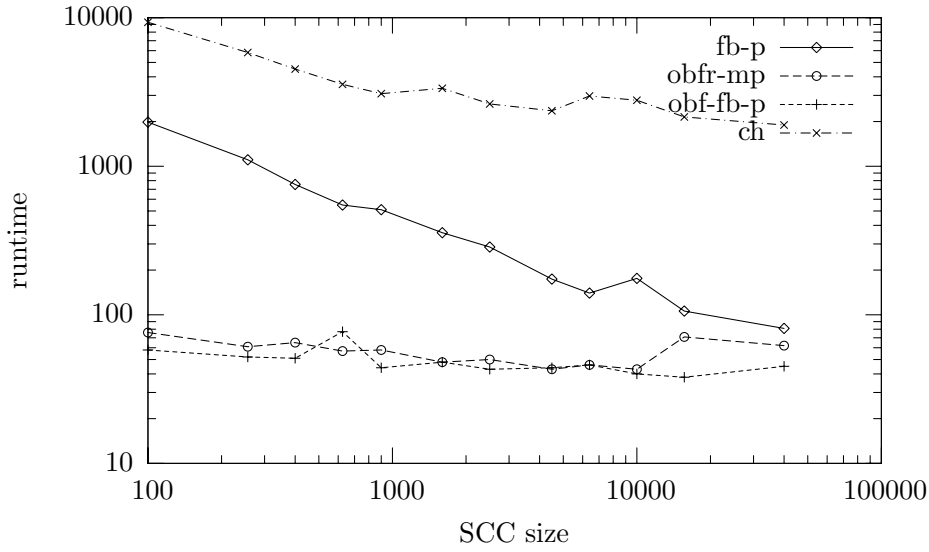

Fig. 7. Dependency of runtime on SCC size, type 2 synthetic graphs (logarithmic scale)

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 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 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) τ-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

[1] Barnat, J. and P.Moravec, *Parallel algorithms for finding SCCs in implicitly given graphs*, in: *Proceedings of the 5th International Workshop on Parallel and Distributed Methods in Verification (PDMC 2006)*, LNCS (2007).

[2] Blom, S. and H. Garavel, *The VLTS benchmark suite*, Available at http://www.inrialpes.fr/vasy/cadp/resources/benchmark_bcg.html (2003).

[3] Cormen, T. H., C. E. Leiserson and R. L. Rivest, "Introduction to Algorithms," MIT, 1990.

[4] Fisler, K., R. Fraer, G. Kamhi, M. Y. Vardi and Z. Yang, *Is there a best symbolic cycle-detection algorithm?*, in: *Proc. Tools and Algorithms for Construction and Analysis of Systems*, LNCS **2031** (2001), pp. 420–434.

[5] Fleischer, L. K., B. Hendrickson and A. Pinar, *On identifying strongly connected components in parallel*, in: *Parallel and Distributed Processing, IPDPS Workshops*, Lecture Notes in Computer Science **1800**, 2000, pp. 505–511.

[6] Orzan, S., "On Distributed Verification and Verified Distribution," Ph.D. thesis, Free University of Amsterdam (2004).

[7] Tarjan, R., *Depth first search and linear graph algorithms*, SIAM Journal on computing (1972), pp. 146–160.