Parallel Algorithms for Finding SCCs in Implicitly Given Graphs*

Jiří Barnat and Pavel Moravec

Department of Computer Science, Faculty of Informatics
Masaryk University Brno, Czech Republic

Abstract. We examine existing parallel algorithms for detection of strongly connected components and discuss their applicability to the case when the graph to be decomposed is given implicitly. In particular, we list individual techniques that parallel algorithms for SCC detection are assembled from and show how to assemble a new more efficient algorithm for solving the problem. In the paper we also report on a preliminary experimental study we did to evaluate the new algorithm.

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 [20] who showed that given a graph with n vertices and m edges, it is possible to identify and list all strongly connected components of the graph in $O(n+m)$ time and $O(n)$ space. Besides Tarjan's serial algorithm, several parallel algorithms have been designed to solve the problem. Tarjan's algorithm (and its variants) strongly rely on the depth-first search post-ordering of vertices whose computation is known to be P-complete [19], and thus, difficult to be parallelized. Therefore, parallel algorithms avoid the depth-first search of the graph and build on different approaches.

A parallel algorithm relying on matrix multiplication was described in [14] and further improved in [10, 1]. The algorithm works in $O(log^2 n)$ time in the worst case, however, to achieve the complexity it requires $O(n^{2.376})$ parallel processors. As graphs that we are typically dealing with in practice contain millions of vertices the algorithm is practically unusable and is only interesting from the theoretical point of view. Another parallel algorithm for finding SCCs was given in [12]. Its general idea is to repeatedly pick a vertex of the graph and identify the component the vertex belongs to using two parallel reachability procedures. The algorithm proved to be efficient enough in practice, which resulted in several theoretical improvements of it [17, 15]. The worst time complexity of the algorithm is $O(n \cdot (n+m))$, nevertheless, the algorithm exhibits $O(m \cdot log\, n)$ expected time [12].

* This work has been partially supported by the Grant Agency of Czech Republic grant No. 201/06/1338.

In this paper, we discuss known as well as suggest new techniques used for parallel SCC decomposition, and we explore their restrictions if they are applied to implicitly given graphs. Efficient parallel algorithms for SCC decomposition will find their application in distributed formal verification tools such as `DiVinE` [2], `CADP` [13], `DUPPAAL` [4], `LiQuor` [8], etc. Namely, they will allow the tools to verify stochastic systems, compute τ-confluence, or verify systems with fairness constraints or properties given by other than Büchi automata.

The rest of the paper is organized as follows. We recapitulate basic terms and definitions in Section 2, describe known and new techniques used in parallel algorithms for solving the problem in Section 3, and list known parallel algorithms along with their pseudo-codes in Section 4. In Section 5 we report on an experimental study we performed, and in Section 6 we conclude the paper with several remarks and plans for future work.

2 Preliminaries

We start by brief summary of basic terms and definitions. Let V be a set of vertices, $E \subseteq V \times V$ a set of directed edges, and $v_0 \in V$ a vertex. We denote by $G = (V, E, v_0)$ a directed graph with initial vertex v_0.

Let $G = (V, E, v_0)$ be a directed graph. A sequence of edges $(u_0, u_1), (u_1, u_2), \ldots, (u_{n-1}, u_n)$ is called a *path* from vertex u_0 to vertex u_n. We say that vertex v *is reachable* from vertex u if there is a path from u to v or $u = v$. A *strongly connected component* (SCC) is a subset $C \subseteq V$ such that for any vertices $u, v \in C$ u is reachable from v. A strongly connected component C is *maximal* if there is no strongly connected component C' such that $C \subsetneq C'$. A maximal strongly connected component C is *trivial* if C is made of a single vertex c and $(c, c) \notin E$. Henceforward, we speak of maximal strongly connected components as of strongly connected components.

Let W_G be the set of all strongly connected components of graph $G = (V, E, v_0)$. A directed graph of strongly connected components of graph G is defined as $SCC(G) = (W_G, H_G, w_0)$, where w_0 is the component that contains the initial vertex v_0, and $H_G \subseteq W_G \times W_G$ is the set of edges between members of W_G. $(w_1, w_2) \in H_G$ if there are vertices $u_1 \in w_1$ and $u_2 \in w_2$ such that $(u_1, u_2) \in E$. Note that the graph of strongly connected components of any directed graph is acyclic.

A graph could be given in many ways. For purpose of this paper (and according to our needs) we consider graphs that are given implicitly. A graph is given implicitly if it is defined by its initial vertex and a function returning immediate successors of arbitrary vertex. Within the context of implicitly given graphs there are some restrictions the algorithms have to follow. If an algorithm requires any piece of information that cannot be concluded from the implicit definition of the graph, the algorithms have 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 all the predecessors must be computed and stored first.

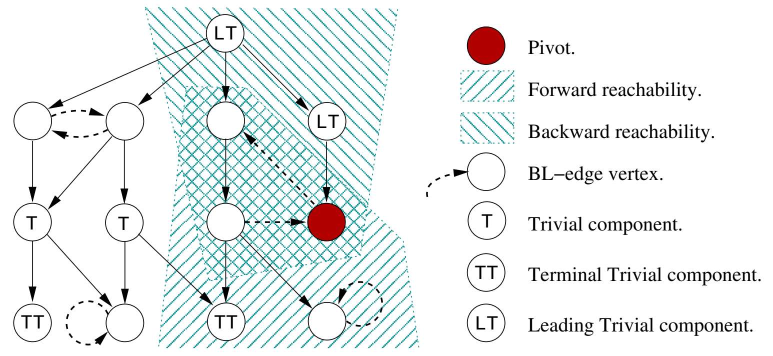

Fig. 1. Component detection, identified subgraphs, and trivial components.

Similarly, to number vertices of an implicitly given graph means to enumerate all its vertices first. For numbering vertices of implicitly given graphs a parallel procedure was introduced in [13]. Note that vertices of an implicitly given graph are trivially reachable from the initial vertex.

The reason for dealing with implicitly given graphs comes from practice. In many cases, the description of rules according to which the graph could 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 [9], the implicit definition of the graph is up to exponentially more succinct compared to the explicit one. This is commonly referred to as the state explosion problem [9].

3 List of Techniques

Before describing individual parallel algorithms we give the basic principles and list common techniques that all later given algorithms use. We hope this allow us to describe the algorithms and analyze their behavior in more compact and clear way.

Basic principle

All parallel algorithms we present in Section 4 build on the same technique that was originally presented in [12]. The graph to be decomposed is split into two parts. The *decomposed* part of the graph consisting of already identified components, and the *not-yet-decomposed* part of the graph consisting of vertices that have not been classified into strongly connected components yet. The basic step of each algorithm consists of picking a vertex from the *not-yet-decomposed* part of the graph, the so called *pivot*, and identifying the component the selected vertex belongs to. Having a pivot, the strongly connected component the pivot belongs to is determined as the intersection of sets of all predecessors and successors of the given pivot [12]. The structure of all algorithms is then a simple loop in which the basic step is repeated until the *not-yet-decomposed* part becomes empty. The basic step is illustrated on the example graph depicted in Figure 1. Note that the *not-yet-decomposed* part of the graph is further structured as explained below.

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. 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 procedures are the first place where parallelism appears in the algorithms. The parallelization of a reachability procedure became the standard technique [6, 7, 21, 16]. The so called *partition function* is used to assign every vertex of the graph to a single processor that is responsible for exploration of the vertex. Every processor participating the parallel computation maintains its own set of already explored vertices and its own list of vertices to be explored. If a vertex has been explored previously (it is in the set of explored 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 we describe in the next section use, in addition to the notion of *forward reachability*, the notion of *backward 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 behavior of the procedure for the forward reachability except it uses immediate predecessors instead of immediate successors during graph traversal. While forward reachability can be performed using only the implicit definition of the graph, the backward reachability, as explained above, requires a list of immediate predecessors to be computed and stored for every vertex first.

Trivial strongly connected components

Considering the basic algorithmic approach to SCC decomposition, the detection of trivial components is quite inefficient. If the pivot itself is a trivial component, both forward and backward reachability procedures perform useless work. There is rather small improvement in omitting the backward reachability procedure in the case the forward procedure did not hit the pivot, however, the forward procedure still performs $O(n+m)$ work. Therefore, any technique that prevents trivial components from becoming pivots has significant impact on practical performance of the algorithm.

A possible approach for doing so builds on the elimination of leading and terminal trivial components from the *not-yet-decomposed* part of the graph. In particular, 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 *not-yet-decomposed* part of 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 to the *One-Way-Catch-Them-Young* elimination (OWCTY) [11]. The technique can be applied in the analogue way also to vertices without successors (Reversed OWCTY). The improved version of the basic parallel algorithm that perform OWCTY elimination procedures before selection of the pivot was described in [15]. We stress that only leading and terminal trivial components may be identified in this way. Trivial components that are neither leading nor terminal may still be chosen as pivots. The graph depicted in Figure 1 contains all three types of trivial components: leading trivial components (`LT`), terminal trivial components (`TT`), and trivial components that are neither leading nor terminal (`T`).

Regarding implicitly given graphs the OWCTY elimination techniques suffer from the difficulty of identifying vertices with zero predecessors or zero successors. Basically, a complete reachability of the *not-yet-decomposed* part of the graph has to be performed to list those vertices. This reachability does not increase the theoretical complexity, however, it may play significant role in the practical performance of the algorithm.

Finally, let us mention that in many cases trivial components of the graph are of a little interest. Therefore, it make sense to save running time by avoiding their explicit enumeration that can be done using a single additional reachability procedure.

Pivot selection

Pivot selection plays a significant role in the complexity of the algorithm. Imagine we always pick a pivot belonging to a component that has no descendant components in the component graph of the *not-yet-decomposed* part. 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 and restricted to the *not-yet-decomposed* part of the graph. 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 introduction, difficult to be done in parallel. Since the optimal pivot selection is difficult, pivots are typically selected randomly. A random pivot selection leads to $O(m \cdot log\, n)$ expected time as claimed in [12].

In the explicit case, we can presuppose that vertices are numbered. Therefore, picking a random pivot corresponds to the generation of a random number. However, the problem occurs if a pivot has to be selected among vertices of the *not-yet-decomposed* part of the graph. As we are not aware of any $O(1)$ time and $O(1)$ space technique for a single pivot selection, we suggest a technique whose complexity is $O(n)$ space and $O(n)$ time if time and space complexity are summed for all pivot selection procedures called within a single run of the algorithm. The technique is applicable to implicitly given graphs as well. First,

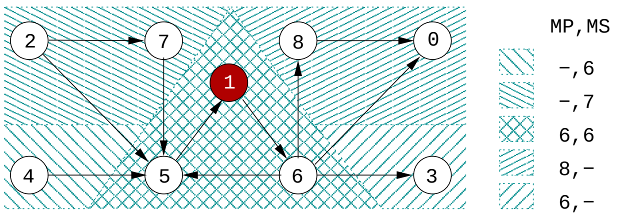

Fig. 2. Subgraphs identified with maximal predecessors (MP) and maximal successors (MS) if the propagations of MPs and MSs are initiated at pivot vertex.

each participating processor enqueues newly discovered vertices in a local queue when doing the very first forward reachability of the graph. Then, a new pivot can be selected among the heads of the local queues. However, if the vertex on the head of a local queue belongs to the *decomposed* part of the graph, it is dequeued and the next head is considered to be a candidate for pivot selection. Moreover, in the case of implicitly given graphs, the procedure organizing vertices into local queues can be combined with the procedure computing the immediate predecessors of vertices producing thus no overhead at all.

As we are typically not interested in trivial components, we suggest a completely new improvement in pivot selection. The idea is to prevent some trivial strongly connected components from being selected as pivots. We achieve this with the definition of the so called *candidate set*, i.e. the set of vertices among which pivots are chosen. Our intention is to terminate the algorithm once all candidate pivots have been selected and the corresponding components identified. If the candidate set contains initially at least one vertex for every non-trivial component of the graph, it must be the case that after the algorithm terminates the remaining *not-yet-decomposed* part of the graph is made of trivial components only. Generally, the smaller the candidate set is, the fewer trivial components are chosen as pivots. What we use for computing the candidate set is the concept of the so called *back-level edge* [3]. It is known that every cycle, and thus every non-trivial strongly connected component, contains at least one back-level edge, which is an edge that leads from a vertex with some distance from the initial vertex of the graph to a vertex with equal or smaller distance from the initial vertex of the graph. Let us call the destination vertex of a back-level edge a *BL-edge* vertex. We suggest the candidate set to be the set of *BL-edge* vertices. Note that *BL-edge* vertices can be computed during the initial reachability procedure using the level-synchronized breadth-first search of the graph [3]. As depicted in the graph in Figure 1, some trivial components can never become pivots considering *BL-edge* vertices as pivot candidates.

Independent subgraphs

In every iteration of the outermost loop of the basic algorithm the *not-yet-decomposed* part of the graph is split into several disjoint subgraphs. Let alone the identified component, these are the subgraph induced by vertices out of the

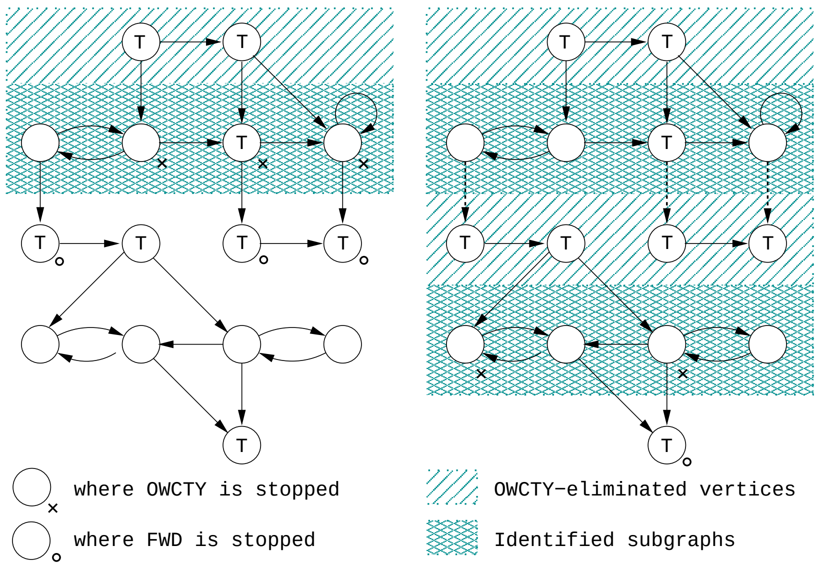

Fig. 3. Two steps of OWCTY-BWD-FWD independent subgraph identification.

component but explored during the forward reachability, subgraph induced by vertices out of the component but explored during the backward reachability, and subgraph induced by vertices that were not explored in that iteration at all. See example in Figure 1. An important observation [12] is that decomposing one of these subgraphs into strongly connected components is completely independent of the decomposition of other subgraphs. Therefore, the subgraphs may be viewed as if they were three independent graphs for the next step of the algorithm, which introduces two major improvements. First, three independent decomposition procedures may be performed in parallel increasing thus the amount of parallelism, second, every independent procedure may be restricted to explore vertices within the subgraph avoiding thus useless exploration of vertices out of the subgraph. Let us call the number of independent subgraphs produced in every iteration of the outermost loop (excluding the identified component) the *degree of parallelism* of the algorithm. Note that the number of the independent subgraphs grows exponentially with the number of iterations. Thus, if p is the number of available processors and d is the degree of parallelism, then after $log_d(p)$ iterations the number of independent subgraphs may exceed the number of available processors.

In the case of implicitly given graphs, vertices of a given subgraph are partitioned among processors according to the partition function. Therefore, all the single decomposition procedures share all processors participating the computation. Unfortunately, this requires to perform as many independent distributed termination detection procedures as there are single decomposition procedures running in parallel, which is quite technically involved and may actually be a reason for preventing individual decomposition procedures from being executed concurrently in practice. Also note that considering independent subgraphs, efficient pivot selection becomes complicated. In particular, we are not aware of any technique that could be used for selection of a random pivot from a subgraph without actually performing the whole subgraph exploration first. Also identifying leading and terminal trivial components in a subgraph results in a reachability procedure performed on the subgraph before the subgraph is decomposed. That is why we did not considered the leading and terminal trivial components elimination in all of the algorithms.

There is a technique that allows to identify more than three subgraphs in a single iteration [17]. Suppose the vertices of the graph are arbitrarily linearly ordered. Then, the maximal preceding vertex and maximal succeeding vertex can be computed for any vertex of the graph using an $O(n \cdot m)$ procedure [5]. If the forward and backward reachability procedures are extended to propagate maximal preceding and succeeding vertices, respectively, new subgraphs can be identified according to the maximal predecessors and successors associated to vertices in the subgraph. All vertices of the strongly connected component that the selected pivot belongs to, must have the same maximal predecessor and successor. Due to the pivot selection the maximal predecessors are computed only in the subgraph induced by vertices reachable from the pivot (forward reachability) while the maximal successors are computed for vertices that can reach the pivot (backward reachability). A possible result after a single iteration on a subgraph is depicted in Figure 2. In the original approach described in [17], the maximal predecessors and successors were computed over the complete graph. Regarding the number of identified components none of the approaches is better.

In the following we suggest a completely new technique to identify independent subgraphs in $O(n + m)$ time. The technique employs OWCTY elimination technique succeeded with backward and forward reachability procedures. A graph and two steps of the new technique performed on the graph are depicted in Figure 3. The OWCTY elimination procedure, if initiated from the vertex with zero predecessors, eliminates all leading trivial components 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. A backward reachability performed from vertices with the little cross identifies one independent subgraph. Note that this subgraph contains exactly all strongly connected components immediately reachable from the eliminated trivial components. Having the subgraph a forward reachability procedure restricted to the subgraph is performed from the vertices with little cross. This procedure stops on vertices outside the subgraph but immediately reachable from the subgraph (vertices with the little circle). Among these vertices there might be some that have predecessors only in the previously identified subgraph. These vertices must be trivial components, and therefore, they can be used as vertices to start the next OWCTY elimination procedure from. Figure 3 shows two successive steps of the subgraph identifying procedure OWCTY-BWD-FWD. We stress that the procedure may detect many independent subgraphs while performing only $O(n + m)$ work.

4 Algorithms

Having described all the techniques, we can now present individual algorithms. All pseudo-codes listed below describe the core parts of the algorithms. We neither list the initial reachability procedure that must be performed in order to compute the predecessor function in the implicit case, nor we list many technical details related to implementation, parallelization, distribution, etc.

F-B

The F-B algorithm [12] is the basic algorithm that all other presented algorithms build on. In the following pseudo-code, we describe a single procedure that is initially called for the complete set of vertices of the graph to be decomposed and then called recursively for identified independent subgraphs. A pivot is selected using procedure PIVOT and the set of vertices reachable in forward and backward manner 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 subgraph, the procedure will explore only immediate successors or predecessor of vertices within the subgraph. 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. Note that if it is necessary, the procedure is able to select pivots among given set of candidates.

```
1  proc F-B($V$, $candidates$)
2    if ($V \neq \emptyset$)
3      then $p$ := PIVOT($V \cap candidates$)
4           $F$ := FWD($p$, $V$)
5           $B$ := BWD($p$, $V$)
6           $SCCs := SCCs \cup \{F \cap B\}$
7           in parallel do
8             F-B($F \setminus B$, $candidates$)
9             F-B($B \setminus F$, $candidates$)
10            F-B($V \setminus (F \cup B)$, $candidates$)
11          od
12   fi
13 end
```

In our experimental study we also considered a slightly modified version of the basic algorithm. In particular, we implemented a version in which the backward reachability procedure was restricted to the vertices discovered by the preceding forward reachability procedure, i.e. line 5 in the pseudo-code is changed to

$$B := \mathrm{BWD}(p, F).$$

This modification decrease the degree of parallelism, but produce a procedure where exploration of some vertices is omitted compared to the original algorithm. One could tend to see this technique as an improvement, however, our experiments proved that neither of the versions was significantly better then the other. Our explanation for the lack of improvement in some cases is that the subgraphs identified as $(V \setminus (F \cup B))$ become actually larger causing thus more work to be done in subsequent recursive calls to the procedure.

MP-MS

Algorithm MP-MS [17] extends the previous algorithm with the maximal predecessors and maximal successors concept. Compared to algorithm F-B, the improvement is in the subgraph detection, see Section 3. In order to compute the maximal predecessors and successors, parallel procedures FWD and BWD have to be replaced with new parallel procedures FWD-MAXPRED and BWD-MAXSUCC, respectively. Besides computing the same reachability relation as procedures FWD and BWD, the new procedures also identify subgraphs according to the maximality of predecessors or successor and return lists of those vertices whose order is used to refer to a subgraph. These *SuccList* and *PredList* are then used to perform parallel recursive calls on identified subgraphs. See the pseudo-code below. Recall that the time complexity of both new procedures is $O(n \cdot m)$, which is worse than if simple reachability procedures are used. However, the bad complexity is paid off with the degree of parallelism being much higher compared to the degree of parallelism of algorithm F-B. Finally, let us mention that also algorithm MP-MS is capable of selecting pivots among given set of candidates.

```
proc MP-MS($V$, $candidates$)
  if ($V \neq \emptyset$)
    then $p := \text{PIVOT}(V \cap candidates)$
         $F, PredList := \text{FWD-MAXPRED}(p, V)$
         $B, SuccList := \text{BWD-MAXSUCC}(p, V)$
         $SCCs := SCCs \cup \{F \cap B\}$
         in parallel do
           MP-MS($V \setminus (F \cup B), candidates$)
           MP-MS($F[k, -], candidates$) foreach $k \in PredList$
           MP-MS($B[-, k], candidates$) foreach $k \in SuccList$
         od
  fi
end
```

O-B-F

Algorithm O-B-F is completely a new algorithm we suggest in this paper. The core idea of the algorithm is to partition the component graph to the so called *O-B-F levels* and then call any algorithm (F-B in our case) to decompose individual levels of the component graph into strongly connected components. Recall that the component graph can be partitioned to those levels in linear time using the new technique described in Section 3.

The procedure O-B-F performs the detection of levels in the level by level manner. It is started with the complete set of vertices as the graph to be decomposed, and with the initial vertex as the vertex to start the exploration from. In every single call of the procedure one O-B-F level is detected. The set of remaining vertices, denoted with V, is appropriately shrunk, candidates for initial vertices of V (the so called *Seeds*) are computed, and two procedures are initiated in parallel. First, a procedure to decompose the subgraph identified with the O-B-F level, second, procedure O-B-F to identify other levels in the remaining set V. Recursive calls to procedure O-B-F terminates when all the levels are recognized and the set of remaining vertices is empty.

Every single O-B-F level is detected using standard procedures. First of all, leading trivial components of the remaining graph are eliminated using procedure OWCTY. The procedure computes the set of leading trivial components (*Eliminated*) and the set of vertices on which the elimination process stopped (*Reached*). Eliminated vertices are removed from set V of remaining vertices and the standard backward reachability procedure is performed from vertices in *Reached* and restricted to vertices in V. As the backward procedure is restricted to V, it computes exactly vertices belonging to the top most level in the component graph of V. These vertices (denoted with B) are removed from set V of remaining vertices and the decomposition of the level is initiated as an independent parallel procedure. Note that the set of potential pivots can be restricted to vertices in *Reached* because every strongly connected component belonging to the level must contain at least one vertex from *Reached*. A forward reachability (FWD-SEEDS) is also performed on vertices in B in order to identify vertices immediately below the current level, which are exactly vertices that become *Seeds* for the next call to procedure OWCTY in the next recursive call of the procedure O-B-F. Note that vertices in *Seeds* that belong to non-trivial strongly connected components are moved directly to set *Reached* within procedure OWCTY.

```
proc O-B-F($V$, $Seeds$)
  if ($V \neq \emptyset$)
    then $Eliminated, Reached$ = OWCTY($Seeds$, $V$)
         $V := V \setminus Eliminated$
         $B :=$ BWD($Reached$, $V$)
         $V := V \setminus \{B\}$
         $F, Seeds :=$ FWD-SEEDS($Reached$, $B$)
         in parallel do
           F-B($B$)
           O-B-F($V$, $Seeds$)
         od
  fi
end
```

5 Experimental Evaluation

We have implemented and experimentally evaluated quite a few algorithms described in this paper. The algorithms were implemented using the DiVinE Library [2] as the library providing support for parallel and distributed generation of implicitly given graphs. The common library used gives approximately the same level of enhancement of all implementations, thus, the experimental comparison is quite fair. All experiments were conducted on a network of ten Intel Pentium 4 2.6 GHz workstations each having 1 GB of RAM and 100Mbps switched Ethernet connection.

The graphs we use to evaluate our implementations come from DiVinE Library distribution. They are listed in Table 1 along with their important characteristics, namely, the number of vertices (**Vertices**), number of edges (**Edges**), numbers of trivial and non-trivial strongly connected components (**T. SCCs**, **N. SCCs**), and the time needed for sequential decomposition into strongly connected components using Tarjan's algorithm (**Tarjan**). Value *n.a.* means that the sequential decomposition algorithm exceeded available amount of RAM. For the purpose of the distributed experiments, all the graphs were distributed using the default hash-based partition function implemented in DiVinE Library.

We implemented and experimentally evaluated six different algorithms. Algorithms **F-B**, **MP-MS**, and **O-B-F** directly correspond to algorithms presented in Section 4. Algorithm **F-RB** is the modified version of algorithm **F-B**, i.e. the version where the backward reachability procedure is restricted to vertices explored during the preceding forward reachability procedure. If the name of the algorithm is extended with suffix **(B)**, then the algorithm was initiated considering *BL-edge* vertices as pivot candidates. We have not implemented the modification of algorithms **F-B** and **MP-MS** that includes elimination of leading and terminating trivial components on the given subgraph before the subgraph is decomposed [15, 18]. The reason is that we are not aware of any technique that would identify vertices with zero predecessors or zero successors in the given subgraph without actually exploring the subgraph first, which makes the approach inefficient in the case of implicitly given graphs.

All our implementations explicitly avoid concurrent performance of the decomposition procedures on independent subgraphs. In particular, if an independent decomposition procedure is about to be initiated, its assignment is stored and its initiation postponed. There are several reasons for this. First, the number of processors we have at our hand is very limited. Therefore, parallel procedures would very soon produce a non-trivial overhead caused by switching the context of CPUs depreciating thus the measured values. Second, as already mentioned in Section 3, appropriate termination detection becomes technically involved if independent parallel procedures share CPUs. Moreover, pivot selection within the given subgraph would generally introduce additional reachability procedure performed on every discovered independent subgraph if the subgraphs should be decomposed in parallel. And third, as the algorithms perform parallel reachability procedures most of the time, we have not observed idling of individual workstations. Therefore, we believe that the parallelism of the decomposition

Name	Vertices	Edges	T. SCCs	N. SCCs	Tarjan
DrivingPhilsK3	6307240	12950475	16	1	4:51
DrivingPhilsK3_4	10301529	24055321	3170354	2680	10:27
Elevator12_2	8591334	89419176	2004966	2	13:21
Lifts6	16364845	50088312	7231789	8052	n.a.
LookUpProc10_3	16562363	33464135	1603283	2	n.a.
MutBak4	9384762	31630895	1881088	15	30:07
MutMcs4	1241948	4456310	9718	39	33
Phils14_1	9565935	124357142	531442	28	n.a.
Pet6err	1060048	6656522	208436	25075	4:29
Rether9_2	7663993	9624242	81831	5	16:38
Train8_2	11740214	37389502	5273750	50858	3:10:44

Table 1. Summary of graphs.

Graph	F-B	F-B (B)	F-RB (B)	MP-MS	MP-MS (B)	O-B-F
DrivingPhilsK3	2:13	1:58	2:08	17:20	22:43	1:57
DrivingPhilsK3_4	n.a.	3:41:37	n.a.	n.a.	n.a.	4:30:36
Elevator12_2	n.a.	n.a.	n.a.	n.a.	n.a.	9:06
Lifts6	n.a.	5:15:46	n.a.	n.a.	n.a.	5:47:44
LookUpProc10_3	n.a.	n.a.	n.a.	n.a.	n.a.	16:36
MutBak4	n.a.	2:31:31	1:42:31	n.a.	n.a.	1:29:09
MutMcs4	7:32	37	23	26:21	34:32	23
Phils14_1	2:42:03	18:36	18:30	n.a.	n.a.	21:31
Pet6err	n.a.	n.a.	n.a.	n.a.	n.a.	4:53:47
Rether9_2	1:13:01	27:57	8:23	4:13:29	2:14:05	17:11
Train8_2	n.a.	2:09:54	1:52:21	n.a.	n.a.	n.a.

Table 2. Runtimes (hours:minutes:seconds).

procedures would bring nothing but increased complexity of the implementations. Actual runtimes needed by all the algorithms to decompose the graphs are reported in Table 2. Value *n.a.* denotes now that the runtime of the algorithm exceeded 10 hours time limit.

We find the experimental results very interesting. First, we were slightly surprised with the practical inefficiency of the algorithm based on maximal predecessors and maximal successors. Its performance is far beyond performance of other algorithms proving that the decomposition into many subgraphs is not worth unless it is done in $O(n+m)$ time. Second, quite interesting result is that the restriction of the set of vertices that can be selected for pivots play significant role in practice. Note that in the case of algorithm **F-B**, the *BL-edge* vertices yielded roughly speed up of to ten. In the case of algorithm **MP-MS** they did not generally help at all, for which we blame the procedures with $O(n \cdot m)$ time complexity whose bad performance cut off any improvements made in pivot selection. Third, algorithm **O-B-F** proved to have big potential as it was the fastest algorithm in many cases, and sometimes even the only algorithm that was able to perform the decomposition within the given time limit. Finally, let us mention that according to our preliminary experiments, there were cases where the parallel algorithms if executed on ten workstations, outperformed even the optimal Tarjan's algorithm.

6 Conclusion and Future Work

In this paper we tried to list and evaluate all known techniques used in parallel algorithms for decomposition of implicitly given graphs into strongly connected components, and compare the parallel algorithms that exploit them. We also introduced two completely new techniques that the parallel algorithms can employ. In particular, we suggested how *BL-edge* vertices can be profited from if they are used as pivot candidates, and how the graph can be decomposed into subgraphs preserving SCCs using linear time and parallel technique OWCTY-BWD-FWD. Both newly suggested techniques have shown their superior strength in our experimental study.

We would especially like to emphasize that the newly suggested procedure shows not only practical usefulness, but also a theoretically interesting behavior. In particular, it may be proved that graphs whose components exhibit a chain-like structure, can be decomposed in parallel in the optimal linear time. Generally, using the technique we are able to give a parallel algorithm for solving the SCC decomposition problem working in $O(h \cdot (n + m))$ time, where h is the maximal number of strongly connected components on an acyclic path in a single *O-B-F* level.

Although, the preliminary results are encouraging, we are well aware of the immaturity of our experimental evaluation. We intend to perform thorough experimental study on larger set of inputs including algorithms with elimination of leading and terminal trivial components in the future. We also intend to improve implementations of the algorithms, in particularly, we would like to come up with a reasonable pivot selection procedure that would allow us to implement and experimentally evaluate virtually concurrent decomposition of the independent subgraphs. Finally, we intend to incorporate the best algorithms in the distributed verification environment DiVinE, so that the tool is capable of distributed and parallel verification of stochastic systems as well as verification of properties given by other than Büchi automata.

Let us also mention that we have tried to come up with some algorithms that avoid backward reachability procedure being thus perfectly suitable for the decomposition of implicitly given graphs. However, all our attempts resulted in algorithms whose practical performance was quite poor and discouraging.

References

1. Nancy Amato. Improved processor bounds for parallel algorithms for weighted directed graphs. *Inf. Process. Lett.*, 45(3):147–152, 1993.
2. J. Barnat, L. Brim, I. Černá, P. Moravec, P. Ročkai, and P. Šimeček. Divine – a tool for distributed verification. To appear in proceedings of CAV 2006.

3. J. Barnat, L. Brim, and J. Chaloupka. Parallel Breadth-First Search LTL Model-Checking. In *18th IEEE International Conference on Automated Software Engineering (ASE'03)*, pages 106–115. IEEE Computer Society, Oct. 2003.
4. G. Behrmann. A performance study of distributed timed automata reachability analysis. In *Proc. Workshop on Parallel and Distributed Model Checking*, volume 68 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science Publishers, 2002.
5. L. Brim, I. Černá, P. Moravec, and J. Šimša. Accepting Predecessors are Better than Back Edges in Distributed LTL Model-Checking. In *5th International Conference on Formal Methods in Computer-Aided Design (FMCAD'04)*, volume 3312 of *LNCS*, pages 352–366. Springer-Verlag, 2004.
6. S. Caselli, G. Conte, and P. Marenzoni. Parallel state space exploration for GSPN models. In G. De Michelis and M. Diaz, editors, *Applications and Theory of Petri Nets 1995*, volume 935 of *LNCS*, pages 181–200. Springer-Verlag, 1995.
7. G. Ciardo, J. Gluckman, and D.M. Nicol. Distributed State Space Generation of Discrete-State Stochastic Models. *INFORMS Journal of Computing*, 1997.
8. Frank Ciesinski and Christel Baier. LiQuor: A tool for Qualitative and Quantitative Linear Time analysis of Reactive Systems, 2006.
9. E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, Cambridge, Massachusetts, 1999.
10. Richard Cole and Uzi Vishkin. Faster optimal parallel prefix sums and list ranking. *Inf. Comput.*, 81(3):334–352, 1989.
11. K. Fisler, 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*, volume 2031 of *LNCS*, pages 420–434. Springer, 2001.
12. Lisa K. Fleischer, Bruce Hendrickson, and Ali Pinar. On identifying strongly connected components in parallel. *Lecture Notes in Computer Science*, 1800:505–511, 2000.
13. H. Garavel, R. Mateescu, and I.M Smarandache. Parallel State Space Construction for Model-Checking. In *Proceedings of the 8th International SPIN Workshop on Model Checking of Software (SPIN'01)*, volume 2057 of *LNCS*, pages 200–216. Springer-Verlag, 2001.
14. Hillel Gazit and Gary L. Miller. An improved parallel algorithm that computes the bfs numbering of a directed graph. *Inf. Process. Lett.*, 28(2):61–65, 1988.
15. William McLendon III, Bruce Hendrickson, Steven J. Plimpton, and Lawrence Rauchwerger. Finding strongly connected components in distributed graphs. *J. Parallel Distrib. Comput.*, 65(8):901–910, 2005.
16. F. Lerda and R. Sisto. Distributed-memory model checking with SPIN. In *Proceedings of the 6th International SPIN Workshop on Model Checking of Software (SPIN'99)*, volume 1680 of *LNCS*. Springer-Verlag, 1999.
17. S. Orzan. *On Distributed Verification and Verified Distribution*. PhD thesis, Free University of Amsterdam, 2004.
18. S.M. Orzan and J.C. van de Pol. Detecting strongly connected components in large distributed state spaces. Technical Report SEN-E0501, CWI, 2005.
19. John H. Reif. Depth-first search is inherently sequential. *Information Processing Letters*, 20(5):229–234, June 1985.
20. R. Tarjan. Depth first search and linear graph algorithms. *SIAM Journal on computing*, pages 146–160, 1972.
21. U.Stern and D. L. Dill. Parallelizing the murφ verifier. In O. Grumberg, editor, *Proceedings of Computer Aided Verification (CAV '97)*, volume 1254 of *LNCS*, pages 256–267. Springer-Verlag, 1997.