Moscow State University

The Faculty of Computational Mathematics and Cybernetics

Overview of algorithms for search of SCC in directed graph.

Ivanov Denis

M118

Moscow, 2018

Contents

1 Introduction

Graph theory is the field of mathematics which studies graphs. A graph is a structure that models relations between objects. The objects are called vertices (or nodes), while the relations are represented by edges. For example, the users of a social network can be represented as vertices, while the friendship between two users is represented by an edge between the two. If the relationship can be asymmetric, then the edge will point into a direction and the graph will be called directed. A fundamental concept in graph theory is a strongly connected component (SCC) [13]. An SCC in a directed graph G is a maximal subset of vertices U such that there is a path in G from every vertex in U to every other vertex in U. An example of a graph partitioned into SCCs can be found in Figure 1.1, where each marked region represents an SCC. For example, the vertices 1, 2, 3 are in the same SCC, because they can all reach each other through a path of edges.

Algorithms for detecting SCCs can be applied to a wide variety of problems from different domains [25]. An example of this is 2-SAT, a special case of the boolean satisfiability problem in which one attempts to assign values to boolean variables in a formula such that it is satisfied [6]. Connected components also are an important concept in compiler construction, where strongly connected components within a call graph are identified to determine the order in which functions should be processed [30]. Yet another example comes from the field of bio-informatics, for the problems in which multiple sequence alignment are performed to decipher the meaning of sequences of biological compounds [21].

Speed enhancements of single-core microprocessors are stagnating as the miniaturization of chips approaches its physical limits [29]. As a result, processors that run multiple cores in parallel have become increasingly relevant. These developments call for concurrent algorithms that can exploit this parallelism in an

optimal way. Specifically, graph algorithms, which are used on increasingly large data sets, need to be adjusted to take advantage of parallelism. Given that the detection of SCC is a problem with a wide variety of applications, a lot of research has been dedicated to devising parallel algorithms for it. This paper aims to give overview and description of known algorithms.

We provide analysis of all the existing parallel algorithm for detecting SCCs. Many of the published papers report a small number of results and only compare theirs results to one or two other algorithms without updating the original implementation. Moreover, the authors of the papers could be biased when reporting the results, and thus some algorithms can be reported to perform better. As such, a fair comparison between these algorithms can only be made by reimplementing the most promising ones and testing them on a set of real-world graphs with various structures, which lies beyond the scope of this paper. There was one algorithm proposed by Schudy [23] that did not report any data. Ebendal gave an implementation of this algorithm and proposed a couple of improvements such that showed very promising results, especially on sparse graphs [9].

2. Algorithms overview

Several algorithms have been developed that can detect SCCs in linear time [8] [10] [26]. In this chapter we will focus only on parallelizable algorithms. Our aim is to give a complete overview and an intuitive description of the different approaches developed for finding SCCs in parallel.

2.1 Divide-and-Conquer Algorithms

Most of the sequential algorithms rely on depth-first search. However, this technique has been proven to be difficult to parallelize [12] as it is P-complete [20]. This is a strong motivation for a divide-and-conquer approach. In a divide-and-conquer approach, the graph is recursively divided in sub-graphs such that an SCC only spans one such sub-graph.

2.1.1 Forward-Backward Algorithm

One of the most popular parallel algorithms for finding SCCs is the so-called forward-backward (FB) algorithm, by Fleischer et al. [11]. This algorithm forms the basis of most other parallel algorithms for finding SCCs. The algorithm uses the concept of forward and backward reachability. In a directed graph, a vertex u is forward reachable from a vertex v, if there is a path from v to u. Likewise, a vertex u is backward reachable from a vertex v, if there is a path from u to v. The forward reachability set of a vertex v consists of all the vertices that are forward reachable from v. The backward reachability set is defined in the same fashion. The idea is to choose any vertex v, find its backward and forward reachability sets and the common subset of these will be a unique SCC.

2.1.1.1 Trimming Step.

The algorithm above was improved by McLendon et al., by adding one step to the design to trim away trivial SCCs [17]. Trivial SCCs are those that contain only one vertex. In the literature this step is also called OWCTY (One-Way-Catch-Them-Young) [1] [3] [7]. The justification of this step comes from the fact that the FB algorithm has a very high recursion depth on sparse graphs. For example, on graphs with no edges the recursion depth is $\Theta(|V|)$. In a sparse graph it is often the case that many of the resulting sets are empty, which will lead to a much slower distribution of the computation over the available processors than desired. The optimisation is based on the observation that vertices which do not have any incoming or outgoing edges can only be in an SCC by themselves. If a vertex has no incoming edges, no other vertex can reach it, while if it has no outgoing edges it cannot reach any other vertex. Thus in the trimming step, each vertex v that has no incoming or outgoing edges, is placed in an SCC of its own, and it is removed from the pool of vertices to be further inspected. If any vertex was removed, the procedure is repeated again, as removing one trivial SCC could create another. An example of trimming is shown in Figure 2.2. In the first step of the trimming step, vertices c and d will be removed, since c has no incoming vertices and d has no outgoing ones. Then the procedure will be called again, where vertex b will be removed, and finally in the next call a will also get removed.

2.1.2 Schudy's Algorithm

Schudy addresses the lack of parallelism of the FB algorithm on sparse graphs as well [23]. He uses a slightly modified version of the FB algorithm as a starting point. The backward reachability search from pivot v is performed only on the already computed forward reachability set F. Thus, no fully backward reachable set is computed.

The main idea of Schudy's improvement, is that there is not one single pivot selected, but a set of pivots. A random unique index from 1 to $|V|$ is assigned to all

the vertices. Through a binary search we find the smallest index s, such that the pivots from 1 to s will together reach a subset of at least half of the vertices plus edges. In Algorithm 1 the procedure pivotSet performs these computations, and the found pivots are stored in P. The pivot procedure retrieves the largest index s from P, after which s is removed from P. Starting from s, the forward reachability set F is computed. Considering only the vertices in F, a backward reachability set C is computed. The set C is an SCC. These steps are exactly the same as in the adapted FB algorithm. Next, a forward reachability set Q is computed from the set P of pivots. Since both F and Q are closed under forward reachability, they are used to further partition the vertices into four sets that have no overlapping SCCs.

The main result of Schudy is that the FB multi-pivot optimisation improves, with high probability, the recursion depth from O(|V |) to O(log |V |). An important note is that from all the studies presented here, Schudy is the only one who only evaluated his work theoretically. He does not report on any experimental results.

2.1.3 Hong's Algorithm

Hong et al. present and explore a property of real-world graphs, namely the small-world property [15]. It has been shown that in graphs with the small-world property there exists a giant SCC of size O(|V |). Moreover, the distribution of the SCCs follow a power distribution: the number of SCCs grows exponentially as the size of the SCCs decreases, meaning there is a great number of smaller SCCs [14].

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

parallelism. That is, use a forward and backward parallel reachability algorithm until a percentage of vertices are assigned to a SCC, then continue doing only task-parallelism with the regular FB algorithm. Their measurements indicate that after classifying more than 1% of the vertices, the huge SSC has also been identified. After the huge SCC has been removed, there are a lot of smaller weakly connected components (WCCs) left. Two vertices are said to be in the same WCC if there would be a path between them if the edges would be undirected. As discussed previously, the FB algorithm then will not produce new tasks fast enough and so threads can stay idle. To solve this issue, Hong et al. apply a fast algorithm to identify each WCC and then FB can run independently on each WCC. Figure 2.6 depicts the decomposition of WCCs.

2.1.4 OWCTY-Backward-Forward (OBF)

The main idea behind the OBF algorithm is to divide the graph in slices, such that there is no SCC spanning two slices [4]. Because of this property, the FB algorithm can be applied on each slice in parallel.

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

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

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

F Remove the slice B from the total pool of vertices, and let the FW-BW algorithm run on it. Recursively call the OBF procedure on the remaining vertices using the successors of the vertices in B as seeds. The procedure terminates when all the vertices have been added to a slice.

2.1.4.1 Recursive OBF

Instead of calling the FB algorithm on each slice, the recursive OBF algorithm will again call itself [2]. However, the OBF algorithm needs a rooted graph to work on. This is solved by dividing the slice in rooted chunks. A rooted chunk is created by picking a vertex at random from the slice and computing its forward closure within the slice. Then the OBF algorithm will be run in parallel on each rooted chunk. Moreover there needs to be a termination criterion in case the whole chunk is one SCC. This is simply if the B step visits all the vertices from the chunk.

2.1.5 Coloring/Heads Off (CH)

The CH algorithm uses the concept of graph coloring [19]. Every vertex has a unique number as identifier, which is called the color of the vertex. The algorithm starts with a forward search which can start in parallel from a different number of vertices. When a vertex v has a higher color than a successor u, u will get the color of v. We say that v propagates its color forward. It is possible for a vertex to have its color changed several times. This phase of the algorithm stops when no vertex can change its color. At this point all vertices of an SCC have the same color, thus all edges between vertices of different colors can be removed.

The vertices that kept their colors are considered the root of an SCC. Now from every vertex v a backward search is performed. If this search reaches the root of its color, then v is in the same SCC as the root. So we can say that all vertices that are backward reachable from the root and have the same color as the root form one SCC. The discovered SCCs are removed and the algorithm continues by assigning new unique numbers to the remaining vertices. The algorithm terminates when all vertices have been assigned to an SCC.

2.1.6 Multistep

The Multistep algorithm is a combination of other algorithms [27]. The motivation of merging different algorithms comes from the observation that the performance of algorithms for detecting SCCs is heavily dependent of the graph's structure, i.e., the relative size and distribution of SCCs. When SCCs are detected and removed, the graph's structure changes, so it can make sense to also apply a different algorithm. Multistep starts with the trimming step. Then, similar to Hong et al., it uses FB to find the huge SCC present in graphs with the small world property. Once the huge SCC has been removed, CH is applied until a partition reaches a threshold of nodes. To avoid parallelization overhead, remaining nodes are sorted using an efficient sequential algorithm, in this case Tarjan's algorithm [28]. Experiments have shown that for a graph with a few millions nodes, a good threshold for switching to the sequential algorithm is 100,000. However, the resulting performance is highly dependent on the graph's structure and also on the machine architecture.

2.2 Depth-First Search Based Algorithms

One important application for detecting SCCs is in the field of formal verification, namely the model checking problem. That is, the problem of exhaustively and automatically verifying certain properties on the state space of a system's model [5]. For this problem in particular, it is useful to be able to detect SCCs on-the-fly, that is, during the construction of the state space. This is because a counter-example for a property, might be found on the basis of an SCC without having to generate the whole state space [24]. The algorithms used for detecting SCCs on-the-fly use depth-first-search (DFS) and are based on Tarjan's algorithm.

2.2.1 Tarjan's Algorithm

Tarjan's algorithm is a sequential recursive algorithm. Even though it does not utilize parallelization, we describe it here as it will make the explanation of the

following algorithms easier to follow. Every vertex keeps track of two values index and lowlink that are initially set to zero. The index variable indicates in which order this vertices were visited, so once assigned it will never change. The lowlink is used to find edges to ancestors and it can be changed during recursive DFS calls. Moreover the algorithm uses a stack to keep track of all the visited vertices. The algorithm starts at an arbitrary vertex v. Its index and lowlink variables are set to the maximum index value of any other node plus one and the node gets added to the stack. A DFS is performed from v, and every visited node will have its fields changed in the same way and get pushed onto the stack. At some point a vertex u is reached with no unvisited successors. If u has a successor w on the stack with a lower lowlink value, then it will copy w's lowlink value. While tracking back from the DFS, if a vertex v's lowlink equals its index value, it means that none of the vertices v can reach can get to a vertex discovered before v, so all these vertices form an SCC. Thus all vertices up to v need to be popped from the stack.

2.2.2 Lowe's Algorithm

Lowe's algorithm parallelizes Tarjan's algorithm by deploying multiple instances which run in parallel, starting from different vertices [16]. The algorithm explores vertices and marks SCCs in a similar fashion as Tarjan's algorithm, however it is modified in order to detect and resolve possible crashes between searches. Every instance of the algorithm uses a local stack, but the fields of every vertex, index and lowlink, are shared data between the threads. An important invariant of the algorithm is that a vertex can only belong to one stack. To make sure this is respected, every vertex has a status field that can be either unseen, live or dead. Initially the status of all vertices is unseen. A vertex can be visited only if its status is unseen, after which the status will change to live. A search that encounters a vertex with the live status is suspended until the vertex changes its status to dead. A vertex changes its status to dead only when it is marked as belonging to an SCC. This can only be done by the

thread that changed its status to live. It is not hard to see that this procedure can lead to deadlocks. For example, a search from thread p changes the status of vertex v to live and next reaches vertex u which has the status live set by a search from thread q. Thread p suspends its search until thread q marks u dead. However, in its search q might encounter vertex v, which only p can change the status of, and so also thread q will suspend its search. In order to recover from deadlocks, the algorithm keeps track of a Suspended map. This stores for each suspended search p, the vertex u at which the search has to stop and its predecessor, vertex v. When a path is found from u to v in the Suspended map, belonging to the suspended search q, the two searches are merged by having all the vertices belonging to the p stack pushed onto the q stack. The search is now continued by one thread. The algorithm deploys as many instances as there are threads available. Whenever a thread is suspended, the algorithm deploys another instance in order to save resources.

2.2.3 Renault's Algorithm

Just like in Lowe's algorithm, in Renault's algorithm multiple instances of Tarjan's algorithms are deployed [22] at the same time. However, the essence of the algorithm is quite different. In Renault's algorithm threads only communicate the fully discovered SCCs. That is, there is only one shared union-find data structure [18], across all the threads containing the already detected vertices belonging to an SCC. Unlike in Lowe's algorithm, in Renault's algorithm if two threads start the discovery of an SCC at the same time, they will both discover the SCC individually. Thus, when the graph contains only one SCC, each thread will work to discover the SCC by itself. While Lowe's strategy may seem to be more efficient at the first glance, it does not always need to be so. Renault's algorithm has the advantage that it can visit a vertex without acquiring a lock. Moreover, it does not have the overhead of deadlocks.

Bibliography

[1] Jiri Barnat, Luboˇs Brim, and Milan Ceˇska. DiVinE-CUDA-a tool for GPU ˇ accelerated LTL model checking. arXiv preprint arXiv:0912.2555, 2009.

[2] Jiri Barnat, Jakub Chaloupka, and Jaco van de Pol. Improved distributed algorithms for SCC decomposition. Electronic Notes in Theoretical Computer Science, 198(1):63–77, 2008.

[3] Jiri Barnat, Jan Havl´ıˇcek, and Petr Roˇckai. Distributed LTL model checking with hash compaction. Electronic Notes in Theoretical Computer Science, 296:79–93, 2013. [4] Jiˇr´ı Barnat and Pavel Moravec. Parallel algorithms for finding SCCs in implicitly given graphs. In International Workshop on Formal Methods for Industrial Critical Systems, pages 316–330. Springer, 2006.

[5] B´eatrice B´erard, Michel Bidoit, Alain Finkel, Fran¸cois Laroussinie, Antoine Petit, Laure Petrucci, and Philippe Schnoebelen. Systems and software verification: model-checking techniques and tools. Springer Science & Business Media, 2013.

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

[7] Lubos Brim, Jir´ı Barnat, et al. Platform dependent verification: On engineering verification tools for 21st century. In PDMC, pages 1–12, 2011.

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

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

[10] Sami Evangelista, Alfons Laarman, Laure Petrucci, and Jaco Van De Pol. Improved multi-core nested depth-first search. In International Symposium on Automated Technology for Verification and Analysis, pages 269–283. Springer, 2012. [11] Lisa K Fleischer, Bruce Hendrickson, and Ali Pınar. On identifying strongly connected components in parallel. In International Parallel and Distributed Processing Symposium, pages 505–511. Springer, 2000.

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

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

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

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

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

[17] William McLendon III, Bruce Hendrickson, Steven J Plimpton, and Lawrence Rauchwerger. Finding strongly connected components in distributed graphs. Journal of Parallel and Distributed Computing, 65(8):901– 910, 2005.

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

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

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

[21] Elisabeth Remy, Brigitte Moss´e, Claudine Chaouiya, and Denis Thieffry. A description of dynamical graphs associated to elementary regulatory circuits. Bioinformatics, 19(suppl 2):172–178, 2003. 37

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

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

[24] Stefan Schwoon and Javier Esparza. A note on on-the-fly verification algorithms. In Nicolas Halbwachs and Lenore D. Zuck, editors, Tools and Algorithms for the Construction and Analysis of Systems, 11th International Conference, volume 3440 of Lecture Notes in Computer Science, pages 174– 190. Springer, 2005.

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

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

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

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

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

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