Technical Report: On Fast Parallel Detection of Strongly Connected Components (SCC) in Small-World Graphs

Sungpack Hong
Oracle Labs
Redwood Shores, CA
sungpack.hong@oracle.com

Nicole C. Rodia
Pervasive Parallelism Laboratory
Stanford University
Stanford, CA
nrodia@stanford.edu

Kunle Olukotun
Pervasive Parallelism Laboratory
Stanford University
Stanford, CA
kunle@stanford.edu

***Abstract*—**
Detecting strongly connected components (SCCs) in a directed graph is a fundamental graph analysis algorithm that is used in many science and engineering domains. Traditional approaches in parallel SCC detection, however, show limited performance and poor scaling behavior when applied to large *real-world* graph instances. In this paper, we investigate the shortcomings of the conventional approach and propose a series of extensions that consider the fundamental properties of *real-world* graphs, e.g. the *small-world* property. Our scalable implementation offers excellent performance on diverse, *small-world* graphs resulting in a 4.44x to 23.98x parallel speedup over the optimal sequential algorithm with 8 cores and 16 hardware threads.

***Keywords*-strongly connected components (SCC); multi-core; parallel algorithms; graph algorithms; small-world graphs**

I. Introduction

In graph theory, a strongly connected component (SCC) of a directed graph is a maximal subgraph where there exists a path between any two vertices in the subgraph. Since any directed graph can be decomposed into a set of disjoint SCCs, the study of large graphs frequently uses SCC detection of the target graph as a fundamental analysis step. Target *real-world* graphs include the Web graph and social networks [11], [12], [17], and those found in diverse scientific applications, including formal verification [14], reinforcement learning [16], 3D mesh element refinement [22], and complex food web analysis [3].

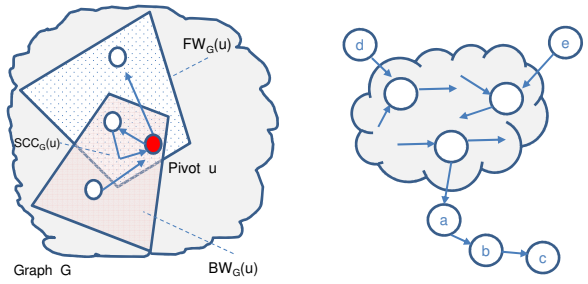
Tarjan's algorithm [27], the classic sequential method for SCC detection, is an asymptotically optimal linear-time algorithm. Unfortunately, Tarjan's algorithm is difficult to parallelize because it extends the depth-first search (DFS) traversal of the graph, which is inherently sequential.

Several studies [13], [22], [9], [8] have investigated parallel or distributed SCC algorithms. Fleischer et al. [13] devised a practical parallel algorithm, the Forward-Backward (FW-BW) algorithm, which motivated further enhancements in following research. The FW-BW algorithm achieves parallelism by partitioning the given graph into three disjoint subgraphs which can be processed independently in a recursive manner. McLendon et al. [22] added a simple extension to this algorithm, the Trim step, which resulted in a significant performance improvement. Barnat et al. [9] proposed another algorithm to improve the degree of parallelism compared to the original FW-BW algorithm. However, their method did not give a large performance improvement over McLendon et al.'s when applied to real-world graphs with few large-sized SCCs [8].

Although these algorithms show a degree of parallel performance in distributed environments, their parallel performance in shared-memory environments is much lower than that of the optimal sequential algorithm, especially when applied to large *real-world* graph instances. As shown in this paper, this is because the characteristics of *real-world* graphs differ substantially from synthetic graphs, such as trees or meshes, for which those algorithms were originally designed. Recent studies [11], [7], [28] have identified several fundamental characteristics of *real-world* graphs, in particular the *small-world* property (Section II-B).

In this paper, we first review McLendon et al.'s parallel algorithm (FW-BW-Trim) before we explain the characteristics of *real-world* graph instances (Section II). Next, we introduce our series of extensions to the conventional FW-BW-Trim algorithm, which account for those characteristics (Section III). In our experiments (Section IV), we run our extended algorithm on a set of *small-world* graph instances and observe the effectiveness of each extension for the characteristics of those instances. Our results show that our methods not only improve the absolute performance of the original FW-BW-Trim algorithm, but also extract a higher degree of parallelism. For interested readers, we discuss details of the implementation in the Appendix.

Our specific contributions are as follows:



(a) (b)

Figure 1. The two main ideas of the conventional FW-BW-Trim algorithm: (a) Forward and Backward reachability and (b) Trimming.

- We identify the performance limitations of the conventional FW-BW-Trim algorithm on large *real-world* graph instances (Sections II and III).
- We propose a set of extensions to the conventional algorithm, which consider characteristics of those *real-world* instances, including the *small-world* property (Section III).
- We analyze the effect of our extensions with varying *small-world* graph shapes (Section IV). To our knowledge, we demonstrate the first parallel SCC algorithm which outperforms Tarjan's algorithm on a shared-memory multi-processor machine on such graphs.

II. Background

A. Conventional FW-BW-Trim Algorithm

In this section, we review FW-BW-Trim, a conventional parallel SCC detection algorithm [22]. The FW-BW-Trim algorithm extends its predecessor, the original FW-BW algorithm [13], by adding the Trim step.

The original FW-BW algorithm is based on the observations in Lemma 1 [13]. Given a directed graph G, let $FW_G(i)$ be the subset of vertices in G which are reachable from vertex i. Let $BW_G(i)$ be the subset of vertices in G from which i is reachable.

Lemma 1. *Let $G = (V, E)$ be a directed graph with $i \in V$ a vertex in G. Then $FW_G(i) \cap BW_G(i)$ is a unique SCC in G. Moreover, for every other SCC s in G, either $s \subset FW_G(i) \setminus BW_G(i)$, $s \subset BW_G(i) \setminus FW_G(i)$, or $s \subset V \setminus (FW_G(i) \cup BW_G(i))$.*

Lemma 1 states that from any node i in graph G, $SCC_G(i)$, the unique SCC that contains i, can be identified from the intersection of two sets: the forward reachable set of i and the backward reachable set of i. Furthermore, the remaining nodes can now be partitioned into three subgraphs (forward reachable only, reverse reachable only, and non-reachable) where each subgraph can be processed independently in a recursive manner. Figure 1(a) provides a visual explanation of this idea.

The parallelism of the FW-BW algorithm comes from its recursive application to each partitioning. Since there cannot be an SCC that belongs to more than one partition, each partition can be processed independently, in parallel. Furthermore, since each partition produces three additional partitions, it is expected that quickly, there would be sufficient independent tasks to consume all of the parallel processing elements in a system.

Parallelism from such independent tasks can be easily exploited via work queues. Note that, however, any of these three partitions can be an empty set; if empty set production is the frequent case, the number of independent tasks may grow more slowly than expected.

Algorithm 1: FW-BW-Trim(G, SCC)

```
Trim(G, SCC)
if |Nodes(G)| = 0 then return;
u ← pick any node in G ;                    /* pivot */
FW ← Forward-Reach(G, u)
BW ← Backward-Reach(G, u)
S ← FW ∩ BW
SCC ← push S
begin in parallel
    FW-BW-Trim(FW \ S, SCC)
    FW-BW-Trim(BW \ S, SCC)
    FW-BW-Trim(G \ (FW ∪ BW), SCC)
end
```

The key observation behind the Trim [22] step is that a trivial SCC (i.e. SCC of size one) is easy to identify: it has either zero incoming edges or zero outgoing edges in the current partition. Therefore, one can easily identify such trivial SCCs only by looking at the number of neighbors, rather than by computing two reachable sets.

The Trim step can be repeated iteratively, since trimming a node can cause other nodes to become trivial SCCs, illustrated in Figure 1(b). In the figure, nodes c, d, and e can be identified as trivial SCCs quickly, as they have zero in- or out- degree and thus cannot form a cycle. The trimming of node c in turn makes node b a trivial SCC, whose trimming also makes node a trivial.

The FW-BW-Trim algorithm is described in Algorithm 1; Algorithm 2 shows details of the Trim step. Although Trim is a simple idea, it greatly improves the performance of the previous FW-BW algorithm, especially for *real-world* graphs [8]. Therefore, to understand its effectiveness, one must comprehend the characteristics of *real-world* graphs.

B. Fundamental Characteristics of Real-World Graphs

Recently, it has been revealed that *real-world graphs* have fundamentally different characteristics than traditional artificial graphs such as trees, meshes, or hypercubes [28], [7], [11], [17]. Real-world graphs are empirical graphs in which no explicit structure has been

Algorithm 2: Trim(G, SCC)

```
repeat
    foreach n ∈ G do
        if In-deg_G(n) = 0 ∨ Out-deg_G(n) = 0 then
            SCC ← push {n}
            G ← G \ {n}
until G not changed
```

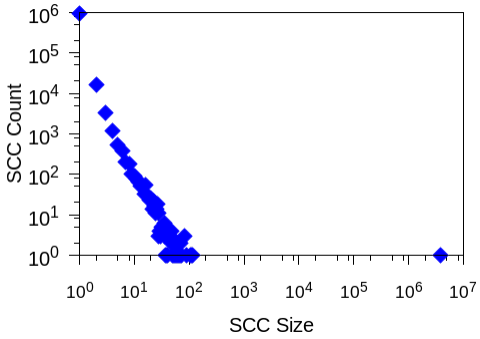


Figure 2. Distribution of SCC sizes in the LiveJournal network.

enforced, but which naturally originate and arbitrarily grow. Examples of such graphs are social networks, web-graphs, citation networks, and protein molecule interaction graphs.

Several interesting properties of these real-world graphs have been identified so far. Of particular importance, the *small-world* property states that the diameter of such graphs is very small even for very large graph instances [28]. This is not a mere observation: it has been shown that by simply re-wiring only a few edges in an arbitrary way, the diameter of any graph rapidly shrinks. This explains why the vast majority of large *real-world* graphs have this property – by nature, they are constructed from arbitrary relationships [28]. [1]

Additionally, in such *real-world* graphs there exists one giant SCC whose size is $O(N)$, where N is the number of nodes in the graph [11]. The remaining SCCs are small-sized, and the distribution of SCC size is skewed such that tiny-sized SCCs are much more frequent than large-sized ones [17].

As an illustrative example, Figure 2 shows a histogram of the SCC sizes in a real-world graph instance, which is the link relationship of a blog sphere named LiveJournal [21]. This figure shows two aforementioned characteristics of *real-world* graph SCC structure: the existence of a single giant SCC and the power-law distribution of SCC sizes. The size of the largest SCC (3,828,682) has the same order as the number of nodes in the graph (4,847,571), and the graph has the same order of size-1 SCCs (947,776). The large number of size-1 SCCs explains why the simple Trim step is so effective for SCC detection – it very quickly identifies size-1 SCCs, which are most prevalent in *real-world* graph instances.

[1]The exceptions are the graphs that represent physical entities, e.g. road networks, because these graphs do not grow arbitrarily. Note that such graphs tend to have limited sizes.

III. Our Extensions

In this section, we discuss our extensions to the conventional FW-BW-Trim algorithm, which account for the characteristics of *real-world* graphs.

A. Baseline Implementation using Parallel Trim

One possible parallelization strategy is to perform the Trim operation in parallel, as shown in Algorithm 3. Note that parallel trimming is applied iteratively, since each trim operation may enable further trimming. To reduce performance overhead, the algorithm performs *marking* and *coloring* of nodes rather than directly modifying the underlying graph.

Algorithm 3: Par-Trim(G, $Color$)

```
repeat
    foreach n ∈ G, n not marked do in parallel
        c ← Color(n)
        /* count neighbors of the same color */
        if In-deg(n,c,Color) = 0 ∨
        Out-deg(n,c,Color) = 0 then
            Color(n) ← a new color; mark n
until Color not changed
```

The Baseline algorithm (Algorithm 4) has two phases: first, the graph is trimmed in parallel, and second, the conventional recursive FW-BW algorithm is applied. Since there are many size-1 SCCs in a *real-world* graph, the parallel trim step achieves a greater degree of parallelism.

Algorithm 4: Baseline(G, $Color$)

```
Par-Trim(G, Color) ;                    /* Parallel Trim */
begin recursion in parallel
    FW-BW(G, Color, SCC) ;   /* Recursive FW-BW */
end
```

B. Method 1: Two-Phase Parallelization

Section II-B introduced two properties of SCC structures in *real-world* graphs: (1) there exists a giant SCC whose size is $O(N)$ and (2) there are many small sized SCCs, where the number of SCCs of a given size decreases drastically as the size grows. Moreover, studies of the SCC structure in *small-world* graphs also revealed that the giant SCC can be considered the *center*, to which most of the other small SCCs are attached [11], [17].

What is the implication of this SCC structure to the performance of the conventional FW-BW-Trim algorithm? First, it causes load imbalance in the algorithm. The conventional implementation of the FW-BW-Trim algorithm lets each thread find one SCC at a time, though there exists one $O(N)$-sized giant SCC in the graph. Worse, it is very likely that this giant SCC is identified at the beginning because other small SCCs are weakly connected to this giant SCC. Consequently, while the large SCC is being identified by one thread, other threads stay idle since there are no other tasks.

Based on the above observations, we adopt another two-phase parallelization strategy. In phase 1, we exploit data-level parallelism, letting every thread work on the same partition of the graph; all threads are used to find reachable sets. In phase 2, we return to the conventional implementation, which exploits task-level parallelism. The transition between phase 1 and phase 2 occurs when the giant SCC has been identified (i.e. an SCC containing, say 1% of the nodes of the original graph), or after a predefined number of iterations.

This strategy is summarized as Method 1 in Algorithm 5. Parallel FW-BW can be implemented with parallel breadth-first search (BFS). Note that a BFS on *small-world* graphs results in a small number of BFS levels, but a large number of nodes in each level that can be visited in parallel [15]. Also, the algorithm applies parallel Trim once more after the Par-FW-BW step, because detection of the giant-SCC presents an opportunity for further trimming.

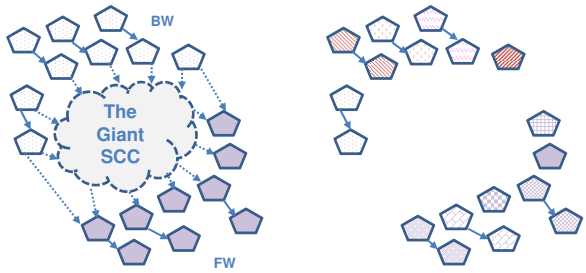
Algorithm 5: Method1(G, $Color$)

```
Par-Trim(G, Color) ;                    /* Parallel Trim */
Par-FW-BW(G, Color) ;                   /* Parallel FW-BW */
Par-Trim(G, Color) ;                    /* Parallel Trim */
begin recursion in parallel
  | FW-BW(G, Color, SCC) ;              /* Recursive FW-BW */
end
```

C. Finding Weakly Connected Components

Method 1 in the previous subsection successfully parallelizes detection of SCCs for most *real-world* graph instances, as will be shown in the experiments (Section IV). This occurs because most of the nodes in *real-world* graphs are processed in a data-parallel phase of the algorithm.

However, the second phase of the algorithm, the recursive FW-BW step, is scarcely parallelized even when a large number of SCCs (e.g. 100,000) are identified in this phase. In fact, especially when a large proportion of nodes are processed in the second phase, such limited parallelism diminishes the overall parallel speedup of Method 1.



(a) (b)

Figure 3. SCC structure of small-world graphs: (a) when the giant SCC is identified and removed, and (b) after the weakly connected component detection algorithm has been applied. Each polygon represents a small-sized SCC. In (a), same color polygons belong to the same set.

The first clue for this phenomenon can be found from the work queue logs; the recorded maximum queue-depth with single threaded execution is only six, indicating insufficient task-level parallelism. This was counter-intuitive at first, because the FW-BW algorithm is designed to produce three more tasks for each task being processed. To understand why, again we must consider the shape of *small-world* graphs.

Figure 3(a) depicts the small SCCs connected around the giant SCC, as in a small-world graph, according to previous studies [11], [17]. Now consider the moment when the giant SCC has been identified by the FW-BW algorithm. Ignoring non-connected SCCs for the time being, the remaining SCCs are grouped into two sets (colors): the FW-set and the BW-set. However, many of these SCCs are not connected to each other. Therefore, recursive application of the FW-BW algorithm to each set (color) will only identify one SCC to which the pivot belongs, but does not provide further partitioning. Consequently, the execution is serialized.

Following is the log of the first five task executions in the recursive FW-BW step when Method 1 is applied to a large graph instance named Flickr (Section IV). The SCC column indicates the size of SCC identified in the iteration, and FW, BW, and Remain indicate the resulting forward, backward, and remaining set sizes, respectively. The log verifies that our observation above indeed occurs in Method 1; each task execution identifies only a small SCC and fails to create additional tasks (i.e. FW set and BW set).

```
SCC FW BW Remain
2   0 0 125432
5   0 0 125427
11  0 0 125416
3   3 0 125410
...
```

The above observation, however, also suggests a way to solve this problem. Once the giant SCC has been identified, the remaining graph is composed of many

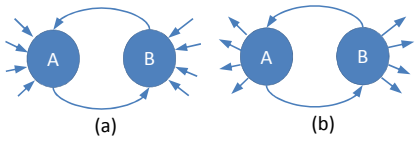


Figure 4. Patterns of size-2 SCCs detected by Trim2. There is a tight cycle between A and B but either (a) there are no other outgoing edges from A and B, or (b) there are no other incoming edges to A and B.

small components that are disconnected from each other. Therefore, we identify all of the weakly connected components (WCCs) over the whole graph in parallel, and assign each WCC a different color. Each WCC becomes a separate entry in the work queue, resulting in a substantial improvement in the degree of task-level parallelism in the recursive FW-BW phase. Figure 3(b) illustrates this idea. Algorithm 6 details how to find weakly connected components in parallel.

Algorithm 6: Par-WCC(G, $Color$)

```
forall n ∈ G, n not marked do in parallel
    WCC(n) ← n
repeat
    forall n ∈ G, n not marked do in parallel
        foreach k ∈ OutNbr(n) do
            if WCC(k) < WCC(n) ∧
              Color(k) = Color(n) then
                WCC(n) ← WCC(k)
    forall n ∈ G, n not marked do in parallel
        k ← WCC(n)
        if k ≠ n ∧ WCC(k) ≠ k then
            WCC(n) ← WCC(k)
until WCC not changed
```

D. Trim2: Fast Detection of Size-2 SCCs

We also add another fast parallel detection mechanism for size-2 SCCs, namely Trim2. The idea is that a large subset of size-2 SCCs can be detected easily by looking only at the neighbors of a given node. Figure 4 illustrates the patterns of size-2 SCCs identified by this algorithm. The algorithm first identifies all of the nodes which have a single neighborhood node that is both an incoming neighbor and an outgoing neighbor, i.e. the shaded nodes in Figure 4. Then the algorithm examines the original node's sole neighbor. If the neighbor has no incoming (or outgoing) edges other than to the original node, the algorithm identifies these two nodes as an SCC because there cannot be any larger cycle that contains both these two nodes. The detailed Trim2 algorithm is summarized in Algorithm 8 in Appendix A.

E. Method 2: Putting It Together

Our Method 2, summarized in Algorithm 7, includes all of the above steps applied one after another. Here, Par-Trim′ includes the application of Par-Trim (iteratively), Par-Trim2 (only once), and Par-Trim (iteratively). We only apply Par-Trim2 once because it is computationally more expensive than Par-Trim.

Algorithm 7: Method2(G, $Color$)

```
Par-Trim(G, Color) ;          /* Parallel Trim */
Par-FW-BW(G, Color) ;         /* Parallel FW-BW */
Par-Trim′(G, Color) ;         /* Parallel Trim′ */
Par-WCC(G, Color) ;           /* Parallel WCC */
begin recursion in parallel
    FW-BW(G, Color, SCC) ;    /* Recursive FW-BW */
end
```

IV. Experiments

In this section, we evaluate the performance of our methods on several large *real-world* graph instances that are available from public repositories [21], [2]. We have chosen graph instances that are large enough to parallelize (i.e. more than 10 million edges). Table I summarizes the size of each graph and provides a short description of the graph instance.

We implement efficiently in C++ our two methods and the Baseline algorithm from Section III and Tarjan's algorithm. There are several pitfalls in implementing these algorithms and a careless implementation might result in an order of magnitude lower performance. We provide implementation notes for interested readers in Appendix A.

All of our experiments were performed on a commodity server-class machine with two Intel Xeon X5550 (2.66GHz) CPUs, each of which has 4 cores and 8 hardware threads. There are in total 8MB of last-level cache and 96 GB of main memory. For all implementations, we used OpenMP for the threading library and compiled our code with g++ version 4.4.6 with the -O3 option. Finally, our servers are running the Ubuntu Linux (2.6.38) operating system.

The plots in Figure 5 summarize the performance of our methods on the real-world graph instances in Table I. The y-axis is the speedup against Tarjan's optimal sequential algorithm.

A first look over all instances (except CA-road, which we will discuss later) reveals that our methods not only improve the performance of the baseline implementation of FW-BW-Trim algorithm, but also exploit a greater degree of parallelism. Excluding CA-road, the speedup result varies from 4.44x (Flickr) to 23.98x (Orkut) using 8 cores and 16 hardware threads. The geometric mean speedup is 9.85. Also, we can see that Method 2

Name	Description	# Nodes	# Edges	Largest SCC Size	Diameter
Livej	Links in LiveJournal (Web) [5],[20]	4,848,571	68,993,773	3,828,682	18
Flickr	Connection of Flickr users (Social) [24]	2,302,925	33,140,018	1,605,184	7
Baidu	Links in Baidu Chinese online encyclopedia (Web) [25]	2,141,300	17,794,839	609,905	5
Wiki	Links in English Wikipedia (Web) [4]	15,172,740	131,166,252	4,736,008	6
Friend*	Connection of Friendster users (Social) [29]	124,836,180	1,806,067,135	46,941,703	25
Twitter	Connection of Twitter users (Social) [18]	41,652,230	1,468,365,182	33,479,734	6
Orkut*	Connection of Orkut users (Social) [29]	3,072,627	11,718,583	2,963,298	8
Patents	Citation among US Patents [19]	3,774,768	16,518,948	1	22
CA-road*	Road network of California [20]	1,965,206	5,533,214	1,168,580	850

Table I
REAL-WORLD GRAPH DATASETS USED IN THE EXPERIMENTS. * INDICATES THAT THE ORIGINAL GRAPH IS UNDIRECTED; WE RANDOMLY ASSIGN A DIRECTION FOR EACH EDGE.

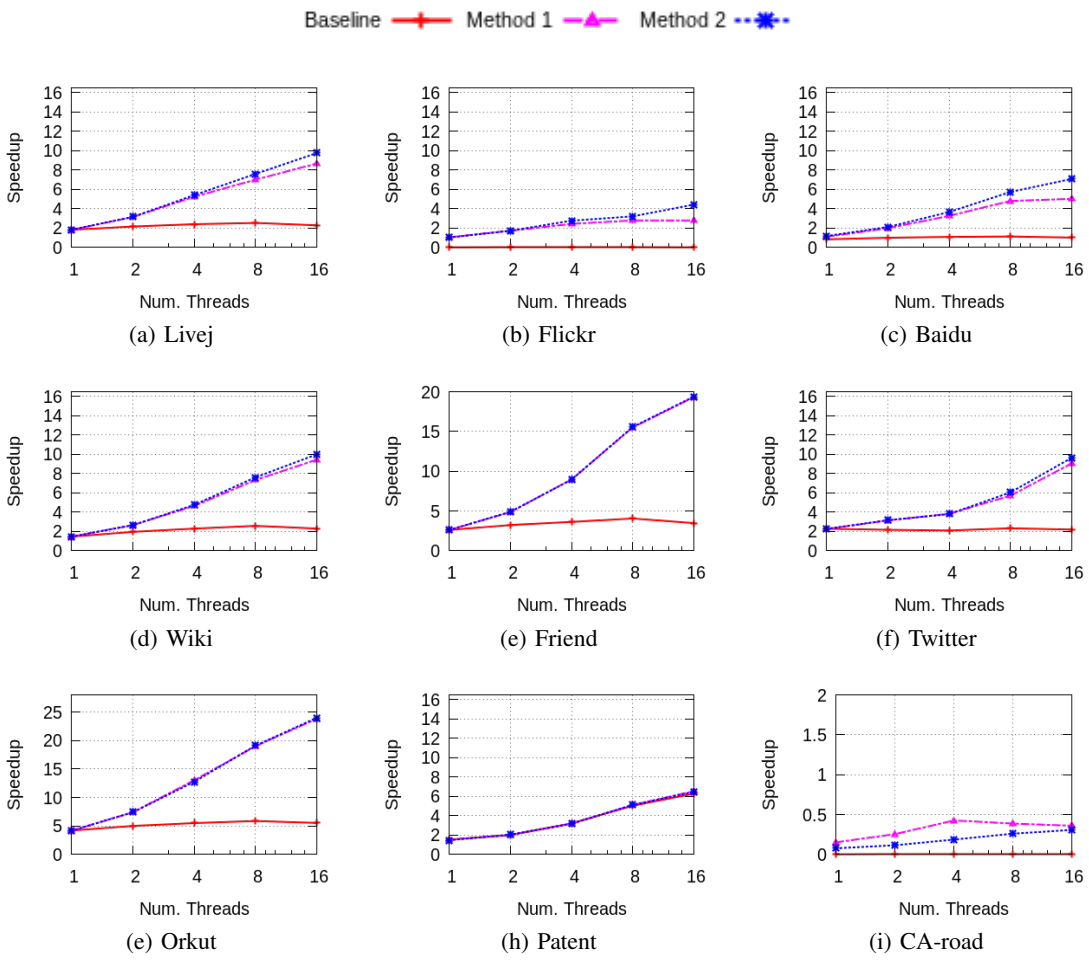


Figure 5. Performance results on real-world graph instances: the y-axis is speedup compared to the optimal sequential algorithm (i.e. Tarjan's).

provides further performance improvement over Method 1 for certain graph instances.

To better understand this performance behavior, we plot the execution time breakdown of each method for all the graph instances in Figure 6. The y-axis in the plots is the execution time measured in milliseconds. Thus, each vertical bar segment represents the time spent in each phase of the algorithm.

Figures 5 and 6 first show that the Baseline method does not scale. As explained in Section III, the gigantic SCC is processed by a singe thread and parallelism is rarely exploited in the recursive FW-BW phase (the topmost segment).

To the contrary, the parallel FW-BW phase of Method 1 (Section III-B) detects the largest SCC of the graph in parallel, which is essential to achieve overall speedup.

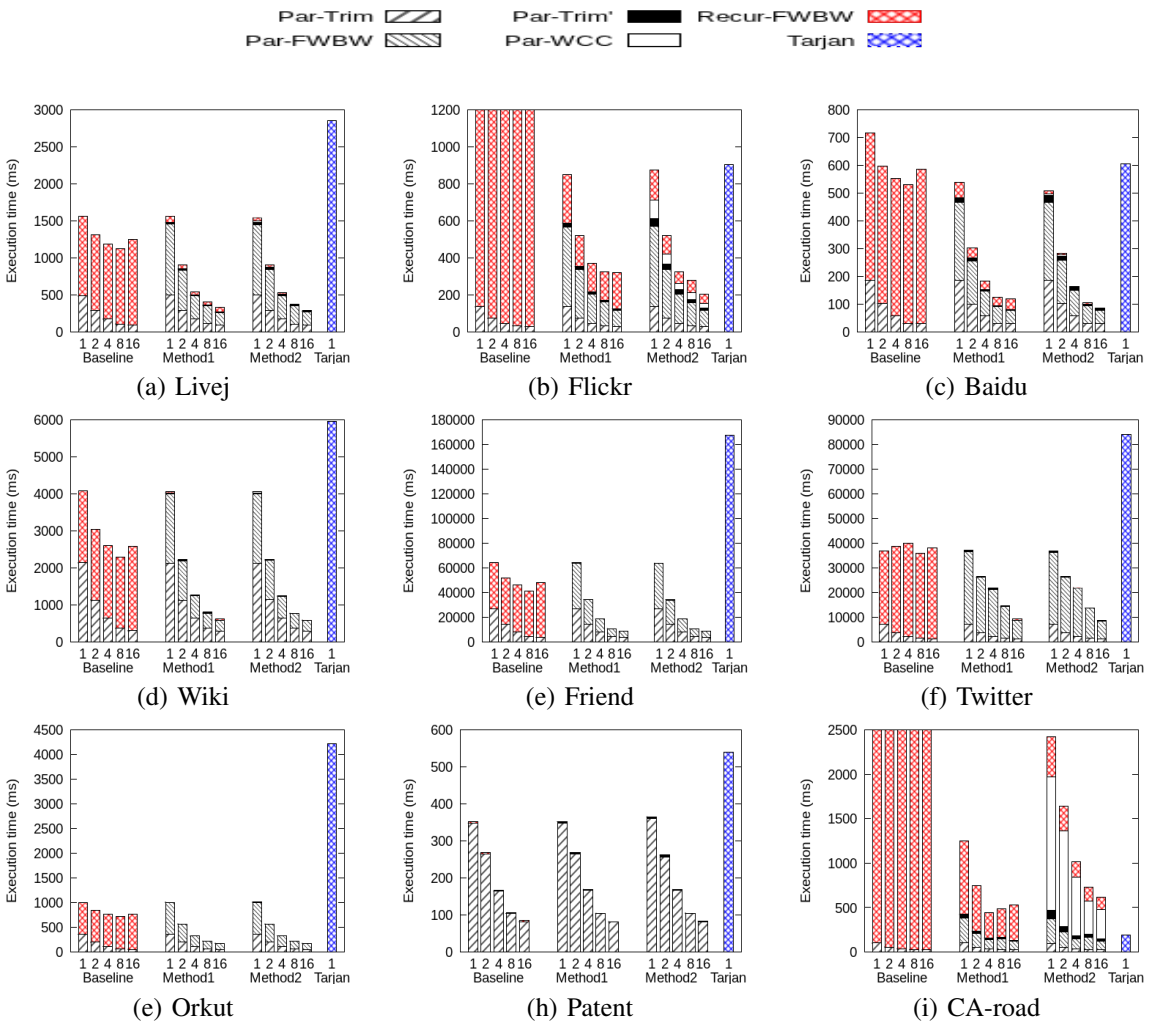


Figure 6. Execution time breakdown for all methods on all graph instances. Par-Trim′ accounts for applying Trim only for Method 1 but applying Trim, Trim2 and Trim in sequence for Method 2.

You can see this in Figure 6, where the second to bottom segments scale down as we increase the number of threads, representing a diminishing fraction of the total execution time. Consequently, Method 1 provides a fair amount of parallel speedup as shown in Figure 5.

Next we look at the cases where Method 2 provides an additional performance benefit over Method 1, including Livej, Flickr, and Baidu. Notice that in Figure 6(b), the execution time of the recursive FW-BW phase (the topmost segment) for Method 1 does not scale down even with more threads. The reason for this phenomenon has been explained in Section III-C: each step in the recursive FW-BW phase does not partition the remaining graph well, failing to provide sufficient parallelism.

Figures 5 and 6 also confirm that Method 2 successfully solves this issue. As can be seen in Figure 6(b), the execution time of the recursive FW-BW phase now scales down in Method 2, after introduction of the parallel WCC phase. Our execution log also confirms that at the beginning of the recursive FW-BW phase there are about 10,000 work items in the queue. Moreover, the parallel WCC phase itself is well parallelized, as its execution time decreases with increasing number of threads.

Therefore, the actual benefits of Method 2 over Method 1 depend on the structure of the graph instance. To illustrate this point, Figure 7 shows the fraction of nodes that are SCC-identified by each phase. Noticeably, the more nodes identified by the recursive FW-BW step, the more performance benefits are achieved by Method 2.

Finally, we discuss the case of the CA-road graph. The graph does not share the same characteristics as the other graph instances because it is (almost) planar by its nature. Therefore, the assumptions that we have made in

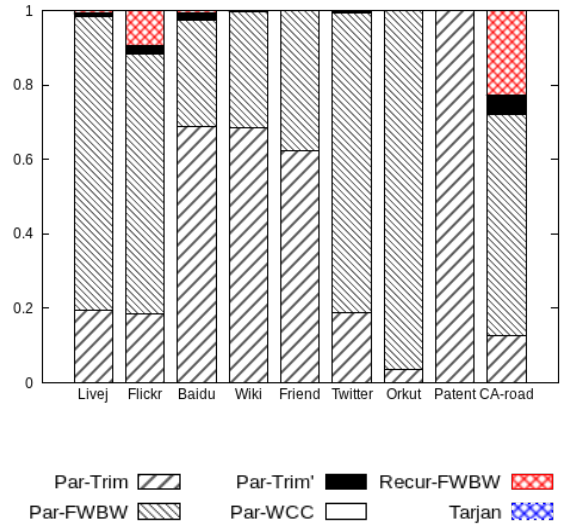


Figure 7. Fraction of nodes whose SCC is identified at each phase of execution.

Section III do not stand for this graph instance. First, the graph has a large diameter (~ 1000) and thus does not possess the small-world property. Second, even though the graph still has a giant SCC, it also has many more large-sized SCCs than small-world graphs (see Figure 8 in Appendix A).

As a result, the parallel FW-BW step provides rather limited parallel speedup in this case because the level-synchronous BFS does not scale up well in such graphs [15]. Moreover, the performance of Method 2 decreases as the execution time of the WCC algorithm increases; the algorithm requires a large number of iterations for convergence when applied on *non-small-world* graphs.

Thus, both methods, although they still scale, do not perform as well as Tarjan's method. Nevertheless, we remind the reader that in the common case, users have a priori knowledge about the property of their graphs, small-world or not. Also, small-world graphs draw more research interest because they are the dominant class of natural large graph instances for many important applications where the graphs are constructed by arbitrary relationships. For example, all of the large graph instances other than the road networks, in public repositories [21], [2] have the *small-world* property.

In summary, our experiments validate the success of our methods in parallelizing SCC detection algorithms for *small-world* graphs because our methods effectively exploit fundamental characteristics of those graphs.

V. Conclusions

In this paper, we analyze the performance shortcomings of the conventional FW-BW-Trim algorithm when applied to *small-world* graph instances. We propose three simple extensions to the conventional algorithm that take advantage of *small-world* graph properties to address the deficiencies in existing algorithms. Consequently, our extensions result in significant parallel and sequential performance improvements on *small-world* graph instances to deliver state-of-the-art parallel SCC detection performance.

References

[1] Graph 500 benchmark. http://graph500.org.

[2] Koblenz network collection. http://konect.uni-koblenz.de/.

[3] S. Allesina, A. Bodini, and C. Bondavalli. Ecological subsystems via graph theory: the role of strongly connected components. *Oikos*, 110(1):164–176, 2005.

[4] SÃűren Auer, Christian Bizer, Georgi Kobilarov, Jens Lehmann, Richard Cyganiak, and Zachary Ives. DBpedia: A nucleus for a web of open data. In *Proc. Int. Semantic Web Conf.*, pages 722–735, 2008.

[5] L. Backstrom, D. Huttenlocher, J. Kleinberg, and X. Lan. Group formation in large social networks: membership, growth, and evolution. In *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 44–54. ACM, 2006.

[6] D.A. Bader and K. Madduri. Snap, small-world network analysis and partitioning: An open-source parallel graph framework for the exploration of large-scale networks. In *IEEE IPDPS*, 2008.

[7] A.L. Barabási and R. Albert. Emergence of scaling in random networks. *Science*, 286(5439):509–512, 1999.

[8] J. Barnat, P. Bauch, L. Brim, and M. Ceška. Computing strongly connected components in parallel on cuda. In *Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International*, pages 544–555. IEEE, 2011.

[9] J. Barnat, J. Chaloupka, and J. van de Pol. Improved distributed algorithms for scc decomposition. *Electronic Notes in Theoretical Computer Science*, 198(1):63–77, 2008.

[10] S. Beamer, K. Asanović, and D. Patterson. Direction-optimizing breadth-first search. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, page 12. IEEE Computer Society Press, 2012.

[11] A. Broder, R. Kumar, F. Maghoul, P. Raghavan, S. Rajagopalan, R. Stata, A. Tomkins, and J. Wiener. Graph structure in the web. *Computer networks*, 33(1):309–320, 2000.

[12] F. Chung and L. Lu. Connected components in random graphs with given expected degree sequences. *Annals of combinatorics*, 6(2):125–145, 2002.

[13] L. Fleischer, B. Hendrickson, and A. Pınar. On identifying strongly connected components in parallel. *Parallel and Distributed Processing*, pages 505–511, 2000.

[14] R. Hojati, R. Brayton, and R. Kurshan. Bdd-based debugging of designs using language containment and fair ctl. In *Computer Aided Verification*, pages 41–58. Springer, 1993.

[15] Sunpack Hong, Tayo Oguntebi, and Kunle Olukotun. Efficient parallel graph exploration for multi-core cpu and gpu. In *IEEE PACT 2011*.

[16] S. Kazemitabar and H. Beigy. Automatic discovery of subgoals in reinforcement learning using strongly connected components. *Advances in Neuro-Information Processing*, pages 829–834, 2009.

[17] R. Kumar, J. Novak, and A. Tomkins. Structure and evolution of online social networks. In *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining KDD 06*, volume 106. ACM Press, 2006.

[18] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. What is Twitter, a social network or a news media? In *Proc. Int. World Wide Web Conf.*, pages 591–600, 2010.

[19] J. Leskovec, J. Kleinberg, and C. Faloutsos. Graphs over time: densification laws, shrinking diameters and possible explanations. In *Proceedings of the eleventh ACM SIGKDD international conference on Knowledge discovery in data mining*, pages 177–187. ACM, 2005.

[20] J. Leskovec, K.J. Lang, A. Dasgupta, and M.W. Mahoney. Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters. *Internet Mathematics*, 6(1):29–123, 2009.

[21] Jure Leskovec. Stanford network analysis library. http://snap.stanford.edu/snap.

[22] W. McLendon III, B. Hendrickson, S.J. Plimpton, and L. Rauchwerger. Finding strongly connected components in distributed graphs. *Journal of Parallel and Distributed Computing*, 65(8):901–910, 2005.

[23] D. Merrill, M. Garland, and A. Grimshaw. Scalable gpu graph traversal. In *Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, pages 117–128. ACM, 2012.

[24] Alan Mislove, Hema Swetha Koppula, Krishna P. Gummadi, Peter Druschel, and Bobby Bhattacharjee. Growth of the flickr social network. In *Proceedings of the 1st ACM SIGCOMM Workshop on Social Networks (WOSN'08)*, August 2008.

[25] Xing Niu, Xinruo Sun, Haofen Wang, Shu Rong, Guilin Qi, and Yong Yu. Zhishi.me – weaving Chinese linking open data. In *Proc. Int. Semantic Web Conf.*, pages 205–220, 2011.

[26] N. Satish, C. Kim, J. Chhugani, and P. Dubey. Large-scale energy-efficient graph traversal: a path to efficient data-intensive supercomputing. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, page 14. IEEE Computer Society Press, 2012.

[27] R. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972.

[28] D.J. Watts and S.H. Strogatz. Collective dynamics of small-world networks. *Nature*, 393(6684), 1998.

[29] J. Yang and J. Leskovec. Defining and evaluating network communities based on ground-truth. In *Proceedings of the ACM SIGKDD Workshop on Mining Data Semantics*, page 3. ACM, 2012.

APPENDIX

Algorithm 8: Par-Trim2(G, $Color$)

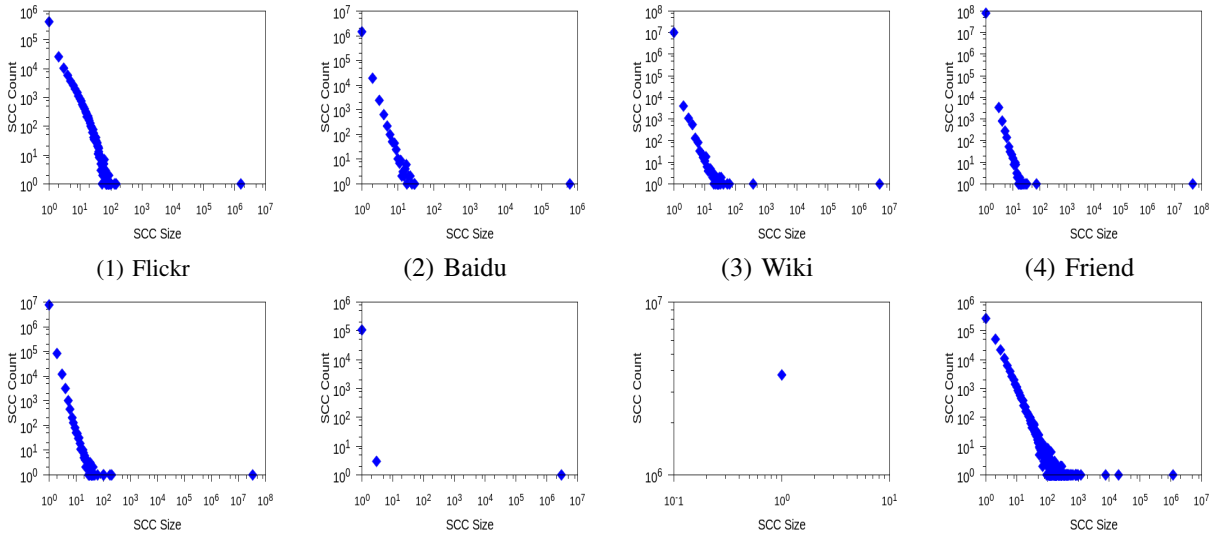
foreach $n \in G$, n *not marked* **do** in parallel
  if $|InNbr_G(n)| = 1$ **then**
    $k \leftarrow$ the only $InNbr_G(n)$
    if $k \in OutNbr_G(n) \wedge |InNbr_G(k)| = 1$ **then**
      $Color(n, k) \leftarrow$ a new color; mark n, k;
  if $|OutNbr_G(n)| = 1$ **then**
    $k \leftarrow$ the only $OutNbr_G(n)$
    if $k \in InNbr_G(n) \wedge |OutNbr_G(k)| = 1$ **then**
      $Color(n, k) \leftarrow$ a new color; mark n, k;

Algorithm 8 shows the details of the Trim2 operation introduced in Section III-D. Unlike Trim, which is applied multiple times iteratively, we apply Trim2 only once since it is computationally more expensive. According to our experiments, although the Trim2 step itself provides only a marginal speedup, it sometimes reduces the execution time of the following WCC step by up to 50% because it can cut out a long chain of weakly connected size-2 SCCs. For this reason, we include Trim2 for Method 2 only.

Figure 8 shows the SCC structure of all graphs used in the experiments (Section IV). Notice that there is a single giant connected component whose size is $O(N)$, the most frequent SCCs are size one, and there are SCCs of other sizes in between, for all graph instances except Patent.

Patent is a special case with no cycles in the graph. However, this is a natural phenomenon due to the way the graph is constructed: a patent can only cite other patents that come before it, thus preventing any cycles. Note that the SCC structure of this graph was identified by the Trim operation (Section IV).

CA-road also shows a noticeably different distribution, since it is not a *small-world* graph. Having a large diameter, the graph has many more non-trivial SCCs



(1) Flickr (2) Baidu (3) Wiki (4) Friend

(5) Twitter (6) Orkut (7) Patent (8) CA-road

Figure 8. Distribution of SCC-sizes in experimental graph instances.

than the other graphs. Moreover, the size of these SCCs is larger as well.

In this section, we discuss implementation issues for all of the algorithms discussed in this paper.

Graph and Set Representation

We implemented all of the algorithms in the paper using C++. For the in-memory graph data structure, we used the compressed sparse row (CSR) adjacency matrix data structure, which uses two arrays to represent the graph. Note that CSR is favored in high performance graph analysis problems [6], [15], [8] because it is compact, memory bandwidth-friendly, and thus best suited for graph traversals.

The CSR representation, however, performs poorly when modifying the graph structure itself. Therefore, instead of actually removing nodes that are trimmed or whose SCCs are identified, we maintain an extra $O(N)$ integer array named $Color$. Whenever a node is trimmed or its SCC identified, we assign a special color (e.g. -1) to the position of the node in the array. In the rest of the algorithm, nodes with this special color value are always ignored.

The same $Color$ array is used to represent disjoint sets of nodes; i.e. the FW-set and BW-set in Algorithm 1. Nodes having the same *color*, or a unique number, are considered to be in the same set; this is represented conceptually by coloring each set of the graph with a different color. Therefore, neighborhood nodes whose color is different from the current node are considered detached.

Still, this approach presents an issue when selecting a pivot from a specific set (Algorithm 1). To pick out any one node in a specific subset (i.e. *color*), the complete *Color* array must be scanned, which is a very expensive operation.

Therefore, we take a hybrid approach; we maintain both the set representation (i.e. `std::set`) and the *Color* array. The former representation is used to choose pivots in the FW-BW step, while the latter representation is used to look up membership of a set. According to our experiments, such a hybrid approach resulted $\sim$10x better performance than using one representation only.

In addition, we also maintain a compact representation for non-marked nodes, i.e. nodes whose SCC has not yet been identified. This compact representation is used by Algorithms 3, 6, and 8 when those algorithms iterate over all nodes that are not marked. Again, we use a hybrid data structure for this representation: a vector of nodes and a boolean array. The vector representation is updated only when each trimming operation has successfully cut out more than 10% of the previously remaining nodes.

Implementing Tarjan's and FW-BW Algorithms

The classic Tarjan's SCC algorithm is based on a DFS (depth-first search) traversal of the graph. However, the required recursion depth for DFS traversal is the size of the largest SCC, which is $O(N)$ for large real-world graphs. Thus, one must increase the size of the program stack accordingly, to hundreds of MBs or even a few GBs. Moreover, Tarjan's algorithm requires an extra stack (other than the program stack) on which the `IsIn` operation is applied. Again, we use both a vector and a boolean array for this stack for fast execution.

For the reachable set computation in the parallel FW-BW step, we used an efficient implementation of the breadth-first search (BFS) order graph traversal [15], [10]. Note that after the advent of the graph500 benchmark suite [1], many efficient implementations of the

BFS traversal have been proposed [23], [26], which may improve our performance results even further.

On the other hand, for the same computation in the recursive FW-BW step, we use DFS instead of BFS. This is because the BFS implementation above, optimized for parallel traversal, has a larger fixed cost than simple sequential DFS. Also, during reachable set exploration in the parallel FW-BW step, we do not maintain an unbounded set representation (i.e. `std::set`), but use the *Color* array only. This is based on the following observations: (1) the traversal will go through a huge fraction of nodes in the graph (i.e. $O(N)$) and thus the size of each set (FW-set, BW-set, and remaining set) will be large as well, and (2) those sets will be modified by the following trimming and compacting operations. Therefore, we defer the construction of sets until the end of the trimming phase, when we perform a scan of non-marked nodes to construct the initial work items.

Worker Threads and the Work Queue

For the threading library, we used OpenMP for all experiments. As a reminder, we exploited data-level parallelism in the first phase of our algorithms, but task-level parallelism in the second phase. The data-level parallelism is implemented with the `parallel for` statement, and the task-level parallelism with a custom work queue implementation.

For the data-level parallelism, however, it was critical to specially handle the workload imbalance problem. Note that there is another fundamental characteristic of *real-world* graphs, the scale-free property, which means that the graph's degree distribution follows a power law [7]. In other words, there exist a few nodes that have a huge number of neighbors while many nodes have only a few neighbors. Therefore, statically assigning the same number of nodes to each thread naturally induces workload imbalance if the work involves neighborhood exploration. Thus, we used dynamic load-balancing for the components that involve neighborhood exploration, but static workload distribution otherwise.

For the task-level parallelism, we used our custom work queue implementation, which is composed of two levels of queues: a global queue and per-thread private queues. Initially, each thread fetches up to K work items from the global queue into its local queue; whenever the local queue becomes empty, it fetches more work from the global work queue. On the other hand, a newly generated work item goes to a local queue first. When the size of local queue grows to $2K$, K items are moved to the global queue. We set K to 1 for the Baseline and Method 1, because those algorithms suffer from a lack of task level parallelism; for Method 2, we set K to 8.