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 5.01x to 29.41x parallel speedup over the optimal sequential algorithm with 16 cores and 32 hardware threads.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming—*parallel programming*; G.2.2 [**Discrete Mathematics**]: Graph Theory—*graph algorithms*

General Terms

Algorithms, Performance

Keywords

strongly connected components (SCC), multicore, parallel algorithms, graph algorithms, small-world graphs

1. 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].

SC '13, November 17-21, 2013, Denver, CO, USA

http://dx.doi.org/10.1145/2503210.2503246

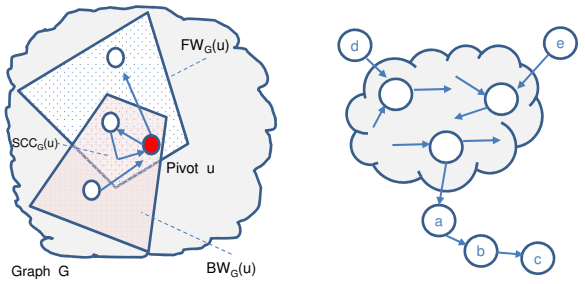
Tarjan's algorithm [28], the classic sequential method for SCC detection, is an asymptotically optimal linear-time algorithm. Unfortunately, Tarjan's algorithm is difficult to parallelize because it extends the depth-first search (DFS) traversal of the graph, which is inherently sequential [26].

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 the recursive OBF algorithm to improve the degree of parallelism compared to the original FW-BW algorithm. However, their method [8] did not give a large performance improvement over McLendon et al.'s when applied to real-world graphs with few large-sized SCCs. Barnat et al. [8] demonstrate a CUDA implementation based on forward reachability that outperforms the sequential Tarjan's algorithm, but concede that none of their implementations on a quad-core system were able to outperform Tarjan's algorithm.

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. Studies [11, 7, 29] have identified several fundamental characteristics of *real-world* graphs, in particular the *small-world* property (Section 2.2).

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 2). Next, we introduce our series of extensions to the conventional FW-BW-Trim algorithm, which account for those characteristics (Section 3). We discuss issues in implementing these algorithms, which can significantly impact performance (Section 4). In our experiments (Section 5), 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 algo-



(a) (b)

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

rithm, but also extract a higher degree of parallelism.

Our specific contributions are as follows:

- We identify the performance limitations of the conventional FW-BW-Trim algorithm on large *real-world* graph instances (Sections 2 and 3).
- We propose a set of extensions to the conventional algorithm, which consider characteristics of those *real-world* instances, including the *small-world* property (Section 3).
- We explain the performance-critical implementation details of the conventional algorithm and extensions (Section 4).
- We analyze the effect of our extensions with varying *small-world* graph shapes (Section 5). To our knowledge, we demonstrate the first parallel SCC algorithm which outperforms Tarjan's algorithm on a shared-memory multiprocessor machine on such graphs.

2. BACKGROUND

2.1 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, which detects size-1 SCCs to improve performance.

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, where we call i the pivot node. 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 computational complexity of the FW-BW algorithm is $O(n + m)$ for each partition, which detects a single SCC [8].

The parallelism of the FW-BW algorithm comes from its recursive application to each partition. 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, where each task in the queue can be assigned to an available compute element. Note that any of these three partitions of the graph can be an empty set; if empty set production is the frequent case, the number of independent tasks may grow more slowly than expected.

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, which is computationally more expensive.

2.2 Fundamental Characteristics of Real-World Graphs

The Trim step can be repeated iteratively, since trimming a node can cause other nodes to become trivial SCCs. Figure 1(b) illustrates this idea. 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.

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

```
In-Out: G: a graph (a subgraph of the original input graph)
In-Out: SCC: a collection of node sets; each set corresponds
        to an SCC of the original graph
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 ← SCC ∪ {S}
begin in parallel
    FW-BW-Trim(FW \ S, SCC)
    FW-BW-Trim(BW \ S, SCC)
    FW-BW-Trim(G \ (FW ∪ BW), SCC)
end
```

Algorithm 2: Trim(G, SCC)

```
In-Out: G: a graph (a subgraph of the original input graph)
In-Out: SCC: a collection of node sets; each set corresponds
        to an SCC of the original graph
repeat
    foreach n ∈ G do
        if In-degree_G(n) = 0 ∨ Out-degree_G(n) = 0 then
            SCC ← SCC ∪ {{n}}
            G ← G \ {n}
until G not changed
```

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.

Recently, it has been revealed that *real-world* graphs have fundamentally different characteristics than traditional artificial graphs

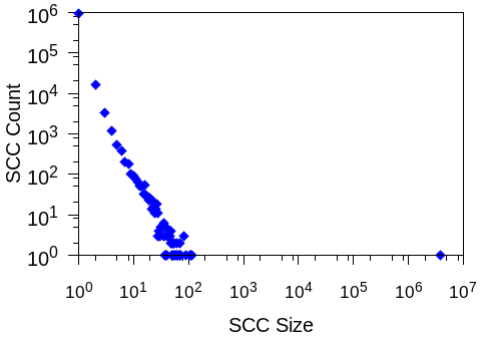


Figure 2: Distribution of SCC sizes in the LiveJournal network.

such as trees, meshes, or hypercubes [29, 7, 11, 17]. Real-world graphs are empirical graphs in which no explicit structure has been 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 [29]. 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 [29]. [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 [19]. 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.

3. 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.

3.1 Baseline Implementation using Parallel Trim

We prepare an efficient implementation of the conventional FW-BW-Trim algorithm and set it as our baseline (Algorithm 3); it has a few small enhancements over Algorithm 1. These improvements include parallelization of the Trim step, the use of additional data structures to avoid directly modifying the input graph, and a work queue to support parallelism in recursion, described below.

[1]On the other hand, graphs that represent physical entities, e.g. road networks, do not have the *small-world* property; Note that it is not allowed to add an edges between any two arbitrary nodes in such graphs. Also such graphs tend to have rather limited sizes.

Algorithm 3: Baseline(G, SCC)

Input : G, the original input graph
Output: SCC, a collection of node sets
Local : $Color$, color value assigned to each node in G
Local : $mark$, boolean value assigned to each node in G
/* initialization */
$\forall\, n \in G$: $Color(n) \leftarrow 0, mark(n) \leftarrow false$

Par-Trim(G, SCC, $Color$, $mark$)
Recur-FWBW(G, 0, SCC, $Color$, $mark$)
until *work queue is empty* **do in parallel**
  $c \leftarrow$ pop a color from the work queue
  Recur-FWBW(G, c, SCC, $Color$, $mark$)

Algorithm 4: Par-Trim(G, SCC, $Color$, $mark$)

Input : G, the original input graph
In-Out: SCC, a collection of node sets
In-Out: $Color$, color value assigned to each node in G
In-Out: $mark$, boolean value assigned to each node in G
repeat
  foreach $n \in G$, $mark(n) = false$ **do** in parallel
    /* only count neighbors with same color as n */
    if *In-degree(n,Color) = 0* $\vee$
    Out-degree(n,Color) = 0 **then**
      $Color(n) \leftarrow$ -1
      $SCC \leftarrow SCC \cup \{\{n\}\}$; $mark(n) \leftarrow true$
until $Color$ *not changed*

The Baseline algorithm (Algorithm 3) has two phases: first, it performs the Trim operation in parallel on multiple disconnected nodes, shown in Algorithm 4, and second, it applies the conventional recursive FW-BW algorithm, shown in Algorithm 5, using a work queue. Since there are many size-1 SCCs in a *real-world* graph, the parallel trim step greatly increases the degree of parallelism by identifying these SCCs before executing the FW-BW algorithm. Although Par-Trim is invoked once at the beginning of Algorithm 3, the actual trimming is iteratively applied inside the Par-Trim kernel. The example in Figure 1(b) demonstrates this idea: the trimming of nodes c, d, and e can be completed in parallel, followed by iterative trimming of nodes b and a.

For performance reasons, we do not mutate the input graphs directly. Instead, we use two auxiliary data structures: *mark* and *Color*. When the SCC of a node is identified, instead of detaching the node from the rest of the graph, we simply set the *mark* value of the node to *true*, and the node is considered detached thereafter. Similarly, when we partition the graph, we assign the same *Color* value to nodes belonging to the same partition; each partition is assigned a unique *Color* value. Therefore, two nodes of different *Color* values are considered disconnected, even when there exists an edge between them in the original graph. Algorithm 4 and Algorithm 5 show how these data structures are used, and their implementation is discussed in Section 4.

Recursion parallelism for the Recur-FW-BW kernel is implemented through a work queue. At the end of Algorithm 5, the three remaining partitions (c, c_{fw}, c_{bw}), other than the newly identified SCC, are pushed into the shared work queue. Every worker thread in the system grabs a partition (i.e. *Color*) from the work queue and processes it concurrently with respect to other worker threads. The program is finished when all the workers become idle and no work items remain in the queue.

Algorithm 5: Recur-FWBW(G, c, SCC, $Color$, $mark$)

Input : G, the original input graph
Input : c, a color value
In-Out: SCC, a collection of node sets
In-Out: $Color$, color value assigned to each node in G
In-Out: $mark$, boolean value assigned to each node in G
$pivot \leftarrow$ choose a random node $s.t.\ Color(pivot) = c$
if $pivot$ = *NIL* **then return**
$c_{fw}, c_{bw}, c_{scc} \leftarrow$ a new color value (each)
$S \leftarrow \emptyset$
traverse G from $pivot$ using forward edges
  when visiting *node* n **do**
    if $Color(n) = c$ **then**
      $Color(n) \leftarrow c_{fw}$
    else prune traversal beyond n;
end
traverse G from $pivot$ using reverse edges
  when visiting *node* n **do**
    if $Color(n) = c$ **then**
      $Color(n) \leftarrow c_{bw}$
    else if $Color(n) = c_{fw}$ **then**
      $S \leftarrow S \cup \{n\}$; $mark(n) \leftarrow True$
      $Color(n) \leftarrow c_{scc}$
    else prune traversal beyond n;
end
$SCC \leftarrow SCC \cup \{S\}$
push c, c_{fw}, c_{bw} into the work queue

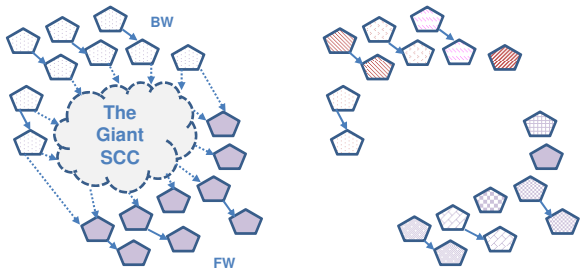
3.2 Method 1: Two-Phase Parallelization

Section 2.2 introduced two important 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? Most of all, it causes workload 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, all the 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 6. We omit the detailed description of Par-FWBW since it is almost identical to Algorithm 5 except that the traversal of the graph is implemented with parallel breadth-first search, and the parallel BFS is repeated until the giant SCC (e.g. an SCC containing more than 1% of nodes) is identified or given the maximum number of trials. 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-FWBW step because detection of the giant SCC may present an opportunity for further trimming.



(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.

Algorithm 6: Method1(G, SCC)

Input : G, the original input graph
Output: SCC, a collection of node sets
Local : $Color$, color value assigned to each node in G
Local : $mark$, boolean value assigned to each node in G
/* Initialization */
$\forall\ n \in G$: $Color(n) \leftarrow 0, mark(n) \leftarrow false$

/* Phase 1: parallelism in trims and traversals */
Par-Trim(G, SCC, $Color$,$mark$)
Par-FWBW(G, 0, SCC, $Color$, $mark$)
Par-Trim(G, SCC, $Color$,$mark$)

/* Phase 2: parallelism in recursion */
until *work queue is empty* **do in parallel**
  $c \leftarrow$ pop a color from the work queue
  Recur-FWBW(G, c, SCC, $Color$, $mark$)

3.3 Finding Weakly Connected Components

Method 1 in the previous subsection successfully parallelizes detection of SCCs for most *real-world* graph instances, as shown in the experiments (Section 5). 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.

The first clue to explain this phenomenon was found in 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) illustrates a typical SCC structure of small-world

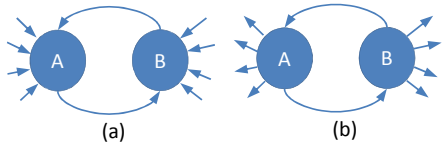


Figure 4: Patterns of size-2 SCCs detected by Trim2. There is a tight cycle between A and B but either (a) there are no other outgoing edges from A and B, or (b) there are no other incoming edges to A and B.

graphs according to previous studies [11, 17], where the small SCCs are connected around the giant SCC. 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 5). The SCC column indicates the size of the 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 and BW sets).

```
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 small components that are disconnected from each other. Groups of one or more of these disconnected SCCs form weakly connected components (WCCs), where a WCC is defined as a maximal group of nodes that are mutually reachable by converting directed edges to undirected edges. Therefore, we identify all of the weakly connected components over the whole graph in parallel, and assign each WCC a different color. Then, 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 7 details how to find weakly connected components in parallel. Once the WCCs are identified, they are pushed into the work queue. We use *Color* and *mark* in the same way as in Algorithm 4, i.e. the Par-WCC algorithm assigns a node's out-neighbors to its WCC only when they are the same color.

3.4 Trim2: Fast Detection of Size-2 SCCs

We also add a 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. nodes A and B 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 nodes.

Algorithm 7: Par-WCC(G, $Color$, $mark$)

Input : G, the original input graph
In-Out: $Color$, color value assigned to each node in G
In-Out: $mark$, boolean value assigned to each node in G
Local : WCC, head-node value assigned to each node in G
forall $n \in G$, $mark(n) = false$ **do** in parallel
  $WCC(n) \leftarrow n$
repeat
  foreach $n \in G$, $mark(n) = false$ **do** in parallel
    foreach $k \in$ *OutNbr*(n), $Color(k) = Color(n)$ **do**
      if $WCC(k) < WCC(n)$ **then**
        $WCC(n) \leftarrow WCC(k)$
  foreach $n \in G$, $mark(n) = false$ **do** in parallel
    $k \leftarrow WCC(n)$
    if $k \neq n \ \wedge \ k \neq WCC(k)$ **then**
      $WCC(n) \leftarrow WCC(k)$
until WCC *not changed*
foreach $n \in G$, $WCC(n) = n$ **do** in parallel
  $c \leftarrow$ a new color
  foreach $k \in G$, $WCC(k) = n$ **do**
    $Color(k) \leftarrow c$
  push color c into the work queue

The detailed Trim2 algorithm is summarized in Algorithm 8. Unlike Trim, which is applied multiple times iteratively, we apply Trim2 only once since it is computationally more expensive. Our experiments revealed that the Trim2 step provides only a marginal speedup by itself; however it reduces the execution time of the following WCC step by up to 50% because it cuts out a chain of weakly connected size-2 SCCs. For this reason, we include Trim2 only for Method 2, described in Section 3.5.

3.5 Method 2: Putting It Together

Our Method 2, summarized in Algorithm 9, includes all of the above steps applied in sequence. 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. The primary difference between Method 1 and Method 2 is the inclusion of the Par-Trim2 and Par-WCC steps. The performance differences between the two methods are discussed in Section 5.

4. ISSUES IN IMPLEMENTATION

We implement efficiently in C++ our two methods and the Baseline algorithm from Section 3 as well as Tarjan's algorithm. There are several pitfalls in implementing these algorithms and a careless implementation could result in an order of magnitude lower performance.

4.1 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) format, which uses two arrays to represent the graph. A $O(N)$-sized array stores a pointer to the beginning of each node's adjacency list, stored in a single $O(M)$-sized array (see Figure 5). Note that CSR is favored in high performance

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

Input : G, the original input graph
In-Out: SCC, a collection of node sets
In-Out: $Color$, color value assigned to each node in G
In-Out: $mark$, boolean value assigned to each node in G
foreach $n \in G$, $mark(n) = false$ **do** in parallel
  if *In-degree(n,Color) = 1* **then**
    $k \leftarrow$ the only *InNbr* of n
    if $k \in OutNbr(n) \wedge$ *In-degree(k,Color) = 1* **then**
      $Color(n), Color(k) \leftarrow$ -1
      $mark(n), mark(k) \leftarrow true$
      $SCC \leftarrow SCC \cup \{\{n, k\}\}$
  else if *Out-degree(n,Color) = 1* **then**
    $k \leftarrow$ the only *OutNbr* of n
    if $k \in InNbr(n) \wedge$ *Out-degree(k,Color) = 1* **then**
      $Color(n), Color(k) \leftarrow$ -1
      $mark(n), mark(k) \leftarrow true$
      $SCC \leftarrow SCC \cup \{\{n, k\}\}$

Algorithm 9: Method2(G, $Color$)

Input : G, the original input graph
Output: SCC, a collection of node sets
Local : $Color$, color value assigned to each node in G
Local : $mark$, boolean value assigned to each node in G
/* Initialization */
$\forall\, n \in G$: $Color(n) \leftarrow 0, mark(n) \leftarrow false$

/* Phase 1: parallelism in trims, traversals and WCC */
Par-Trim(G, SCC, $Color$, $mark$)
Par-FWBW(G, 0, SCC, $Color$, $mark$)
Par-Trim′(G, SCC, $Color$, $mark$)
Par-WCC(G, $Color$,$mark$)

/* Phase 2: parallelism in recursion */
until *work queue is empty* **do in parallel**
  $c \leftarrow$ pop a color from the work queue
  Recur-FWBW(G, c, SCC, $Color$, $mark$)

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 the extra data structures $mark$ and $Color$.

$mark$ is a O (N) boolean array which represents the nodes whose SCCs are identified and thus are detached from the original graph. Our algorithm always ignores nodes whose $mark$ value is $true$; therefore, setting the $mark$ value of a node has the same effect as removing the node from the graph representation.

Similarly, $Color$ is a O (N) integer array which represents the current partitioning of the graph. That is, all the nodes in a (non-trivial) partition of the graph have the same *color* value, and a unique *color* value is assigned to each partition. For instance, the construction of the FW-set and BW-set in Algorithm 5 simply assigns the same *color* value to every reachable node. Therefore, neighborhood nodes whose color is different from the current node are considered detached.

However, this approach presents an issue when selecting a pivot from a specific set (Algorithm 5). To select any single node of a specific *color*, the complete *Color* array must be scanned; this

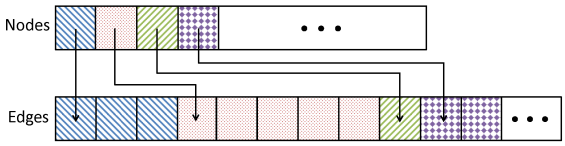


Figure 5: CSR adjacency matrix data structure: the node array stores pointers to each node's adjacency list in the edge array.

becomes a very expensive operation when there exist only a few nodes of that *color*.

To solve this issue, we adopt a hybrid representation. That is, while constructing FW-set and BW-set in Algorithm 5, we maintain a compact representation (i.e. `std::set`) for each set, in addition to the *Color* array. The former is used to choose pivots in the FW-BW step, while the latter is used to look up membership of a set. However, we use the hybrid representation only for phase 2 (i.e. Recur-FWBW) but not for phase 1, since the set size of each color is sufficiently large in phase 1. Our experiments revealed that such a hybrid approach resulted in ∼10x better performance than using one representation only.

4.2 Implementing Graph Traversals

The classic Tarjan's SCC algorithm is based on a depth-first search (DFS) 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 additional stack (other than the program stack) on which it places nodes in the order in which they are visited; the algorithm must check if a node is in this stack. Like the $Color$ array and `std::set` representations described in 4.1, we implement this stack using both a vector and a boolean array 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, 27], 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.

4.3 Managing Parallel Work Items

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 using 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 spe-

Name	Description	# Nodes	# Edges	Largest SCC Size	Diameter
Livej	Links in LiveJournal (Web) [5],[21]	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) [30]	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) [30]	3,072,627	11,718,583	2,963,298	8
Patents	Citation among US Patents [20]	3,774,768	16,518,948	1	22
CA-road*	Road network of California [21]	1,965,206	5,533,214	1,168,580	850

Table 1: Real-world graph datasets used in the experiments. * indicates that the original graph is undirected; we randomly assign a direction for each edge with 50% probability for each direction. The graph diameters are estimated from a random sampling of nodes; the actual diameters are likely somewhat larger due to outlier nodes.

cially 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, more work is fetched from the global queue. Each newly generated work item goes to a local queue first. When the size of a 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 these algorithms suffer from a lack of task level parallelism; for Method 2, we set K to 8.

5. EXPERIMENTS

In this section, we evaluate the performance of our methods on several large *real-world* graph instances that are available from public repositories [19, 2]. We have chosen graph instances that are large enough to parallelize (i.e. more than 10 million edges). Table 1 summarizes the size of each graph and provides a short description of the graph instance.

All of our experiments were performed on a commodity server-class machine with two Intel Xeon E5-2660 (2.20GHz) CPUs, each of which has 8 cores and 16 hardware threads. There are in total 20 MB of last-level cache and 256 GB of main memory. For all implementations, we used OpenMP for the threading library and compiled our code with g++ version 4.4.7 with the -O3 option. Finally, our servers are running the CentOS Linux (6.4 Final) operating system.

The plots in Figure 6 summarize the performance of our methods on the real-world graph instances in Table 1. The y-axis is the speedup against Tarjan's optimal sequential algorithm, and the log scale x-axis is the number of threads.

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 the FW-BW-Trim algorithm, but also exploit a greater degree of parallelism. Excluding CA-road, the speedup result varies from 5.01x (Flickr) to 29.41x (Twitter) using 16 cores and 32 hardware threads. The geometric mean speedup is 14.05x. Also, we can see that Method 2 provides further performance improvement over Method 1 for certain graph instances.

We remind the reader that the machine has two CPU sockets, where each CPU has 8 cores only. As a result, there is a natural knee in Figure 6 between 8 threads and 16 threads, since the latter crosses the socket boundary, i.e. NUMA effect. Similarly, there is another knee between 16 threads and 32 threads, because the latter exploits simultaneous multithreading (SMT) using two hardware threads in each physical core.

To better understand the performance behavior shown in Figure 6, we plot in Figure 7 the execution time breakdown of each method for all of the graph instances. 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 6 and 7 first show that the Baseline method does not scale. As explained in Section 3, a singe thread processes the gigantic SCC in each graph, thus the recursive FW-BW phase (the topmost segment) rarely exploits parallelism.

To the contrary, the parallel FW-BW phase of Method 1 (Section 3.2) detects the largest SCC of the graph in parallel, which is essential to achieve overall speedup. You can see this in Figure 7, where the second to bottom segments (Par-FWBW) 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 6.

Next we look at the cases where Method 2 provides an additional performance benefit over Method 1, including Livej, Flickr, Baidu, and Twitter. Notice that in Figure 7(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 3.3: each step in the recursive FW-BW phase does not partition the remaining graph well, failing to provide sufficient parallelism.

Figures 6 and 7 also confirm that Method 2 successfully solves this issue. As can be seen in Figure 7(b), the execution time of the recursive FW-BW phase now scales down in Method 2, due to 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, providing sufficient task-level parallelism. 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 8 shows the fraction of nodes whose SCCs are identified by each phase. Noticeably, the more nodes identified by the recur-

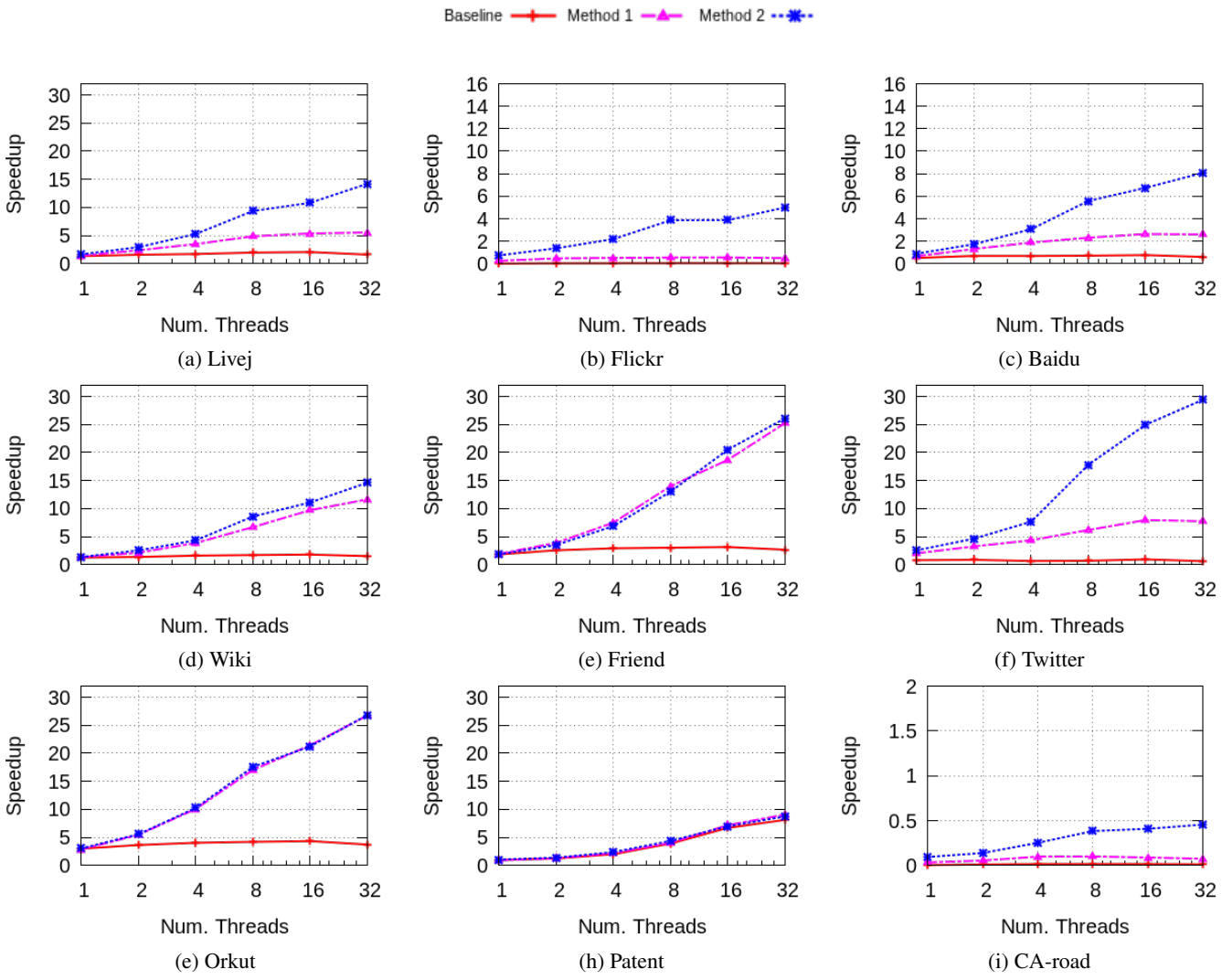


Figure 6: Performance results on real-world graph instances. The y-axis is speedup compared to the optimal sequential algorithm (i.e. Tarjan's). The x-axis is in log scale. Note that each of two CPU sockets has only 8 cores; 16-thread execution exploits two sockets and 32-thread execution uses simultaneous multithreading. The Baseline (Algorithm 3) uses parallel trim and the recursive FW-BW algorithm; Method 1 (Algorithm 6) utilizes two-phase parallelization (data-level and task-level); Method 2 (Algorithm 9) adds parallel trim2 and parallel WCC.

sive 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 Section 3 do not stand for this non-small-world 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 9).

Figure 9 shows the SCC structure of all graphs used in the experiments discussed in this section. 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. These in-between-sized SCCs differentiated scalability between Method1 and Method2 (Figure 6), as described in Sections 3.3 and 3.4.

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:

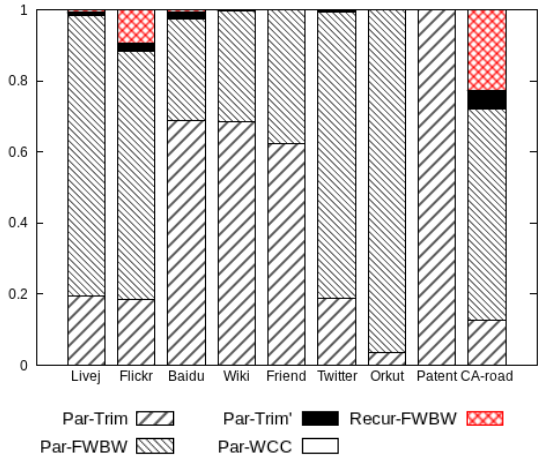


Figure 8: Fraction of nodes whose SCC is identified at each phase of execution for Method 2.

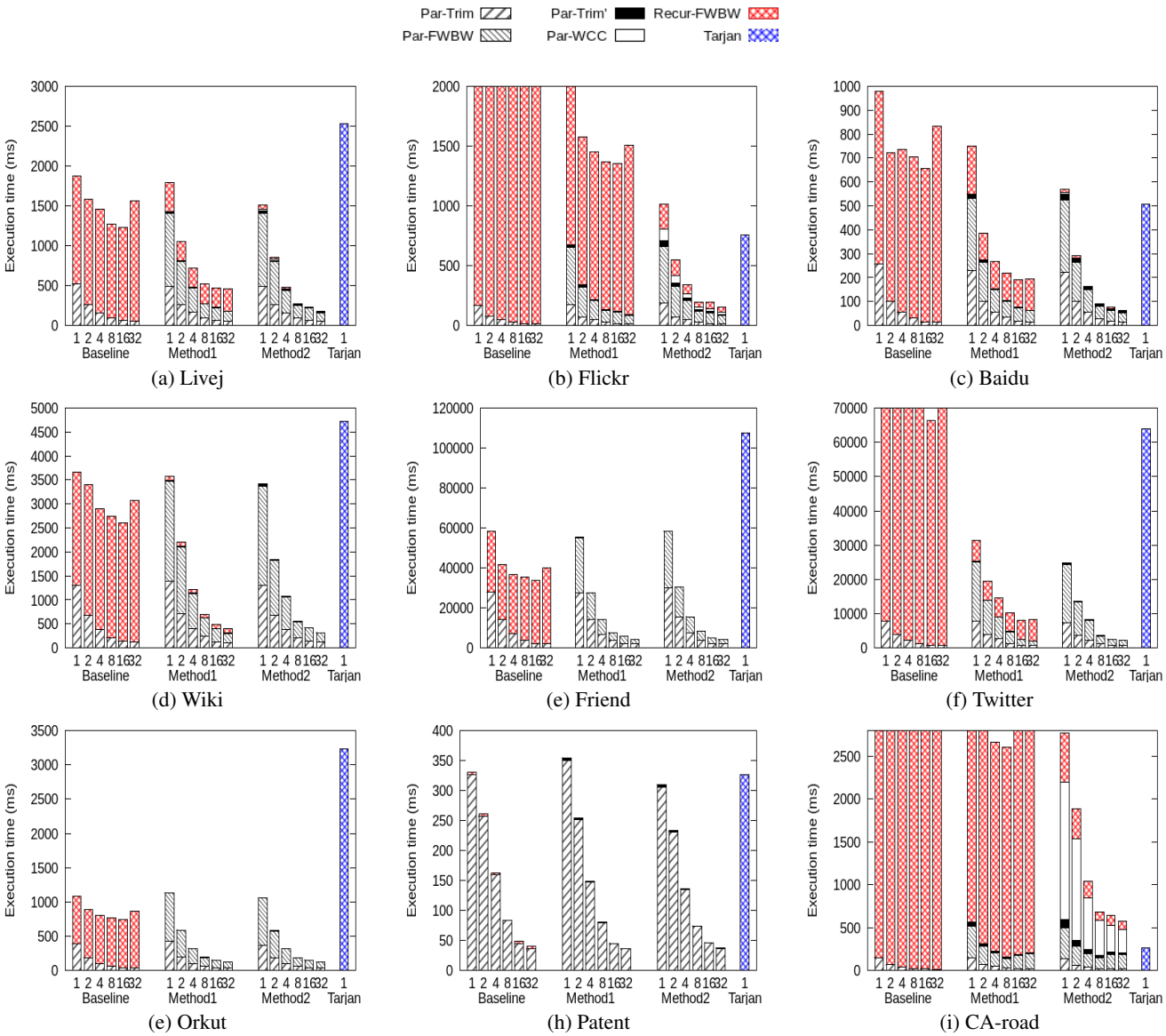


Figure 7: Execution time breakdown for all methods on all graph instances. Par-Trim′ accounts for applying Trim only for Method 1 but applying Trim, Trim2 and Trim in sequence for Method 2.

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.

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

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 for CA-road. 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 [19, 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.

6. 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. Conse-

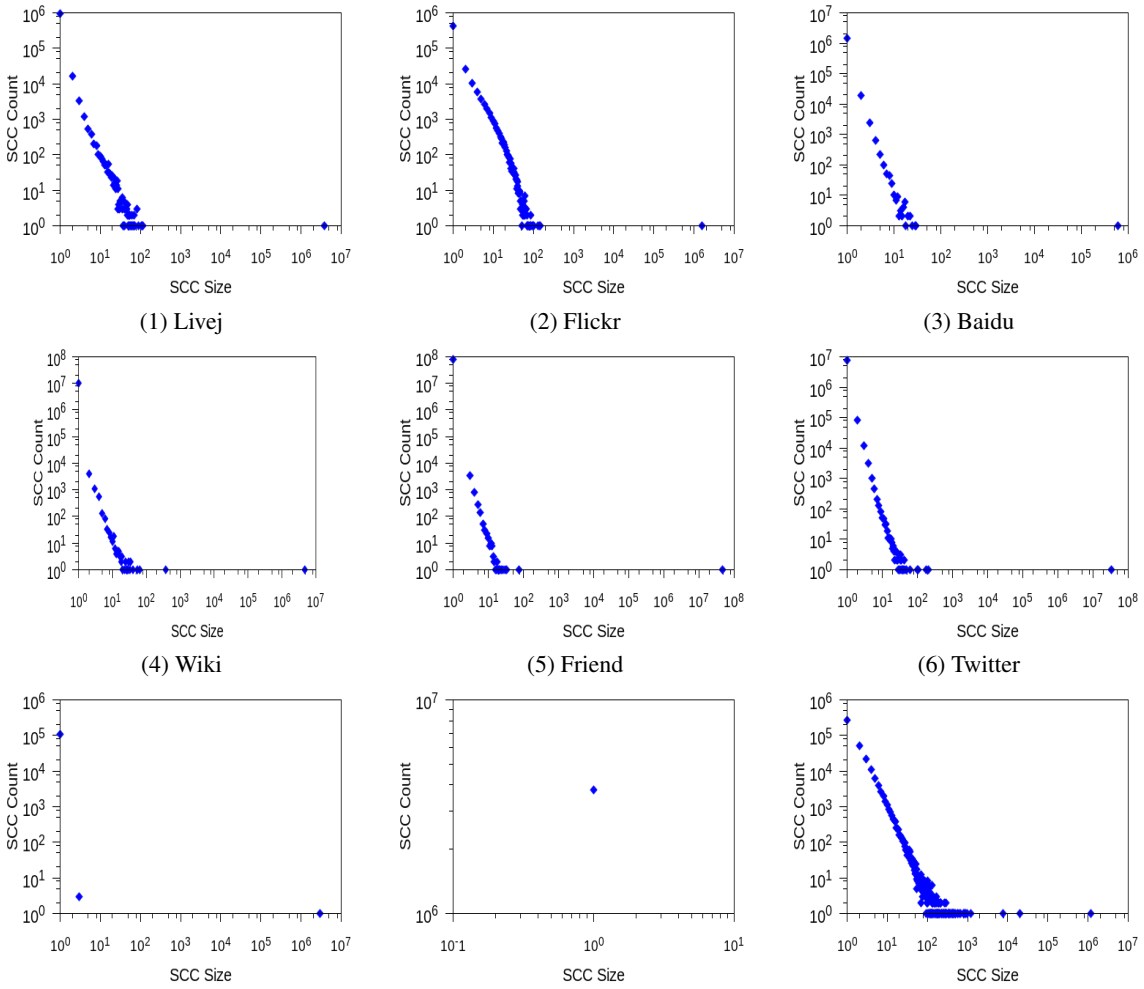


Figure 9: Distribution of SCC sizes of the graph instances that are used in our experiments.

quently, 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.

As a next step, we plan to implement our algorithm in a distributed environment. Our extensions can be easily implemented in such an environment as they only require data from direct neighbors.

7. ACKNOWLEDGEMENTS

This research is supported by DARPA contract, Oracle order US1226344; DARPA Contract, SEEC: Specialized Extremely Efficient Computing, Contract #HR0011-11-C-0007; DARPA Contract, Xgraphs; Language and Algorithms for Heterogeneous Graph Streams, FA8750-12-2-0335; Army contract AHPCRC W911NF-07-2-0027-1; NSF grant, BIGDATA: Mid-Scale: DA: Collaborative Research: Genomes Galore - Core Techniques, Libraries, and Domain Specific Languages for High-Throughput DNA Sequencing, IIS-1247701; NSF grant, SHF: Large: Domain Specific Language Infrastructure for Biological Simulation Software, CCF-1111943; Stanford PPL affiliates program, Pervasive Parallelism Lab: Oracle, AMD, Intel, NVIDIA, and Huawei. Authors also acknowledge additional support from Oracle. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DARPA or the U.S. Government.

8. 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] S. Auer, C. Bizer, G. Kobilarov, J. Lehmann, R. Cyganiak, and Z. 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. 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. 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] S. Hong, T. Oguntebi, and K. 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] H. Kwak, C. Lee, H. Park, and S. 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. Stanford network analysis library. `http://snap.stanford.edu/snap`.

[20] 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.

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

[22] W. McLendon III, B. Hendrickson, S. 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] A. Mislove, H. S. Koppula, K. P. Gummadi, P. Druschel, and B. Bhattacharjee. Growth of the flickr social network. In *Proceedings of the 1st ACM SIGCOMM Workshop on Social Networks (WOSN'08)*, August 2008.

[25] X. Niu, X. Sun, H. Wang, S. Rong, G. Qi, and Y. Yu. Zhishi.me – weaving Chinese linking open data. In *Proc. Int. Semantic Web Conf.*, pages 205–220, 2011.

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

[27] 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.

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

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

[30] 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*, pages 3:1–3:8, 2012.