BFS and Coloring-based Parallel Algorithms for Strongly Connected Components and Related Problems*

George M. Slota[1], Sivasankaran Rajamanickam[2], and Kamesh Madduri[1]
[1]The Pennsylvania State University
[2]Sandia National Laboratories
Email: gslota@psu.edu, srajama@sandia.gov madduri@cse.psu.edu

Technical Report CSE #14–004

February 26, 2014

Abstract

Finding the strongly connected components (SCCs) of a directed graph is a fundamental graph-theoretic problem. Tarjan's algorithm is an efficient serial algorithm to find SCCs, but relies on the hard-to-parallelize depth-first search (DFS). We observe that implementations of several parallel SCC detection algorithms show poor parallel performance on modern multicore platforms and large-scale networks. This paper introduces the Multistep method, a new approach that avoids work inefficiencies seen in prior SCC approaches. It does not rely on DFS, but instead uses a combination of breadth-first search (BFS) and a parallel graph coloring routine. We show that the Multistep method scales well on several real-world graphs, with performance fairly independent of topological properties such as the size of the largest SCC and the total number of SCCs. On a 16-core Intel Xeon platform, our algorithm achieves a 20× speedup over the serial approach on a 2 billion edge graph, fully decomposing it in under two seconds. For our collection of test networks, we observe that the Multistep method is 1.92× faster (mean speedup) than the state-of-the-art Hong et al. SCC method. In addition, we modify the Multistep method to find connected and weakly connected components, as well as introduce a novel algorithm for determining articulation vertices of biconnected components. These approaches all utilize the same underlying BFS and coloring routines.

*Extended technical report version of the conference paper to appear in the Proceedings of the 28th IEEE International Parallel & Distributed Processing Symposium (IPDPS).

1 Introduction

The problem of strongly connected components (SCCs) refers to detection of all maximal strongly connected subgraphs in a large directed graph. Informally, a strongly connected subgraph is a subgraph in which there is a path from every vertex to every other vertex. SCC decomposition is a useful preprocessing and data reduction strategy when analyzing large web graphs [1] and networks constructed from online social network data [2]. SCC detection also has several uses in formal verification tools, including model checking in state space graphs [3]. Other application areas include computer-aided design [4] and scientific computing [5].

Tarjan's algorithm [6] is an efficient serial algorithm for SCC detection. It uses a recursive depth first search (DFS) to form a search tree of explored vertices. The roots of subtrees of the search tree form roots of strongly connected components. Although it is possible to extract parallelism from DFS while retaining proper vertex ordering, this is often met with limited success [7]. Therefore, most parallel SCC algorithms have avoided its use.

The Forward-Backward (FW-BW) method [8] and Orzan's coloring method [3] are two SCC detection algorithms that are amenable to both shared-memory and distributed-memory implementation. These methods use very different subroutines and were proposed in different contexts, FW-BW for graphs arising in scientific computing and coloring in the context of formal verification tool design. We observe that shared-memory parallelization of both methods perform poorly in comparison to the serial algorithm for SCC decomposition in social networks and web crawls.

In this work, we also present parallel algorithms for problems related to SCC detection: connected components (CC), weakly connected components (WCC), and biconnected components (BiCC). Known efficient serial and parallel algorithms for these problems can be very different from SCC detection algorithms. However, we observe that the approach we present, the **Multistep method**, is easily extended to solve these problems as well.

1.1 Contributions

Our new Multistep method is designed for SCC detection in large real-world graphs, such as online social networks and web crawls, using current shared-memory multicore platforms. We utilize variants of FW-BW and Orzan's coloring methods in subroutines. We minimize synchronization and avoid use of fine-grained locking. Our BFS subroutine incorporates several recently-identified optimizations for low-diameter graphs and multicore platforms [9, 10, 11]. We perform an extensive experimental study on a 16-core Intel Xeon server with Sandy Bridge processors, and the following are our main observations:

- For low-diameter networks (e.g., Twitter, LiveJournal crawls), the single-threaded Multistep approach is significantly faster than the serial Tarjan algorithm.
- Multistep is faster and exhibits better scaling than our implementations of the FW-BW and coloring algorithms.

- Multistep is up to 8.9× faster than the state-of-the-art Hong et al. method [12] on ItWeb, a network with a large number of SCCs (30 million).
- Multistep modified for CC is consistently faster than the coloring-based algorithm implementation in Ligra [13].
- Our novel BiCC algorithm demonstrates up to an 8× parallel speedup over a serial DFS-based approach.
- Our modified atomic-free and lock-free BFS averages a traversal rate of 1.4 GTEPS (Giga traversed edges per second) over all tested networks.

2 Background

2.1 Strongly Connected Components

There are several existing serial and parallel algorithms that are used to determine the SCCs of a graph. This section will give a general overview of a number of important ones. Serial methods include Tarjan's and Kosaraju's algorithms, while parallel methods are the Forward-Backward algorithm and Coloring.

2.1.1 Serial Algorithms

Two common serial algorithms used for SCC detection are Tarjan's [6] and Kosaraju's [14] algorithms. Both of these algorithms perform linear $O(n + m)$ work in the RAM model of computation, where n is the number of vertices and m is the number of edges in an input graph. However, since Tarjan's algorithm only requires a single DFS as opposed to Kosaraju's two, it is often faster in practice.

2.1.2 Forward-Backward

The Forward-Backward (FW-BW) algorithm [8] (see Algorithm 1) can be described as follows. Let V denote the set of vertices in the graph, $E(V)$ the set of outgoing edges, and $E'(V)$ the set of incoming edges. Given the graph $G(V, E(V))$, a *pivot* vertex u is selected. This can be done either randomly or through simple heuristics. A BFS (or DFS) search is conducted starting from this vertex to determine all vertices which are reachable from u (the *forward sweep*). These vertices form the *descendant* set (D). Another BFS is performed from u, but on $G(V, E'(V))$. This search (the *backward sweep*) will find the set (P) of all vertices than can reach u, called the *predecessor* set. The intersection of these two sets forms an SCC ($S = D \cap P$) that has the pivot u in it. If we remove all vertices in S from the graph, we can have up to three remaining disjoint vertex sets: $(D \setminus S), (P \setminus S)$, and the *remainder* R, which is the set of vertices that we have not explored during either search from u. The FW-BW algorithm can then be recursively called on each of these three sets.

Algorithm 1 Forward-Backward Algorithm

```
1: procedure FW-BW(V)
2:     if V = ∅ then
3:         return ∅
4:     Select a pivot u ∈ V
5:     D ← BFS(G(V, E(V)), u)
6:     P ← BFS(G(V, E'(V)), u)
7:     R ← (V \ (P ∪ D)
8:     S ← (P ∩ D)
9:     new task do FW-BW(D \ S)
10:    new task do FW-BW(P \ S)
11:    new task do FW-BW(R)
```

There is parallelism on two levels. As the three sets are disjoint, they can each be explored in parallel. Also, note that we do not require any vertex ordering within each set, just reachability. Therefore, each of the forward and backward searches can be easily parallelized or run concurrently. For graphs with bounded constant vertex degree, FW-BW is shown to perform $O(n \log n)$ expected case work.

A routine called *trimming* is commonly performed before executing FW-BW. The trimming procedure was initially proposed as an extension to FW-BW [15] to remove all trivial SCCs. The procedure is quite simple: all vertices that have an in-degree or out-degree of zero (excluding self-loops) are removed. Trimming can also be performed recursively, as removing a vertex will change the effective degrees of its neighbors. In this paper, we refer to a single iteration of trimming as just *simple* trimming and iterative trimming as *complete* trimming. This procedure is very effective in improving the performance of the FW-BW algorithm, but can be beneficial for other approaches as well.

2.1.3 Coloring

The coloring algorithm for SCC decomposition is given in Algorithm 2. This algorithm is similar to FW-BW in that it uses forward and backward traversals. However, the approach is also quite different, as it uses multiple pivots in the forward phase and only looks at a subset of edges for each pivot in the backward phase.

Assume that the graph vertices are numbered from 1 to n. The algorithm starts by initializing elements of the array $Colors$ to these vertex identifiers. The values are then propagated outward from each vertex in the graph, until there are no further changes to $Colors$. This effectively partitions the graph into disjoint sets. As we initialized $Colors$ to vertex identifiers, there is a unique vertex corresponding to every distinct c in $Colors$. We consider $u = c$ as the root of a new SCC, SCV_c. The set of reachable vertices in the backward sweep from u of vertices of the same color(V_c) belong to this SCV_c. We then remove all these vertices from V and proceed to the next color/iteration. The two subroutines amenable to parallelization are the color propagation step and the backward sweep. In a graph with a

Algorithm 2 Coloring Algorithm

```
1: procedure ColorSCC(G(V, E))
2:     while G ≠ ∅ do
3:         for all u ∈ V do Colors(u) ← u
4:         while at least one vertex has changed colors do
5:             for all u ∈ V in parallel do
6:                 for all ⟨u, v⟩ ∈ E do
7:                     if Colors(u) > Colors(v) then
8:                         Colors(v) ← Colors(u)
9:         for all unique c ∈ Colors in parallel do
10:            V_c ← {u ∈ V : Colors(u) = c}
11:            SCV_c ← BFS(G(V_c, E'(V_c)), u)
12:            V ← (V \ SCV_c)
```

very large SCC and high diameter, the color of the root vertex has to be propagated to all of the vertices in the SCC, limiting the efficiency of the color propagation step.

2.1.4 Other Parallel SCC Approaches

There has been other recent work aimed at improving FW-BW and coloring. One example is the OBF algorithm [16] of Barnat et al., which, like coloring, aims to further decompose the graph into multiple distinct partitions at every iteration. The OBF decomposition step can be performed much quicker than coloring. However, it does not necessarily result in as many partitions. Barnat et al. implement OBF, FW-BW, and coloring on NVIDIA GPUs [17], and demonstrate considerable speedup over equivalent CPU implementations.

More recently, Hong et al. [18, 12] present several improvements to the FW-BW algorithm and trimming procedure by expanding trimming to find both 1-vertex and 2-vertex SCCs, decomposing the graph after the first SCC is found by partitioning based on weakly connected components, and implementing a dual-level task-based queue for the recursive step of FW-BW to improve runtimes by reducing overhead for the task-based parallelism. We present detailed comparisons to their approach in Section 6.

2.2 Connected and Weakly Connected Components

The connected components of an undirected graph are the maximal subgraphs where every vertex in the subgraph has a path to every other vertex. Weakly connected components in a directed graph are the equivalent to connected components in undirected graphs if edge directivity is ignored.

The approaches for determining connected components and weakly connected components in graphs are similar. There are two primary parallel methods, using techniques similar to those described in the preceding sections. First, a parallel BFS can be used for connected components. Any vertices reachable by the BFS traversal will be in the same

component. We continue selecting new unvisited vertices as BFS roots until all vertices have been visited and all connected components identified. The procedure is the same for weakly connected components, but it is required to examine both in and out edges.

We can also use a coloring approach. Each vertex is initialized with a unique color, and the maximal colors are propagated throughout the network. Once the colors reach a stable point, all vertices contained in each discrete component will have the same color. The number of propagation iterations is bounded by $O(\log n)$ when using the pointer-jumping technique. There is also load balancing phase to handle giant connected components [19].

2.3 Biconnected Components

The biconnected components (BiCCs) of a graph are the maximal biconnected subgraphs. which are subgraphs that will remain connected on the removal of any given vertex in the subgraph. Determining biconnectivity of a graph is especially important in networking and routing to determine redundancy.

The optimal linear-time sequential algorithm for determining the biconnected components of a graph is the Hopcroft-Tarjan algorithm, which is based on DFS [20]. The Tarjan-Vishkin parallel algorithm [21] avoids DFS. It constructs an auxiliary graph such that the connected components of the auxiliary graph form biconnected components in the original graph. Several improvements have since been made to the Tarjan-Vishkin approach to reduce work and improve parallelism [22, 7].

3 Multistep Method For SCC Decomposition

In this section, we introduce our algorithm for graph SCC decomposition, the Multistep method. The name comes from the fact that it is a combination of some of the previously described parallel algorithms, with algorithmic changes to each one of them, stepped through in a certain order. This section gives some of the details of our algorithm and justifies the algorithmic choices we have made.

3.1 Observations

The FW-BW algorithm can be quite efficient if a graph has relatively small number of large and equally-sized SCCs, as the leftover partitions in each step would, on average, result in similar amounts of task-parallel work. The FW and BW searches could also be efficiently parallelized in this instance.

However, the structure of most real-world graphs is very different. Most real-world graphs have one giant SCC containing a large fraction of the total vertices and a large number of small SCCs [2]. The small SCCs are often disconnected once the large SCC is removed. Running a naïve implementation of FW-BW would result in a large work imbalance after the large SCC is found, as most vertices will be in the remainder set when discovering the small SCCs. In addition, using a naïve task-parallel model would add considerable overhead,

as each new task will be finding a SCC with only a few vertices. As a result, the overall runtime of the FW-BW algorithm is dominated by the total number of SCCs in the initial graph. As we will show, even implementations that use a smarter tasking model [18, 12] can still suffer when the graph and number of SCCs get large enough.

Conversely, the coloring algorithm is quite efficient when the graph has a large number of small and disconnected SCCs. The runtime of each coloring step is proportional to the diameter of the largest connected component in the graph. This leads to poor initial performance on real-world graphs, as the time for each step can be very high when the largest SCCs remain, and there is no guarantee that these SCCs will be removed in any of the first few iterations.

It is also important to note that when the number of vertices in the graph is less than a certain threshold, all parallel algorithms perform poorly over Tarjan's or Kosaraju's serial algorithms due to parallel overhead.

3.2 Description of Method

Based on the above observations, we have developed the Multistep method that aims at maximizing the advantages and minimizing the drawbacks of prior algorithms. The four primary phases are shown in Algorithm 3. We apply simple trimming (MS-SimpleTrim), a FW-BW stage (steps 4-7 of Algorithm 3), coloring (MS-Coloring, steps 8-10), and Tarjan's algorithm in sequence to decompose large real-world graphs into their SCCs.

We only use simple trimming during the first phase, as experiments have shown that vertices trimmed in second or subsequent iterations during complete trimming can be better handled by the coloring or serial phases.

Algorithm 3 Multistep Algorithm

1: **procedure** MULTISTEP($G(V, E)$)
2: $T \leftarrow$ MS-SimpleTrim(G)
3: $V \leftarrow V \setminus T$
4: Select $v \in V$ for which $d_{in}(v) * d_{out}(v)$ is maximal
5: $D \leftarrow \text{BFS}(G(V, E(V)), v)$
6: $S \leftarrow D \cap \text{BFS}(G(D, E'(D)), v)$
7: $V \leftarrow V \setminus S$
8: **while** NumVerts(V) $> n_{\text{cutoff}}$ **do**
9: $C \leftarrow$ MS-Coloring($G(V, E(V))$)
10: $V \leftarrow V \setminus C$
11: Tarjan($G(V, E(V))$)

In the second phase, we try to increase the chance that our initial pivot is in the largest SCC by selecting as pivot the vertex that has the largest product of its in degree and out degree. Although there is no guarantee that this will be the case, the heuristic works very well in practice with real-world graphs.

With the chosen pivot, we perform a forward and backward traversal to find one SCC. Unlike FW-BW, we do not explicitly compute the three sets D, P, and R. We modify the backward search from BFS($G(V, E'(V), v)$ to BFS($G(D, E'(D)), v$). If we encounter a vertex that is not in the descendent set D, we can safely ignore it and avoid exploring its predecessors, as they will not be part of the current SCC. For a simple proof by contradiction, given a pivot v and its predecessor p_i that is not a descendant of v (as p_i was not encountered in forward search) assume p_i has a predecessor p_j that is part of the SCC with v in it. If p_j is part of the SCC, then p_j is a descendant of v, and in turn p_i will be a descendant of v as p_j is its predecessor. This allows us to safely ignore p_i and all of its predecessors in this phase. For certain graphs, this optimization considerably reduces the vertices we inspect during the backward phase.

In the third phase, we pass all remaining vertices to MS-Coloring. We use coloring until the number of remaining vertices goes below a certain threshold. In our final phase, we use Tarjan's serial algorithm to compute the rest of the SCCs.

3.3 Work Complexity

The execution time of Multistep is dependent on the performance of its constituent subroutines. The upper bound for FW-BW is known to be $O(mn)$ [8], since each recursive step will remove at least one of the n vertices (after potentially examining all n vertices), and there can be at most n recursive invocations of FW-BW. In the best case, the graph is fully strongly connected and only two complete searches are performed for an overall $O(n + m)$ work. In Multistep, we perform two searches in the FW-BW stage to find one SCC and do not recurse. Hence the work bound for this stage will be $O(n + m)$.

Consider the trimming subroutine. For the worst case graph instance of a chain, naïve complete trimming will trim one vertex over each of n iterations while examining n vertices, resulting in an upper bound of $O(n^2)$ work. Multistep instead uses simple trimming, or performing only a single pass, resulting in $O(n)$ work regardless of input structure. As trimming does not need to explicitly examine edges, the m term is omitted.

The upper bound for coloring is $O(n^2)$, and this occurs when only a single vertex is removed per iteration. The best case instance only requires a single color propagation step and a single search over n vertices and m edges, resulting in a $O(n + m)$ bound. In general, the work performed is dependent on the number of coloring iterations and the number of edges touched in each iteration. Multistep uses a predefined cutoff to switch to the serial algorithm, and so this further reduces the total number of iterations. Also, note that the performance of the coloring step is dependent on the integer values assigned to the colors. Since we initialize colors to the vertex identifiers, the ordering of vertices has an impact on the performance of the algorithm.

The final stage of Tarjan's algorithm has a well-known linear $O(n+m)$ work bound. Thus, for Multistep, the best network instance would be a fully disconnected graph, incurring $O(n)$ work during simple trimming. The worst case instance for Multistep will be a mostly acyclic graph, and this would require $O(n^2)$ work for the coloring stage. We will demonstrate in Section 6 that the performance of Multistep is better than FW-BW and simple Coloring on

all the real-world and most of the tested synthetic graphs.

4 Applying the Multistep Method

This section provides more implementation details about various phases, and also discusses extensions for WCC, CC, and articulation point detection. All our algorithms were implemented in C++, using OpenMP for multithreading. We use the compressed sparse row (CSR) representation for graph storage, and use additional arrays for storing incoming edges. To avoid modifying the graph, we have a boolean array termed *valid* which signifies if a vertex is yet to be placed in an SCC. We also have an additional integer array which gives a numeric identifier of the SCC to which each vertex belongs. We avoid locking or atomic operations when possible through thread-owned queues, mitigation of race conditions, and by utilizing various techniques to reduce work.

4.1 Trim Step

We consider two different approaches for parallel trimming. We use a boolean array to mark vertices that are trimmed. Simple trimming requires the degrees of vertices in the original graph. Therefore, it requires a single pass through all vertices to find their in/out degrees, and flip their *valid* boolean if either degree is zero.

Complete trimming is a bit more complex. To speed up parallel complete trimming, in addition to the boolean trimmed array, we create current and future queues and an additional boolean array of values (mark) to signify if a vertex is currently placed in the future queue. All vertices are in the current queue to begin with. We then determine the effective in- and out-degrees for all vertices in the current queue and mark trimmed vertices as such. In addition, any untrimmed child or parent of the trimmed vertex is placed in the future queue and marked as such. Once the current queue is empty, the queues are swapped with the marks reset.

This process is repeated for as many iterations as necessary. The queues avoid having to look through all vertices at each iteration, as it has been observed that long tendrils of vertices in several real-world graphs [1] tend to result in numerous iterations where only a few vertices are removed at a time. The marking is done to prevent a vertex from being placed in the future queue multiple times. To avoid the synchronization overhead that would be required with a parallel queue, we maintain separate queues for each thread and combine them into the next level queue at the end of each iteration of complete trimming.

Although complete trimming is easily parallelizable and can be quite fast, with the queues and marking being done similar to our BFS and coloring steps (described below), it does not offset the additional cost. We note that the simple trimming step removes the vast majority of vertices that can be removed by trimming, and the additional iterative steps have a high overhead.

4.2 Breadth-First Search

The main subroutine in the FW-BW step is the parallel breadth-first search. We utilize a level-synchronous and hybrid bottom-up parallel approach, with threads concurrently exploring the vertices in the current frontier. Further, each thread maintains a queue of visited vertices to represent the frontier, and these queues are merged at the end of each iteration. Using thread-local queues instead of a shared queue avoids the synchronization overhead of insertions.

A key data structure required in BFS is a lookup array of size n, to check if a vertex has been visited or not. A typical BFS optimization is to use a bitmap (1 bit per vertex) to avoid further exploring visited vertices. A bitmap will fit completely in the last-level cache of modern server-grade CPUs for graphs of up to tens of millions of vertices. However, as we observed, a boolean *visited* array (one byte per vertex) actually outperforms a bitmap in our test environment. The likely reason for this is three-fold: less arithmetic to figure out the vertex index within the bitmap, the additional accesses needed for a SCC algorithm as opposed to running a pure BFS, and guaranteed atomic reads/writes at the byte level on our test system [23]. A much more complicated read/write function is required to guarantee atomic updates for a bitmap [11]. We note that the effectiveness of a bitmap, in practice, will depend on last-level cache utilization, which is dependent on the size and structure of the network being explored.

Recent results show that for certain levels of a BFS in low-diameter graphs, it is more efficient to look in the reverse direction [10]. In this *direction-optimizing* approach to BFS, all unvisited vertices attempt to find a parent that is in the frontier, instead of the typical way of inspecting adjacencies of frontier vertices. We used this optimization with similar settings as the original paper ($\alpha = 15, \beta = 25$) and notice considerable speedup. However, we had to maintain the thread queues in the bottom-up hybrid as opposed to explicitly rebuilding the queue from scratch when we turn the hybrid mode off. This is due to the fact that we do not maintain the BFS tree and lack the ability to track BFS level on a per-vertex basis, as we only maintain the visited array for determining the SCC.

We also investigated a per-socket graph partitioning and exploration scheme similar to the ones described in Agarwal et al. [9] and Chhugani et al. [11]. Although these partitioning approaches improved parallel scaling, it was only in a limited number of instances that actual runtimes improved due to the additional overhead. We do not include it in our final results. Overall, our BFS implementation achieves an mean traversal rate of 1.4 GTEPS (billion traversed edges per second) on the graphs given in Table 1.

4.3 Coloring

The pseudocode for the parallel vertex coloring step MS-Coloring is given in Algorithm 4. Initially, all active vertices are assigned a color which is the same as their vertex identifier and placed into a frontier queue Q. The adjacencies of all vertices in Q are inspected in parallel, and we check to see if an adjacency's color is lower than the color of v, the current vertex. If it is, the color is passed to the child, and both the parent and child are placed in

the thread's next level queue and globally marked as such.

Algorithm 4 Pseudocode for MS-Coloring

```
1: for all v ∈ V do
2:     Color(v) ← v
3:     Add v to Q
4:     Visited(v) ← false
5: while Q ≠ ∅ do
6:     for all v ∈ Q do in parallel on thread t
7:         for all ⟨v, u⟩ ∈ E(V) do
8:             if Color(v) > Color(u) then
9:                 Color(u) ← Color(v)
10:                if Visited(u) = false then
11:                    Visited(u) ← true
12:                    Add u to Q_t
13:        if any u changed color then
14:            if Visited(v) = false then
15:                Visited(v) ← true
16:                Add v to Q_t
17:    for all v ∈ Q_t do in parallel on thread t
18:        Visited(v) ← false
19:    Barrier synchronization
20:    Q ← ∪_t Q_t                          ▷ Master thread performs merge
```

We place the parent in the queue to avoid explicit locking. It is possible that two parents will have higher colors than a shared child, creating a race condition. Both parents will once again examine their children in the next iteration to make sure that either the color that was given by them, or a higher one, has been placed. Additionally, since only a higher color can be assigned, we can ignore the race condition created if a parent has their own color overwritten before they assign their previous one to the child.

We also tried to avoid locks by trying a bottom-up scheme: having children look at their parents' and own color and take the largest, avoiding the race condition entirely. However, this is much slower in practice, because either all vertices need to be examined at each iteration, or the out vertices of the child need to be examined to create the queue, effectively doubling the amount of memory transfers for each iteration.

Our parallel SCC finding on the reverse step is fairly standard, as it is a trivial algorithm to parallelize. We simply determine the root vertices by finding all unique colors in the graph, and then run a serial DFS on the transverse graph from each root, only looking at vertices with the same color as the root. We use a DFS here, since there isn't further room for parallelism, and experimentation has shown our serial DFS to be faster than our serial BFS.

4.4 Serial Step

We use the recursive Tarjan's algorithm for the serial step. Previous work has demonstrated little difference in runtime between recursive and non-recursive implementations [24]. Additionally, Tarjan's runtime should ideally be within a small factor of the runtime of a plain DFS, and our implementation runs within a factor of 1.65× on average across our test graphs in Table 1.

We experimentally determined that a cutoff of about 100,000 remaining vertices is a relatively good heuristic for switching to the serial algorithm, although this is hardware-specific. Some graphs benefit from running coloring all the way to completion, while some others would benefit more from switching to serial sooner. However, determining this cutoff without prior knowledge of the graph may be quite difficult. The cutoff threshold can be set based on the number of steps needed to fully color the graph, and we will investigate this in future work.

4.5 Connected Components and Weakly Connected Components

Our Multistep method can be easily extended to detect weakly connected components in directed graphs, and connected components in undirected graphs. We initially determine the massive (weakly) connected component through a single parallel BFS from the pivot (instead of two in case of SCC), and then subsequently perform coloring on the remaining vertices.

4.6 Biconnected Components

We now introduce a new parallel approach for BiCC decomposition by identifying articulation vertices in the graph. An articulation vertex u can be identified by the fact that it has at least one child vertex that does not have a simple path in $G(V \setminus \{u\}, E(V \setminus \{u\}))$ to another vertex with the same BFS level as u. A simple proof is as follows: if there was some path from the child vertex to another vertex on the same level as its parent, this other vertex and the parent would have to share one common ancestor at a lower level up the BFS tree, which would by definition imply that all edges connecting these vertices are in the same biconnected component.

The pseudocode for our algorithm is listed in Algorithm 5 and relies purely on multiple BFSes. We first perform a BFS from a source vertex s and track the BFS tree by maintaining parents of visited vertices and levels of each vertex from the source. Our goal is to examine every vertex $u \in G$ to check if it is an articulation vertex. We take every child that u has, as indicated by the BFS tree, and run a new search from it (called BFS-ML in Algorithm 5) on the graph induced by $V \setminus \{u\}$. If we are not able to identify any vertex during that search which is on the same level as u, then we can mark u as an articulation vertex. This algorithm is efficiently parallelized across all $u \in G$.

Although the vast number of searches in the inner loops may seem like a lot of work, it is minimized by the fact that only a very small fraction of vertices actually have any

Algorithm 5 BFS-based algorithm to identify articulation points in BiCC decomposition.

```
1: procedure MULTISTEP-BICC(G(V, E))
2:     for all v ∈ V do Art(v) ← false
3:     Select a root vertex s
4:     Determine P and Levels from BFS(G, s)
5:     for all u(≠ s) ∈ G in parallel do
6:         if Art(v) ≠ true then
7:             for all ⟨u, v⟩ ∈ E(V) and P[v] = u do
8:                 ml ← BFS-ML(G(V \ {u}, E(V \ {u})), v)
9:                 if ml = Levels[u] then
10:                    Art(u) ← true
11:                    break
12:    Check if s is an articulation point
```

children in the BFS tree for real-world graphs. Most vertices are leaf nodes in the BFS tree. Additionally, ruling out a vertex which does have a child is quite fast, since a vertex with a higher level is typically encountered after only one or two frontier expansions in BFS-ML, and we do not need to fully execute the entire BFS in BFS-ML before we return ml.

To explicitly check whether the root of the BFS tree is an articulation vertex or not, we need to examine whether one of its children can reach all of the others. This can be a costly procedure. However, it is also easily mitigated. For almost all real-world graphs, there are vertices with a degree of one. A vertex that is the sole neighbor of one of these vertices is then easily identified as an articulation point. All we then need to do is begin our BFS traversal from a known articulation point so we are not required to explicitly check if it is one. Another option is simply to rerun a new BFS from scratch using new roots and only check our original roots for being articulation points. Because the initial parallel BFS search is the fastest part of the procedure, this can be a valid option as well.

5 Experimental Setup

Experiments were performed on *Compton*, a dual-socket system with 64 GB main memory and Intel Xeon E5-2670 (Sandy Bridge) CPUs at 2.60 GHz, each having 20 MB last-level cache. Compton was running RHEL 6.1 and all programs were compiled with the Intel C++ compiler, version 13.1.2. The `-O3` optimization parameter was used with the `-openmp` flag. The environment variable `KMP_AFFINITY` was used to control thread locality when needed.

For comparison to recent work, we also run SCC code provided by Hong et al. [18, 12] and CC code from the Ligra framework, released by Shun and Blelloch [13]. We used the same compilation procedures and runtime environment when possible, with the exception of using Cilk Plus for parallelizing Ligra code instead of OpenMP. This was observed to be faster in practice.

Several large real-world and synthetic graphs were used in the course of this work (see

Network	n	m	*deg*		$\widetilde{D}$	(S)CCs	
			avg	max		count	max
Twitter	53M	2000M	37	780K	19	12M	41M
ItWeb	41M	1200M	28	10K	830	30M	6.8M
WikiLinks	26M	600M	23	39K	170	6.6M	19M
LiveJournal	4.8M	69M	14	20K	18	970K	3.8M
XyceTest	1.9M	8.3M	4.2	246	93	400K	1.5M
RDF_Data	1.9M	130M	70	10K	7	1.9M	1
RDF_linkedct	15M	34M	2.3	72K	13	15M	1
R-MAT_20	0.56M	8.4M	15	24K	9	210K	360K
R-MAT_22	2.1M	34M	16	60K	9	790K	1.3M
R-MAT_24	7.7M	130M	17	150K	9	3.0M	4.7M
GNP_1	10M	200M	20	49	7	1	10M
GNP_10	10M	200M	20	49	7	10	5.0M
Friendster	66M	1800M	53	5.2K	34	70	66M
Orkut	3.1M	117M	76	33K	11	1	3.1M
Cube	2.1M	62M	56	69	157	47K	2.1M
Kron_21	1.5M	91M	118	213K	8	94	1.5M

Table 1: Information about test networks. Columns are # vertices, # edges, average and max. degree, approximate diameter, # of (S)CCs, and size of the largest (S)CC.

Table 1). The first twelve graphs listed in the table are undirected while the rest are directed. These graphs were retrieved from a number of sources, namely the SNAP database [25], the Koblenz Network Collection [26], the 10th DIMACS Implementation Challenge [27], and the University of Florida Sparse Matrix Collection [28]. The R-MAT [29] and $G(n, p)$ networks were generated with the GTGraph [30] suite using the default parameters.

Friendster, LiveJournal, Orkut, and Twitter are crawls of social networks [31, 32, 33]. Italy Web is a web crawl of the .it domain [34]. WikiLinks is the cross-link network between articles on Wikipedia [26]. XyceTest is a Sandia National Labs electrical simulation network and Cube is 3D coupled consolidation problem of a cube discretized with tetrahedral finite elements. R-MAT_20/22/24 are R-MAT graphs of scale 20, 22, and 24, respectively. RDF_Data is constructed from RDF triples in a *data.gov* data set (# 1527), and RDF_linkedct is a semantic data set from clinical trials. Note that these RDF datasets contain no non-trivial SCCs because they are mostly bipartite and acyclic. WCC detection is a useful preprocessing step for partitioning these data sets. The Kron_21 graph is a SCALE 21 graph created from the Kronecker generator of the Graph500 benchmark [35, 36]. Finally, GNP_1 and GNP_10 refer to Erdős-Rényi random graphs with 1 and 10 large SCCs, respectively.

These graphs were selected to represent a wide mix of graph sizes and topologies. The number of SCCs/CCs and max SCC/CC both play an important role in the general performance of decomposition algorithms, while the average degree and graph diameter can have a large effect on the BFS subroutine that is necessarily used for these algorithms.

Network	Execution time (s)					MS Speedup	
	Serial	MS	Hong	FW-BW	Color	Serial	All
Twitter	33.0	**1.60**	2.6	120.00	40.0	20.0×	1.6×
ItWeb	6.7	**1.80**	16.0	1400.00	7.1	3.6×	3.6×
WikiLinks	4.9	**0.90**	0.98	270.00	9.3	5.5×	1.1×
LiveJournal	1.3	**0.11**	0.20	4.10	1.6	12.0×	1.9×
XyceTest	0.2	**0.04**	0.08	0.07	0.37	4.7×	1.9×
R-MAT_24	2.4	**0.25**	**0.25**	0.62	2.4	9.5×	1.0×
GNP_1	7.2	**0.15**	0.30	1.60	6.5	47.0×	1.9×
GNP_10	5.5	2.90	5.10	**1.20**	3.5	1.9×	0.6×

Table 2: Comparison of serial Tarjan's algorithm with parallel Multistep, Hong et al. , Naïve FW-BW, and Coloring, running on 16 cores.

6 Experimental Results

In this section, we compare our Multistep SCC algorithm execution time and scaling to our implementations of the baseline FW-BW and coloring algorithms, as well as the Hong et al. SCC algorithm. Furthermore, we compare our Multistep CC algorithm to baseline coloring and the coloring approach implemented in the Ligra graph processing framework. We then compare our weakly connected components algorithm to the coloring-based parallel approach, and our biconnected components algorithm to the optimal serial algorithm. We justify algorithmic choices and measure their influence on parallel performance for different graphs.

6.1 Strongly Connected Component Decomposition

Table 2 gives the absolute execution time on 16 cores for baseline coloring, FW-BW with complete trimming, Multistep with simple trimming and the Hong et al. *Method 2* on several directed graphs. The fastest method for each network in highlighted in bold. We also give the speedup achieved by Multistep over the serial approach and the fastest approach for that network.

Both Multistep and Hong et al. are considerably faster than the parallel FW-BW and Coloring approaches. The performance of the baseline approaches is also very dependent on graph structure. The graphs with a large proportion of their vertices in the massive SCC, such as the $G(n, p)$, R-MAT, and Xyce graphs, show very poor performance with Coloring, due to the long time needed to fully propagate the colors. Further, networks with a large absolute number of SCCs show poor performance with FW-BW, due to the recursive and tasking overhead. FW-BW demonstrates the strongest performance on GNP_10, as this graph was designed to result in very even partitions for each recursive call. It should be noted that with the Hong et al. code on the GNP_10 graph we utilize their BFS subroutine for the recursive SCC calls instead of their standard DFS, as this is more amenable to finding large scale SCCs and gives vastly superior performance and a more fair comparison to our

work.

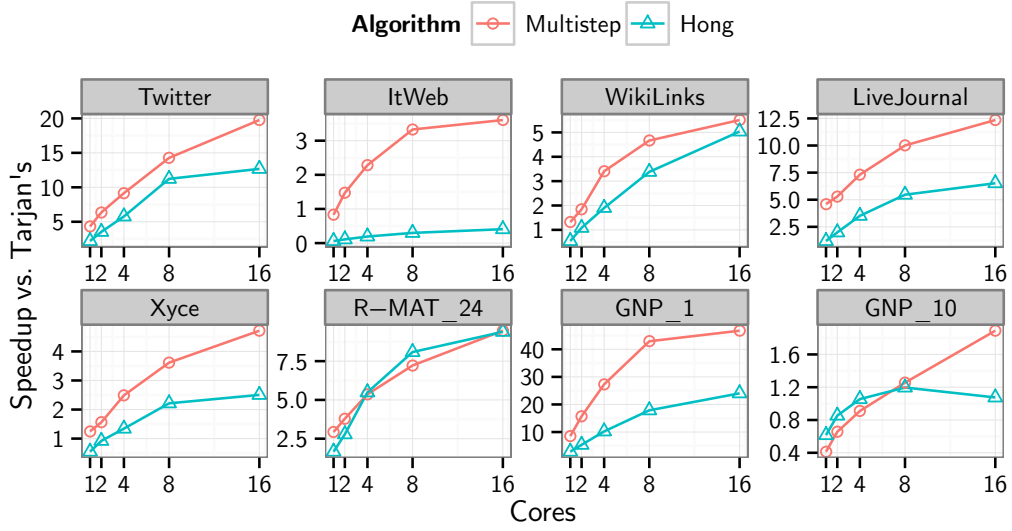

Figure 1: Finding SCCs: Parallel scaling of Multistep and Hong et al. relative to Tarjan's serial algorithm.

Although the Hong et al. method attempts to minimize the impact of the recursive and tasking overhead with a partitioning step based on WCCs and a smart tasking queue, on graphs with a very high number of small but non-trivial SCCs, such as ItWeb, the overhead inherent in the FW-BW algorithm can still dominate the running time. It can also be noted that our coloring step will, at each iteration, partition the graph into *at least* as many discrete partitions that their WCC decomposition will. Overall, for 16-core runs, Multistep gives a geometric mean speedup of 1.92× over Hong et al. on these graphs.

Figure 1 gives the scaling of Multistep and Hong et al. for parallel runs, relative to the serial Tarjan implementation. Both Multistep and Hong et al. demonstrate good scaling on most test instances, and the overall speedup on 16 cores is dependent on the single-threaded performance. The Hong et al. running time on ItWeb is greatly affected by the number of SCCs. Additionally, on ItWeb, there are long strings of trivial and non-trivial SCCs, which results in a relatively long time spent in the multiple trimming iterations that are in the Hong et al. approach, as well as longer time spent in their WCC decomposition step.

Figure 2 gives the breakdown for each stage of Multistep as a proportion of total parallel running time. We observe that the execution time proportion for the FW-BW and coloring steps is mostly dependent on graph structure, with coloring taking a larger proportion of time for graphs for graphs with a higher diameter (e.g., ItWeb vs Twitter). In case of GNP_10, most of the time is spent in the serial step due to the fixed cutoff employed in our case.

Figure 2 also gives further justification for our choice of doing simple trimming versus complete trimming with Multistep. In general, the time spent doing iterative trimming does not sufficiently decrease the execution times of the FW-BW or coloring steps for the overall running time to be lower. As is shown on LiveJournal and Twitter, doing no trimming at all can end up being faster than fully trimming the graph with our Multistep approach. ItWeb

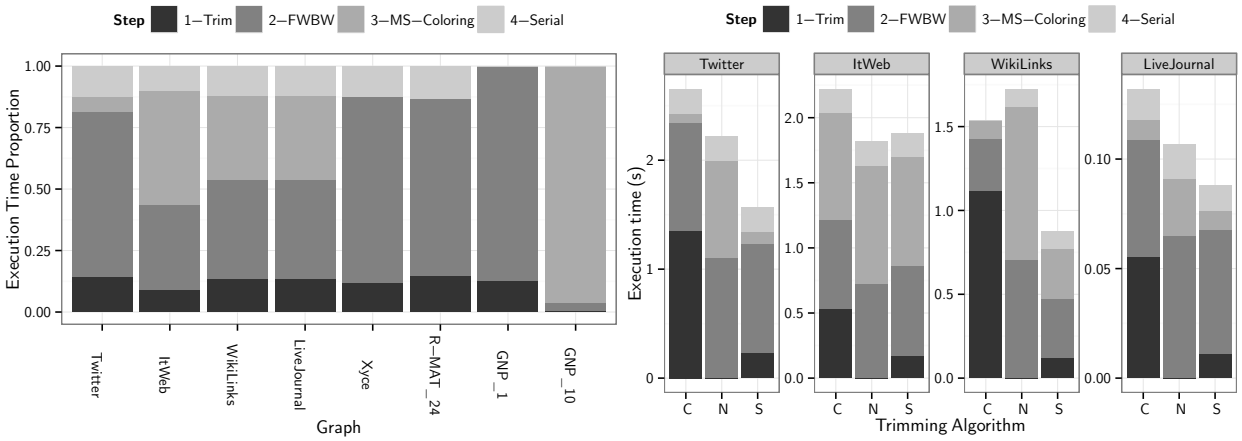

Figure 2: Left: Proportion of time spent in each subroutine of the Multistep algorithm. Right: Comparing possible trimming procedures (S: Simple, N: None, C: Complete) in Multistep for several networks.

shows that no trimming can even be faster than simple trimming, although this appears to be an exception. While running Multistep across a wide variety of graphs, fully trimming the graph never improved execution times versus only doing a single iteration. However, complete trimming is important for naïve FW-BW.

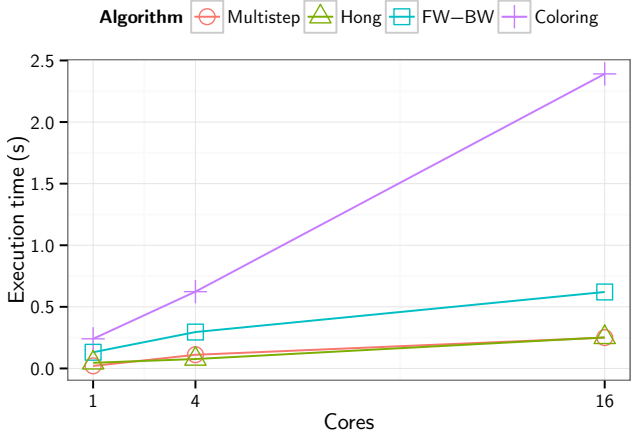

Figure 3: Approximate weak scaling of Multistep compared to coloring and naïve FW-BW on R-MAT graphs.

Figure 3 gives approximate weak scaling for three R-MAT test graphs (R-MAT_20/22/24). The test graphs' number of vertices, edges, number of SCCs, and size of largest SCC all increase by approximately a factor of 4×. From Figure 3, we see that Multistep scales better than simple FW-BW or Coloring, and Hong et al. performance is comparable to Multistep for this instance.

6.2 Connected and Weakly Connected Component Decomposition

We also compare our approach to Ligra for the problem of determining connected components, for the four undirected networks in our collection. Ligra implements a parallel coloring-based algorithm. We show scaling relative to a serial DFS. From Figure 4, we observe that Multistep greatly outperforms the other approaches on all tested graphs.

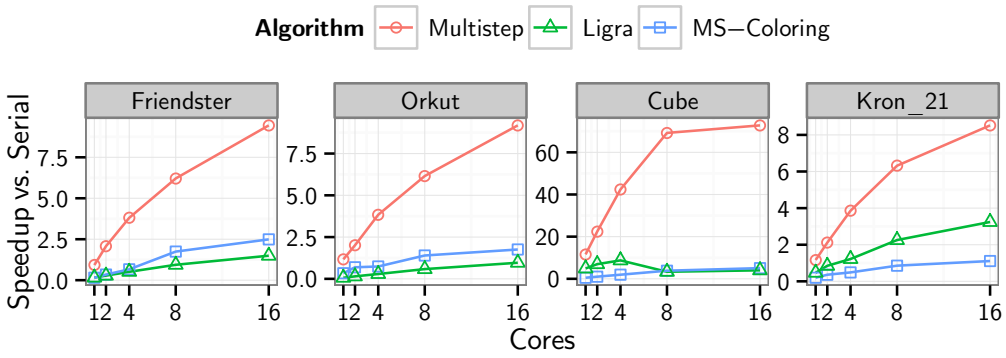

Figure 4: Finding CCs: Parallel scaling of Multistep CC, Ligra, and MS-Coloring relative to the serial DFS approach.

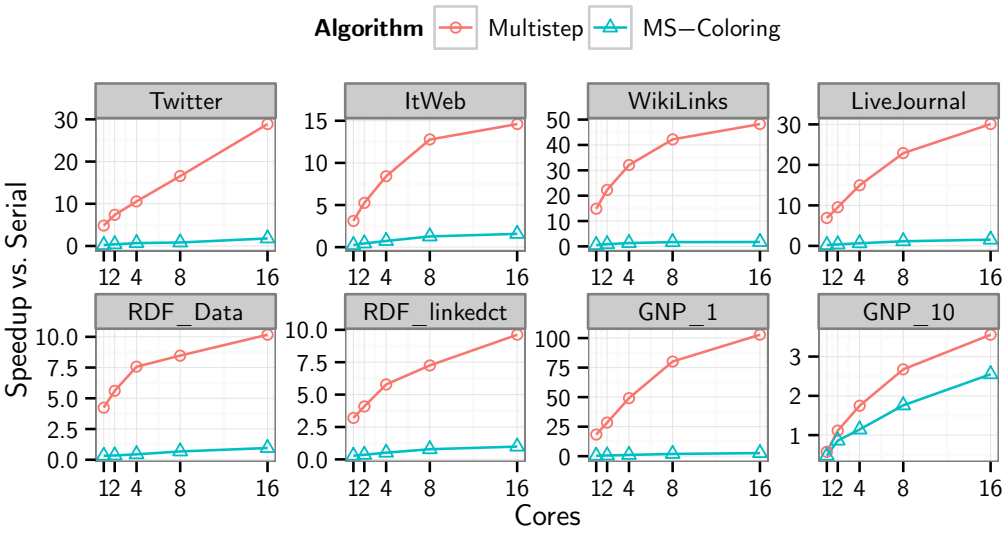

Figure 5: Finding WCCs: Comparison of WCC-Multistep and Coloring scaling relative to the serial DFS approach.

Figure 5 gives the speedup of the Multistep method and our coloring approach for determining the weakly connected components of several graphs. We give speedup relative to the serial DFS approach. Once again, we observe good scaling of Multistep relative to both Coloring and the serial code.

6.3 Biconnected Component Decomposition

Figure 6 gives the parallel scaling of our new BFS-based BiCC articulation point detection algorithm compared to the serial Hopcroft-Tarjan approach. On the three largest non-fully biconnected graphs, we achieve up to an 8× speedup and good scaling across 16 cores. However, our approach does not scale well with the fully-biconnected Cube graph. This is likely due to its regular and simple structure, which limit the inner-loop searches to a single iteration on most instances. While the approach is faster than the serial algorithm on a single thread, there is no parallel speedup.

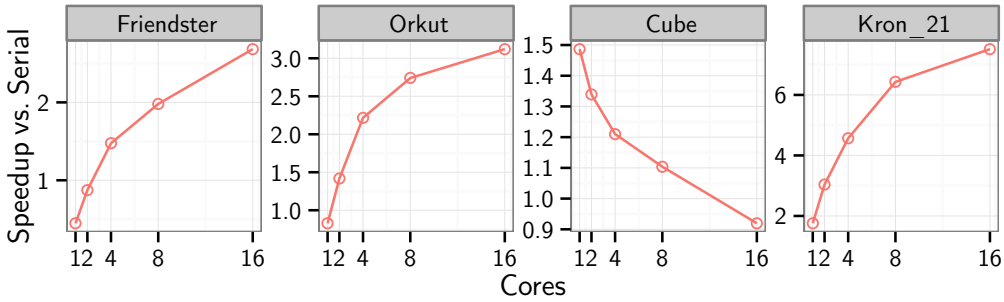

Figure 6: Finding BiCCs: Parallel scaling of BiCC-BFS relative to the serial approach.

7 Conclusion

We present the new Multistep method for SCC detection, and its extensions for solving related problems (CC, WCC, and BiCC). We demonstrate significant speedup over the current state-of-the-art methods on a multicore server, and present scaling results on a wide variety of networks. The Multistep method uses optimized BFS and coloring routines and several heuristics to achieve this performance. In future work, we will study Multistep performance optimizations for manycore systems and for processing larger graph instances.

Acknowledgment

We thank Erik Boman, Karen Devine, and Bruce Hendrickson for encouraging us to focus on this problem. We thank the reviewers for their helpful suggestions. Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000. This work was partially supported by the DOE Office of Science through the FASTMath SciDAC Institute and by NSF grant ACI-1253881.

References

[1] A. Broder, R. Kumar, F. Maghoul, P. Raghavan, S. Rajagopalan, R. Stata, A. Tomkins, and J. Wiener, "Graph structure in the Web," *Computer Networks*, vol. 33, pp. 309–320, 2000.

[2] A. Mislove, M. Marcon, K. Gummadi, P. Druschel, and B. Bhattacharjee, "Measurement and analysis of online social networks," in *Proc. 7th Conf. on Internet Measurement (IMC '07)*, 2007, pp. 29–42.

[3] S. Orzan, "On distributed verification and verified distribution," Ph.D. dissertation, Vrije Universiteit, 2004.

[4] A. Xie and P. Beerel, "Implicit enumeration of strongly connected components and an application to formal verification," *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, vol. 19, no. 10, pp. 1225–1230, 2000.

[5] A. Pothen and C.-J. Fan, "Computing the block triangular form of a sparse matrix," *ACM Trans. on Mathematical Software (TOMS)*, vol. 16, no. 4, pp. 303–324, 1990.

[6] R. E. Tarjan, "Depth first search and linear graph algorithms," *SIAM Journal of Computing*, vol. 1, pp. 146–160, 1972.

[7] J. A. Edwards and U. Vishkin, "Better speedups using simpler parallel programming for graph connectivity and biconnectivity," in *Proc. 2012 Int'l. Workshop on Programming Models and Applications for Multicores and Manycores*. ACM, 2012, pp. 103–114.

[8] L. Fleischer, B. Hendrickson, and A. Pinar, "On identifying strongly connected components in parallel," in *Parallel and Distributed Processing*, ser. LNCS. Springer Berlin Heidelberg, 2000, vol. 1800, pp. 505–511.

[9] V. Agarwal, F. Petrini, D. Pasetto, and D. Bader, "Scalable graph exploration on multicore processors," in *Proc. Supercomputing*, 2010.

[10] S. Beamer, K. Asanović, and D. Patterson, "Direction-optimizing breadth-first search," in *Proc. Supercomputing*, 2012.

[11] J. Chhugani, N. Satish, C. Kim, J. Sewall, and P. Dubey, "Fast and efficient graph traversal algorithm for CPUs: Maximizing single-node efficiency," in *Proc. Supercomputing*, 2012.

[12] S. Hong, N. C. Rodia, and K. Olukotun, "On fast parallel detection of strongly connected components (SCC) in small-world graphs," in *Proc. Supercomputing*, 2013.

[13] J. Shun and G. Blelloch, "Ligra: A lightweight graph processing framework for shared memory," in *Proc. 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2013, pp. 135–146.

[14] A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *Data Structures and Algorithms*. Addison-Wesley, 1983.

[15] W. McLendon III, B. Hendrickson, S. J. Plimpton, and L. Rauchwerger, "Finding strongly connected components in distributed graphs," *Journal of Parallel and Distributed Computing*, vol. 65, no. 8, pp. 901–910, 2005.

[16] J. Barnat and P. Moravec, "Parallel algorithms for finding SCCs in implicitly given graphs," *Formal Methods: Applications and Technology*, vol. 4346, pp. 316–330, 2006.

[17] J. Barnat, P. Bauch, L. Brim, and M. Cevska, "Computing strongly connected components in parallel on CUDA," in *Proc. 25th Int'l. Parallel and Distributed Processing Symp. (IPDPS)*. IEEE, 2011, pp. 544–555.

[18] S. Hong, N. C. Rodia, and K. Olukotun, "Technical report: On fast parallel detection of strongly connected components (SCC) in small-world graphs," Stanford University, Tech. Rep., 2013.

[19] S. J. Plimpton and K. D. Devine, "Mapreduce in MPI for large-scale graph algorithms," *Parallel Comput.*, vol. 37, no. 9, pp. 610–632, Sep. 2011.

[20] J. Hopcroft and R. Tarjan, "Efficient algorithms for graph manipulation," *Communications of the ACM*, vol. 16, no. 6, pp. 374–378, 1973.

[21] R. Tarjan and U. Vishkin, "An efficient parallel biconnectivity algorithm," *SIAM Journal on Computing*, vol. 14, no. 4, pp. 862–874, 1985.

[22] G. Cong and D. A. Bader, "An experimental study of parallel biconnected components algorithms on symmetric multiprocessors (smps)," in *Proc. 19th Int'l. Parallel and Distributed Processing Symp. (IPDPS)*. IEEE, 2005, pp. 45b–45b.

[23] Intel, *Intel 64 and IA-32 Architectures Software Developer's Manual, System Programming Guide, Part 1*. Intel Press, 2013, vol. 3A.

[24] K. Mehlhorn, S. Näher, and P. Sanders, "Engineering DFS-based graph algorithms," 2007.

[25] J. Leskovec, "SNAP: Stanford network analysis project," http://snap.stanford.edu/index.html, last accessed 4 Feb 2014.

[26] J. Kunegis, "KONECT - the Koblenz network collection," konect.uni-koblenz.de, last accessed 4 Feb 2014.

[27] D. A. Bader, H. Meyerhenke, P. Sanders, and D. Wagner, "Graph partitioning and graph clustering, 10th DIMACS implementation challenge workshop," *Contemporary Mathematics*, vol. 588, 2013.

[28] T. A. Davis and Y. Hu, "The University of Florida sparse matrix collection," *ACM Transactions on Mathematical Software*, vol. 38, no. 1, pp. 1–25, 2011.

[29] D. Chakrabarti, Y. Zhan, and C. Faloutsos, "R-MAT: A recursive model for graph mining," in *SDM*, 2004.

[30] K. Madduri and D. A. Bader, "GTgraph: A suite of synthetic graph generators," http://www.cse.psu.edu/~madduri/software/GTgraph/, last accessed 4 Feb 2014.

[31] J. Yang and J. Leskovec, "Defining and evaluating network communities based on ground-truth," in *ICDM*, 2012.

[32] L. Backstrom, D. Huttenlocher, J. Kleinberg, and X. Lan, "Group formation in large social networks: Membership, growth, and evolution," in *KDD*, 2006.

[33] M. Cha, H. Haddadi, F. Benevenuto, and K. P. Gummadi, "Measuring User Influence in Twitter: The Million Follower Fallacy," in *Proc. 4th Int'l. AAAI Conf. on Weblogs and Social Media (ICWSM)*, May 2010.

[34] P. Boldi, B. Codenotti, M. Santini, and S. Vigna, "Ubicrawler: A scalable fully distributed web crawler," *Software: Practice & Experience*, vol. 34, no. 8, pp. 711–726, 2004.

[35] J. Leskovec, D. Chakrabarti, J. Kleinberg, C. Faloutsos, and Z. Ghahramani, "Kronecker graphs: An approach to modeling networks," *Journal of Machine Learning Research*, vol. 11, pp. 985–1042, 2010.

[36] Graph500, http://www.graph500.org/, last accessed 4 Feb 2014.