

FastSV: A Distributed-Memory Connected Component Algorithm with Fast Convergence

Yongzhe Zhang*

Ariful Azad†

Zhenjiang Hu‡

Abstract

This paper presents a new distributed-memory algorithm called FastSV for finding connected components in an undirected graph. Our algorithm simplifies the classic Shiloach-Vishkin algorithm and employs several novel and efficient hooking strategies for faster convergence. We map different steps of FastSV to linear algebraic operations and implement them with the help of scalable graph libraries. FastSV uses sparse operations to avoid redundant work and optimized MPI communication to avoid bottlenecks. The resultant algorithm shows high-performance and scalability as it can find the connected components of a hyperlink graph with over 134B edges in 30 seconds using 262K cores on a Cray XC40 supercomputer. FastSV outperforms the state-of-the-art algorithm by an average speedup of $2.21 \times$ (max $4.27 \times$) on a variety of real-world graphs.

1 Introduction

This paper presents a distributed-memory parallel algorithm for finding connected components (CC) in an undirected graph $G(V, E)$ where V and E are the set of vertices and edges, respectively. A connected component is a subgraph of G in which every pair of vertices are connected by paths and no vertex in the subgraph is connected to any other vertex outside of the subgraph. Finding connected components has numerous applications in bioinformatics [25], computer vision [29], and scientific computing [20].

Sequentially, connected components of a graph with n vertices and m edges can be easily found by breadth-first search (BFS) or depth-first search in $O(m + n)$ time. While this approach performs linear work, the depth is proportional to the sum of the diameters of the connected components. Therefore, BFS-based parallel algorithms are not suitable for high-diameter graphs or graphs with millions of connected components. Connectivity algorithms based on the “tree hooking” scheme work by arranging the vertices into disjoint trees such

that at the end of the algorithm, all vertices in a tree represent a connected component. Shiloach and Vishkin [22] used this idea to develop a highly-parallel PRAM (parallel random access machine) algorithm that runs in $O(\log n)$ time using $O(n + m)$ processors. Their algorithm is not work efficient as it performs $O(m \log n)$ work, but the availability of $O(m)$ parallel work made it an attractive choice for large-scale distributed-memory systems. Therefore, the Shiloach-Vishkin (SV) algorithm and its variants are frequently used in scalable distributed-memory CC algorithms such as LACC [6], ParConnect [16], and Hash-Min [21].

To the best of our knowledge, LACC [6] is the most scalable published CC algorithm that scales to 262K cores when clustering graphs with more than 50B edges. LACC is based on the Awerbuch-Shiloach (AS) algorithm, which is a simplification of the SV algorithm. The AS algorithm consists of four steps: (a) finding stars (trees of height 1), (b) hooking stars conditionally onto other trees, (c) hooking stars unconditionally onto other trees, (d) shortcutting to reduce the height of trees. LACC mapped these operations to parallel linear-algebraic operations such as those defined in the GraphBLAS standard [10] and implemented them in the CombBLAS [8] library for scalability and performance. We observed that LACC’s requirements of star hooking and unconditional hooking can be safely removed to design a simplified SV algorithm with just two steps: (a) hooking trees conditionally onto other trees and (b) shortcutting. After mapping these two operation to linear algebra and performing other simplifications, we developed a distributed-memory SV algorithm that is both simpler and faster than LACC. Since, each of the four operations in LACC takes about 25% of the total runtime, each iteration of our SV is usually more than $2 \times$ faster than each iteration of LACC when run on the same number of processors. However, the simplified SV requires more iterations than LACC because of the removal of unconditional hooking. To alleviate this problem, we developed several novel hooking strategies for faster convergence, hence the new algorithm is named as FastSV.

The simplicity of FastSV along with its fast conver-

*The Graduate University for Advanced Studies, Japan

†Indiana University Bloomington, USA

‡Peking University, China

gence schemes makes it suitable for distributed-memory platforms. We map different steps of FastSV to linear-algebraic operations and implemented the algorithm using the CombBLAS library. We choose CombBLAS due to its high scalability and the fact that the state-of-the-art connected component algorithm LACC and ParConnect also rely on CombBLAS. We further employ several optimization techniques for eliminating communication bottlenecks. The resultant algorithm is highly parallel as it scales up to 262,144 cores of a Cray XC40 supercomputer and can find CCs from graphs with billions of vertices and hundreds of billions of edges in just 30 seconds. FastSV advances the state-of-the-art in parallel CC algorithm as it is on average $2.21\times$ faster than the previous fastest algorithm LACC. Overall, we made the following technical contributions in this paper:

- We develop a simple and efficient algorithm FastSV for finding connected components in distributed memory. FastSV uses novel hooking strategies for fast convergence.
- We present FastSV using a handful of GraphBLAS operations and implement the algorithm in CombBLAS for distributed-memory platforms and in SuiteSparse:GraphBLAS for shared-memory platforms. We dynamically use sparse operations to avoid redundant work and optimize MPI communication to avoid bottlenecks.
- Both shared- and distributed-memory implementations of FastSV are significantly faster than the state-of-the-art algorithm LACC. The distributed-memory implementation of FastSV can find CCs in a hyperlink graph with 3.27B vertices and 124.9B edges in just 30 seconds using 262K cores of a XC40 supercomputer.

2 Background

2.1 Notations. This paper assumes the connected component algorithm to be performed on an undirected graph $G = (V, E)$ with n vertices and m edges. For each vertex $v \in V$, we use $N(v)$ to denote v 's neighbors, the set of vertices adjacent to v . We use *pointer graph* to refer to an auxiliary directed graph $G_p = (V, E_p)$ for G , where for every vertex $v \in V$ there is exactly one directed edge $(v, v_1) \in E_p$ and $v_1 \leq v$. If we ignore the self-loops $(v, v) \in E_p$, G_p defines a forest of directed rooted trees where every vertex can follow the directed edges to reach the root vertex. In G_p , a tree is called a *star* if every vertex in the tree points to a root vertex (a root points to itself).

2.2 GraphBLAS. Expressing graph algorithms in the language of linear algebra is appealing. By using

a small set of matrix and vector (linear algebra) operations, many scalable graph algorithms can be described clearly, the parallelism is hidden for the programmers, and the high performance can be achieved by performance experts implementing those primitives on various architectures. Several independent systems have emerged that use matrix algebra to perform graph computations [8, 14, 23]. Recently, GraphBLAS [10] defines a standard set of linear-algebraic operations (and C APIs [11]) for implementing graph algorithms. In this paper, we will use the functions from the GraphBLAS API to describe our algorithms due to its conciseness. Our distributed implementation is based on CombBLAS [8].

2.3 The original SV algorithm. The SV algorithm stores the connectivity information in a forest of rooted trees, where each vertex v maintains a field $f[v]$ through which it points to either itself or another vertex in the same connected component. All vertices in a tree belong to the same component, and at termination of the algorithm, all vertices in a connected component belong to the same tree. Each tree has a designated root (a vertex having a self-loop) that serves as the representative vertex for the corresponding component. This data structure is called a pointer graph, which changes dynamically during the course of the algorithm.

The algorithm begins with n single-vertex trees and iteratively merges trees to find connected components. Each iteration of the original SV algorithm performs a sequence of four operations: (a) conditional hooking, (b) shortcutting, (c) unconditional hooking and (d) another shortcutting. Here, hooking is a process where the root of a tree becomes a child of a vertex from another tree. Conditional hooking of a root u is allowed only when u 's id is larger than the vertex which u is hooked into. Unconditional hooking can hook any trees that remained unchanged in the preceding conditional hooking. The shortcutting step reduces the height of trees by replacing a vertex's parent by its grandparent. With these four steps the SV algorithm is guaranteed to finish in $O(\log n)$ iterations, where each iteration performs $O(m)$ parallel work.

The original Shiloach-Vishkin paper mentioned that the last shortcutting is for a simpler proof of their algorithm. Hence, it can be removed without sacrificing correctness or convergence speed. If we remove unconditional hooking, the algorithm is still correct, but it may need $O(n)$ iterations in the worst case. Nevertheless, practical parallel algorithms often remove the unconditional hooking [24, 31] because it needs to keep track of unchanged trees (also known as stagnant trees), which is expensive, especially in distributed memory. We fol-

Algorithm 1 The SV algorithm. **Input:** An undirected graph $G(V, E)$. **Output:** The parent vector f .

```

1: procedure SV( $V, E$ )
2:   for every vertex  $u \in V$  do
3:      $f[u], f_{next}[u] \leftarrow u$ 
4:   repeat
5:      $\triangleright$  Step 1: Tree hooking
6:     for every  $(u, v) \in E$  do in parallel
7:       if  $f[u] = f[f[u]]$  and  $f[v] < f[u]$  then
8:          $f_{next}[f[u]] \leftarrow f[v]$ 
9:      $f \leftarrow f_{next}$ 
10:     $\triangleright$  Step 2: Shortcutting
11:    for every  $u \in V$  do in parallel
12:      if  $f[u] \neq f[f[u]]$  then
13:         $f_{next}[u] \leftarrow f[f[u]]$ 
14:     $f \leftarrow f_{next}$ 
15:  until  $f$  remains unchanged

```

low the same route and use a simplified SV algorithm discussed next.

2.4 A simplified SV algorithm Algorithm 1 describes the simplified SV algorithm, which is the basis of our parallel algorithm. Initially, the parent $f[u]$ of a vertex u is set to itself to denote n single-vertex trees. We additionally maintain a copy f_{next} of the parent vector so that the parallel algorithm reads from f and writes to f_{next} . Given a fixed ordering of vertices, each execution of Algorithm 1 generates exactly the same pointer graph after the i th iteration because of using separate vectors for reading and writing. Hence, the convergence pattern of this parallel algorithm is completely deterministic, making it suitable for massively-parallel distributed systems. By contrast, concurrent reading from and writing to a single vector f still deliver the correct connected components, but the structures of intermediate pointer graphs are not deterministic.

In each iteration, the algorithm performs tree hooking and shortcutting operations in order:

- **Tree hooking (line 6–8):** for every edge (u, v) , if u 's parent $f[u]$ is a root and $f[v] < f[u]$, then make $f[u]$ point to $f[v]$. As mentioned before, the updated parents are stored in a separate vector f_{next} so the the updated parents are not used in the current iteration.
- **Shortcutting (line 11–13):** if a vertex u does not point to a root vertex, make u point to its grandparent $f[f[u]]$.

The algorithm terminates when the parent vector remains unchanged in the latest iteration. At termination, every tree becomes a star, and vertices in a star

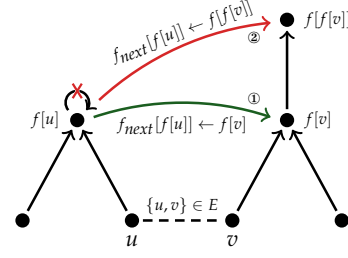


Figure 1: Two different ways of performing the tree hooking. (1) the original algorithm that hooks u 's parent $f[u]$ to v 's parent $f[v]$, (2) hook u 's parent $f[u]$ to v 's grandparent $f[f[v]]$. Both strategies are correct and the latter one improves the convergence.

constitute a connected component. The correctness of this algorithm is discussed in previous work [15]. However, as mentioned before, without the unconditional hooking used in the original SV algorithm, we can no longer guarantee that Algorithm 1 converges in $O(\log n)$ iterations. We will show in Section 5 that Algorithm 1 indeed converges slowly, but does not require the worst case bound $O(n)$ iterations for the practical graphs we considered. Nevertheless, the extra iterations needed by Algorithm 1 increase the runtime of parallel SV algorithms. To alleviate this problem, we develop several novel hooking schemes, ensuring that the improved algorithm FastSV is as simple as Algorithm 1, but the former converges faster than the latter.

3 The FastSV algorithm

In this section, we introduce four important optimizations for the simplified SV algorithm, obtaining FastSV with faster convergence.

3.1 Hooking to grandparent. In the original algorithm, the tree hooking is represented by the assignment $f_{next}[f[u]] \leftarrow f[v]$ (line 8 in Algorithm 1) requiring $f[u]$ to be a root vertex, $(u, v) \in E$ and $f[v] < f[u]$. It is not hard to see, if we perform the tree hooking using v 's grandparent $f[f[v]]$, saying $f_{next}[f[u]] \leftarrow f[f[v]]$, the algorithm will still produce the correct answer. To show this, we visualize both operations in Figure 1.

Suppose (u, v) is an edge in the input graph and $f[v] < f[u]$. The original hooking operation is represented by the green arrow in the figure, which hooks $f[u]$ to v 's parent $f[v]$. Then, our new strategy simply changes $f[v]$ to v 's grandparent $f[f[v]]$, resulting the red arrow from $f[u]$ to $f[f[v]]$. It is not hard to see, as long as we choose a value like $f[f[v]]$ such that it is in the same tree of v , we can easily prove the correctness of the algorithm. One can also expect that any value like $f^k[v]$ (v 's k -th level ancestor) will also work.

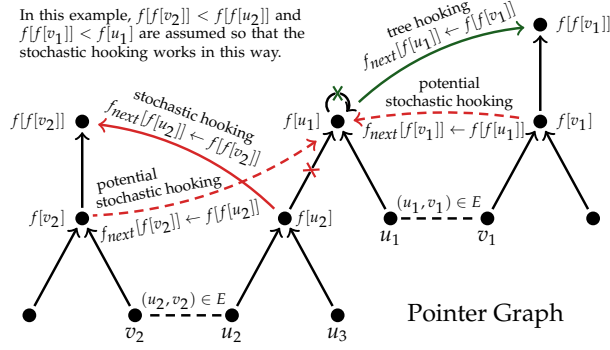


Figure 2: The stochastic hooking strategy. Suppose there are two edges (u_1, v_1) and (u_2, v_2) activating the hooking operation. The red arrows are the potential modifications to the pointer graph due to our stochastic hooking strategy, which tries to hook a non-root vertex to another vertex. The solid line successfully modifies $f[u_2]$'s pointer to $f[f[v_2]]$, but the dashed lines do not take effect due to the ordering on the vertices.

Intuitively, choosing a higher ancestor of v in the tree hooking will likely create shorter trees, leading to faster convergence (all trees are stars at termination). However, finding higher ancestors may incur additional computational cost. Here, we choose grandparents $f[f[v]]$ because they are needed in the shortcutting operation anyway; hence, using grandparents does not incur additional cost in the hooking operation.

3.2 Stochastic hooking. The original SV algorithm and Algorithm 1 always hooked the root of a tree onto another tree (see Figure 1 for an example). Therefore, the hooking operation in Algorithm 1 never breaks a tree into multiple parts and hooks different parts to different trees. This restriction is enforced by the equality check $f[f[u]] = f[u]$ in line 7 of Algorithm 1, which is only satisfied by roots and their children. We observed that this restriction is not necessary for the correctness of the SV algorithm. Intuitively, we can split a tree into multiple parts and hook them independently because these tree fragments will eventually be merged to a single connected component when the algorithm terminates. We call this strategy *stochastic hooking*.

The stochastic hooking strategy can be employed by simply removing the condition $f[f[u]] = f[u]$ from line 7 of Algorithm 1. Then, any part of a tree is allowed to hook onto another vertex when the other hooking conditions are satisfied. It should be noted that after removing the condition $f[f[u]] = f[u]$, it is possible that a tree may hook onto a vertex in the same tree. This will not affect the correctness though. In this case, the effect of stochastic hooking is similar to

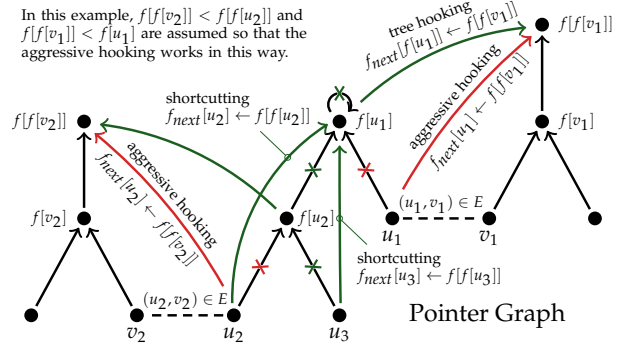


Figure 3: The aggressive hooking strategy. Suppose there are two edges (u_1, v_1) and (u_2, v_2) activating the hooking operation. The green arrows represent the hooking strategies introduced so far, and the red arrows represent our aggressive hooking strategy where a vertex may point on one of its neighbor's grandparent. Some vertices may have multiple arrows (like u_2), and which vertex to hook onto is decided by the ordering on the vertices.

the shortcutting, which hooks a vertex to some other vertex with a smaller identifier.

Figure 2 shows an example of stochastic hooking by the solid red arrow from $f[u_2]$ to $f[f[v_2]]$. In the original algorithm, u_2 does not modify its non-root parent $f[u_2]$'s pointer, but stochastic hooking changes $f[u_2]$'s pointer to one of u 's neighbor's grandparent $f[f[v_2]]$. Suppose $f[u_1]$ points to $f[f[v_1]]$ after the tree hooking, we can see that $f[u_1]$ and $f[u_2]$ might be no longer in the same connected component (assuming $f[f[v_1]]$ and $f[f[v_2]]$ are in different trees). Possible splitting of trees is a policy that differs from the conventional SV algorithm, but it gives a non-root vertex an opportunity to be hooked. In Figure 2, $f[u_2]$'s new parent $f[f[v_2]]$ is smaller than $f[u_1]$, which can expedite the convergence.

Algorithm 2 presents the high-level description of FastSV using the new hooking strategies. Here, \leftarrow^{\min} denotes a compare-and-assign operation that updates an entry of f_{next} only when the right hand side is smaller. The stochastic hooking is shown in line 6–7, and the shortcutting operation in line 12–13 is also affected by the removal of the predicate $f[f[u]] = f[u]$.

3.3 Aggressive hooking. Next, we give a vertex u another chance to hook itself onto another tree whenever possible. This strategy is called *aggressive hooking*, performed by $f_{next}[u] \leftarrow^{\min} f[f[v]]$ for all $(u, v) \in E$. Figure 3 gives an example of aggressive hooking by the red arrow for u_1 and u_2 . Here, u_1 's pointer will not be modified by any hooking operation introduced so far. Then, the aggressive hooking makes u_1 point to its newest grandparents $f[f[v_1]]$, as if an additional short-

Algorithm 2 The FastSV algorithm. **Input:** $G(V, E)$. **Output:** The parent vector f

```

1: procedure FASTSV( $V, E$ )
2:   for every vertex  $u \in V$  do
3:      $f[u], f_{next}[u] \leftarrow v$ 
4:   repeat
5:      $\triangleright$  Step 1: Stochastic hooking
6:     for every  $(u, v) \in E$  do in parallel
7:        $f_{next}[f[u]] \xleftarrow{\min} f[f[v]]$ 
8:      $\triangleright$  Step 2: Aggressive hooking
9:     for every  $(u, v) \in E$  do in parallel
10:       $f_{next}[u] \xleftarrow{\min} f[f[v]]$ 
11:     $\triangleright$  Step 3: Shortcutting
12:    for every  $u \in V$  do in parallel
13:       $f_{next}[u] \xleftarrow{\min} f[f[u]]$ 
14:     $f \leftarrow f_{next}$ 
15:  until  $f[f]$  remains unchanged

```

cutting is performed. We should mention that the cost of an additional shortcutting $f'_{next} \leftarrow f_{next}[f_{next}]$ is expensive due to the recalculation of grandparents, while the aggressive hooking is essentially a cheap element-wise operation over f by reusing some results in the stochastic hooking. We will discuss how they are implemented in Section 4.1.

For u_2 in Figure 3, it only performs the shortcutting operation $f_{next}[u_2] \leftarrow f[f[u_2]]$ in the original algorithm, and now the aggressive hooking performs $f_{next}[u_2] \leftarrow f[f[v_2]]$. Our implementation let $f[u_2]$ point to the smaller one between $f[f[u_2]]$ and $f[f[v_2]]$, which is expected to give the best convergence for vector f .

3.4 Early termination. The last optimization is a generic one that applies to most variations of the SV algorithm. SV’s termination is based on the stabilization of the parent vector f , which means even if f reaches the converged state (where every vertex points to the smallest vertex in its connected component), we need an additional iteration to verify that. We will see in Section 5.2 that for most real-world graphs, FastSV usually takes 5 to 10 iterations to converge. Hence, this additional iteration can consume a significant portion of the runtime. **The removal of the last iteration is possible by detecting the stabilization of the grandparent $f[f]$ instead of f .** The following lemma ensures the correctness of this new termination condition.

LEMMA 3.1. *After an iteration, if the grandparent $f[f]$ remains unchanged, then the vector f will not be changed afterwards.*

Proof. See Appendix A.

In practice, we found that on most practical graphs,

FastSV identifies all the connected components before converged, and the last iteration always performs the shortcutting operation to turn the trees into stars. In such case, the grandparent vector $f[f]$ converges one iteration earlier than f .

4 Implementation of FastSV in linear algebra

In this section, we first give the formal description of FastSV in GraphBLAS, a standardized set of linear algebra primitives for describing graph algorithms. We then present its linear algebra distributed-memory implementation in Combinatorial BLAS [8] and discuss two optimization techniques for improving its performance.

4.1 Implementation in GraphBLAS. In GraphBLAS, we assume that the vertices are indexed from 0 to $|V| - 1$, then the vertices and their associated values are stored as GraphBLAS object **GrB.Vector**. The graph’s adjacency matrix is stored as a GraphBLAS object **GrB.Matrix**. For completeness, we concisely describe the GraphBLAS functions used in our implementation below, where the formal descriptions of these functions can be found in the API document [10]. We use \emptyset to denote **GrB.NULL**, which is fed to those ignored input parameters.

- The function **GrB.mxv**($y, \emptyset, accum, semiring, \mathbf{A}, x, \emptyset$) multiplies the matrix \mathbf{A} with the vector x on a semiring and outputs the result to the vector y . When the accumulator (a binary operation *accum*) is specified, the multiplication result is combined with y ’s original value instead of overwriting it.
- The function **GrB.extract**($y, \emptyset, \emptyset, x, index, n, \emptyset$) extracts a sub-vector y from the specified positions in an input vector x . We can regard this operation as $y[i] \leftarrow x[index[i]]$ for $i \in [0 \dots n - 1]$ where n is the length of the array *index* and also the vector y .
- The function **GrB.assign**($y, \emptyset, accum, x, index, n, \emptyset$) assigns the entries from the input vector x to the specified positions of an output vector y . We can regard it as $y[index[i]] \leftarrow x[i]$ for $i \in [0 \dots n - 1]$ where n is the length of the array *index* and also the vector x . *accum* is the same as the one in **GrB.mxv**.
- The function **GrB.eWiseMult**($y, \emptyset, \emptyset, binop, x_1, x_2, \emptyset$) performs the element-wise (generalized) multiplication on the intersection of elements of two vectors x_1 and x_2 and outputs the vector y .
- The function **GrB.Vector.extractTuples**(*index*, *value*, $\&n, f$) extracts the nonzero elements (tuples of *index*

and value) from vector f into two separate arrays *index* and *value*. It returns the element count to n .

For the rest functions, we have **GrB.Vector_dup** to duplicate a vector, **GrB.reduce** to reduce a vector to a scalar value through a user-specified binary operation, and **GrB.Matrix_nrows** to obtain the dimension of a matrix.

Algorithm 3 describes the FastSV algorithm in GraphBLAS. Before every iteration, we calculate the initial grandparent gf for every vertex. First, we perform the stochastic hooking in line 9–10. GraphBLAS has no primitive that directly implements the parallel-for on an edge list (line 9 in Algorithm 2), so we have to first aggregate v 's grandparent $gf[v]$ to vertex u for every $(u, v) \in E$, obtaining the vector $mngf[u] = \min_{v \in N(u)} gf[v]$. This can be implemented by a matrix-vector multiplication $mngf = \mathbf{A} \cdot gf$ using the (select2nd, min) semiring. Next, the hooking operation is implemented by the assignment $f[f[u]] \leftarrow mngf[u]$ for every vertex u . This is exactly the **GrB.assign** function in line 10 where the indices are the values of vector f extracted in either line 5 before the first iteration or line 16 from the previous iteration. The accumulator **GrB_MIN** prevents the nondeterminism caused by the modification to the same entry of f , and the minimum operation gives the best convergence in practice.

Aggressive hooking is then implemented by an element-wise multiplication $f \leftarrow \min(f, mngf)$ in line 13. Although it is another operation in FastSV that performs the parallel-for on an edge list, it can reuse the vector $mngf$ computed in the previous step, so the aggressive hooking is actually efficient. Shortcutting is also implemented by an the element-wise multiplication $f \leftarrow \min(f, gf)$ in line 15. Next, we calculate the grandparent vector $gf[u] \leftarrow f[f[u]]$. It is implemented by the **GrB.extract** function in line 17 where the indices are the values of f extracted in line 16.

At the end of each iteration, we calculate the number of modified entries in gf in line 19 – 20 to check whether the algorithm has converged or not. A copy of gf is stored in the vector dup for determining the termination in the next iteration.

4.2 Distributed implementation using CombBLAS. The distributed version of FastSV is implemented in CombBLAS [8]. CombBLAS provides all operations needed for FastSV, but its API differs from the GraphBLAS standard. GraphBLAS's *collections* (matrices and vectors) are opaque datatypes whose internal representations (sparse or dense) are not exposed to users, but CombBLAS distinguishes them in the user interface. Then, GraphBLAS's functions often consist of multiple operations (like masking, accumulation and

Algorithm 3 The linear algebra FastSV algorithm.
Input: The adjacency matrix \mathbf{A} and the parent vector f . **Output:** The parent vector f .

```

1: procedure FASTSV( $\mathbf{A}, f$ )
2:   GrB_Matrix_nrows ( $\&n, \mathbf{A}$ )
3:   GrB_Vector_dup ( $\&gf, f$ )  $\triangleright$  initial grandparent
4:   GrB_Vector_dup ( $\&dup, gf$ )  $\triangleright$  duplication of  $gf$ 
5:   GrB_Vector_extractTuples (index, value,  $\&n, f$ )
6:   Sel2ndMin  $\leftarrow$  a (select2nd, Min) semiring
7:   repeat
8:      $\triangleright$  Step 1: Stochastic hooking
9:     GrB_mxv ( $mngf, \emptyset, \text{GrB\_MIN}, \text{sel2ndMin}, \mathbf{A}, gf, \emptyset$ )
10:    GrB_assign ( $f, \emptyset, \text{GrB\_MIN}, mngf, \text{value}, n, \emptyset$ )
11:     $\triangleright$  Step 2: Aggressive hooking
12:    GrB_eWiseMult ( $f, \emptyset, \emptyset, \text{GrB\_MIN}, f, mngf, \emptyset$ )
13:     $\triangleright$  Step 3: Shortcutting
14:    GrB_eWiseMult ( $f, \emptyset, \emptyset, \text{GrB\_MIN}, f, gf, \emptyset$ )
15:     $\triangleright$  Step 4: Calculate grandparents
16:    GrB_Vector_extractTuples (index, value,  $\&n, f$ )
17:    GrB_extract ( $gf, \emptyset, \emptyset, f, \text{value}, n, \emptyset$ )
18:     $\triangleright$  Step 5: Check termination
19:    GrB_eWiseMult ( $diff, \emptyset, \emptyset, \text{GxB\_ISNE}, dup, gf, \emptyset$ )
20:    GrB_reduce ( $\&sum, \emptyset, \text{Add}, diff, \emptyset$ )
21:    GrB_Vector_dup ( $\&dup, gf$ )
22:  until sum = 0

```

the main operation) as described in Section 4.1, while in CombBLAS we usually perform a single operation at a time. Despite these differences, a straightforward implementation of FastSV on CombBLAS can be obtained by transforming each GraphBLAS function to the semantically equivalent ones in CombBLAS, using dense vectors in all scenarios.

The parallel complexity of the main linear algebraic operations used in FastSV (the vector variants of **GrB.extract** and **GrB.assign**, and the **GrB.mxv**), as well as the potential optimizations are discussed in the LACC paper [6]. Due to the similarity of FastSV and LACC in the algorithm logic, they can be optimized by the similar optimization techniques. We briefly summarize them below.

Broadcasting-based implementation for the extract and assign operations. The *extract* and *assign* operations fetch or write data on the specified locations of a vector, which may cause a load balancing issue when there is too much access on a few locations. In FastSV, these locations are exactly the set of parent vertices in the pointer graph, and due to the skewed structure of the pointer graph, the root vertices (especially those belonging to a large component) will have extremely high workload. When using the default *assign* and *extract* implementations in CombBLAS via all-to-all communication, several processes become the bottleneck and slow down the whole operation signif-

icantly. The solution is a manual implementation of these two operations via the detection of the hot spots and broadcasting the entries on those processes.

Taking advantage of the sparsity. The matrix-vector multiplication $mngf = \mathbf{A} \cdot gf$ is an expensive operation in FastSV (see our performance profiling in Section 5.7). The straightforward implementation naturally chooses the sparse-matrix dense-vector (SpMV) multiplication, since all the vectors in FastSV are dense vectors. Alternatively, we can use an incremental implementation by computing $\Delta mngf = \mathbf{A} \cdot (\Delta gf)$, where $\Delta gf = gf - gf_{prev}$ containing only the modified entries of gf is stored as a sparse vector, so the multiplication is the sparse-matrix sparse-vector multiplication (SpM-SpV) [5]. Depending on the sparsity of Δgf , SpM-SpV could have much lower computation and communication cost than SpMV. We use a threshold on the portion of modified entries of gf to decide which method to use in each iteration, which effectively reduces the computation time. Section 5.7 presents a detailed evaluation.

5 Experiments

In this section, we evaluate various aspects of FastSV showing its fast convergence, shared- and distributed-memory performance, scalability and several other performance characteristics. We compare FastSV with LACC [6] that has demonstrated superior performance over other distributed-memory parallel CC algorithms. Table 1 shows a diverse collection of large graphs used to evaluate CC algorithms. To the best of our knowledge, the Hyperlink graph [19] with 3.27B vertices and 124.90B edges is the largest publicly available graph.

5.1 Evaluation platform. We evaluate the performance of distributed algorithms on NERSC Cori supercomputer. Each node of Cori has Intel KNL processor with 68 cores and 96GB of memory. All operations in CombBLAS are parallelized with OpenMP and MPI. Given p MPI processes, we always used a square $\sqrt{p} \times \sqrt{p}$ process grid. In our experiments, we used 16 threads per MPI process. The execution pattern of our distributed algorithm follows the bulk synchronous parallel (BSP) model, where all MPI processes perform local computation followed by synchronized communication rounds.

We also show the shared-memory performance of FastSV implemented using the SuiteSparse:GraphBLAS library [3]. These experiments are conducted on Amazon EC2’s r5.4xlarge instance (128G memory, 16 threads).

5.2 Speed of convergence. At first, we show how different hooking strategies impact the convergence of

SV and FastSV algorithms. We start with the simplified SV algorithm (Algorithm 1) and incrementally add different hooking strategies as shown in Figure 4. The rightmost bars report the number of iterations needed by LACC.

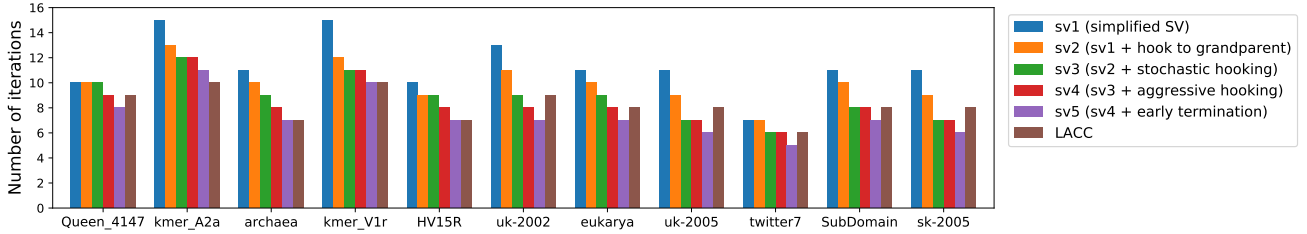
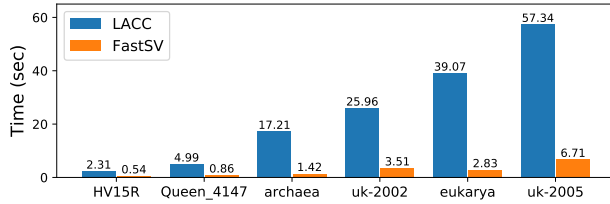
Figure 4 shows that the simplified SV without unconditional hooking can take up to $1.57\times$ more iterations than LACC. We note that despite needing more iterations, Algorithm 1 can run faster than LACC in practice because each iteration of the former is faster than each iteration of the latter. Figure 4 demonstrates that SV converges faster as we incrementally apply advanced hooking strategies. In fact, every hooking strategy improves the convergence of some graphs, and their combination improves the convergence of all graphs. Finally, the early termination discussed in Section 3.4 always removes an additional iteration needed by other algorithms. With all improvements, sv5 which represents Algorithm 2, on average reduces 35.0% iterations (min 20%, max 46.2%) from Algorithm 1. Therefore, FastSV converges as quickly as, or faster than, LACC.

5.3 Performance in shared-memory platform using SuiteSparse:GraphBLAS. To check the correctness of Algorithm 3, we implemented it in SuiteSparse:GraphBLAS, a multi-threaded implementation of the GraphBLAS standard. LACC also has an unoptimized SuiteSparse:GraphBLAS implementation available as part of the LAGraph library [18]. We compare the performance of Figure 5 and LACC in this setting on an Amazon EC2’s r5.4xlarge instance with 16 threads. Figure 5 shows that FastSV is significantly faster than LACC (avg. $8.66\times$, max $13.81\times$). Although both algorithms are designed for distributed-memory platforms, we still observe better performance of FastSV, thanks to its simplicity.

5.4 Performance in distributed-memory platform using CombBLAS. We now evaluate the performance of FastSV implemented using CombBLAS and compare its performance with LACC on Cori supercomputer. Both algorithms are implemented in CombBLAS, so they share quite a lot of common operations and optimization techniques (see Section 4.2), making it a fair comparison between the two algorithms. Generally, FastSV operates with simpler computation logic and uses less expensive parallel operations than LACC. However, depending on the structure of the graph, LACC can detect the already converged connected components on the fly and can potentially use more sparse operations. Hence, the structure of the input graph often influences the relative performance of

Table 1: Graph datasets used to evaluate the parallel connected component algorithms.

Graph	Vertices	Directed edges	Components	Description
Queen_4147	4.15M	166.82M	1	3D structural problem [13]
kmer_A2a	170.73M	180.29M	5353	Protein k-mer graphs from GenBank [13]
archaea	1.64M	204.78M	59794	archaea protein-similarity network [7]
kmer_V1r	214.01M	232.71M	9	Protein k-mer graphs, from GenBank [13]
HV15R	2.02M	283.07M	1	Computational Fluid Dynamics Problem [13]
uk-2002	18.48M	298.11M	1990	2002 web crawl of .uk domain [13]
eukarya	3.24M	359.74M	164156	eukarya protein-similarity network [7]
uk-2005	39.45M	936.36M	7727	2005 web crawl of .uk domain [13]
twitter7	41.65M	1.47B	1	twitter follower network [13]
SubDomain	82.92M	1.94B	246969	1st-level subdomain graph extracted from Hyperlink [19]
sk-2005	50.64M	1.95B	45	2005 web crawl of .sk domain [13]
MOLIERE_2016	30.22M	3.34B	4457	automatic biomedical hypothesis generation system [13]
Metaclust50	282.20M	37.28B	15982994	similarities of proteins in Metaclust50 [7]
Hyperlink	3.27B	124.90B	29360027	hyperlink graph extract from the Common Crawl [19]

**Figure 4:** Number of iterations the simplified SV takes after performing each of the optimizations (sv5 is exactly our FastSV algorithm), and the number of iterations LACC takes.**Figure 5:** Performance of the parallel FastSV and LACC in SuiteSparse:GraphBLAS on six small graphs.

these algorithms.

Figure 6 summarizes the performance of FastSV and LACC on twelve small datasets. We observe that both FastSV and LACC scale to 4096 cores on all the graphs, and for the majority of the graphs (8 out of 12), they continue scaling to 16384 cores. The four graphs on which they stop scaling are just too small that both algorithms finish within 2 seconds. FastSV outperforms LACC on all instances. On 256 cores, FastSV is 2.80 \times faster than LACC on average (min 1.66 \times , max 4.27 \times). When increasing the number of nodes, the performance gap between FastSV and LACC shrinks slightly, but FastSV is still 2.53 \times , 1.97 \times and 1.61 \times faster than LACC on average on 1024, 4096 and 16384 cores, respectively.

To see how the performance of FastSV and LACC are affected by the graph structure, we plot the average degree ($|E|/|V|$) and the speedup of FastSV over

LACC for each graph (using 1024 cores) in Figure 7. Generally, FastSV tends to outperform LACC by a significant margin on denser graphs. This is mainly due to fewer matrix-vector multiplications used in FastSV, whose parallel complexity is highly related to the density of the graph. The outliers *archaea* and *eukarya* are graphs with a large number of small connected components: they have more than 30% converged vertices detected early. On such graphs, LACC’s detection of converged connected components provides it with better opportunities to employ sparse operations, while such detection is not allowed in FastSV. Nevertheless, LACC’s sparsity optimization still cannot compensate its high computational cost in each iteration.

5.5 Performance of FastSV with bigger graphs.

We separately analyze the performance of FastSV and LACC on the two largest graphs in Table 1. Hyperlink is perhaps the largest publicly available graph, making it the largest connectivity problem we can currently solve. Since each of these two graphs requires more than 1TB memory, it may be impossible to process them on a typical shared-memory server. Figure 8 shows the strong scaling of both algorithms and the better performance of FastSV. On the smaller graph Metaclust50, both algorithms scale to 65,536 cores where FastSV is 1.47 \times faster than LACC. On the Hyperlink graph containing 124.9 billion edges, they continue scaling to 262,144

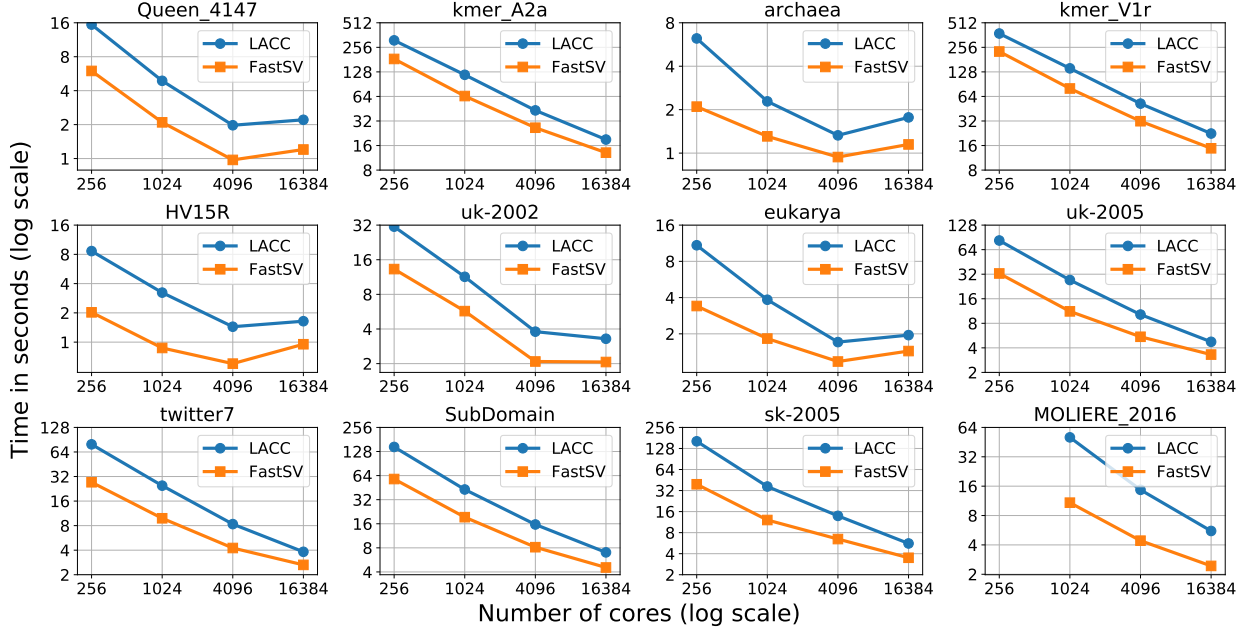


Figure 6: Strong scaling of distributed-memory FastSV and LACC using up to 16384 cores (256 nodes).

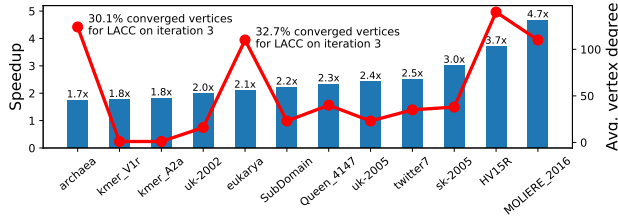


Figure 7: The speedup of FastSV over LACC on twelve small datasets using 256 cores (bar chart) and each graph's density in terms of average vertex degree (line chart). A positive correlation between the two metrics can be observed, except for the two outliers *archaea* and *eukarya* where LACC makes use of the detection of converged connected components to accelerate its execution.

cores, where FastSV achieves an $2.03\times$ speedup over the LACC algorithm.

5.6 Performance characteristics for operations.

Figure 9 shows the execution time of FastSV by breaking the runtime into three parts: finding the grandparent, matrix-vector multiplication, the hooking operations. The time spent on checking the termination is omitted, since it is insignificant relative to other operations. Each of these operations contributes significantly to the total execution time. Finding the grandparent and the hooking operations basically reflect the parallel complexity of the **extract** and **assign** operations, and the ratio of them is relatively stable for all graphs. By contrast, the execution time of SpMV varies considerably across different graphs, because SpMV's complex-

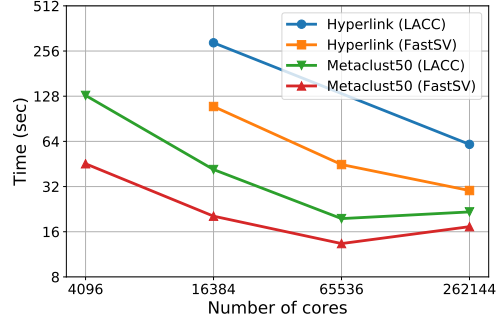


Figure 8: Performance of FastSV and LACC with two large graphs on CoriKNL (up to 262144 cores using 4096 nodes).

ity depends on the density of a graph.

5.7 Execution time reduced by the sparsity optimization.

As mentioned in Section 4.2, FastSV dynamically selects SpMV or SpMSpV based on the changes in the grandparent vector gf . This optimization is particularly effective for high-density graphs where SpMV usually dominates the runtime (see Figure 9). Figure 10 explains the benefit of sparsity with four representative graphs, where we plot the number of vertices modified in each iteration. We observe that only a small fraction of vertices participates in the last few iterations when SpMSpV can be used instead of SpMV. As shown by the red runtime lines in Figure 10, the use of SpMSpV drastically reduces the runtime of the last few iterations.

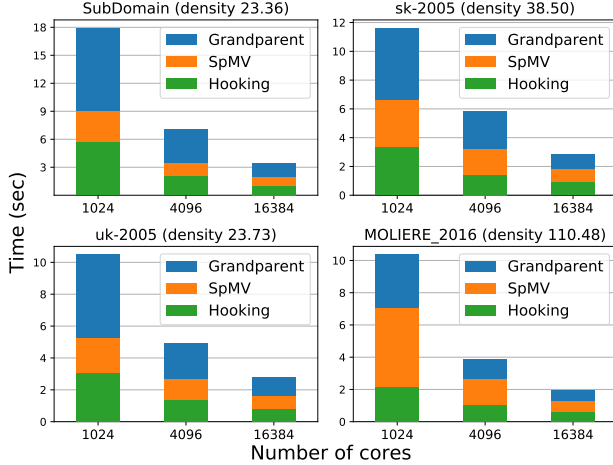


Figure 9: Performance breakdown of FastSV on four representative graphs.

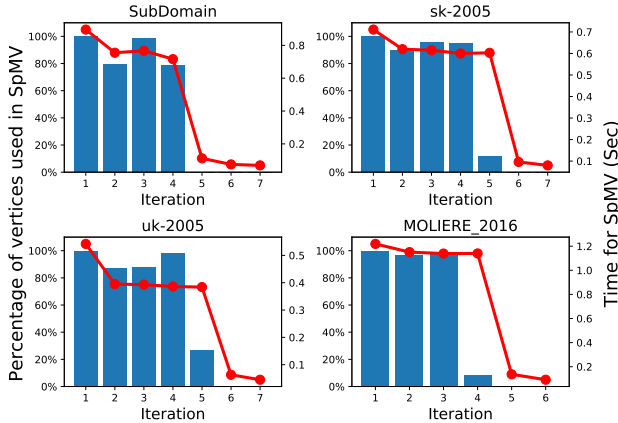


Figure 10: Percentage of vertices that participate the SpMV (sparse matrix-vector multiplication) operation for each iteration (bar chart), and the runtime for SpMV (line chart). A vertex participates the SpMV if its grandparent gf is not changed in the previous iteration.

6 Related work

Finding the connected components of an undirected graph is a well-studied problem in PRAM model. Many of these algorithms such as the Shiloach-Vishkin (SV) algorithm assume the CRCW (concurrent-read and concurrent-write model) model. The SV algorithm [22] takes $O(\log n)$ time on $O(m + n)$ processors. The Awerbuch-Shiloach (AS) algorithm [4] is a simplification of SV by using a different termination condition. Transforming the complete SV or AS to distributed-memory is possible [6, 27], but the detection of stagnant trees in SV’s *unconditional hooking* step is in fact not suitable for a distributed-memory implementation, which introduces considerable computation and communication cost. Therefore, we based our distributed-memory FastSV on a simplified SV algorithm preserving only the

essential steps and introduce efficient hooking steps to ensure fast convergence in practice.

There are several distributed-memory connected component algorithms proposed in the literature. Parallel BFS is a popular method that are implemented and optimized in various systems [9, 12, 26, 32], but its complexity is bounded by the diameter of the graph, so it is mainly used on small-world networks. LACC [6] is the state-of-the-art algorithm prior to our work, which guarantees the convergence in $\log(n)$ iterations by transforming the complete AS algorithm into linear algebra operations. FastSV’s high performance comes from a much simplified computation logic than LACC. ParConnect [16] is another distributed-memory algorithm that adaptively uses parallel BFS and SV and dynamically selects which method to use. For other software architectures, there are Hash-Min [21] for MapReduce systems and S-V PPA [27] for vertex-centric message passing systems [17]. MapReduce algorithms tend to perform poorly in tightly-couple parallel systems our work targets, compared to the loosely-coupled architectures that are optimized for cloud workloads. The S-V PPA algorithm, due to the requirement of communicating between non-neighboring vertices, is only supported by several Pregel-like systems [1, 2, 30], and these frameworks tend to have limited scalability on multi-core clusters due to the lack of multi-threading support.

7 Conclusion

In this paper, we present a new distributed-memory connected component algorithm FastSV that is scalable to hundreds of thousands processors on modern supercomputers. FastSV achieves its efficiency by first keeping the backbones of the Shiloach-Vishkin algorithm and then employing several novel hooking strategies for fast convergence. FastSV attains high performance by employing scalable GraphBLAS operations and optimized communication. Given the generic nature of our algorithm, it can be easily implemented for any computing platforms such as using GraphBLAST [28] for GPUs and using any programming language such as using pygraphblas (<https://github.com/michelp/pygraphblas>) in Python.

Finding CCs is a fundamental operation in many large-scale applications such as metagenome assembly and protein clustering. With the exponential growth of genomic data, these applications will generate graphs with billions of vertices and trillions of edges and will use upcoming exascale computers to solve scientific problems. FastSV is generic enough to be integrated with many existing libraries and scalable enough to be integrated with massively-parallel applications.

References

- [1] Apache Giraph. <http://giraph.apache.org/>.
- [2] Pregel+. <http://www.cse.cuhk.edu.hk/pregelplus/>.
- [3] SuiteSparse:GraphBLAS. <http://faculty.cse.tamu.edu/davis/GraphBLAS.html>.
- [4] B. Awerbuch and Y. Shiloach. New connectivity and MSF algorithms for shuffle-exchange network and PRAM. *IEEE Transactions on Computers*, (10):1258–1263, 1987.
- [5] A. Azad and A. Buluç. A work-efficient parallel sparse matrix-sparse vector multiplication algorithm. In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 688–697. IEEE, 2017.
- [6] A. Azad and A. Buluç. LACC: A linear-algebraic algorithm for finding connected components in distributed memory. In *Proceedings of the IPDPS*, pages 2–12. IEEE, 2019.
- [7] A. Azad, G. A. Pavlopoulos, C. A. Ouzounis, N. C. Kypides, and A. Buluç. HipMCL: a high-performance parallel implementation of the markov clustering algorithm for large-scale networks. *Nucleic Acids Research*, 46(6):e33–e33, 2018.
- [8] A. Buluç and J. R. Gilbert. The combinatorial BLAS: Design, implementation, and applications. *The International Journal of High Performance Computing Applications*, 25(4):496–509, 2011.
- [9] A. Buluç and K. Madduri. Parallel breadth-first search on distributed memory systems. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, page 65. ACM, 2011.
- [10] A. Buluç, T. Mattson, S. McMillan, J. Moreira, and C. Yang. Design of the GraphBLAS API for C. In *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 643–652. IEEE, 2017.
- [11] A. Buluc, T. Mattson, S. McMillan, J. Moreira, and C. Yang. The GraphBLAS C API specification. *GraphBLAS.org, Tech. Rep.*, 2017.
- [12] R. Chen, J. Shi, Y. Chen, and H. Chen. PowerLyra: Differentiated graph computation and partitioning on skewed graphs. In *Proceedings of the Tenth European Conference on Computer Systems*, page 1. ACM, 2015.
- [13] T. A. Davis and Y. Hu. The University of Florida sparse matrix collection. *ACM Transactions on Mathematical Software (TOMS)*, 38(1):1, 2011.
- [14] K. Ekanadham, W. P. Horn, M. Kumar, J. Jann, J. Moreira, P. Pattnaik, M. Serrano, G. Tanase, and H. Yu. Graph programming interface (GPI): a linear algebra programming model for large scale graph computations. In *Proceedings of the ACM International Conference on Computing Frontiers*, pages 72–81. ACM, 2016.
- [15] J. Greiner. A comparison of parallel algorithms for connected components. In *Proceedings of the sixth annual ACM symposium on Parallel algorithms and architectures*, pages 16–25. ACM, 1994.
- [16] C. Jain, P. Flick, T. Pan, O. Green, and S. Aluru. An adaptive parallel algorithm for computing connected components. *IEEE Transactions on Parallel and Distributed Systems*, 28(9):2428–2439, 2017.
- [17] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *SIGMOD*, pages 135–146. ACM, 2010.
- [18] T. Mattson, T. A. Davis, M. Kumar, A. Buluc, S. McMillan, J. Moreira, and C. Yang. Lagraph: A community effort to collect graph algorithms built on top of the graphblas. In *IPDPS Workshops*, pages 276–284. IEEE, 2019.
- [19] R. Meusel, S. Vigna, O. Lehmberg, and C. Bizer. Graph structure in the web—revisited: a trick of the heavy tail. In *Proceedings of the 23rd international conference on World Wide Web*, pages 427–432. ACM, 2014.
- [20] A. Pothén and C.-J. Fan. Computing the block triangular form of a sparse matrix. *ACM Transactions on Mathematical Software (TOMS)*, 16(4):303–324, 1990.
- [21] V. Rastogi, A. Machanavajjhala, L. Chitnis, and A. D. Sarma. Finding connected components in map-reduce in logarithmic rounds. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*, pages 50–61. IEEE, 2013.
- [22] Y. Shiloach and U. Vishkin. An $O(\log n)$ parallel connectivity algorithm. *Journal of Algorithms*, 3:57–67, 1982.
- [23] N. Sundaram, N. Satish, M. M. A. Patwary, S. R. Dulloor, M. J. Anderson, S. G. Vadlamudi, D. Das, and P. Dubey. GraphMat: High performance graph analytics made productive. *Proceedings of the VLDB Endowment*, 8(11):1214–1225, 2015.
- [24] M. Sutton, T. Ben-Nun, and A. Barak. Optimizing parallel graph connectivity computation via subgraph sampling. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 12–21. IEEE, 2018.
- [25] S. M. Van Dongen. *Graph clustering by flow simulation*. PhD thesis, 2000.
- [26] R. S. Xin, J. E. Gonzalez, M. J. Franklin, and I. Stoica. GraphX: A resilient distributed graph system on Spark. In *First International Workshop on Graph Data Management Experiences and Systems*, page 2. ACM, 2013.
- [27] D. Yan, J. Cheng, K. Xing, Y. Lu, W. Ng, and Y. Bu. Pregel algorithms for graph connectivity problems with performance guarantees. *Proceedings of the VLDB Endowment*, 7(14):1821–1832, 2014.
- [28] C. Yang, A. Buluc, and J. D. Owens. Graphblast: A high-performance linear algebra-based graph framework on the gpu. *arXiv preprint arXiv:1908.01407*, 2019.
- [29] X. D. Yang. An improved algorithm for labeling connected components in a binary image. Technical report, Cornell University, 1989.
- [30] Y. Zhang and Z. Hu. Composing optimization tech-

niques for vertex-centric graph processing via communication channels. pages 428–438, 2019.

- [31] Y. Zhang, H.-S. Ko, and Z. Hu. Palgol: A high-level DSL for vertex-centric graph processing with remote data access. In *Proceedings of the 15th Asian Symposium on Programming Languages and Systems*, pages 301–320. Springer, 2017.
- [32] X. Zhu, W. Chen, W. Zheng, and X. Ma. Gemini: A computation-centric distributed graph processing system. In *OSDI*, pages 301–316, 2016.

$f[f[v]]$ will not be performed since for all $(u, v) \in E$ we have $f[f[u]] \leq f[f[v]]$. Shortcutting will not be performed either since every vertex points to a root now (by Lemma A.3). Then, no hooking operation can be performed, and the vector f remains unchanged afterwards. \square

A Correctness of the early termination

Lemma 3.1 states that, in FastSV if the grandparent $f[f]$ remains unchanged after an iteration, then the vector f will not be changed afterwards. The proof makes use of the following lemmas.

LEMMA A.1. *During the whole algorithm, $f[u] \leq u$ holds for all vertices u .*

Proof. Initially, $f[u] = u$ for all vertex u and the lemma holds trivially. The operation $\xleftarrow{\min}$ ensures that f can only decrease, so the lemma always holds. \square

LEMMA A.2. *After an iteration, if the grandparent $f[f]$ remains unchanged, then every vertex hooks onto its grandparent in the previous operation.*

Proof. By contradiction. Suppose u changes its pointer to some v other than $f[f[u]]$, then since it overrides the shortcutting operation $f_{next}[u] \xleftarrow{\min} f[f[u]]$ we know that $v < f[f[u]]$. By Lemma A.1, u 's new grandparent $f_{next}[v] \leq v < f[f[u]]$, then the grandparent of u is changed. \square

LEMMA A.3. *After an iteration, if the grandparent $f[f]$ remains unchanged, then every vertex points to a root now.*

Proof. By contradiction. Suppose u 's new parent v is not a root, then u 's new grandparent is $f_{next}[v] < v = f[f[u]]$ (by Lemma A.1 and Lemma A.2), which means u 's grandparent has changed. \square

Here we prove Lemma 3.1.

Proof. We show that no hooking operation will be performed if $f[f]$ remains unchanged after an iteration. The aggressive hooking in the form of $f_{next}[u] \xleftarrow{\min} f[f[v]]$ is overridden by the shortcutting operation $f_{next}[u] \xleftarrow{\min} f[f[u]]$ in the previous iteration (by Lemma A.2), meaning that $f[f[u]] \leq f[f[v]]$ for all $(u, v) \in E$. Since then, $f[f]$ is not changed, so the aggressive hooking will not be performed in the current iteration either. The stochastic hooking $f_{next}[f[u]] \xleftarrow{\min}$