Parallel Algorithms for Finding Connected Components using Linear Algebra

Yongzhe Zhang[a], Ariful Azad[b], Aydın Buluç[c]

[a]*Department of Informatics, The Graduate University for Advanced Studies, SOKENDAI, Japan*
[b]*Department of Intelligent Systems Engineering, Indiana University, Bloomington, IN, USA*
[c]*Computational Research Division, Lawrence Berkeley National Laboratory, Berkeley, CA, USA*

Abstract

Finding connected components is one of the most widely used operations on a graph. Optimal serial algorithms for the problem have been known for half a century, and many competing parallel algorithms have been proposed over the last several decades under various different models of parallel computation. This paper presents a class of parallel connected-component algorithms designed using linear-algebraic primitives. These algorithms are based on a PRAM algorithm by Shiloach and Vishkin and can be designed using standard GraphBLAS operations. We demonstrate two algorithms of this class, one named LACC for Linear Algebraic Connected Components, and the other named FastSV which can be regarded as LACC's simplification. With the support of the highly-scalable Combinatorial BLAS library, LACC and FastSV outperform the previous state-of-the-art algorithm by a factor of up to 12x for small to medium scale graphs. For large graphs with more than 50B edges, LACC and FastSV scale to 4K nodes (262K cores) of a Cray XC40 supercomputer and outperform previous algorithms by a significant margin. This remarkable performance is accomplished by (1) exploiting sparsity that was not present in the original PRAM algorithm formulation, (2) using high-performance primitives of Combinatorial BLAS, and (3) identifying hot spots and optimizing them away by exploiting algorithmic insights.

1. Introduction

Given an undirected graph $G = (V, E)$ on the set of vertices V and the set of edges E, a connected component (CC) is a subgraph in which every vertex is connected to all other vertices in the subgraph by paths and no vertex in the subgraph is connected to any other vertex outside of the subgraph. Finding all connected components in a graph is a well studied problem in graph theory with applications in bioinformatics [1] and scientific computing [2, 3].

Parallel algorithms for finding connected components also have a long history, with several ingenious techniques applied to the problem. One of the most well-known parallel algorithms is due to Shiloach and Vishkin [4], where they introduced the hooking procedure. The algorithm also uses pointer jumping, a fundamental technique in PRAM (parallel random-access machine) algorithms, for shortcutting. Awerbuch and Shiloach [5] later simplified and improved on this algorithm. Despite the fact that PRAM model is a poor fit for analyzing distributed memory algorithms, we will show in this paper that the Shiloach-Vishkin (SV) and Awerbuch-Shiloach (AS) algorithms admit a very efficient parallelization using proper computational primitives and sparse data structures.

Decomposing the graph into its connected components is often the first step in large-scale graph analytics where the goal is to create manageable independent subproblems. Therefore, it is important that connected component finding algorithms can run on distributed memory, even if the subsequent steps of the analysis need not. Several applications of distributed-memory connected component labeling have recently emerged in the field of genomics. The metagenome assembly algorithms represent their partially assembled data as a graph [6, 7]. Each component of this graph can be processed independently. Given that the scale of the metagenomic data that needs to be assembled is already on the order of several TBs, and is on track to grow exponentially, distributed connected component algorithms are of growing importance.

Another application comes from large scale biological network clustering. The popular Markov clustering algorithm (MCL) [1] iteratively performs a series of sparse matrix manipulations to identify the clustering structure in a network. After the iterations converge, the clusters are extracted by finding the connected components on the symmetrized version of the final converged matrix, i.e., in an undirected graph represented by the converged matrix. We have recently developed the distributed-memory parallel MCL (HipMCL) [8] algorithm that can cluster protein similarity networks with hundreds of billions of edges using thousands of nodes on modern supercomputers. Since computing connected components is an important step in HipMCL, a parallel connected component algorithm that can scale to thousands of nodes is imperative.

In this paper, we present two parallel algorithms based on the SV and AS algorithms. These algorithms are specially designed by mapping different operations to the GraphBLAS [9] primitives, which are standardized linear-algebraic functions that can be used to implement graph algorithms. The linear-algebraic algorithm which is derived from the AS algorithm

is named as LACC for *linear algebraic connected components*. The second algorithm is a simplification of the SV algorithm and is named as FastSV due to its improved convergence in practice. FastSV can also be considered a simplification of LACC. While the initial reasons behind choosing the SV and AS algorithms were simplicity, performance guarantees, and expressibility using linear algebraic primitives, we found that they are never slower than the state-of-the-art distributed-memory algorithm ParConnect [10], and they often outperform ParConnect by several folds.

LACC and FastSV algorithms are published as conference papers [11, 12]. This journal paper extends those algorithms by providing a unified framework for CC algorithms in linear algebra. Designing CC algorithms using a standard set of linear-algebraic operations gives a crucial benefit. After an algorithm is mapped to GraphBLAS primitives, we can rely on any library providing high-performance implementations of those primitives. In this paper we use the Combinatorial BLAS (CombBLAS) library [13], a well-known framework for implementing graph algorithms in the language of linear algebra. Different from the original SV and AS algorithms, our implementations fully exploit vector sparsity and avoids processing on inactive vertices. We perform several additional optimizations to eliminate performance hot spots and provide a detailed breakdown of our parallel performance, both in terms of theoretical communication complexity and in experimental results. These algorithmic insights and optimizations result in a distributed algorithm that scales to 4K nodes (262K cores) of a Cray XC40 supercomputer and outperforms previous algorithms by a significant margin. We also implemented our algorithm using SuiteSparse:GraphBLAS [14], a multi-threaded implementation of the GraphBLAS standard. The performance of the shared-memory implementations is comparable to state-of-the-art CC algorithms designed for share-memory platforms.

Distributed-memory LACC and FastSV codes are publicly available as part of the CombBLAS library[1]. Shared-memory GraphBLAS implementations are also committed to the LAGraph Library [15][2]. This paper is an extension of a conference paper [11] published in IPDPS 2019.

2. Background

2.1. Notations

This paper only considers an undirected and unweighted graph $G = (V, E)$ with n vertices and m edges. Given a vertex v, $N(v)$ is the set of vertices adjacent to v. A tree is an undirected graph where any two vertices are connected by exactly one path. A directed rooted tree is a tree in which a vertex is designated as the root and all vertices are oriented toward the root. The level $l(v)$ of a vertex v in a tree is 1 plus the number of edges between v and the root. The level of the root is 1. A tree is called a *star* if every vertex is a child of the root (the root

is a child of itself). A vertex is called a *star vertex* is it belongs to a star.

2.2. GraphBLAS

The duality between sparse matrices and graphs has a long and fruitful history [16, 17]. Several independent systems have emerged that use matrix algebra to perform graph operations [13, 18, 19]. Recently, the GraphBLAS forum defined a standard set of linear-algebraic operations for implementing graph algorithms, leading to the GraphBLAS C API [20]. In this paper, we will use the functions from the GraphBLAS API to describe our algorithms. That being said, our algorithms run on distributed memory while currently no distributed-memory library faithfully implements the GraphBLAS API. The most recent version of the API (1.2.0) is actually silent on distributed-memory parallelism and data distribution. Consequently, while our descriptions follow the API, our implementation will be based on CombBLAS functions [13], which are either semantically equivalent in functionality to their GraphBLAS counterparts or can be composed to match GraphBLAS functionality.

2.3. Related work

Finding connected components of an undirected graph is one of the most well-studied problems in the PRAM (parallel random-access memory) model. A significant portion of these algorithms assume the CRCW (concurrent-read concurrent-write model). The Awerbuch-Shiloach (AS) algorithm is a simplification of the Shiloach-Vishkin (SV) algorithm [4]. The fundamental data structure in both AS and SV algorithms is a forest of rooted trees. While AS only keeps the information of the current forest, SV additionally keeps track of the forest in the previous iteration of the algorithm as well as the last time each parent received a new child. The convergence criterion for AS is to check whether each tree is a star whereas SV needs to see whether the last iteration provided any updates to the forest.

Randomization is also a fundamental technique applied to the connected components problem. The random-mate (RM) algorithm, due to Reif [21], flips an unbiased coin for each vertex to determine whether it is a parent or a child. Each child then finds a parent among its neighbors. The stars are contracted to supervertices in the next iteration as in AS and SV algorithms. All three algorithms described so far (RM, AS, and SV) are work inefficient in the sense that their processor-time product is asymptotically higher than the runtime complexity of the best serial algorithm.

A similar randomization technique allowed Gazit to discover a work-efficient CRCW PRAM algorithm for the connected components problem [22]. His algorithm runs with $O(m)$ optimal work and $O(\log(n))$ span. More algorithms followed achieving the same work-span bound but improving the state-of-the-art by working with more restrictive models such as EREW (exclusive-read exclusive-write) [23], solving more general problems such as minimum spanning forest [24] whose output can be used to infer connectivity, and providing first implementations [25].

The literature on distributed-memory connected component algorithms and their complexity analyses, is significantly

Algorithm 1 The skeleton of the AS algorithm. **Inputs:** an undirected graph $G(V, E)$. **Output:** The parent vector f

```
 1: procedure AWERBUCH-SHILOACH(G(V, E))
 2:     for every vertex v in V do ▷ Initialize
 3:         f[v] ← v
 4:     repeat
 5:         ▷ Step1: Conditional star hooking
 6:         for every edge {u, v} in E do in parallel
 7:             if u belongs to a star and f[u] > f[v] then
 8:                 f[f[u]] ← f[v]
 9:         ▷ Step2: Unconditional star hooking
10:         for every edge {u, v} in E do in parallel
11:             if u belongs to a star and f[u] ≠ f[v] then
12:                 f[f[u]] ← f[v]
13:         ▷ Step3: Shortcutting
14:         for every vertex v in V do in parallel
15:             gf[v] ← f[f[v]]
16:         for every vertex v in V do in parallel
17:             if v does not belongs to a star then
18:                 f[v] ← gf[v]
19:     until f remains unchanged
20:     return f
```

Algorithm 2 Finding vertices in stars. **Inputs:** a graph $G(V, E)$ and the parent vector f. **Output:** The *star* vector.

```
 1: procedure STARCHECK(G(V, E), f)
 2:     for every vertex v in V do in parallel  ▷ Initialize
 3:         star[v] ← true
 4:         gf[v] ← f[f[v]]
 5:     ▷ Exclude every vertex v with l(v) > 2 and its grandparent
 6:     for every vertex v in V do in parallel
 7:         if f[v] ≠ gf[v] then
 8:             star[v] ← false
 9:             star[gf[v]] ← false
10:     ▷ In nonstar trees, exclude vertices at level 2
11:     for every vertex v in V do in parallel
12:         star[v] ← star[f[v]]
13:     return star
```

where $f[v]$ stores the parent of a vertex v. 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.

The algorithm begins with n single-vertex trees and iteratively merges trees until no such merging is possible. This merging is performed by a process called *tree hooking*, where the root of a tree becomes a child of a vertex in another tree. Between two subsequent iterations, the algorithm reduces the height of trees by a process called *shortcutting*, where a vertex becomes a child of its grandparent.

The original Shiloach-Vishkin algorithm and its successor the Awerbuch-Shiloach algorithm used a conditional hooking step as well as an unconditional hooking step in every iteration. 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. With this general framework, the SV algorithm is guaranteed to finish in $O(\log n)$ iterations, where each iteration performs $O(m)$ parallel work.

sparser than the case for PRAM algorithms. The state-of-the-art prior to our work is the ParConnect algorithm [10], which is based on both the SV algorithm and parallel breadth-first search (BFS). Slota et al. [26] developed a distributed memory Multistep method that combines parallel BFS and label propagation technique. There have also been implementations of connected component algorithms in PGAS (partitioned global address space) languages [27] in distributed memory. Viral Shah's PhD thesis [28] presents a data-parallel implementation of the AS algorithm that runs on Matlab*P, a distributed variant of Matlab that is now defunct. Shah's implementation uses vastly different primitives than our own and solely relies on manipulating dense vectors, hence is limited in scalability.

Kiveras et al. [29] proposed the Two-Phase algorithm for MapReduce systems. Such 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. There is also recent work on parallel graph connectivity within the theory community, using various different models of computation [30, 31]. These last two algorithms are not implemented and its is not clear if such complex algorithms can be competitive in practice on real distributed-memory parallel systems.

3. Variants of the Shiloach-Vishkin (SV) algorithm

At first, we discuss the general framework of the SV algorithms. Based on this framework, we discuss two algorithmic variants that will be designed using linear algebra.

The SV algorithm and its variants maintain a forest (a collection of directed rooted trees), where each tree represents a connected component at the current stage of the algorithm. To represent trees, the algorithm maintains a parent vector f,

3.1. The Awerbuch-Shiloach (AS) algorithm

Awerbuch and Shiloach simplified the SV algorithm by allowing only stars to be hooked on to other trees [5]. Similar to the SV algorithm, the AS algorithm also needs both conditional and unconditional star hooking and shortcutting operations. To track vertices in stars, the algorithm maintains a Boolean vector *star*. For every vertex v, *star*[v] is *true* if v is a star vertex, *star*[v] is *false* otherwise.

Description of the algorithm. Algorithm 1 describes the main components of the AS algorithm. Initially, every vertex is its own parent, creating n single-vertex stars (lines 2-3 of Algorithm 1). In every iteration, the algorithm performs three operations: (a) conditional hooking, (b) unconditional hooking and (c) shortcutting. In the conditional hooking (lines 6-8), every edge {u, v} is scanned to see if (a) u is in a star and (b) the parent of u is greater than the parent of v. If these conditions are satisfied, $f[u]$ is hooked to $f[v]$ by making f[v] the parent
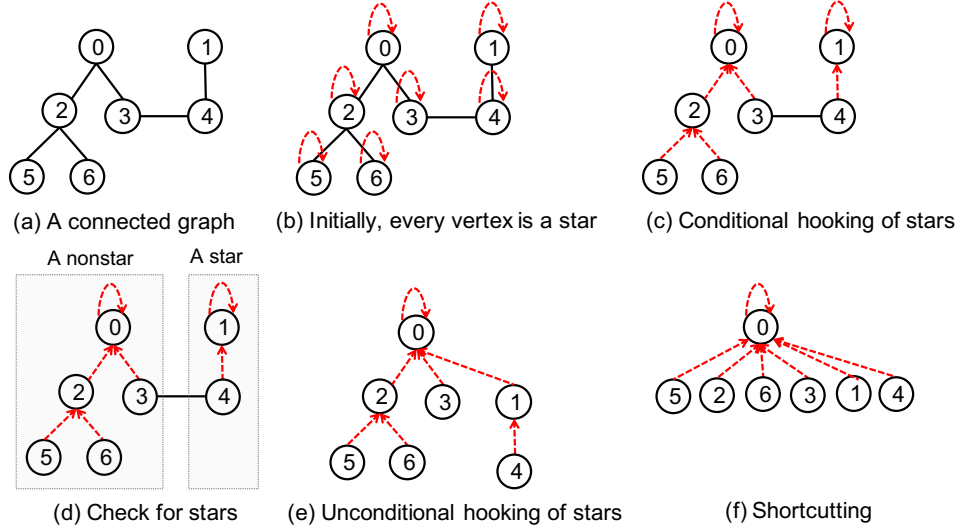
Figure 1: An illustrative example of the AS algorithm. Edges are shown in solid black edges. A dashed arrowhead connects a child with its parent. (a) An undirected and unweighted graph. (b) Initially, every vertex forms a singleton tree. (c) After conditional hooking. Here, we only show edges connecting vertices from different trees. (d) Identifying vertices in stars (see Figure 2 for details). (e) After unconditional hooking: the star rooted at vertex 1 is hooked onto the left tree rooted at vertex 0. (f) After shortcutting, all vertices belong to stars. The algorithm returns with a connected component.
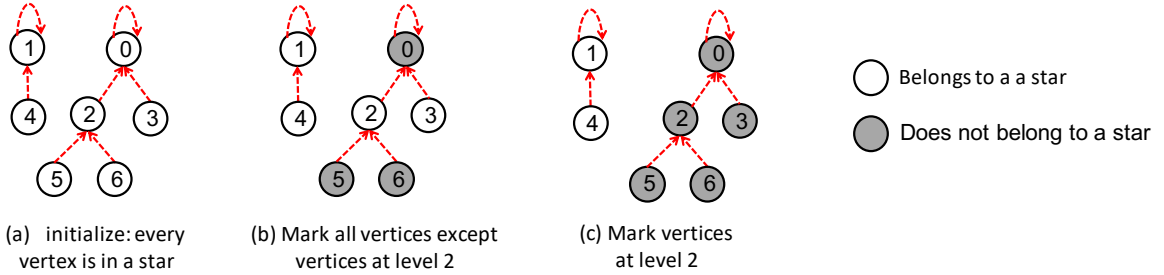

Figure 2: Finding star vertices. Star and nonstar vertices are shown with unfilled and filled circles, respectively. A dashed arrowhead connects a child with its parent. (a) Initially, every vertex is assumed to be a star vertex. (b) Every vertex v with $l(v) > 2$ and its grandparent are marked as nonstar vertices. (c) In a nonstar tree, vertices at level 2 are marked as nonstar vertices.

of f[u]. The remaining stars then get a chance to hook unconditionally (lines 10-12). In the shortcutting step, the grandparent of all vertices are identified and stored in the gf vector (lines 14-15). The gf vector is then used to update parents of all vertices (lines 16-18). Figure 1 shows the execution of different steps of the AS algorithm.

Algorithm 2 and Figure 2 describe how star vertices are identified based on the parent vector. Initially, every vertex v is assumed to be a star vertex by setting $star[v]$ to $true$ (line 3 of Algorithm 2). The algorithm marks vertices as nonstars if any of the following three conditions is satisfied:

- v's parent and grandparent are not the same vertex. In this case, $l(v) > 2$ as shown in Figure 2(b)

- If v is a nonstar vertex, then its grandparent is also a nonstar vertex (Figure 2(b) and and line 9 of Algorithm 2)

- If v's parent is a nonstar, then v is also a nonstar vertex (Figure 2(c) and lines 11-12 of Algorithm 2)

The AS algorithm terminates when every tree becomes a star and the parent vector f is not updated in the latest itera-

tion. The algorithm terminates in $O(\log n)$ iterations. Hence, the algorithm runs in $O(\log n)$ time using $m + n$ processors in the PRAM model.

3.2. A simplified SV algorithm with fast convergence

While the AS algorithm is simpler than the original SV algorithm, it still needs to identify stars before every hooking operation. This star finding step can incur significant performance overhead, especially when the algorithm is run in distributed memory systems. Here, we discuss a variant of the SV algorithm that is simpler and faster than the AS algorithm in practice.

Notice that if we remove unconditional hooking from SV, the algorithm is still correct, but it no longer guarantees the $O(\log n)$ iterations in the worst case. Nevertheless, practical parallel algorithms often remove the unconditional hooking [32, 33] because it needs to keep track of unchanged trees (also known as stagnant trees), which is expensive, especially in distributed memory. In a recent paper, we developed an algorithm called FastSV [12] that has only one hooking phase followed by shortcutting in every iteration. In the hooking phase,

4

Algorithm 3 The skeleton of the FastSV algorithm. **Input:** a graph $G(V, E)$. **Output:** The parent vector f

---

1: **procedure** FASTSV($G(V, E)$)
2:     **for** every vertex $u \in V$ **do in parallel** ▷ Initialize
3:         $f[u], gf[u] \leftarrow v$
4:     **repeat**
5:         ▷ Step 1: Hooking phase
6:         **for** every edge $\{u, v\}$ in E **do in parallel**
7:             **if** $gf[u] > gf[v]$ **then**
8:                 $f[f[u]] \leftarrow gf[v]$
9:                 $f[u] \leftarrow gf[v]$
10:        ▷ Step 2: Shortcutting
11:        **for** every vertex u in V **do in parallel**
12:            **if** $f[u] > gf[u]$ **then**
13:                $f[u] \leftarrow gf[u]$
14:        ▷ Step 3: Calculate grandparent
15:        **for** every vertex u in V **do in parallel**
16:            $gf[u] \leftarrow f[f[u]]$
17:     **until** gf remains unchanged
18:     **return** f

FastSV explores any opportunities to hook a subtree onto another tree if a "suitable" edge between them can be found. Having this simplified constraint, FastSV not only avoids the cycle in the tree updating, but also makes it possible to employ more powerful hooking strategies while retaining the low computation cost. The shortcutting is the same as the AS algorithm, which shortens the distance of each vertex to the root.

Description of the algorithm. Algorithm 3 describes the complete FastSV algorithm[3]. The initialization is the same as the AS algorithm which creates n single-vertex trees, and the grandparent of each vertex is also initialized and stored in the vector gf. In each iteration, the algorithm performs three operations: (a) stochastic hooking, (b) aggressive hooking and (c) shortcutting. Due to the similarity of the first two operations, they are combined into a single step called the hooking phase (line 6-9). In this phase, we compare the grandparent of u and v for every edge $\{u, v\}$ in E. If the condition $gf[u] > gf[v]$ is satisfied, we hook both $f[u]$ and u onto $gf[v]$, v's grandparent in the previous iteration. Here, hooking $f[u]$ to $gf[v]$ corresponds to the *stochastic hooking* and hooking u to $gf[v]$ corresponds to the *aggressive hooking*. Then, in the shortcutting step (line 11-13), every vertex u modifies its pointer to $gf[u]$ if $gf[u] > f[u]$ is satisfied. In the end of each iteration, the grandparents of all vertices are recalculated and stored in the gf vector (lines 15-16), and the algorithm's termination is based on the stabilization of the gf vector instead of f, which is also correct and is proved to be a better termination condition in practice [12].

4. The AS and FastSV algorithms using linear algebra

In this section, we design the AS and FastSV algorithms using the GraphBLAS API [9]. We used GraphBLAS API to

---

[3]Algorithm 3 is presented in a prior work [12]. It is described here for completeness and readability

---

describe our algorithms because the API is more expressive, well-thought-of, and future proof. Below we give an informal description of GraphBLAS functions used in our algorithms. Formal descriptions can be found in the API document [34].

The function GrB_Vector_nvals retrieves the number of stored elements (tuples) in a vector. GrB_Vector_extractTuples extracts the indices and values associated with nonzero entries of a vector. In all other GraphBLAS functions we use, the first parameter is the output, the second parameter is the mask that determines to which elements of the output should the result of the computation be written into, and the third parameter determines the accumulation mode. We will refrain from using an accumulator and instead be performing an assignment in all cases; hence our third parameter is always GrB_NULL.

- The function GrB_mxv multiplies a matrix with a vector on a semiring, outputting another vector. The GraphBLAS API does not provide specialized function names for sparse vs. dense vectors and matrices, but instead allows the implementation to internally call different subroutines based on input sparsity. In our use case, matrices are always sparse whereas vectors start out dense and get sparse rapidly. GrB_mxv operates on a user defined semiring object GrB_Semiring. We refer to a semiring by listing its scalar operations, such as the (multiply, add) semiring. Our algorithm uses the (Select2nd, min) semiring with the GrB_mxv function where Select2nd returns its second input and min returns the minimum of its two inputs.

- The vector variant of GrB_extract extracts a sub-vector from a larger vector. The larger vector from which we are extracting elements from is the fourth parameter. The fifth parameter is a pointer to the set of indices to be extracted, which also determines the size of the output vector.

- The vector variant of the GrB_assign function that assigns the entries of a GraphBLAS vector (u) to another, potentially larger, vector w. The vector whose entries we are assigning to is the fourth parameter u. The fifth parameter is a pointer to the set of indices of the output w to be assigned.

- The vector variant of GrB_eWiseMult performs element-wise (general) multiplication on the intersection of elements of two vectors. The multiplication operation is provided as a GrB_Semiring object in the fourth parameter and the input vectors are passed in the fifth and sixth parameters.

We will refrain from making a general complexity analysis of these operations as the particular instantiations have different complexity bounds. Instead, we will analyze their complexities as they are used in our particular algorithms.

4.1. LACC: The AS algorithm using linear algebra

At first, we design the AS algorithm (Algorithm 1 and 2) using linear algebra. As mentioned in the Introduction, the linear-algebraic design of the AS algorithm is called LACC, following the naming used in the conference paper which this journal

Algorithm 4 Conditional hooking of stars. **Inputs:** an adjacency matrix $\mathbf{A}$, the parent vector f, the star-membership vector *star*. **Output:** Updated f. (NULL is denoted by $\emptyset$)

1: **procedure** CondHook($\mathbf{A}$, f, *star*)
2:    Sel2ndMin ← a (select2nd, min) semiring
3:    ▷ Step1: $f_n[i]$ stores the parent (with the minimum id) of a neighbor of vertex i. Next, $f_n[i]$ is replaced by $\min\{f_n[i], f[i]\}$
4:    GrB_mxv (f_n, *star*, $\emptyset$, Sel2ndMin, $\mathbf{A}$, f, $\emptyset$)
5:    GrB_eWiseMult (f_n, $\emptyset$, $\emptyset$, GrB_MIN_T, f_n, f, $\emptyset$);
6:    ▷ Step2: Parents of hooks (hooks are nonzero indices in f_n)
7:    GrB_eWiseMult (f_h, $\emptyset$, $\emptyset$, GrB_SECOND_T, f_n, f, $\emptyset$)
8:    ▷ Step3: Hook stars on neighboring trees ($f[f_h] = f_n$).
9:    GrB_Vector_nvals(&nhooks, f_n)
10:    GrB_Vector_extractTuples (index, value, nhooks, f_h)
11:    GrB_extract (f_n, $\emptyset$, $\emptyset$, f_n, index, nhooks, $\emptyset$) ▷ Dense
12:    GrB_assign (f, $\emptyset$, $\emptyset$, f_n, value, nhooks, $\emptyset$)

Algorithm 5 Unconditional star hooking. **Inputs:** an adjacency matrix $\mathbf{A}$, the parent vector f, the star-membership vector *star*. **Output:** Updated f. (NULL is denoted by $\emptyset$)

1: **procedure** UncondHook($\mathbf{A}$, f, *star*)
2:    Sel2ndMin ← a (select2nd, min) semiring
3:    ▷ Step1: For a star vertex, find a neighbor in a nonstar. $f_n[i]$ stores the parent (with the minimum id) of a neighbor of i
4:    GrB_extract(f_{ns}, *star*, $\emptyset$, f, GrB_ALL, 0, GrB_SCMP)
5:    GrB_mxv (f_n, *star*, $\emptyset$, Sel2ndMin, $\mathbf{A}$, f_{ns}, $\emptyset$)
6:    ▷ Step 2 and 3 are similar to Algorithm 4

Algorithm 6 The shortcut operation. **Input:** the parent vector f. **Output:** Updated f.

1: **procedure** Shortcut(f)
2:    ▷ find grandparents ($gf \leftarrow f[f]$)
3:    GrB_Vector_extractTuples(idx, value, &n, f) ▷ $n = |V|$
4:    GrB_extract (gf, $\emptyset$, $\emptyset$, f, value, n, $\emptyset$)
5:    GrB_assign (f, $\emptyset$, $\emptyset$, gf, GrB_ALL, 0, $\emptyset$) ▷ $f \leftarrow gf$

---

paper is based upon. Here, we describe various operations of LACC using the GraphBLAS API.

Conditional hooking. Algorithm 4 describes the conditional hooking operation designed using the GraphBLAS API. For each star vertex v, we identify a neighbor with the minimum parent id. This operation is performed using GrB_mxv in line 4 of Algorithm 4 where we multiply the adjacency matrix $\mathbf{A}$ by the parent vector f on the (Select2nd, min) semiring. We only keep star vertices by using the *star* vector as a mask. The output of GrB_mxv is stored in f_n, where $f_n[v]$ stores the minimum parent among all parents of $N(v)$ such that v belongs to a star. If the parent $f[v]$ of vertex v is smaller than $f_n[v]$, we store $f[v]$ in $f_n[v]$ in line 5. Nonzero indices in $f_n[v]$ are called hooks. Next, we identify parents f_h of hooks in line 7 by using the GrB_eWiseMult function that simply copies parents from f based on nonzero indices in f_n. Here, f_h contains roots because only a root can be a parent within a star. In the final step (lines 9-12), we hook f_h to f_n by using the GrB_assign function. In order to perform this hooking, we update parts of the parent vector f by using nonzero values from f_h as indices and nonzero values from f_n as values.

Unconditional hooking. Algorithm 5 describes unconditional hooking. As we will show in Lemma 2, unconditional hooking only allows a star to get hooked onto a nonstar. Hence, in line 4, we extract parents f_{ns} of nonstar vertices (GrB_SCMP denotes structural complement of the mask), which is then used with GrB_mxv in line 5. Here, we break ties using the (Select2nd, min) semiring, but we could have used other semiring addition operations instead of "min". The rest of Algorithm 5 is similar to Algorithm 4.

Shortcut. Algorithm 6 describes the shortcutting operation using two GraphBLAS primitives. At first, we use GrB_extract to obtain the grandparents gf of vertices. Next, we assign gf to the parent vector using GrB_assign.

Starcheck. Algorithm 7 identifies star vertices. At first, we initialize all vertices as stars (line 2). Next, we identify the subset of vertices h whose parents and grandparents are different (lines 4-5) using a Boolean mask vector *hbool*. Nonzero indices and values in h represent vertices and their grandparents,

respectively. In lines 7-10, we mark these vertices and their grandparents as nonstars. Finally, we mark a vertex nonstar if its parent is also a nonstar (lines 12-14).

4.2. Efficient use of sparsity in LACC

As shown in Algorithm 1, every iteration of the original AS algorithm explores all vertices in the graph. Hence, conditional and unconditional hooking explore all edges, and shortcut and starcheck explore all entries in parent and star vectors. If we directly translate the AS algorithm to linear algebra, all of our operations will use dense vectors, which is unnecessary if some vertices remain "inactive" in an iteration. A key contribution of this paper is to identify inactive vertices and sparsify vectors whenever possible so that we can eliminate unnecessary work performed by the algorithm. We now discuss ways to exploit sparsity in different steps of the algorithm.

Tracking converged components. A connected component is said to be *converged* if no new vertex is added to it in subsequent iterations. We can keep track of converged components using the following lemma.

Lemma 1. *Except in the first iteration, all remaining stars after unconditional hooking are converged components.*

Proof. Consider a star S after the unconditional hooking in the ith iteration where $i > 1$. In order to hook S in any subsequent iteration, there must be an edge $\{u, v\}$ such that $u \in S$ and $v \notin S$. Let v belong to a tree T at the beginning of the ith iteration. If T is a star, then the edge $\{u, v\}$ can be used to hook S onto T or T onto S depending on the labels of their roots. If T is a nonstar, the edge $\{u, v\}$ can be used to hook S onto T in unconditional hooking. In any of these cases, S will not be a star at the end of the ith iteration because hooking of a star on another tree always yields a nonstar. Hence, $\{u, v\}$ does not exist and S is a converged component. □

In our algorithm, we keep track of vertices in converged components and do not process these vertices in subsequent

Algorithm 7 Updating star memberships. **Inputs:** the parent vector f, the star vector *star*. **Output:** Updated *star* vector.

1: **procedure** STARCHECK(f, *star*)
2:   GrB_assign (*star*, $\emptyset$, $\emptyset$, true, GrB_ALL, 0, $\emptyset$) ▷ initialize
3:   ▷ vertices whose parents and grandparents are different. See Algorithm 6 for the code for computing grandparents *gf*
4:   GrB_eWiseMult(*hbool*, $\emptyset$, $\emptyset$, GrB_NE_T, f, gf, $\emptyset$)
5:   GrB_extract(*h*, *hbool*, $\emptyset$, gf, GrB_ALL, 0, $\emptyset$)
6:   ▷ mark these vertices and their grandparents as nonstars
7:   GrB_Vector_nvals(&nnz, *h*)
8:   GrB_Vector_extractTuples(index, value, nnz, *h*)
9:   GrB_assign (*star*, $\emptyset$, $\emptyset$, false, index, nnz, $\emptyset$)
10:   GrB_assign (*star*, $\emptyset$, $\emptyset$, false, value, nnz, $\emptyset$)
11:   ▷ $star[v] \leftarrow star[f[v]]$
12:   GrB_Vector_extractTuples(idx, value, &n, f) ▷ $n = |V|$
13:   GrB_extract ($star_f$, $\emptyset$, $\emptyset$, *star*, value, n, $\emptyset$)
14:   GrB_assign (*star*, $\emptyset$, GrB_MIN_T, $star_f$, GrB_All, 0, $\emptyset$)

Table 1: The scope of using sparse vectors at different steps of LACC (does not apply to the first iteration).

Operation	Operate on the subset of vertices in
Conditional hooking	Nonstars after unconditional hooking in the previous iteration
Unconditional hooking	Nonstars after conditional hooking
Shortcut	Nonstars after unconditional hooking
Starcheck	Nonstars after unconditional hooking

iterations. Hence Lemma 1 impacts all four steps of LACC. Since Lemma 1 does not apply to iteration 1, it has no influence in the first two iterations of LACC. Furthermore, a graph with a few components is not benefited significantly as most vertices will be active in almost every iteration.

Lemma 2. *Unconditional hooking does not hook a star on another star [5, Theorem 2(a)].*

Consequently, we can further sparsify unconditional hooking as was described in Algorithm 5. Even though unconditional hooking can hook a star onto another star in the first iteration, we prevent it by removing conditionally hooked vertices from consideration in unconditional hooking.

According to Lemma 1, only nonstar vertices after unconditional hooking will remain active in subsequent iterations. Hence, only these vertices are processed in the shortcut and starcheck operations. Table 1 summarizes the subset of vertices used in different steps of our algorithm.

4.3. The FastSV algorithm using linear algebra

Algorithm 8 describes the complete FastSV algorithm using the GraphBLAS API. We maintain both the parent vector f and the grandparent vector gf in each iteration. The hooking phase is similar to LACC's conditional hooking phase, which uses GrB_mxv to identify for each u the with the minimum $gf[v]$ among all the edges $\{u, v\}$ in E. The matrix multiplication $\mathbf{A} \cdot f$ is parameterized with the same (Select2nd, min) semiring, and the output is stored in f_n. Then, the stochastic hooking

Algorithm 8 The linear algebra FastSV algorithm. **Input:** an adjacency matrix $\mathbf{A}$ and the parent vector f. **Output:** Updated f. (NULL is denoted by $\emptyset$)

1: **procedure** FASTSV($\mathbf{A}$, f)
2:   GrB_Matrix_nrows (&n, $\mathbf{A}$)
3:   GrB_Vector_dup (& gf, f) ▷ initial grandparent
4:   GrB_Vector_dup (& *dup*, gf) ▷ duplication of *gf*
5:   GrB_Vector_extractTuples (index, value, &n, f)
6:   Sel2ndMin ← a (select2nd, Min) semiring
7:   **repeat**
8:     ▷ Step 1: Hooking phase
9:     GrB_mxv (f_n, $\emptyset$, GrB_MIN_T, sel2ndMin, $\mathbf{A}$, gf, $\emptyset$)
10:     GrB_assign (f, $\emptyset$, GrB_MIN_T, f_n, value, n, $\emptyset$)
11:     GrB_eWiseMult (f, $\emptyset$, $\emptyset$, GrB_MIN_T, f, f_n, $\emptyset$)
12:     ▷ Step 2: Shortcutting
13:     GrB_eWiseMult (f, $\emptyset$, $\emptyset$, GrB_MIN_T, f, gf, $\emptyset$)
14:     ▷ Step 3: Calculate grandparents
15:     GrB_Vector_extractTuples (index, value, &n, f)
16:     GrB_extract (gf, $\emptyset$, $\emptyset$, f, value, n, $\emptyset$)
17:     ▷ Step 4: Check termination
18:     GrB_eWiseMult (*diff*, $\emptyset$, $\emptyset$, GxB_ISNE_T, *dup*, gf, $\emptyset$)
19:     GrB_reduce (&sum, $\emptyset$, Add, *diff*, $\emptyset$)
20:     GrB_assign (*dup*, $\emptyset$, $\emptyset$, gf, GrB_ALL, 0, $\emptyset$))
21:   **until** sum = 0

$f[f[u]] \leftarrow f_n[u]$ is implemented by the GrB_assign function in line 10, which assigns the entries of f_n into the specified locations of vector f, and the indices *value* is extracted from the vector f in either line 5 before the first iteration or line 15 from the previous iteration. Next, the aggressive hooking $f[u] \leftarrow f_n[u]$ is implemented by an element-wise multiplication $f \leftarrow \min(f, f_n)$ in line 11, and the shortcutting operation $f \leftarrow \min(f, gf)$ is implemented in the same way in line 13. We recalculate the grandparent vector $gf[u] \leftarrow f[f[u]]$ in line 15-16, and in the end of each iteration, we calculate the number of modified entries in gf in line 18 - 19 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.

5. Parallel implementations of LACC and FastSV

In the previous section, we described two connected component algorithms LACC and FastSV using linear algebraic operations. Given libraries with parallel implementations of those linear algebraic operations, we can easily implement LACC and FastSV. In this section, we describe the implementations of LACC and FastSV using a shared-memory and a distributed-memory parallel library. We primarily focus on distributed-memory implementation and discuss detailed computation and communication complexity.

5.1. Parallel LACC and FastSV for shared memory platforms

Currently, the SuiteSparse:GraphBLAS library[4] provides a full implementation of the GraphBLAS C API with

---

the OpenMP parallelism. Therefore, algorithms described in Section 4 can be directly implemented in the SuiteSparse:GraphBLAS library. In the earlier conference version of this work, we implemented LACC and FastSV using SuiteSparse:GraphBLAS just to test the correctness of the presented algorithms with respect to the GraphBLAS API. In this paper, we show detailed shared-memory performance of LACC and FastSV's SuiteSparse:GraphBLAS implementation in Section 6.3. Our SuiteSparse:GraphBLAS implementation is committed to the LAGraph Library.

5.2. Parallel LACC and FastSV for distributed-memory platforms

We use the CombBLAS library [13] to implement LACC and FastSV for distributed-memory platforms. Since CombBLAS does not directly support the masking operations, we use element-wise filtering after performing an operation when masking is needed.

CombBLAS distributes its sparse matrices on a 2D $p_r \times p_c$ processor grid. Processor $P(i, j)$ stores the submatrix $\mathbf{A}_{ij}$ of dimensions $(m/p_r) \times (n/p_c)$ in its local memory. CombBLAS uses the doubly compressed sparse columns (DCSC) format to store its local submatrices for scalability, and uses a vector of {index, value} pairs for storing sparse vectors. Vectors are also distributed on the same 2D processor grid in a way that ensures that processor boundaries aligned for vector and matrix elements during multiplication.

5.3. Parallel complexity of linear-algebraic operations used in LACC and FastSV

We explain the parallel complexity of LACC and FastSV with respect to the GraphBLAS operations which LACC and FastSV depend upon. Here, we measure communication by the number of *words* moved (W) and the number of *messages* sent (S). The cost of communicating a length m message is $\alpha + \beta m$ where α is the latency and β is the inverse bandwidth, both defined relative to the cost of a single arithmetic operation. Hence, an algorithm that performs F arithmetic operations, sends S messages, and moves W words takes $T = F + \beta W + \alpha S$ time.

Our GrB_mxv internally maps to either a sparse-matrix dense-vector multiplication (SpMV) for the few early iterations when most vertices are active or to a sparse-matrix sparse-vector multiplication (SpMSpV) for subsequent iterations. Given the 2D distribution CombBLAS employs, both functions require two steps of communication: first within column processor groups, and second within row processor groups. The first stage of communication is a gather operation to collect the missing pieces of the vector elements needed for the local multiplication and the second one is a reduce-scatter operation to redistribute the result to the final vector. Both stages can be implemented to take advantage of vector sparsity. In fact, there is exciting research on the sparse reduction problem [35, 36]. We found that a simple allgather is the most performant for both SpMV and SpMSpV for the first stage in our case. For the reduce-scatter phase, SpMV uses a simple reduction within a loop (i.e. one for each processor in the row

group) whereas SpMSpV uses an irregular all-to-all operation followed by a local merge.

Assuming a square processor grid $p_c = p_r = \sqrt{p}$ and a load balanced matrix with m nonzeros, one SpMV iteration costs

$$T_{\text{SpMV}} = O\Big(\frac{m}{p} + \beta \frac{n}{\sqrt{p}}\big(\frac{\sqrt{p}-1}{\sqrt{p}} + \lg\sqrt{p}\big) + \alpha\big(\sqrt{p} + \lg\sqrt{p}\big)\Big)$$

using standard MPI implementations [37].

For the SpMSpV case, let the density of input vector be f and the unreduced output vector be g. While f is always less than or equal to 1, this is not necessarily the case for g because the number of nonzeros in the unreduced vector can be larger than n. If that is the case, we resort to a dense reduce-scatter operation similar to the one employed by SpMV. Hence, when we write g, we mean $\min(g, 1)$. Assuming that the nonzeros in vectors are i.i.d. distributed, the cost of SpMSpV is

$$T_{\text{SpMSpV}} = O\Big(\frac{mf}{p} + \beta \frac{nf + ng}{\sqrt{p}}\big(\frac{\sqrt{p}-1}{\sqrt{p}}\big) + \alpha\big(\sqrt{p} + \lg\sqrt{p}\big)\Big).$$

Vector variants of GrB_extract and GrB_assign are fairly general functions that can be exploited to perform very different computations. That being said, our use of them are sufficiently constrained that we can perform a reasonably tight analysis. The cost of GrB_extract primarily depends on the numbers of nonzeros in the output vector w. In contrast, the cost of GrB_assign primarily depends on the numbers of nonzeros in the input vector u. They both use the irregular all-to-all primitive for communication. With similar load balance assumptions as before, which can be theoretically achieved using a cyclic vector distribution, the cost of GrB_assign is:

$$T_{\text{ASSIGN}} = O\Big(\frac{nnz(u)}{p} + \beta \frac{nnz(u)}{p} + \alpha(p - 1)\Big).$$

The cost of GrB_extract is identical except that $nnz(u)$ is replaced by $nnz(w)$. Remember that $nnz(u), nnz(w) \le n$.

In practice, we achieve high performance in all-to-all operations by employing other optimizations to CombBLAS' block distributed vectors, described in Section 5.4, instead of using a cyclic distribution.

Despite our sparsity aware analysis of individual primitives, we could not prove bounds on aggregate sparsity across all iterations. We can, however, still provide an overall complexity assuming the worst case $nnz(u), nnz(w) = n$ and $f, g = 1$. Given that there are a constant number of calls to GraphBLAS primitives in each iteration and the algorithm converges in $\lg(n)$ iterations, LACC's sparsity-agnostic parallel cost is:

$$T_{\text{LACC}} = O\Big(\frac{m\lg(n)}{p} + \beta \frac{n\lg(n)\lg(\sqrt{p})}{\sqrt{p}} + \alpha(p - 1)\lg(n)\Big).$$

Each iteration of FastSV (Algorithm 8) has the same asymptotic complexity as LACC. In practice, an iteration of FastSV can run faster because of its simplicity. For example, FastSV
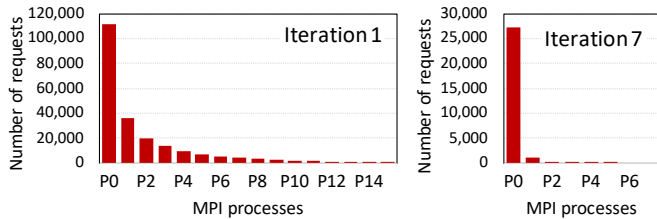
Figure 3: Number of requests received by every process when accessing grand-parents. We show iteration 1 and 7 when running LACC with 16 processes. Only even numbered processes are labeled on the x-axis. Lower ranked processes receive more requests than higher ranked process in all iterations. Later iterations are more imbalanced than earlier iterations.

needs one SpMV operation, where as LACC needs two Sp-MVs. However, FastSV can take $O(n)$ iterations in the worst case. Hence, FastSV's parallel cost is:

$$T_{\text{FastSV}} = O(\frac{mn}{p} + \beta \frac{n^2 \lg(\sqrt{p})}{\sqrt{p}} + \alpha(p-1)n).$$

Because of our aggressive hooking strategies, FastSV needed approximately the same number of iterations as needed by LACC in all problems we experimented in this paper. Consequently, FastSV runs faster than LACC in practice despite its higher asymptotic complexity.

5.4. Load balancing and communication efficiency

In CombBLAS, we randomly permute the rows and columns of the adjacency matrix, resulting in load-balanced distribution of the matrix and associated dense vectors. Hence, GrB_mxv is a load-balanced operation both in terms of computation and communication. However, GrB_assign and GrB_extract can be highly imbalanced when a vector is indexed by parents. For example, Figure 3 shows the number of requests received by every process when extracting grand-parents using GrB_extract in two different iterations of LACC. This imbalance is caused primarily by the conditional hooking (via the (select2nd, min) semiring), where parents have smaller ids than their children. Since CombBLAS employs a block distribution of vectors, lower-ranked processes receive more data than higher-ranked processes in all-to-all communication, which may result in poor performance. Many of these received requests need to access the same data at the recipient process, incurring redundant data access and communication.

To alleviate this problem with highly skewed all-to-all communication, we broadcast entries from few low-ranked processes and then remove those processes from all-to-all collective operations. If a processor receives h times more requests than the total number of elements it has, it broadcasts its local part of a vector rather than participating in an all-to-all collective call. Here, h is a system-dependent tunable parameter. If more than one process broadcasts data in an iteration, we use nonblocking MPI_Ibcast so that they can proceed independently.

We also used two more optimizations to make all-to-all communication more efficient. First, when data is highly imbalanced as shown in Figure 3, we noticed that all-to-all operations

Table 2: Overview of evaluation platforms. [2]Memory bandwidth is measured using the STREAM copy benchmark per node.

	Cori (Intel KNL)	Edison (Intel Ivy Bridge)
Core		
Clock (GHz)	1.4	2.4
L1 Cache (KB)	32	32
L2 Cache (KB)	1024	256
DP GFlop/s/core	44	19.2
Node Arch.		
Sockets/node	1	2
Cores per socket	68	12
STREAM BW[2]	102 GB/s	104 GB/s
Memory per node	96 GB	64 GB
Prog. Environment		
Compiler	gcc 7.3.0	gcc 7.3.0
Optimization	-O2	-O2

in Cray's MPI library at NERSC are not scaling beyond 1024 MPI ranks. A possible reason could be the use of the pairwise-exchange algorithm that has $\alpha(p-1)$ latency cost [37]. Hence, we replace all MPI_Alltoallv calls with a hypercube-based implementation by Sundar et al. [38], which has $\alpha \log(p)$ latency cost. Second, in iteration 7 of Figure 3, processes 7-15 have no data to communicate. In that case (after P0 broadcasts its data), we use a sparse variant of all-to-all implementation [38], where only P1-P5 exchange data. All of these optimizations made our implementations of GrB_assign and GrB_extract highly scalable as seen in Figure 9.

6. Results

6.1. Evaluation platforms

Shared-memory platform. We evaluate the shared-memory performance of LACC, FastSV, and other CC algorithms on an Amazon AWS r5.8xlarge instance with Intel Xeon Platinum 8000 CPU (3.1GHz, 249G memory). We use up to 16 threads in our shared-memory experiments.

Distributed-memory platform. Distributed-memory experiments were conducted on NERSC Edison and Cori KNL supercomputers as described in Table 2. Even though the Edison supercomputer is not longer in service, we keep results from Edison to keep this paper consistent with the preceding conference paper [11]. Our distributed-memory implementations uses OpenMP for multithreaded execution within an MPI process. In our experiments, we only used square process grids because rectangular grids are not supported in Comb-BLAS [13]. When p cores are allocated for an experiment, we create a $\sqrt{p/t} \times \sqrt{p/t}$ process grid where t is the number of threads per process. All of our experiments used 16 and 6 threads per MPI process on Cori and Edison, respectively. In our hybrid OpenMP-MPI implementation, all MPI processes perform local computation followed by synchronized communication rounds. Only one thread in every process makes MPI calls in the communication rounds.

9

Table 3: Test problems used to evaluate parallel connected component algorithms. We report directed edges because the symmetric adjacency matrices are stored in LACC. We cite the sources from where we obtained the graphs.

Graph	Vertices	Directed edges	Components	Description
archaea	1.64M	204.79M	59,794	archaea protein-similarity network [8]
queen_4147	4.15M	329.50M	1	3D structural problem [39]
eukarya	3.23M	359.74M	164,156	eukarya protein-similarity network [8]
uk-2002	18.48M	529.44M	1,990	2002 web crawl of .uk domain [39]
M3	531M	1.047B	7.6M	Soil metagenomic data [10]
twitter7	41.65M	2.405B	1	twitter follower network [39]
sk-2005	50.64M	3.639B	45	2005 web crawl of .sk domain [39]
MOLIERE_2016	30.22M	6.677B	4,457	automatic biomedical hypothesis generation system [39]
Metaclust50	282.2M	42.79B	15.98M	similarities of proteins in Metaclust50 [8]
iso_m100	68.48M	67.16B	1.35M	similarities of proteins in IMG isolate genomes [8]

6.2. Test problems

Table 3 describes ten test problems used in our experiments. These graphs contain a wide range of connected components and cover a broad spectrum of applications. The protein-similarity networks are generated from the IMG database at the Joint Genome Institute and are publicly available as part of the HipMCL software [8].

6.3. Shared-memory performance

The shared-memory implementations of LACC and FastSV can be directly obtained from SuiteSparse:GraphBLAS, a multi-threaded implementation of the GraphBLAS standard. We compare them with a popular shared-memory graph processing framework Ligra [40], which implements the propagation algorithm using parallel breadth-first search (BFS). We chose not to compare with other hand-tuned connected component codes because our objective is to show that linear-algebraic CC algorithms are competitive to popular graph processing frameworks such as Ligra. As mentioned before, our shared-memory experiments are conducted on an Intel Xeon Platinum 8000 CPU with up to 16 threads.

Figure 4 shows the shared-memory performance of LACC and FastSV implemented using SuiteSparse:GraphBLAS and Ligra's CC implementation. Here, we only experimented with the first eight graphs from Table 3 because the last two graphs did not fit on the memory of our single node server. LACC and FastSV are originally designed for reducing the communication cost on distributed-memory, but their shared-memory implementations are also competitive with Ligra. Especially, the simplicity of FastSV makes it efficient on multithreaded environments. For six out of eight graphs in Figure 4 , FastSV's runtime is within 17% of Ligra's CC implementation. This results indicates that SuiteSparse:GraphBLAS has scalable implementations of linear algebraic operations that we used in our algorithms. However, for some problems like the very sparse M3 graph, FastSV can be much slower than Ligra. M3 is an extremely sparse graph and it is possible that SuiteSparse:GraphBLAS is not well optimized for these type of very sparse graphs (especially if we use SpMV in every iteration of the SV algorithm). By contrast, Ligra's implementation carefully handles the BFS's frontier so that it does not span the
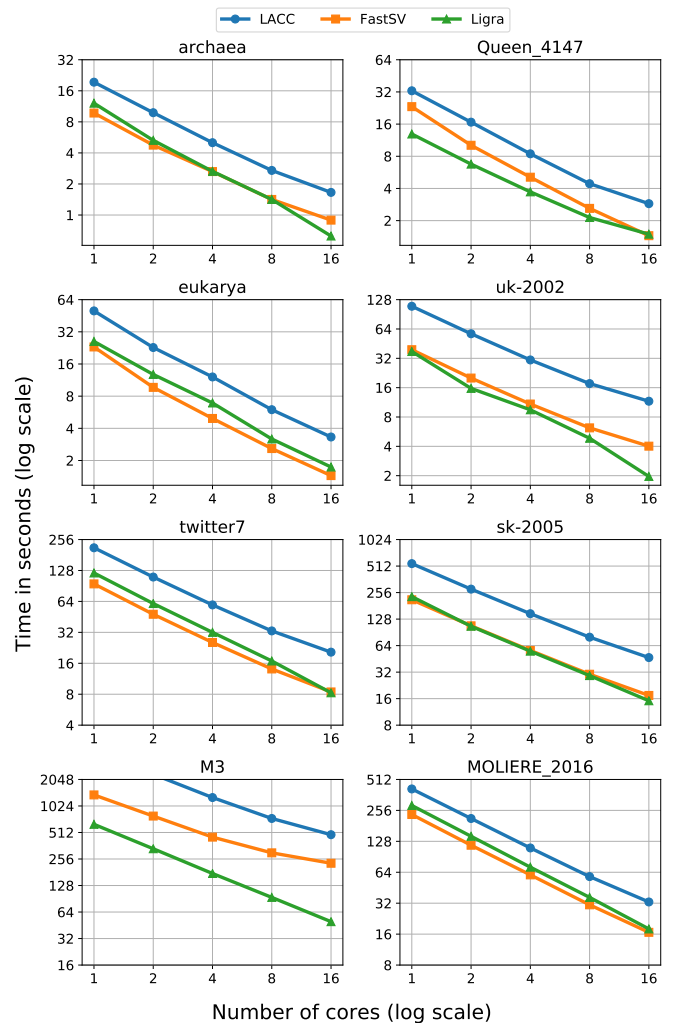

Figure 4: Scalability of LACC, FastSV and Ligra with regarding to the number of threads on eight small datasets using 16 threads.

whole vertex set in every iteration. Another reason for Ligra's superior performance on very sparse graph might be due to its direction optimization. The GrB_mxv function within SuiteSparse:GraphBLAS does not automatically implement this fea-

10

Table 4: Comparison of the runtime of CombBLAS and LAGraph (developed on top of SuiteSparse:GraphBLAS) using 16 threads (in seconds).

Graph	LACC		FastSV	
	LAGraph	CombBLAS	LAGraph	CombBLAS
archaea	1.66	2.18	0.89	1.30
Queen_4147	2.88	5.43	1.45	2.84
eukarya	3.32	4.93	1.44	2.47
uk-2002	11.61	21.41	4.02	6.17
twitter7	20.53	49.75	8.42	14.91
sk-2005	46.83	120.50	17.40	24.87
M3	483.30	449.14	229.13	339.53

ture that would switch search direction depending on the sparsity of input and output vectors. This feature, however, is implemented in GraphBLAST [41] and we expect it to be available soon in other GraphBLAS-inspired libraries.

LACC is approximately 2× to 3× slower than FastSV and Ligra. This is primarily due to LACC's use of two SpMV operations (that is the GrB_mxv function) needed in the conditional and unconditional hooking (as opposed to one SpMV needed by FastSV). Note that for most graphs, SpMV is the most expensive operation, since it needs to traverse all edges of the graph. Therefore, each iteration of LACC is about 2× slower than a FastSV iteration that uses SpMV only once in its stochastic hooking.

Figure 4 demonstrates that all three shared-memory implementations scale almost linearly up to 16 threads. On average, LACC, FastSV and Ligra achieves 11.4×, 12.3×, 15.09× speedup on 16 threads with respect to their sequential runtimes. Overall, we highlight the fact that connected component algorithms implemented using linear-algebraic operations can attain high performance on shared memory platforms. This performance is comparable to the state-of-the-art shared-memory platforms. However, true benefits of linear-algebraic operations can be realized in the distributed-memory systems, as we will demonstrate in the next few sections.

6.4. Single node performance of LACC and FastSV when implemented using SuiteSparse:GraphBLAS and CombBLAS

Even though CombBLAS is designed for distributed-memory platforms, it can also attain good performance on a single node. Since we implemented LACC and FastSV using both SuiteSparse:GraphBLAS and CombBLAS, we compare their performance on a single node with multicore processors. For this experiment, we used the same configuration used in the experiments in Section 6.3.For CombBLAS, we always use a 4 × 4 process grid with each process having one thread during the computation. Table 4 presents the runtime of LACC and FastSV implemented in CombBLAS and SuiteSparse:GraphBLAS using 16 threads. We only show results for graphs that fit in the memory of a node. On all the seven graphs, we observe that CombBLAS is on average 1.78× and 1.62× slower than GraphBLAS for LACC and FastSV, respectively. This slowdown of CombBLAS is possibly due its distributed-memory overheads such as MPI communication and

buffer copies. Therefore, we observe that CombBLAS itself is an efficient linear algebraic library on shared-memory, and the extra overhead is worth paying to obtain the extraordinary scalability and high-performance on large supercomputers.

6.5. Performance of LACC and FastSV in distributed-memory

As mentioned before, we implemented distributed memory LACC and FastSV using the CombBLAS library. Here, we compare the performance of distributed LACC and FastSV with ParConnect [10], the state-of-the art algorithm prior to our work. Similar to our algorithms, ParConnect also depends on CombBLAS; hence, both of them require a square process grid. Since ParConnect does not use multithreading, we place one MPI process per core in ParConnect experiments.

Figure 5 shows the performance of LACC and ParConnect with the smaller eight test problems on Edison (we did not show FastSV on Edison because Edison went out of service when FastSV was developed). Both LACC and ParConnect scale well up to 6144 cores (256 nodes), but LACC runs faster than ParConnect on all concurrencies. On 256 nodes, LACC is 5.1× faster than ParConnect on average (min 1.2×, max 12.6×). LACC is expected to perform better when a graph has many connected components because, for these graphs, we have better opportunities to employ sparse operations. Consequently, LACC performs the best for archaea and eukarya. For M3, LACC performs comparably to ParConnect, which will be explained in detail in Section 6.7.

The relative performance of LACC and ParConnect on Cori KNL is similar to Edison as can be seen in Figure 6. We additionally show the strong scaling of FastSV, which is generally faster than both LACC and ParConnect. As with Edison, LACC outperforms ParConnect on all core counts on Cori for all graphs except M3, for which the performance is comparable. FastSV is faster than LACC because of the former using only one SpMV operation per iteration. However, we also obverse that LACC scales slightly better than FastSV. For example, in Figure 6, the gap between LACC and FastSV decreases as we increase cores for eukarya and twitter7. Better scalability of LACC is due to its use of sparse operations which in turn reduce the communication on high concurrency. Generally, all three connected component algorithms ran faster on Edison than Cori given the same number of nodes. This behavior is common for sparse graph manipulations where few faster cores (e.g., Intel Ivy Bridge on Edison) are more beneficial than more slower cores (e.g., KNL on Cori) [42].

6.6. Performance of CC algorithms for bigger graphs

In the previous section, we presented results for smaller graphs, each of which can be stored in less than 150GB memory (ignoring MPI overheads). It is often possible to store these graphs on a shared-memory server and compute connect components using an efficient shared-memory algorithm [33]. However, the last two graphs in Table 3 need more than 1TB memory, requiring distributed-memory processing. We show the performance of LACC, FastSV and ParConenct for these big graphs in Figure 7. We observed that LACC and FastSV
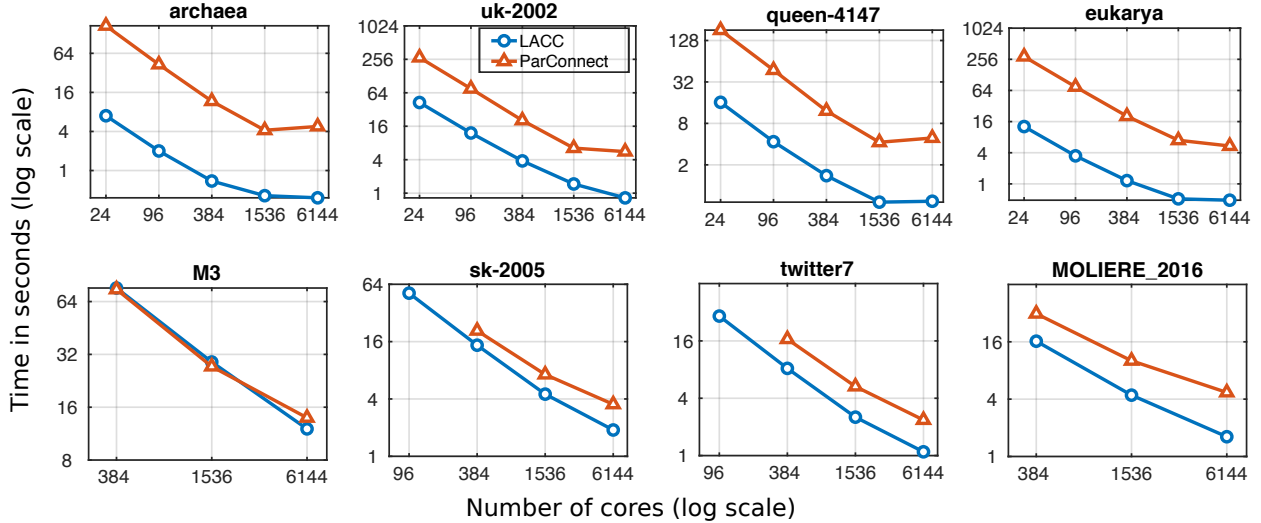
Figure 5: Strong scaling of LACC and ParConnect on Edison on (up to 6144 cores on 256 nodes). LACC uses 4 MPI processes per node and ParConnect uses 24 MPI processes per node.
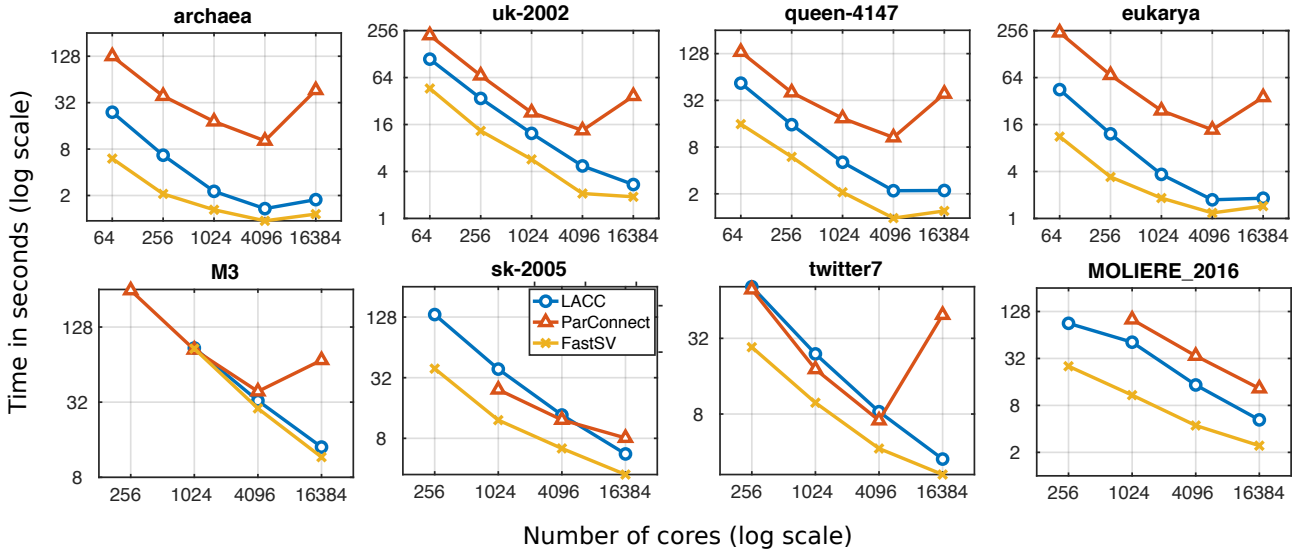

Figure 6: Strong scaling of LACC and ParConnect on Cori KNL (up to 16,384 cores on 256 nodes). LACC uses 4 MPI processes per node and ParConnect uses 64 MPI processes per node. Graphs with large numbers of connected components are shown.

continue scaling to 4096 nodes (262,144 cores) on Cori and computes connected components in these large networks in less than 16 seconds. By contrast, ParConnect does not scale beyond 16,384 cores for these two graphs. One reason of ParConnect not performing well on high core counts could be its reliance on flat MPI. On 262,144 cores, ParConnect creates 262,144 MPI processes and needs more than two hours to find connected components. Once again, we observe that LACC scales slightly better than FastSV on high concurrency. The remarkable ability of LACC and FastSV to process graphs with tens of billions of edges on hundreds of thousands cores makes it well suited for large-scale applications such as high-performance Markov clustering [8]. We will discuss this in more detail in Section 6.8.

6.7. Understanding the performance of LACC

We now explore different features of LACC and describe why it achieves good performance for most of the test graphs[5].

(a) Number of active vertices (vector sparsity). When fewer vertices remain active in an iteration, LACC performs less work and communicate less data. Hence, identifying and eliminating converged forests boost the performance of LACC significantly. In our GraphBLAS-style implementation, this translates into sparser vectors, which impacts the performance of GrB_mxv, GrB_assign, and GrB_extract. However, LACC can take advantage of the vector sparsity only if the input graph

---

[5]Here, we only discuss detail performance of LACC since this paper is an extension of the LACC paper [11]. Detailed performance of FastSV can be found in [12].
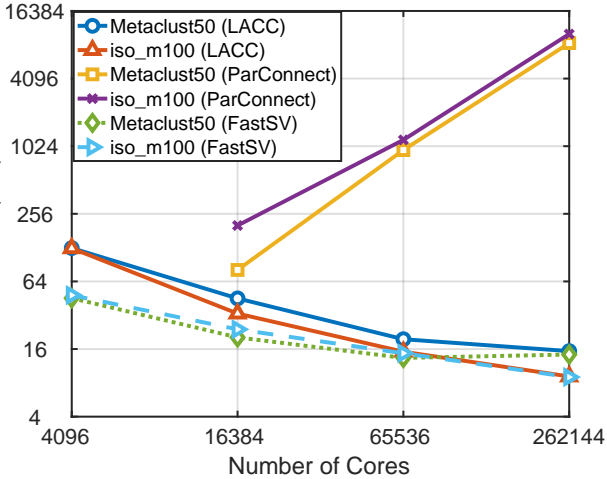
Figure 7: Performance of LACC and ParConnect with two large protein-similarity networks on Cori KNL (up to 262,144 cores on 4096 nodes). LACC and ParConnect use 4 and 64 MPI processes per node, respectively. While LACC scales to 262,144 cores, ParConnect stopped scaling at this extreme scale. ParConnect ran out of memory on 64 nodes.
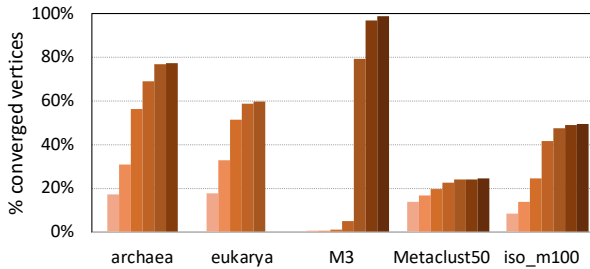

Figure 8: Percentage of vertices in converged connected components at different iterations of LACC. For every graph, iterations are shown incrementally from left to right.

has a large number of connected components. To demonstrate this, Figure 8 plots the percentage of vertices in converged components for five graphs with the highest number of components. We observe that a significant fraction of vertices becomes inactive after few iterations. Hence, LACC is expected to perform better (both sequential and parallel cases) for these graphs. Figure 5 and Figure 7 confirm this expectation except for M3. For M3, LACC needs 11 iterations, eight of which have less than 5% converged vertices. Hence, LACC can not take advantage of vector sparsity in most of the iterations, which can partially explains the observed performance of LACC on the M3 graph. For a connected graph, LACC can not take advantage of vector sparsity at all.

(b) Sparsity of the input graph. The sparsity of the input graph also impacts the performance and scalability of LACC. When a dense vector is used, the computational cost of GrB_mxv is $O(m)$, while all other operations take $O(n)$ time. Since GrB_assign and GrB_extract may communicate $O(n)$ data, the computation to communication ratio of LACC is $O(m/n)$. For very sparse graphs similar to M3, communication starts to dominate the overall runtime, affecting the performance of our GraphBLAS kernels. High graph sparsity and

lack of vector sparsity in most iterations play roles in the performance of LACC on the M3 graph. By contrast, queen_4147 (with average degree of 82) is denser than M3. Consequently, LACC performs much better on queen_4147 despite it having a single component.

(c) Scalability of different parts of LACC. Figure 9 shows the performance breakdown of LACC for three representative graphs, where all four parts of LACC scale well on Edison and Cori. For smaller graphs like eukarya, LACC stops scaling after 64 nodes (1,536 cores) because of the relatively high communication overhead on high concurrency. We also observe that conditional hooking is usually more expensive than unconditional hooking because the latter can utilize additional vector sparsity as shown in Lemma 2. Finally, our adaptive communication scheme discussed in Section 5.4 makes the shortcut and starcheck operations highly scalable.

6.8. Performance of LACC when used in Markov clustering

As discussed in the introduction, finding connected components is an important step in the popular Markov clustering algorithm. LACC is already incorporated with HipMCL where LACC can be 3288× faster (on 1024 nodes of Edison) than the shared-memory parallel connected component algorithm used in the original MCL software [1]. HipMCL is an ongoing project with an aim to scale to upcoming exascale systems and cluster more than 50B proteins in the IMG database (https://img.jgi.doe.gov/). A massively-parallel LACC boosts HipMCL's performance and helps us cluster massive biological networks with billions of vertices and trillions of edges.

7. Opportunities and limitations of linear-algebraic CC algorithms

The primary advantage of linear-algebraic CC algorithms like LACC and FastSV is their reliance on off-the-shelf functions from high-performance libraries like CombBLAS and SuiteSparse:GraphBLAS. As a result, we were able to implement LACC and FastSV rapidly after designing them in the language of linear algebra. This paper demonstrated the effectiveness of our approach where LACC and FastSV scaled to hundreds of thousands of cores with the support of the highly-optimized CombBLAS library.

Even though LACC and FastSV achieve remarkable scalability, it is certainly possible to develop highly optimized hand-tuned codes that run faster than LACC and FastSV. Notably, customized CC algorithms perform well on shared-memory platforms using improved data locality and sampling techniques. For example, a recent algorithm called Afforest [33] used sampling to develop a shared-memory parallel CC algorithm. The implementation of Afforest in the GAP benchmark [43] runs up to 5× faster than the shared-memory FastSV code implemented on top of SuiteSparse:GraphBLAS. Similarly, the iSpan algorithm [44] uses various asynchronous schemes and direction-optimized BFS to find connected components quickly and can run faster than our algorithms on multicore processors. Recently, Dhulipala et al. [45] developed a
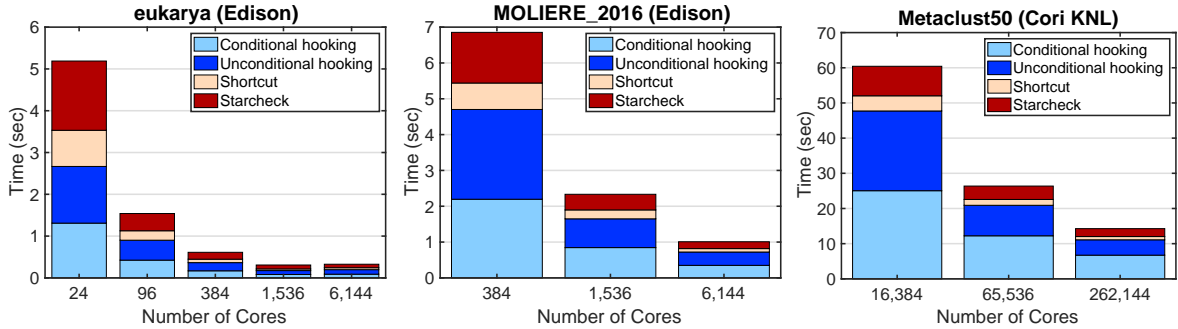
13

Figure 9: Performance breakdown of LACC for three representative graphs.

class of shared-memory parallel graph algorithms that achieve state-of-the-art performance on multicore servers with large memory. Their CC algorithm is based on a prior work [25] where they compared with Ligra's direction-optimized CC implementation. Since FastSV can be up to 2× slower than Ligra's CC implementation (see Fig. 4), it can be at most 4× slower than the CC algorithm reported by Dhulipala et al. [45]. Overall, it is expected that customized shared-memory codes [33, 44, 45] would perform better than our algorithms based on general-purpose linear algebra libraries.

LACC and FastSV achieve the state-of-the-art performance in distributed memory. This performance is achieved by using highly-optimized distributed primitives available in Comb-BLAS. However, on a single shared-memory platform, LACC and FastSV implemented on CombBLAS are up to 2× slower than their implementations on SuiteSparse:GraphBLAS (see Table 4). This slowdown is observed due to the overheads in CombBLAS which is optimized for distributed-memory platforms. Highly-scalable algorithms like LACC and FastSV are especially valuable when the graph does not fit in the memory of a single node or when the graph is already distributed as part of another application such as HipMCL [8].

It is generally hard to relate the performance of a distributed algorithm to the input graph characteristics. In our experiments, we observed various performance gains with different graphs. Generally, LACC and FastSV perform better for graphs with a large number of connected components. For example, on a low-diameter graph with a single connected component, a BFS-based algorithm is expected to perform better than LACC and FastSV. In a graph with many connected components, LACC avoids already-found connected components in subsequent iterations by using sparse vectors. This approach is only useful when a sizable fraction of connected components is found in early iterations (see Fig. 8). However, it is often not possible to predict the number of components or the rate of component discoveries in advance from some summary statistics of the graph. Hence, we did not find a clear correlation between the performance of LACC and FastSV and the input graphs.

8. Conclusions

We present two distributed-memory connected component algorithms LACC and FastSV that are implemented using

sparse linear algebra and are based on the Shiloach-Vishkin algorithm. Both algorithms achieve unprecedented scalability to 4K nodes (262K cores) of a Cray XC40 supercomputer and outperforms previous state-of-the-art by a significant margin. There are three key reasons for the observed performance: (1) our algorithms rely on linear algebraic kernels that are highly optimized for distributed memory graph analysis, (2) whenever possible, our algorithms employ sparse vectors in the hooking, shortcutting and star finding steps, eliminating redundant computation and communication, and (3) our algorithms detect imbalanced collective communication patterns inherent in the CC algorithm and remove them with customized all-to-all operations.

Extreme scalability achieved by linear-algebraic connected component algorithms such as LACC and FastSV can boost the performance of many large-scale applications. Metagenome assembly and protein clustering are two such applications that compute connected components in graphs with hundreds of billions or even trillions of edges on hundreds of thousands of cores.

The use of sparsity (Lemma 1 and 2 in Section 4) is a property of the Awerbuch-Shiloach algorithm and can be applied to any Awerbuch-Shiloach implementation. The customized communications are related to the way CombBLAS distributes sparse matrices and vectors. As future work, we plan to improve our vector operations so that they can avoid communication hot spots and work better on very sparse graphs similar to the M3 graph in Table 3. Using cyclic distributions of vectors, instead of the current block distribution used in CombBLAS, is one possible approach to distribute load more evenly and make LACC even more scalable.

In terms of reducing the number of actual operations performed, we plan to utilize the automatic direction optimization feature of GraphBLAST [41] in order to be competitive with Ligra on very sparse graphs as well. Porting our code to Graph-BLAST will also enable LACC to seamlessly run on GPUs.

9. Acknowledgments

References

[1] S. M. Van Dongen, "Graph clustering by flow simulation," Ph.D. dissertation, 2000.

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

[3] H. K. Thornquist, E. R. Keiter, R. J. Hoekstra, D. M. Day, and E. G. Boman, "A parallel preconditioning strategy for efficient transistor-level circuit simulation," in *Intl. Conf. on Computer-Aided Design*. New York, NY, USA: ACM, 2009, pp. 410–417.

[4] Y. Shiloach and U. Vishkin, "An O(logn) parallel connectivity algorithm," *Journal of Algorithms*, vol. 3, no. 1, pp. 57–67, 1982.

[5] B. Awerbuch and Y. Shiloach, "New connectivity and MSF algorithms for shuffle-exchange network and PRAM," *IEEE Transactions on Computers*, vol. 10, no. C-36, pp. 1258–1263, 1987.

[6] E. Georganas, R. Egan, S. Hofmeyr, E. Goltsman, B. Arndt, A. Tritt, A. Buluc, L. Oliker, and K. Yelick, "Extreme scale de novo metagenome assembly," in *Proceedings of SC*, 2018.

[7] S. Nurk, D. Meleshko, A. Korobeynikov, and P. A. Pevzner, "metaSPAdes: a new versatile metagenomic assembler," *Genome research*, pp. gr–213 959, 2017.

[8] A. Azad, G. A. Pavlopoulos, C. A. Ouzounis, N. C. Kyrpides, and A. Buluç, "HipMCL: A high-performance parallel implementation of the Markov clustering algorithm for large-scale networks," *Nucleic Acids Research*, vol. 46, no. 6, pp. e33–e33, 2018.

[9] A. Buluç, T. Mattson, S. McMillan, J. Moreira, and C. Yang, "Design of the GraphBLAS API for C," in *IPDPS Workshops*, 2017, pp. 643–652.

[10] 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*, vol. 28, no. 9, pp. 2428–2439, 2017.

[11] A. Azad and A. Buluç, "LACC: A linear-algebraic algorithm for finding connected components in distributed memory," in *Proceedings of the IPDPS*. IEEE, 2019, pp. 2–12.

[12] Y. Zhang, A. Azad, and Z. Hu, "FastSV: A distributed-memory connected component algorithm with fast convergence," in *Proceedings of the 2020 SIAM Conference on Parallel Processing for Scientific Computing*. SIAM, 2020, pp. 46–57.

[13] A. Buluç and J. R. Gilbert, "The Combinatorial BLAS: Design, implementation, and applications," *The International Journal of High Performance Computing Applications*, vol. 25, no. 4, pp. 496–509, 2011.

[14] T. A. Davis, "Algorithm 1000: SuiteSparse: GraphBLAS: Graph algorithms in the language of sparse linear algebra," *ACM Transactions on Mathematical Software (TOMS)*, vol. 45, no. 4, pp. 1–25, 2019.

[15] T. Mattson, T. A. Davis, M. Kumar, A. Buluç, S. McMillan, J. Moreira, and C. Yang, "LAGraph: A community effort to collect graph algorithms built on top of the GraphBLAS," in *IPDPS Workshops*, 2019, pp. 276–284.

[16] A. George, J. R. Gilbert, and J. W. Liu, *Graph theory and sparse matrix computation*. Springer Science & Business Media, 2012, vol. 56.

[17] J. Kepner and J. Gilbert, *Graph algorithms in the language of linear algebra*. SIAM, 2011.

[18] 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 *Computing Frontiers (CF)*, 2016, pp. 72–81.

[19] 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

[20] graph analytics made productive," *Proceedings of the VLDB Endowment*, vol. 8, no. 11, pp. 1214–1225, 2015.

[20] A. Buluç, T. Mattson, S. McMillan, J. Moreira, and C. Yang, "Design of the GraphBLAS API for C," in *IPDPS Workshops*, 2017.

[21] J. Reif, "Optimal parallel algorithms for integer sorting and graph connectivity." Harvard Univ., Cambridge, MA (USA)., Tech. Rep., 1985.

[22] H. Gazit, "An optimal randomized parallel algorithm for finding connected components in a graph," *SIAM Journal on Computing*, vol. 20, no. 6, pp. 1046–1067, 1991.

[23] S. Halperin and U. Zwick, "An optimal randomised logarithmic time connectivity algorithm for the EREW PRAM," *Journal of Computer and System Sciences*, vol. 53, no. 3, pp. 395–416, 1996.

[24] S. Pettie and V. Ramachandran, "A randomized time-work optimal parallel algorithm for finding a minimum spanning forest," *SIAM Journal on Computing*, vol. 31, no. 6, pp. 1879–1895, 2002.

[25] J. Shun, L. Dhulipala, and G. Blelloch, "A simple and practical linear-work parallel algorithm for connectivity," in *Proceedings of SPAA*, 2014, pp. 143–153.

[26] G. M. Slota, S. Rajamanickam, and K. Madduri, "A case study of complex graph analysis in distributed memory: Implementation and optimization," in *Proceedings of IPDPS*, 2016, pp. 293–302.

[27] G. Cong, G. Almasi, and V. Saraswat, "Fast PGAS connected components algorithms," in *Third Conference on PGAS Programing Models*. ACM, 2009, p. 13.

[28] V. B. Shah, "An interactive system for combinatorial scientific computing with an emphasis on programmer productivity," Ph.D. dissertation, University of California, Santa Barbara, 2007.

[29] R. Kiveris, S. Lattanzi, V. Mirrokni, V. Rastogi, and S. Vassilvitskii, "Connected components in MapReduce and beyond," in *Proceedings of the ACM Symposium on Cloud Computing*. ACM, 2014, pp. 1–13.

[30] A. Andoni, Z. Song, C. Stein, Z. Wang, and P. Zhong, "Parallel graph connectivity in log diameter rounds," in *2018 IEEE 59th Annual Symposium on Foundations of Computer Science (FOCS)*. IEEE, 2018, pp. 674–685.

[31] G. Pandurangan, P. Robinson, and M. Scquizzato, "Fast distributed algorithms for connectivity and MST in large graphs," *ACM Transactions on Parallel Computing (TOPC)*, vol. 5, no. 1, p. 4, 2018.

[32] 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*. Springer, 2017, pp. 301–320.

[33] M. Sutton, T. Ben-Nun, and A. Barak, "Optimizing parallel graph connectivity computation via subgraph sampling," in *Proceedings of IPDPS*, 2018, pp. 12–21.

[34] A. Buluç, T. Mattson, S. McMillan, J. Moreira, and C. Yang, "The GraphBLAS C API specification," version 1.2.0. Technical report, The GraphBLAS Signatures Subgroup, Tech. Rep., 2018.

[35] H. Zhao and J. Canny, "Kylix: A sparse allreduce for commodity clusters," in *Proceedings of ICPP*. IEEE, 2014, pp. 273–282.

[36] J. L. Träff, "Transparent neutral element elimination in MPI reduction operations," in *European MPI Users' Group Meeting*. Springer, 2010, pp. 275–284.

[37] R. Thakur, R. Rabenseifner, and W. Gropp, "Optimization of collective communication operations in MPICH," *Intl. Jour. of High Perf. Comp. App.*, vol. 19, no. 1, pp. 49–66, 2005.

[38] H. Sundar, D. Malhotra, and G. Biros, "Hyksort: a new variant of hypercube quicksort on distributed memory architectures," in *Proceedings ICS*. ACM, 2013, pp. 293–302.

[39] T. A. Davis and Y. Hu, "The University of Florida sparse matrix collection," *ACM Transactions on Mathematical Software (TOMS)*, vol. 38, no. 1, p. 1, 2011.

[40] J. Shun and G. E. Blelloch, "Ligra: A lightweight graph processing framework for shared memory," in *ACM Sigplan Notices*, vol. 48, no. 8. ACM, 2013, pp. 135–146.

[41] C. Yang, A. Buluç, and J. D. Owens, "GraphBLAST: A high-performance linear algebra-based graph framework on the GPU," *arXiv preprint arXiv:1908.01407*, 2019.

[42] M. Halappanavar, A. Pothen, A. Azad, F. Manne, J. Langguth, and A. Khan, "Codesign lessons learned from implementing graph matching on multithreaded architectures," *Computer*, no. 8, pp. 46–55, 2015.

[43] S. Beamer, K. Asanović, and D. Patterson, "The GAP benchmark suite,"

arXiv preprint arXiv:1508.03619, 2015.

[44] Y. Ji, H. Liu, and H. H. Huang, "iSpan: Parallel identification of strongly connected components with spanning trees," in *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2018, pp. 731–742.

[45] L. Dhulipala, G. E. Blelloch, and J. Shun, "Theoretically efficient parallel graph algorithms can be fast and scalable," in *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures*, 2018, pp. 393–404.