LACC: A Linear-Algebraic Algorithm for Finding Connected Components in Distributed Memory

Ariful Azad
Department of Intelligent Systems Engineering
Indiana University, Bloomington
azad@iu.edu

Aydın Buluç
Computational Research Division
Lawrence Berkeley National Laboratory
abuluc@lbl.gov

***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 parallel connected-components algorithm that can run on distributed-memory computers. Our algorithm uses linear algebraic primitives and is based on a PRAM algorithm by Awerbuch and Shiloach. We show that the resulting algorithm, named LACC for Linear Algebraic Connected Components, outperforms competitors by a factor of up to 12x for small to medium scale graphs. For large graphs with more than 50B edges, LACC scales to 4K nodes (262K cores) of a Cray XC40 supercomputer and outperforms 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.**

I. Introduction

Given an undirected graph $G = (V, E)$ on the set of vertices V and the set of edges E, a connected component 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 Awerbuch-Shiloach (AS) algorithm admits 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 a distributed-memory implementation of the AS algorithm [5]. We mapped the AS algorithm to GraphBLAS [9] primitives, which are standardized linear-algebraic functions that can be used to implement graph algorithms. Our algorithm is named as LACC for *linear algebraic connected components*. While the initial reasons behind choosing the AS algorithm were simplicity, performance guarantees, and expressibility using linear algebraic primitives, we found that it is never slower than the state-of-the-art distributed-memory algorithm ParConnect [10], and it often outperforms ParConnect by several fold.

The actual implementation of our algorithm uses the Combinatorial BLAS (CombBLAS) library [11], a well-known framework for implementing graph algorithms in the language of linear algebra. Different from the AS algorithm, our implementation fully exploits 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.

Distributed-memory LACC code that is used in our experiments is publicly available as part of the CombBLAS library[1]. A simplified unoptimized serial GraphBLAS implementation is also committed to the LAGraph Library[2] for educational purposes.

II. Background

A. 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.

B. GraphBLAS

The duality between sparse matrices and graphs has a long and fruitful history [12], [13]. Several independent systems have emerged that use matrix algebra to perform graph operations [11], [14], [15]. Recently, the GraphBLAS forum defined a standard set of linear-algebraic operations for implementing graph algorithms, leading to the GraphBLAS C API [16]. 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 [11], which are either semantically equivalent in functionality to their GraphBLAS counterparts or can be composed to match GraphBLAS functionality.

C. 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). We based our distributed-memory implementation on the Awerbuch-Shiloach (AS) algorithm, which itself 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. Due to its simpler data structures, AS is a better candidate for GraphBLAS-style implementation.

Randomization is also a fundamental technique applied to the connected components problem. The random-mate (RM) algorithm, due to Reif [17], 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.

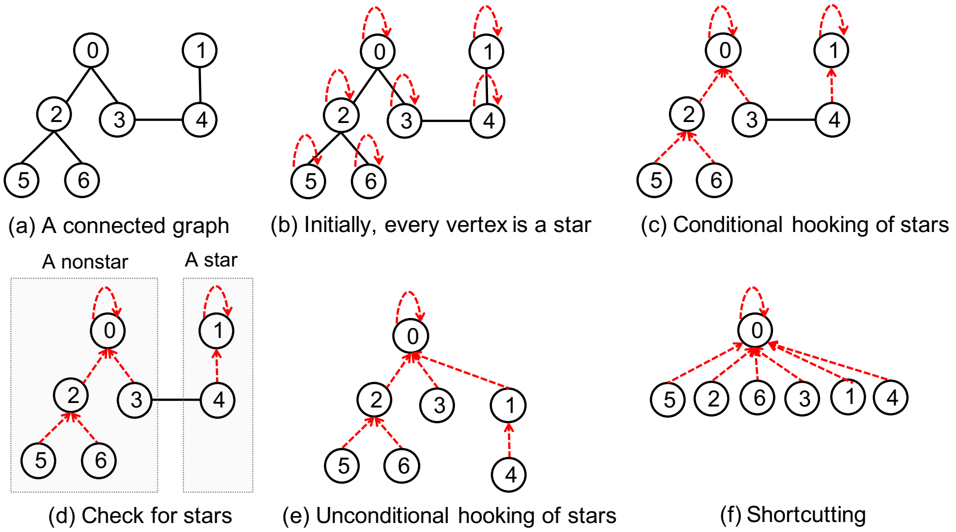
A similar randomization technique allowed Gazit to discover a work-efficient CRCW PRAM algorithm for the connected components problem [18]. His algorithm runs with $O(m)$ optimal work and $O(\log(n))$ span. More algorithms followed achieving the same work-span bound but improving the state-of-the-art by working with more restrictive models such as EREW (exclusive-read exclusive-write) [19], solving more general problems such as minimum spanning forest [20] whose output can be used to infer connectivity, and providing first implementations [21].

The literature on distributed-memory connected component algorithms and their complexity analyses, is significantly 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. [22] 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 [23] in distributed memory. Viral Shah's PhD thesis [24] 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. [25] 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 [26], [27]. 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.

[1] https://bitbucket.org/berkeleylab/combinatorial-blas-2.0/

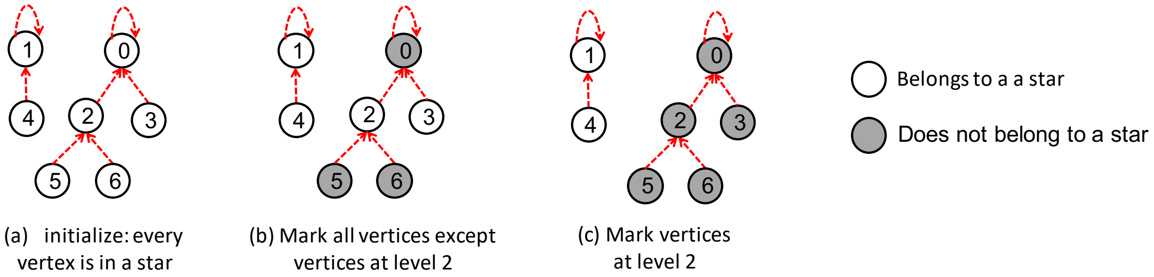
[2] https://github.com/GraphBLAS/LAGraph



(a) A connected graph (b) Initially, every vertex is a star (c) Conditional hooking of stars

(d) Check for stars (e) Unconditional hooking of stars (f) Shortcutting

Fig. 1: An illustrative example of the 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.



(a) initialize: every vertex is in a star (b) Mark all vertices except vertices at level 2 (c) Mark vertices at level 2

Fig. 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.

III. The Awerbuch-Shiloach (AS) algorithm

Overview. The AS algorithm maintains a forest (a collection of directed rooted trees), where each tree represents a connected component at the current stage of the algorithm. The algorithm begins with n single-vertex stars. In every iteration, the algorithm merges stars with other trees (both stars and nonstars) until no such merging is possible. This merging is performed by a process called *star hooking*, where the root of a star becomes a child of a vertex from another tree. Hooking stars unconditionally to other trees may create a cycle of trees, violating the forest requirement of the algorithm [5]. Hence, the algorithm performs a conditional hooking, followed by an unconditional hooking. 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 algorithm maintains a parent vector f, where $f[v]$ stores the parent of a vertex v. 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 the latter to be the parent of the former. 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

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

```
procedure AWERBUCH-SHILOACH($G(V, E)$)
    for every vertex $v$ in $V$ do                      ▷ Initialize
        $f[v] \leftarrow v$
    repeat
        ▷ Step1: Conditional star hooking
        for every edge $\{u, v\}$ in $E$ do in parallel
            if $u$ belongs to a star and $f[u] > f[v]$ then
                $f[f[u]] \leftarrow f[v]$
        ▷ Step2: Unconditional star hooking
        for every edge $\{u, v\}$ in $E$ do in parallel
            if $u$ belongs to a star and $f[u] \neq f[v]$ then
                $f[f[u]] \leftarrow f[v]$
        ▷ Step3: Shortcutting
        for every vertex $v$ in $V$ do in parallel
            $gf[v] \leftarrow f[f[v]]$
        for every vertex $v$ in $V$ do in parallel
            if $v$ does not belongs to a star then
                $f[v] \leftarrow gf[v]]$
    until $f$ remains unchanged
    return $f$
```

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 iteration. 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.

IV. THE AS ALGORITHM USING MATRIX ALGEBRA

In this section, we design the AS algorithm using the GraphBLAS API [9]. We used GraphBLAS API to 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 [28].

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.

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:         $star[v] \leftarrow true$
 4:         $gf[v] \leftarrow f[f[v]]$
 5:     ▷ Exclude every vertex $v$ with $l(v) > 2$ and its grandparent
 6:     for every vertex $v$ in $V$ do in parallel
 7:         if $f[v] \neq gf[v]$ then
 8:             $star[v] \leftarrow false$
 9:             $star[gf[v]] \leftarrow false$
10:     ▷ In nonstar trees, exclude vertices at level 2
11:     for every vertex $v$ in $V$ do in parallel
12:         $star[v] \leftarrow star[f[v]]$
13:     return $star$
```

- 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.

A. Designing the AS algorithm using GraphBLAS primitives

Conditional hooking. Algorithm 3 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 3 where we multiply the adjacency matrix $\mathbf{A}$ by the parent vector f on the (Select2nd, min) semiring.

Algorithm 3 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 4 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 3
```

Algorithm 5 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, $\emptyset$)        ▷ $f \leftarrow gf$
```

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 4 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 4 is similar to Algorithm 3.

Shortcut. Algorithm 5 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 6 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).

Implementing LACC using the SuiteSparse:GraphBLAS library. Currently, only the SuiteSparse:GraphBLAS library[3] provides a full sequential implementation of the GraphBLAS C API. We developed a simplified unoptimized serial GraphBLAS implementation of LACC using the SuiteSparse:GraphBLAS library to test the correctness of the presented algorithms with respect to the GraphBLAS API. Our SuiteSparse:GraphBLAS implementation is committed to the LAGraph Library[4] for educational purposes. In this paper, we do not report any performance numbers from the SuiteSparse:GraphBLAS implementation because it lacks several key optimizations (e.g., use of sparsity) that we implemented in CombBLAS.

B. *Efficient use of sparsity*

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

[3] http://faculty.cse.tamu.edu/davis/GraphBLAS.html

[4] https://github.com/GraphBLAS/LAGraph

Algorithm 6 Updating star memberships. **Inputs:** the parent vector f, the star vector $star$. **Output:** Updated $star$ vector.

```
procedure STARCHECK($f$, $star$)
    GrB_assign ($star$, ∅, ∅, true, GrB_ALL, 0, ∅)   ▷ initialize
    ▷ vertices whose parents and grandparents are different. See Algorithm 5 for the code for computing grandparents $gf$
    GrB_eWiseMult($hbool$, ∅, ∅, GrB_NE_T, $f$, $gf$, ∅)
    GrB_extract($h$, $hbool$, ∅, $gf$, GrB_ALL, 0, ∅)
    ▷ mark these vertices and their grandparents as nonstars
    GrB_Vector_nvals(&nnz, $h$)
    GrB_Vector_extractTuples(index, value, nnz, $h$)
    GrB_assign ($star$, ∅, ∅, false, index, nnz, ∅)
    GrB_assign ($star$, ∅, ∅, false, value, nnz, ∅)
    ▷ $star[v] \leftarrow star[f[v]]$
    GrB_Vector_extractTuples(idx, value, &n, $f$)   ▷ $n = |V|$
    GrB_extract ($star_f$, ∅, ∅, $star$, value, n, ∅)
    GrB_assign ($star$, ∅, ∅, $star_f$, GrB_All, 0, ∅)
```

TABLE I: 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

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 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 benefitted 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 4. 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 I summarizes the subset of vertices used in different steps of our algorithm.

V. IMPLEMENTING THE AS ALGORITHM IN COMBBLAS

We use the CombBLAS framework [11] to implement the GraphBLAS primitives needed to implement the Awerbuch-Shiloach algorithm. 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.

A. Parallel complexity

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 [29], [30]. 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 [31].

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

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

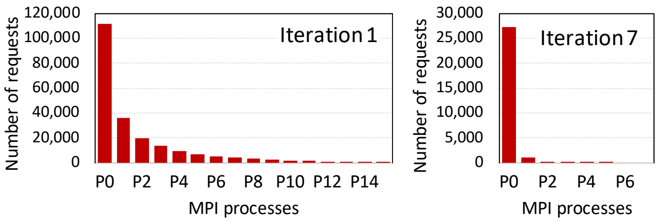


Fig. 3: Number of requests received by every process when accessing grandparents. We show iteration 1 and 7 when running LACC with 16 processes. Only even numbered processes are labeled on the x-axis. Lower ranked processes receive more requests than higher ranked process in all iterations. Later iterations are more imbalanced than earlier iterations.

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 $nnz(u), nnz(w) \leq n$.

In practice, we achieve high performance in all-to-all operations by employing other optimizations to CombBLAS' block distributed vectors, described in Section V-B, 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:

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

B. 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 grandparents 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.

TABLE II: 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

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 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 [31]. Hence, we replace all MPI_Alltoallv calls with a hypercube-based implementation by Sundar et al. [32], 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 [32], 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 8.

VI. Results

A. Evaluation platforms

We evaluate the performance of LACC on NERSC Edison and Cori KNL systems as described in Table II. We used

TABLE III: 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 [33]
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 [33]
M3	531M	1.047B	7.6M	Soil metagenomic data [10]
twitter7	41.65M	2.405B	1	twitter follower network [33]
sk-2005	50.64M	3.639B	45	2005 web crawl of .sk domain [33]
MOLIERE_2016	30.22M	6.677B	4,457	automatic biomedical hypothesis generation system [33]
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]

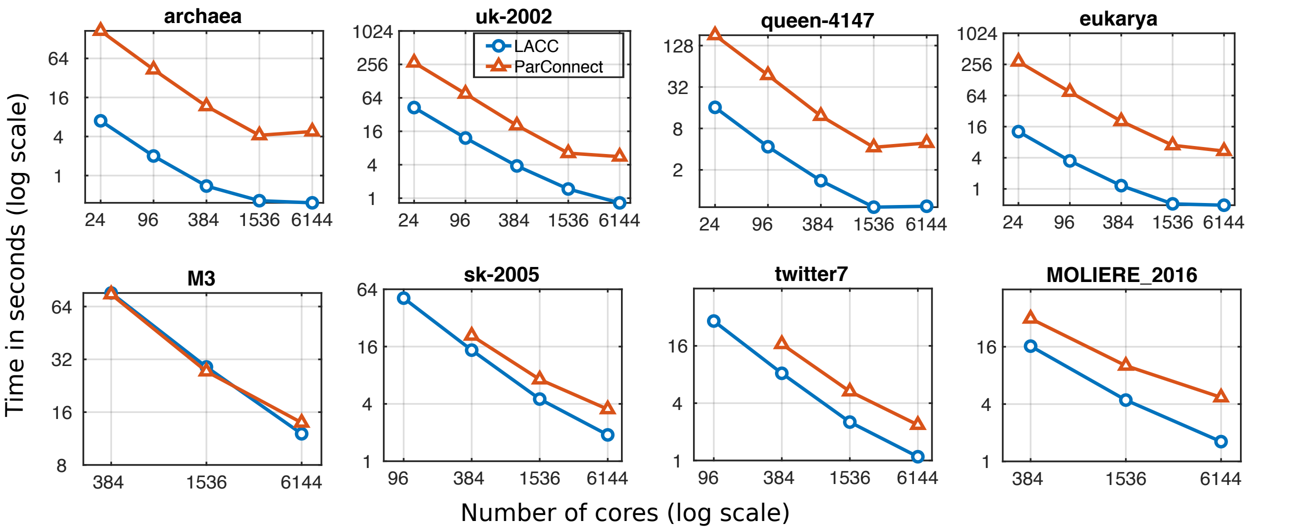


Fig. 4: Strong scaling of LACC and ParConnect on Edison on (up to 6144 cores on 256 nodes). LACC uses 4 MPI processes per node and ParConnect uses 24 MPI processes per node.

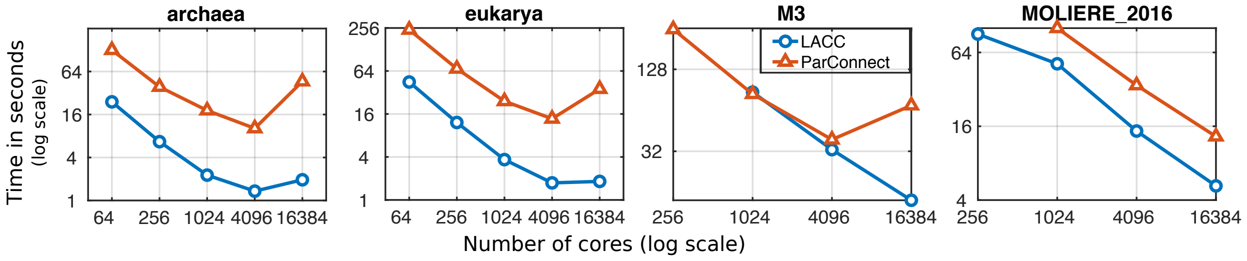


Fig. 5: Strong scaling of LACC and ParConnect on Cori KNL (up to 16,384 cores on 256 nodes). LACC uses 4 MPI processes per node and ParConnect uses 64 MPI processes per node. Graphs with large numbers of connected components are shown.

OpenMP for multithreaded execution within an MPI process. In our experiments, we only used square process grids because rectangular grids are not supported in CombBLAS [11]. 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.

B. Test problems

Table III 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].

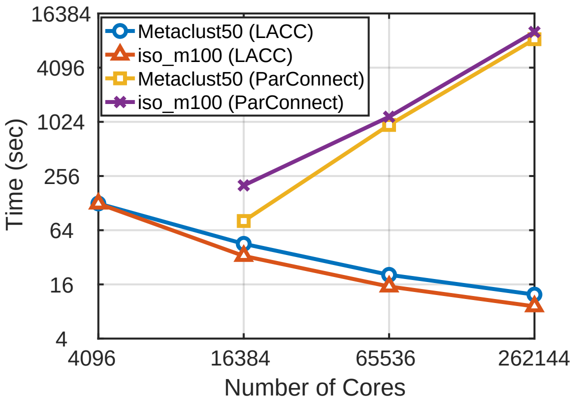


Fig. 6: Performance of LACC and ParConnect with two large protein-similarity networks on Cori KNL (up to 262,144 cores on 4096 nodes). LACC and ParConnect use 4 and 64 MPI processes per node, respectively. While LACC scales to 262,144 cores, ParConnect stopped scaling at this extreme scale. ParConnect ran out of memory on 64 nodes.

C. Performance of LACC with respect to the state-of-the-art

We compare the performance of LACC with ParConnect [10], the state-of-the art algorithm prior to our work. Similar to LACC, 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 4 shows the performance of LACC and ParConnect with the smaller eight test problems on Edison. 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 VI-E.

The relative performance of LACC and ParConnect on Cori is similar to Edison. Figure 5 shows results for four graphs that have the highest number of components. As with Edison, LACC outperforms ParConnect on all core counts on Cori for all graphs except M3, for which the performance is comparable. We observe that both LACC and ParConnect run 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) are more beneficial than more slower cores (e.g., KNL) [34].

D. Performance of LACC with 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 [35].

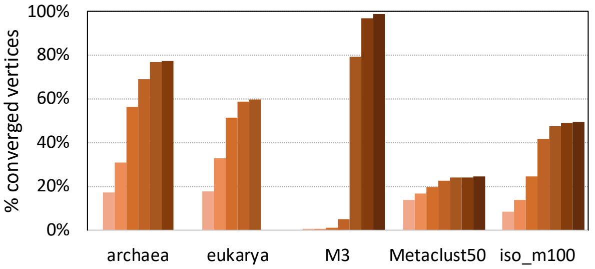


Fig. 7: Percentage of vertices in converged connected components at different iterations of LACC. For every graph, iterations are shown incrementally from left to right.

However, the last two graphs in Table III need more than 1TB of memory, requiring distributed-memory processing. We show the performance of LACC and ParConenct for these big graphs in Figure 6. We observed that LACC continues scaling to 4096 nodes (262,144 cores) on Cori and computes connected components in these large networks in just ten 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. The remarkable ability of LACC to process graphs with hundreds 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 VI-F.

E. Understanding the performance of LACC

We now explore different features of LACC and describe why it achieves better performance for most of the test graphs.

(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 has a large number of connected components. To demonstrate this, Figure 7 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 4 and Figure 6 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.

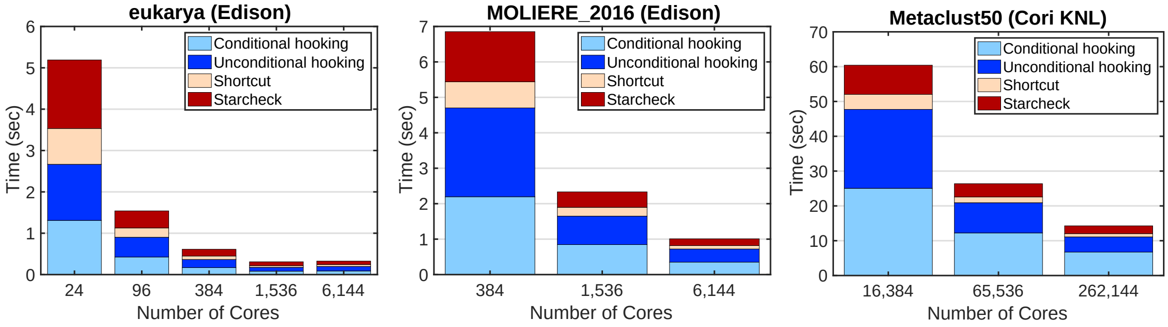


Fig. 8: Performance breakdown of LACC for three representative graphs.

(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 8 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 V-B makes the shortcut and starcheck operations highly scalable.

F. 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.

VII. Conclusions

We presented a distributed-memory connected component algorithm (LACC) that is implemented using sparse linear algebra and is based on the Awerbuch-Shiloach algorithm. LACC achieves 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) LACC relies on linear algebraic kernels that are highly optimized for distributed memory graph analysis, (2) whenever possible, LACC employs sparse vectors in the hooking, shortcutting and star finding steps, eliminating redundant computation and communication, and (3) LACC detects imbalanced collective communication patterns inherent in the AS algorithm and removes them with customized all-to-all operations.

Extreme scalability achieved by LACC 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 IV) 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 III. 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.

VIII. Acknowledgments

Partial funding for AA was provided by the Indiana University Grand Challenge Precision Health Initiative. AB was supported in part by the Applied Mathematics program of the DOE Office of Advanced Scientific Computing Research under Contract No. DE-AC02-05CH11231, and in part by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration.

We would like to thank Scott McMillan for providing valuable comments on an earlier draft of this paper. We used resources of the NERSC supported by the Office of Science of the DOE under Contract No. DE-AC02-05CH11231.

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. 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.

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

[13] J. Kepner and J. Gilbert, *Graph algorithms in the language of linear algebra*. SIAM, 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 *Computing Frontiers (CF)*, 2016, pp. 72–81.

[15] 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*, vol. 8, no. 11, pp. 1214–1225, 2015.

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

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

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

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

[24] 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.

[25] 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.

[26] 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.

[27] 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.

[28] 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.

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

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

[31] 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.

[32] 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.

[33] 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.

[34] 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.

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