An Adaptive Parallel Algorithm for Computing Connected Components

Chirag Jain, Patrick Flick, Tony Pan, Oded Green, and Srinivas Aluru, *Fellow, IEEE*

Abstract—We present an efficient distributed memory parallel algorithm for computing connected components in undirected graphs based on Shiloach-Vishkin's PRAM approach. We discuss multiple optimization techniques that reduce communication volume as well as load-balance the algorithm. We also note that the efficiency of the parallel graph connectivity algorithm depends on the underlying graph topology. Particularly for short diameter graph components, we observe that parallel Breadth First Search (BFS) method offers better performance. However, running parallel BFS is not efficient for computing large diameter components or large number of small components. To address this challenge, we employ a heuristic that allows the algorithm to quickly predict the type of the network by computing the degree distribution and follow the optimal hybrid route. Using large graphs with diverse topologies from domains including metagenomics, web crawl, social graph and road networks, we show that our hybrid implementation is efficient and scalable for each of the graph types. Our approach achieves a runtime of 215 seconds using 32 K cores of Cray XC30 for a metagenomic graph with over 50 billion edges. When compared against the previous state-of-the-art method, we see performance improvements up to 24 $\times$.

Index Terms—Parallel algorithms, distributed memory, breadth first search, undirected graphs

1 Introduction

Computing connected components in undirected graphs is a fundamental problem in graph analytics. The sizes of graph data collections continue to grow in multiple scientific domains, motivating the need for high performance distributed memory parallel graph algorithms, especially for large networks that cannot fit into the memory of a single compute node. For a graph $G(V, E)$ with n vertices and m edges, two vertices belong to the same *connected component* if and only if there is a path between the two vertices in G. Sequentially, this problem can be solved in linear $O(m + n)$ time, e.g., by using one of the following two approaches. One approach is to use graph traversal algorithms, i.e., either Breadth First (BFS) or Depth First Search (DFS). A single traversal is necessary for each connected component in the graph. Another technique is to use a union-find based algorithm, where each vertex is initially assumed to be a different graph component and components connected by an edge are iteratively merged.

Parallel BFS traversal algorithms have been invented that are work-optimal and practical on distributed memory systems for small-world graphs [1], [2]. While parallel BFS algorithms have been optimized for traversing a short diameter big graph component, they can be utilized for finding connected components. However, connectivity can be determined for only one component at a time, as BFS cannot merge the multiple partial search trees in the same component that are likely to arise during concurrent runs. For an undirected graph with a large number of small components, parallel BFS thus has limited utility. On the other hand, BFS is an efficient technique for scale-free networks that are characterized by having one dominant short diameter component.

The classic Shiloach-Vishkin (SV) algorithm [3], a widely known PRAM algorithm for computing connectivity, simultaneously computes connectivity of all the vertices and promises convergence in logarithmic iterations, making it suitable for components with large diameter, as well as for graphs with a large number of small sized components. Note that compared to simple label propagation techniques, the SV algorithm bounds the number of iterations to $O(\log n)$ instead of $O(n)$, where each iteration requires $O(m + n)$ work. In this work, we provide a novel edge-based parallel algorithm for distributed memory systems based on the SV approach. We also propose optimizations to reduce data volume and balance load as the iterations progress.

To achieve the best performance for different graph topologies, we introduce a dynamic pre-processing phase to our algorithm that guides the algorithm selection at runtime. In this phase, we try to classify the graph as scale-free by estimating the goodness of fit of its degree distribution to a power-law curve. If and only if the graph is determined to be scale-free, we execute one BFS traversal iteration from a single root to find the largest connected component with high probability, before switching to the SV algorithm to process the remaining graph. While the pre-processing phase introduces some overhead, we are able to improve the overall performance by using a combination of parallel BFS and SV algorithms, with minimal parameter tuning.

Our primary application driver is metagenomic assembly, where de Bruijn graphs are used for reconstructing,

• *The authors are with the Georgia Institute of Technology, Atlanta, GA 30332. E-mail: {cjain, patrick.flick, tony.pan, ogreen}@gatech.edu, aluru@cc.gatech.edu.*

Manuscript received 10 July 2016; revised 13 Feb. 2017; accepted 19 Feb. 2017. Date of publication 22 Feb. 2017; date of current version 9 Aug. 2017. Recommended for acceptance by A. Benoit.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TPDS.2017.2672739

from DNA sequencer outputs, constituent genomes in a metagenome [4]. A recent scientific study showed that high species-level heterogeneity in metagenomic data sets leads to a large number of weakly connected components, each of which can be processed as independent de Bruijn graphs [5]. This coarse grained data parallelism motivated our efforts in finding connected components in large metagenomic de Bruijn graphs. However, our work is applicable to graphs from domains beyond bioinformatics.

In this study, we cover a diverse set of graphs, both small world and large diameter, to highlight that our algorithm can serve as a general solution to computing connected components for undirected graphs. We experimentally evaluate our algorithm on de Bruijn graphs from publicly available metagenomic samples, road networks of the United States and European Union, scale-free networks from the internet, as well as Kronecker graphs from the Graph500 benchmark [6]. The graphs range in edge count from 82 million to 54 billion. Even though we focus on computing connected components in undirected graphs, ideas discussed in this work are applicable to finding strongly connected components in directed graphs as well. Our C++ and MPI-based implementation is available as open source at https://github.com/ParBLiSS/parconnect.

To summarize the contributions of this paper:

- We provide a new scalable strategy to adapt the Shiloach-Vishkin PRAM connectivity algorithm to distributed memory parallel systems.
- We discuss and evaluate a novel and efficient dynamic approach to compute weakly connected components on a variety of graphs, with small and large diameters.
- We demonstrate the scalability of our algorithm by computing the connectivity of the de Bruijn graph for a large metagenomic dataset with 1.8 billion DNA sequences and 54 billion edges in less than 4 minutes using 32 K cores.
- Depending on the underlying graph topology, we see variable performance improvements up to 24 $\times$ when compared against the state-of-the-art parallel connectivity algorithm.

2 Related Work

Due to its broad applicability, there have been numerous efforts to parallelize the connected component labeling problem. Hirschberg et al. [7] presented a CREW[1] PRAM algorithm that runs in $O(\log^2 n)$ time and does $O(n^2 \log n)$ work, while Shiloach and Vishkin [3] presented an improved version assuming a CRCW[2] PRAM that runs in $O(\log n)$ time using $O(m+n)$ processors. As our parallel SV algorithm is based on this approach, we summarize the SV algorithm in separate section. Krishnamurthy et al. [8] made the first attempt to adapt SV algorithm to distributed memory machines. However, their method is restricted to mesh graphs, which they could naturally partition among the processes [9]. Goddard et al. [10] discussed a practical implementation of SV algorithm for distributed machines with mesh network topology. Their method, however, was shown to exhibit poor scalability beyond 16 processors for sparse graphs [11].

Bader et al. [12] and Patwary et al. [13] discussed shared memory multi-threaded parallel implementations to compute spanning forest and connected components on sparse and irregular graphs. Recently, Shun et al. [14] reported a work optimal implementation for the same programming model. Note that these solutions are not applicable for distributed memory environments due to high frequency of remote memory accesses. Cong et al. [15] proposed a parallel technique for solving the connectivity problem on a single GPU.

There have been several recent parallel algorithms for computing the breadth-first search traversal on distributed memory systems [1], [2], [16]. However, parallel BFS does not serve as an efficient, stand-alone method for computing connectivity. There are also several large-scale distributed graph analytics frameworks that can solve the connectivity problem in large graphs, including GraphX [17], PowerLyra [18], PowerGraph [19], and GraphLab [20]. Iverson et al. [21] proposed a distributed-memory connectivity algorithm using successive graph contraction operations, however, the strong scalability demonstrated for this method was limited to 32 cores.

Slota et al. [22] proposed a shared memory parallel *Multistep* method that combines parallel BFS and label propagation (LP) technique and was reported to perform better than using BFS or LP alone. In their *Multistep* method, BFS is first used to label the largest component before using the LP algorithm to label the remaining components. More recently, they proposed a distributed memory parallel implementation of this method and showed impressive speedups against the existing parallel graph processing frameworks [23]. However, their algorithm design and experimental datasets are restricted to graphs which contain a single massive connected component. While our algorithm likewise employs a combination of algorithms, in contrast to *MultiStep*, we use BFS and our novel SV implementation, and determine dynamically at runtime whether the BFS should be executed.

This paper is an extension of our previous work [24] which described a parallel connected components algorithm based on the SV approach for large diameter metagenomic graphs. Here we propose a hybrid approach using both BFS and SV and present a generalized efficient algorithm for finding connected components in arbitrary undirected graphs. We show that using runtime algorithm selection and our SV implementation, our method generalizes to diverse graph topologies and achieves superior performance.

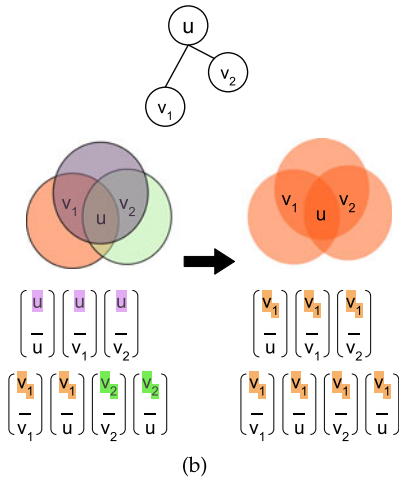
2.1 The Shiloach-Vishkin Algorithm

The Shiloach-Vishkin connectivity algorithm was designed assuming a PRAM model. It begins with singleton trees corresponding to each vertex in the graph and maintains this auxiliary structure of rooted directed trees to keep track of the connected components discovered so far during the execution. Within each iteration, there are two phases referred to as $shortcutting$ and $hooking$. $Shortcutting$ involves collapsing the trees using pointer doubling. On the other hand, $hooking$ connects two different connected components when they share an edge in the input graph. This algorithm requires $O(\log n)$ iterations each taking constant time. Since

1. CREW = Concurrent Read Exclusive Write.
2. CRCW = Concurrent Read Concurrent Write.

Symbol	Description	Definition
V	Vertices in graph G	
E	Edges in graph G	
$\langle p,q,r\rangle$	Tuple	$p,q,r \in \mathbb{Z}$
$\mathcal{A}_i$	Array of tuples in iteration i	
$\mathcal{P}_i$	Unique partitions	$\{p \mid \langle p,q,r\rangle \in \mathcal{A}_i\}$
$\mathcal{PB}_i(p)$	Partition bucket for partition p	$\{\langle \hat{p},q,r\rangle \in \mathcal{A}_i \mid \hat{p}=p\}$
$\mathcal{VB}_i(u)$	Vertex bucket for vertex u	$\{\langle p,q,r\rangle \in \mathcal{A}_i \mid r=u\}$
$\mathcal{V}_i(p)$	Vertex members in partition p	$\{r \mid \langle p,q,r\rangle \in \mathcal{PB}_i(p)\}$
$\mathcal{C}_i(p)$	Candidate partitions for partition p	$\{q \mid \langle p,q,r\rangle \in \mathcal{PB}_i(p)\}$
$\mathcal{M}_i(u)$	Partitions in vertex bucket for vertex u	$\{p \mid \langle p,q,r\rangle \in \mathcal{VB}_i(u)\}$
$\mathcal{N}_i(p)$	Neighborhood partitions of partition p	$\cup_{u\in\mathcal{V}_i(p)}\mathcal{M}_i(u)$

(a)



(b)

Fig. 1. (a) Summary of the notations used in Section 3. (b) Initialization of array $\mathcal{A}$ for a small connected component with three vertices u, v_1, v_2 in our algorithm. Partitions are highlighted using different shades. Desired solution, assuming $v_1 = \min(u, v_1, v_2)$, shown on the right will be to have all three vertices in a single component v_1. Accordingly, all the tuples associated with this component should contain the equal partition id v_1.

this approach uses $O(m+n)$ processors, the total work complexity is $O((m+n)\cdot \log n)$.

3 Algorithm

3.1 Parallel SV Algorithm

3.1.1 Notations

Given an undirected graph $G=(V,E)$ with $n=|V|$ vertices and $m=|E|$ edges, our algorithm identifies its connected components, and labels each vertex $v \in V$ with its corresponding component. Our algorithm works on an array of three-tuples $\langle p,q,r\rangle$, where p, q, and r are integers. The first two elements of these tuples will be updated in each iteration of the algorithm. The third element r corresponds to a vertex $r \in V$ of the graph and is not changed throughout the algorithm. This element will also be used to identify the vertices of G with their final connected components after termination.

Let $\mathcal{A}_i$ denote the array of tuples in iteration i. We initialize $\mathcal{A}_0$ as follows: for each vertex $x \in V$, we add the tuple $\langle x,\_,x\rangle$, and for each undirected edge $\{x,y\} \in E$, we add tuples $\langle x,\_,y\rangle$ and $\langle y,\_,x\rangle$. The middle elements will be initialized later during the algorithm.

We denote the set of unique values in the first entry of all the tuples in $\mathcal{A}_i$ by $\mathcal{P}_i$, therefore $\mathcal{P}_i = \{p \mid \langle p,q,r\rangle \in \mathcal{A}_i\}$. We refer to the unique values in $\mathcal{P}_i$ as *partitions*, which represent intermediate groupings of tuples that eventually coalesce into connected components. We say that a tuple $\langle p,q,r\rangle$ is a member of the partition p. Once the algorithm converges, all tuples for a vertex r will have a single unique partition p, which is also the unique connected component label for this vertex.

In order to refer to the tuples of a partition p, we define the *partition bucket* $\mathcal{PB}_i(p)$ of p as those tuples which contain p in their first entry: $\mathcal{PB}_i(p) = \{\langle \hat{p},q,r\rangle \in \mathcal{A}_i \mid \hat{p}=p\}$. Further, we define the *candidates* or the next potential partitions $\mathcal{C}_i(p)$ of p as the values contained in the second tuple position of the partition bucket for p: $\mathcal{C}_i(p) = \{q \mid \langle p,q,r\rangle \in \mathcal{PB}_i(p)\}$. We denote the minimum of the candidates of p as $p_{min} = \min \mathcal{C}_i(p)$. A partition p for which $p_{min} = p$ is called a *stable partition*. Further, to identify all the vertices in a partition, we define the *vertex members* of a partition p as $\mathcal{V}_i(p) = \{r \mid \langle p,q,r\rangle \in \mathcal{PB}_i(p)\}$.

Each vertex $u \in V$ is associated with multiple tuples in $\mathcal{A}_i$, possibly in different partitions p. We define *vertex bucket* $\mathcal{VB}_i(u)$ as those tuples which contain u in their third entry: $\mathcal{VB}_i(u) = \{\langle p,q,r\rangle \in \mathcal{A}_i \mid r=u\}$. We define the partitions $\mathcal{M}_i(u)$ as the set of partitions in the vertex bucket for u: $\mathcal{M}_i(u) = \{p \mid \langle p,q,r\rangle \in \mathcal{VB}_i(u)\}$. The minimum partition in $\mathcal{M}_i(u)$, i.e., $\min \mathcal{M}_i(u)$ is called *nominated partition* by u.

For a small example graph with vertices u, v_1, v_2, (Fig. 1), we show the array of tuples $\mathcal{A}$. At the initialization stage, the vertex bucket $\mathcal{VB}_0(u)$ of u is the set of tuples $\{\langle u,\_,u\rangle, \langle v_1,\_,u\rangle, \langle v_2,\_,u\rangle\}$. The set of unique partitions $\mathcal{P}_0$ equals $\{u, v_1, v_2\}$. The partition bucket $\mathcal{PB}_0(u)$ for partition u is given by the set $\{\langle u,\_,u\rangle, \langle u,\_,v_1\rangle, \langle u,\_,v_2\rangle\}$. At termination of our algorithm, all tuples will have the same common partition id, which for this example will be $\min(u, v_1, v_2)$.

Each partition is associated with a set of vertices, and the tuples for a vertex can be part of multiple partitions. We define the neighborhood for a partition p as those partitions which share at least one vertex with p, i.e., those which share tuples with a common identical value in the third tuple element. More formally, we define the *neighborhood* partitions of p as $\mathcal{N}_i(p) = \cup_{u\in\mathcal{V}_i(p)}\mathcal{M}_i(u)$. In the above example, the neighborhood partitions $\mathcal{N}_0(v_1)$ for the partition v_1 are u, v_1 and v_2. All the notations introduced in this section are summarized in Table 1 a for quick reference.

3.1.2 Algorithm

We first describe the sequential version of our algorithm, outlined in Algorithm 1 (see Fig. 2). Our algorithm is structured similar to the classic *Shiloach-Vishkin* algorithm. However, our algorithm is implemented differently, using an edge-centric representation of the graph.

At a high level, every vertex begins in its own partition, and partitions are connected via the edges of the graph. In each iteration, we join each partition to its numerically minimal neighbor, until the partitions converge into the connected components of the graph. In order to resolve

Algorithm 1: Connected components labeling

Input: Undirected graph $G = (V, E)$
Output: Labeling of Connected Components

1 $\mathcal{A}_0 = []$
2 **for** $x \in V$ **do** $\mathcal{A}_0$.append($\langle x, \_, x\rangle$);
3 **for** $\{x, y\} \in E$ **do** $\mathcal{A}_0$.append($\langle x, \_, y\rangle$, $\langle y, \_, x\rangle$);
4 $i \leftarrow 1$
5 $converged \leftarrow$ false
6 **while** $converged \neq true$ **do**
7 $converged \leftarrow$ true
8 $\mathcal{A}_i \leftarrow \mathcal{A}_{i-1}$
9 $\mathcal{M}_i(u) \leftarrow$ sort($\mathcal{A}_i$ by third element)
10 **for** $u \in V$ **do**
11 $u_{min} \leftarrow \min \mathcal{M}_i(u)$
12 **for** ***each*** $\langle p, q, r\rangle \in \mathcal{VB}_i(u)$ **do**
13 $\langle p, \boldsymbol{q}, r\rangle \leftarrow \langle p, \boldsymbol{u_{min}}, r\rangle$
14 **end**
15 **end**
16 $\mathcal{C}_i(p) \leftarrow$ sort($\mathcal{A}_i$ by first element)
17 **for** $p \in \mathcal{P}_i$ **do**
18 $p_{min} \leftarrow \min \mathcal{C}_i(p)$
19 **if** $p \neq p_{min}$ **then**
20 $converged \leftarrow$ false
21 **end**
22 **for** ***each*** $\langle p, q, r\rangle \in \mathcal{PB}_i(p)$ **do**
23 $\langle \boldsymbol{p}, q, r\rangle \leftarrow \langle \boldsymbol{p_{min}}, q, r\rangle$
24 **end**
25 $\mathcal{A}_i$.append($\langle p_{min}, \_, p_{min}\rangle_{tmp}$)
26 **end**
27 **redo** steps 9 - 15
28 **redo** steps 16 - 24
29 **for** ***each*** $\langle p, q, r\rangle_{tmp} \in \mathcal{A}_i$ **do**
30 $\mathcal{A}_i$.erase($\langle p, q, r\rangle_{tmp}$)
31 **end**
32 $i \leftarrow i + 1$
33 **end**

Fig. 2. Our parallel SV algorithm, presented using sequential semantics.

large diameter components quickly, we utilize the pointer doubling technique during shortcutting. To implement pointer doubling, we will require the *parent* partition id of the newly joined partition in each iteration. We use *temporary* tuples $\langle p, q, r\rangle_{tmp}$ to fetch this information. These tuples will be created and erased within the same iteration.

As laid out in Section 3.1.1, we first create an array of tuples $\mathcal{A}$, containing one tuple per vertex and two tuples per edge (Algorithm 1 (see Fig. 2)). In each iteration i, we perform four sorting operations over $\mathcal{A}_i$. During the first two sorting operations, we compute and join each partition p to its minimum neighborhood, i.e., $\min \mathcal{N}_i(p)$. Sorting $\mathcal{A}_i$ by the third entry, namely the vertex ids enables easy and cache efficient processing of each vertex bucket $\mathcal{VB}_i(u), u \in V$, since the tuples of a bucket are positioned contiguously in $\mathcal{A}_i$ due to the sorted order (line 9-15). For each vertex bucket $\mathcal{VB}_i(u)$, we scan all the partition ids containing u, i.e., $\mathcal{M}_i(u)$ and compute the nominated partition u_{min} which becomes the candidate (potential next partition). We save the candidate partition id in the second element of the tuples.

After computing all the candidate partitions, we perform a second global sort of $\mathcal{A}_i$ by the first tuple element in order to process the partition buckets $\mathcal{PB}_i$ (line 16-24). Each partition $p \in \mathcal{P}_i$ then computes and joins the minimum candidate partition, i.e., $p_{min} = \min \mathcal{C}_i(p)$. In other words, partition p joins its minimum neighbor p_{min}. We loop over these two sort-and-update steps until partitions converge into the connected components of the graph. Convergence for a partition p is reached when its *neighborhood* $\mathcal{N}_i(p)$ contains p as its only member. Consequently, we can determine when to terminate the algorithm by checking whether all the partitions have fully converged, i.e., if they do not have any further neighboring partitions. For any partition p, $p \neq p_{min}$ implies the existence of at least one neighbor partition around p (line 19).

Iteratively invoking lines 7-24 until convergence produces connected components of the graph within $O(n)$ iterations in the worst-case. By following the pointer doubling technique described in the SV algorithm [3], we achieve logarithmic convergence. We summarize the role of all the four sorting operations in Fig. 3. After joining partition p to p_{min}, we revise p_{min} to $\min \mathcal{M}_i(p_{min})$. The revision is effected by introducing temporary tuples $\langle p_{min}, \_, p_{min}\rangle_{tmp}$ in $\mathcal{A}_i$ (line 25), then repeating the two sorts by the third and first element respectively (line 27, 28). In a way similar to the first two sorts of this iteration, the third sort forces the vertex p_{min} to nominate $\min \mathcal{M}_i(p_{min})$ as the candidate partition id in the second element of the temporary tuples. Partition p_{min}, then, joins the partition id $\min \mathcal{M}_i(p_{min})$ after the final sort. The temporary tuples are removed from $\mathcal{A}_i$ after the pointer doubling phase is completed (line 30).

Note that the global count of the temporary tuples equals $|\mathcal{P}_i|$ in each iteration, and we know $|\mathcal{P}_i| \leq |V|$ (by the definition of $\mathcal{P}_i$). Therefore, the $O(m + n)$ bound holds for $|\mathcal{A}_i|$ throughout the execution. After the algorithm converges, the unique connected component label c of a vertex $u \in V$ can be projected from the first element of any tuple $\langle c, \_, u\rangle$ in $\mathcal{A}$.

3.1.3 Parallel Algorithm

We now describe our parallel implementation of the above algorithm for connected components labeling in a distributed memory environment. In this setting, each processor in the environment has its own locally addressable memory space. Remote data is accessible only through well defined communication primitives over the interconnection network. The algorithm consists of three components: data distribution, parallel sorts, and bucket updates. We designed our algorithm and its components using MPI primitives.

Data Distribution. All data, including the input, intermediate results, and final output, are equally distributed across all available processors. As specified in Section 3.1.2, the pipeline begins by generating tuples of the form $\langle p, q, r\rangle$ from the block distributed input $G(V, E)$ as edge list. By the end of this operation, each of the ρ processes contains its equal share of $|\mathcal{A}|/\rho$ tuples.

Parallel Sorts. The bulk step of the algorithm is the sorting of tuples by either their third or first element in order to form the buckets $\mathcal{VB}_i$ or $\mathcal{PB}_i$, respectively. Parallel distributed memory sorting has been studied extensively. Blelloch et al. [25] give a good review of different methods. With sufficiently large count of elements per process, which is often true while processing large datasets, the study concluded that samplesort is the fastest. Accordingly, we implement a

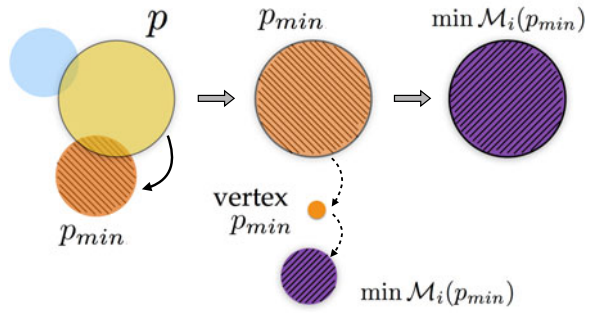


Fig. 3. Role of the four sorting phases used in each iteration of the algorithm. Using the first two sorts, partition p joins p_{min}. The next two sorts enable pointer-jumping as p_{min} joins $\min \mathcal{M}_i(p_{min})$. The temporary tuple $\langle p_{min}, -, p_{min} \rangle_{tmp}$ used in the algorithm simulates a link between the partition p_{min} and the vertex p_{min} to allow jumping.

variant of samplesort with regular sampling, where each processor first sorts its local array independently, and then picks equally spaced samples. The samples are then again sorted and $\rho - 1$ of these samples are used as splitters for distributing data among processors. In a final step, the sorted sequences are merged locally.

Bucket Updates. After each sort, we need to determine the minimum element for each bucket, either u_{min} for $\mathcal{VB}_i(u)$ or p_{min} for $\mathcal{PB}_i(p)$. As a result of the parallel sorting, all the tuples $\langle p, q, r \rangle$ belonging to the same bucket are stored consecutively. However, a bucket might span multiple processors. Therefore, the first and last bucket of each processor require global communication during processing, while the internal buckets are processed in the same way as in the sequential case. Note, the first and last bucket on a processor may be the same if a bucket spans an entire processor. Communicating the minimum of buckets with the previous and next processor would require $O(\rho)$ communication steps in the worst case, since large $O(|\mathcal{A}|)$ size partitions can span across $O(\rho)$ processes. We thus use two parallel prefix (scan) operations with custom operators to achieve independence from the size of partitions, requiring at most $O(\log \rho)$ communication steps in addition to the local linear time processing time.

We describe the custom reduction operation to compute the p_{min} within the partition buckets $\mathcal{PB}_i(p)$. Note that when computing p_{min} in the algorithm, $\mathcal{A}_i$ is already sorted by the first element of the tuples and p_{min} is the minimum second element for tuples in each bucket. We first perform an exclusive scan, where each processor participates with the minimum tuple from its last bucket. This operation communicates the minimum of buckets from lower processor rank to higher rank. The binary reduction operator chooses from two tuples the tuple $\langle p, q, r \rangle$ with the maximum p, and between those with equal p, the minimum q. Next we perform a reverse exclusive prefix scan to communicate the minimum from high rank to low rank. Here, each processor participates with its minimum tuple of its first bucket. Given the two results of the scan operations, we can compute for each processor the overall minimum p_{min} for both the first and the last buckets. Computing u_{min} follows a similar procedure.

Runtime Complexity. The runtime complexity of each iteration is dominated by sorting $\mathcal{A}$, and the number of iterations is bounded by $O(\log n)$. If $T(k, \rho)$ is the runtime to sort k elements using ρ processes, the runtime of our algorithm for computing connectivity of graph $G(V, E)$ equals $O(log(n) \cdot T(m + n, \rho))$.

3.1.4 Excluding Completed Partitions

As the algorithm progresses through iterations, certain partitions become *completed*. A partition p is *completed* if p has no neighbor partition except itself, i.e., $\mathcal{N}_i(p) = \{p\}$. Even though we have described how to detect the global convergence of the algorithm, detecting as well as excluding the *completed* partitions reduces the active working set throughout successive iterations.

By the definition of $\mathcal{N}_i(p)$ in Section 3.1.1, $\mathcal{N}_i(p) = \{p\}$ implies that $\cup_{u \in \mathcal{V}_i(p)} \mathcal{M}_i(u) = \{p\}$. Since the third elements of the tuples are never altered, each vertex is associated with at least one partition throughout the algorithm, therefore $|\mathcal{M}_i(u)| > 0 \, \forall u \in V$. Using these arguments, we claim the following: p is *completed* $\Leftrightarrow \mathcal{M}_i(u) = \{p\} \, \forall u \in \mathcal{V}_i(p)$. Once the partition is *completed*, it takes us one more iteration to detect its completion. While processing the vertex buckets after the first sort of the algorithm, we label all the tuples in $\mathcal{VB}_i(u), u \in V$ as *potentially completed* if $|\mathcal{M}_i(u)| = 1$. While processing the partition buckets subsequently, partition p is marked as *completed* if all the tuples in $\mathcal{PB}_i(p)$ are *potentially completed*.

Completed partitions are marked as such and swapped to the end of the local array. All following iterations treat only the first, non-completed part of its local array as the local working set. As a result, the size of the active working set shrinks throughout successive iterations. This optimization yields significant reduction in the volume of active data, particularly for graphs with a large number of small components, since many small connected components are quickly identified and excluded from future processing.

3.1.5 Load Balancing

Although we initially start with a block decomposition of the array $\mathcal{A}$, exclusion of *completed partitions* introduces an increasing imbalance of the active elements with each iteration. Since we join partitions from larger *ids* to smaller *ids*, a large partition will have smaller final partition ids than small partitions probabilistically. As the sort operation maps large id partitions to higher rank processes, the higher rank processes retain fewer and fewer active tuples over time, while lower rank processes contain growing partitions with small ids. Our experiments in Section 5 study this imbalance of data distribution and its effect on the overall run time. We resolve this problem and further optimize our algorithm by evenly redistributing the active tuples after each iteration. Our results show that this optimization yields significant improvement in the total run time.

3.2 Hybrid Implementation Using BFS

Connected components can be found using a series of BFS traversals, one for each component. The known parallel BFS algorithms are asymptotically work-optimal, i.e., they maintain $O(m + n)$ parallel work for small-world networks [1]. Parallel BFS software can be adapted to achieve the same objective as our parallel SV algorithm, namely to compute all the connected components in a graph. To do so, parallel BFS can be executed iteratively, each time selecting a new seed vertex from among the vertices that were not visited

during any of the prior BFS iterations. However, we note the following strengths and weaknesses associated with using BFS methods for the connectivity problem:

- *Pro:* For a massive connected component with a small diameter, the large number of vertices at each level of the traversal yields enough data parallelism for parallel BFS methods to become bandwidth bound, and thus efficient.
- *Con:* When the diameter of a component is large and vertex degrees are small, for instance in mesh graphs, the number of vertices at each level of BFS traversal is small. The application becomes latency-bound due to the lack of data parallelism. This leads to under-utilization of the compute resources and the loss of efficiency in practice [1].
- *Con:* For graphs with a large number of small components, parallel BFS needs to be executed repeatedly. The application becomes latency-bound as the synchronization and remote communication latency costs predominate the effective work done during the execution. In this case, BFS method's scalability is greatly diminished. Slota et al. [22] draw a similar conclusion while parallelizing the strongly connected components problem using shared memory systems.

A small world scale-free network contains a single large connected component [26]. To compute the connectivity of these graphs, we note that identifying the first connected component using a BFS traversal is more efficient than using the SV algorithm over the complete graph. For parallel BFS, we use Buluç et al.'s [1] state-of-the-art implementation available as part of the CombBLAS library [27] and integrate this software as an alternative preprocessing step to our parallel SV algorithm.

Scale-free networks are characterized by a power-law vertex degree distribution [28]. Therefore, we classify the graph structure as scale-free by checking if the degree distribution follows a power-law distribution. We use the statistical framework described by Clauset et al. [29] to fit a power-law curve to the discrete graph degree distribution, and estimate the goodness of fit with one-sample Kolmogorov-Smirnov (K-S) test. The closer the K-S statistic value is to 0, the better is the fit. If this value is below a user specified threshold τ, then we execute a BFS iteration before invoking our parallel SV algorithm. Algorithm 2 (see Fig. 4) gives the outline of our hybrid approach.

In our implementation, we choose to store each undirected edge (u, v) as two directed edges (v, u) and (u, v) in our edge list. This simplifies the computation of the degree distribution of the graph (line 2). We compute the degree distribution $\mathcal{D}$ of the graph by doing a global sort of edge list by the source vertex. Through a linear scan over the sorted edge list, we compute the degree of each vertex $u \in V$. In practice, it is safe to assume that the maximum vertex degree c is much smaller than number of edges $|E|$ ($c \ll |E|$). Thus each process can compute the local degree distribution in an array of size c, and a parallel reduction operation is used to solve for $\mathcal{D}$. Once $\mathcal{D}$ is known, evaluating the degree distribution statistics takes insignificant time as size of $\mathcal{D}$ equals c. Therefore, we compute the K-S statistics as described before, sequentially on each process.

Algorithm 2: Connected components labeling

```
   Input: Undirected graph G = (V, E)
   Output: Labeling of Connected Components
1  // Graph structure prediction
2  D ← Degree_Distribution(G)
3  if K-S statistic (D) < τ then
4  |  // Relabel vertices
5  |  G(V, E) ← G(V, E) s.t. u ∈ [0, |V| − 1] ∀u ∈ V
6  |  //Execute BFS
7  |  choose a seed s ∈ V
8  |  VI ← Parallel-BFS(s)
9  |  //Filter out the traversed component
10 |  V ← V \ VI
11 |  E ← E \ {(u, v)|u ∈ VI}
12 end
13 Parallel-SV(G(V, E))
```

Fig. 4. Hybrid approach using parallel BFS and SV algorithms to compute connected components.

If the K-S statistic is below the set threshold, we choose to run the parallel BFS on $G(V, E)$ (line 3). Buluç's BFS implementation works with the graph in an adjacency matrix format. Accordingly, we relabel the vertices in $G(V, E)$ such that vertex ids are between 0 to $|V| - 1$ (line 5). This process requires sorting the edge list twice, once by the source vertices and second by the destination vertices. After the first sort, we perform a parallel prefix (scan) operation to label the source vertices with a unique id $\in [0, V - 1]$. Similarly, we update the destination vertices using the second sort.

Next, we execute the parallel BFS from a randomly selected vertex in $G(V, E)$ and get a distributed list of visited vertices $\mathcal{VI}$ as the result. Note that the visited graph component is expected to be the largest one as it spans the majority of $G(V, E)$ in the case of scale-free graphs. To continue solving for other components, we filter out the visited component $\mathcal{VI}$ from $G(V, E)$ (line 10,11). $\mathcal{VI}$ is distributed identically as V, therefore vertex filtering is done locally on each process. We already have the edge list E in the sorted order by destination vertices due to the previous operations, therefore we execute an all-to-all collective operation to distribute $\mathcal{VI}$ based on the sorted order and delete the visited edges locally on each processor. Finally, irrespective of whether we use BFS or not, we run the parallel SV algorithm on $G(V, E)$ (line 13). In our experiments, we show the overall gain in performance using the hybrid approach as well as the additional overhead incurred by the prediction phase. We also report the proportion of time spent in each of the prediction, relabeling, parallel-BFS, filtering, and parallel-SV stages.

4 Experimental Setup

4.1 Hardware

For the experiments, we use Edison, a Cray XC30 supercomputer located at Lawrence Berkeley National Laboratory. In this system, each of the 5,576 compute nodes has two 12-core Intel Ivy Bridge processors with 2.4 GHz clock speed and 64 GB DDR3 memory. To perform parallel I/O, we use the scratch storage supported through the Lustre

TABLE 1
List of the Nine Graphs and Their Sizes Used for Conducting Experiments

Id	Dataset	Type	Vertices	Undirected Edges	Components	Approx. diameter	Largest component	Source
M1	Lake Lanier	Metagenomic	1.1 B	1.1 B	2.6 M	3,763	53%	NCBI (SRR947737)
M2	Human Metagenome	Metagenomic	2.0 B	2.0 B	1.0 M	3,989	91.1%	NCBI (SRR1804155)
M3	Soil (Peru)	Metagenomic	531.2 M	523.6 M	7.6 M	2,463	0.3%	MG-RAST (4477807.3)
M4	Soil (Iowa)	Metagenomic	53.7 B	53.6 B	319.2 M	-	44.2%	JGI (402461)
G1	Twitter	Social	52.6 M	2.0 B	29,533	16	99.99%	[30]
G2	sk-2005	Web Crawl	50.6 M	1.9 B	45	27	99.99%	[31]
G3	eu-usa-osm	Road Networks	74.9 M	82.9 M	2	25,105	65.2%	[31]
K1	Kronecker (scale = 27)	Kronecker	63.7 M	2.1 B	19,753	9	99.99%	Synthetic [6]
K2	Kronecker (scale = 29)	Kronecker	235.4 M	8.6 B	73,182	9	99.99%	Synthetic [6]

Edge between two vertices is counted once while reporting the graph sizes. Largest component's size is computed in terms of percentage of count of edges in the largest component relative to complete graph.

file system. We assign one MPI process per physical core for the execution of our algorithm. Further, we only use square process grids as CombBLAS [27] requires the process count to be a perfect square.

4.2 Datasets

Table 1 lists the nine graphs used in our experiments. These include four de Bruijn graphs constructed from different metagenomic sequence datasets, one social graph from Twitter, one web crawl, one road network and two synthetic Kronecker graphs from the Graph500 benchmark. The sizes of these graphs range from 83 million edges to 54 billion edges.

For each graph, we report the relevant statistics in Table 1 to correlate them with our performance results. Computing the exact diameter is computationally expensive and often infeasible for large graphs [32]. As such, we compute their approximate diameters by executing a total of 100 BFS runs from a set of random seed vertices. For all the graphs but M4, this approach was able to give us an approximation. However, the size of M4 required a substantial amount of time for completing this task and as such it did not complete. We estimate that only 4 of the 9 tested graphs are small world networks.

4.2.1 Metagenomic de Bruijn Graphs

M1-M4 are built using publicly available metagenomics samples from different environments. We obtained the sequences in FASTQ format. We discarded the sequences with unknown nucleotides using the fastx_clipper utility supported in the FASTX toolkit [33]. The size of the sequence dataset depends upon the amount of sampling done for each environment. We build de Bruijn graphs from these samples using the routines from the parallel distributed memory k-mer indexing library Kmerind [34]. It is worth noting that in de Bruijn graphs, vertex degrees are bounded by 8 [4]. One motivation for picking samples from different environments is the difference in graph properties associated with them such as the number of components and relative sizes. These are dependent on the degree of microbial diversity in the environments. Among the environments we picked, it has been estimated that the soil environments are the most diverse while the human microbiome samples are the least diverse of these environments [35]. This translates to large number of connected components in the soil graphs M3 and M4.

4.2.2 Other Graphs

Graphs K1-K2 and G1-G3 are derived from widely used graph databases and benchmarks. We use the synthetic Kronecker graph generator from the Graph500 benchmark specifications [6] to build Kronecker graphs with scale 27 (K1) and 29 (K2). Graphs G1-G3 are downloaded directly from online databases in the edge list format. G1 and G2 are small world scale-free networks from twitter and online web crawl respectively. G3 consists of two road networks from Europe and USA, downloaded from the Florida Sparse Matrix Collection [31]. Among all our graphs, G3 has the highest estimated diameter of 25 K. To read these data files in our program, a file is partitioned into equal-sized blocks, one per MPI process. The MPI processes concurrently read the blocks from the file system and generate distributed arrays of graph edges in a streaming fashion.

5 Performance Analysis

In all our experiments, we exclude file I/O and de Bruijn graph construction time from our benchmarks, and begin profiling after the block-distributed list of edges are loaded into memory. Profiling terminates after computing the connected component labels for all the vertices in the graph. Each vertex id in the input edge list is assumed to be a 64 bit integer. The algorithm avoids any runtime bias on vertex naming of the graph by permuting the vertex ids using Robert Jenkin's 64 bit mix invertible hash function [36].

5.1 Load Balancing

We first show the impact of the two optimizations performed by our parallel SV algorithm (Sections 3.1.4 and 3.1.5) for reducing and balancing the work among the processes. Our algorithm used 10 iterations to compute the connectivity of M1. Fig. 5 shows the minimum (min), maximum (max), and mean size of the distributed tuple array per process as iterations progress in three variants of our algorithm, using 256 cores. The max load is important as it determines the parallel runtime. A smaller separation between the min and max values indicates better load balance. The first implementation, referred to as Naive (Section 3.1.3), does not remove the completed components along the iterations and therefore the work load remains constant. Removing the stable components reduces the size of the working set per each iteration as illustrated by the desirable decrease in mean

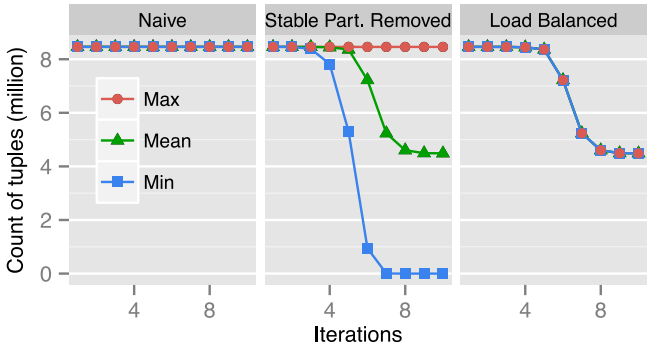


Fig. 5. Work load balance in terms of tuples per processes during each iteration of the three algorithm variants for parallel SV algorithm. Illustrated are the maximum, average, and minimum count of tuples on all the processes. The experiments were conducted using the M1 graph and 256 processor cores. Each edge is represented as two tuples internally in the algorithm.

tuple count. The difference between min and max grows significantly after four iterations. With our load balanced implementation, we see an even distribution of tuples across processors, as the minimum and maximum count are the same for each iteration. We see that the mean drops to about 50 percent of the initial value because the largest component in M1 contains 53 percent of the total edges (Table 1).

Consequently, we see improvement in the execution time for M1 and M3 in Fig. 6 as a result of these optimizations. Of the three implementations, the load balanced implementation consistently achieves better performance against the other two approaches. For the M2 graph, we get negligible gains using our load-balanced approach against the Naive approach because the largest component in M2 covers 91 percent of the graph. Therefore, the total work load stays roughly the same across the iterations.

5.2 Hybrid Implementation Analysis

As discussed in Section 3.2, BFS is more efficient for computing the first component in the small world scale-free graphs. We use an open-source C++ library [37] which fits the power-law distributions to discrete empirical data based on the procedure described by Clauset et al. [29]. Table 2 shows the K-S statistic value computed using the degree distribution for all our graphs. For each of the graphs with scale-free topology (G1, G2, K1, K2), there is a clear distinction of these values against rest of the graphs. Based on

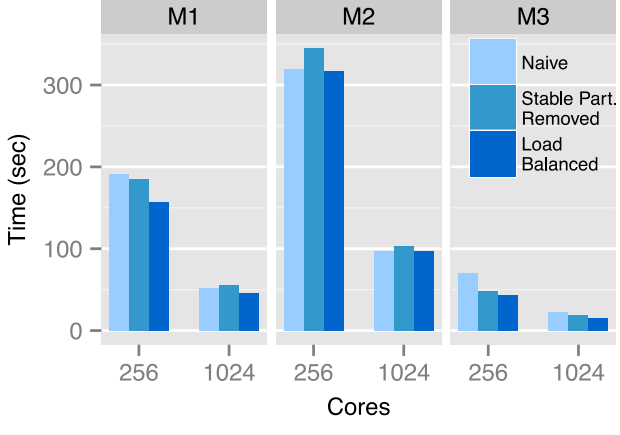


Fig. 6. Performance gains due to load balancing for graphs M1-M3 using 256, 1,024 processor cores.

TABLE 2
Kolmogorov-Smirnov Test Values Used to Estimate the Goodness of Power Law Curve Fit to the Degree Distribution of Each Graph

Dataset	K-S statistic	Run BFS iteration?	Correct Decision
M1	0.41	×	✓
M2	0.24	×	×
M3	0.39	×	✓
M4	0.31	×	✓
G1	**0.01**	✓	✓
G2	**0.03**	✓	✓
G3	0.21	×	✓
K1	**0.01**	✓	✓
K2	**0.01**	✓	✓

BFS is executed if K-S statistic value is less than 0.05.

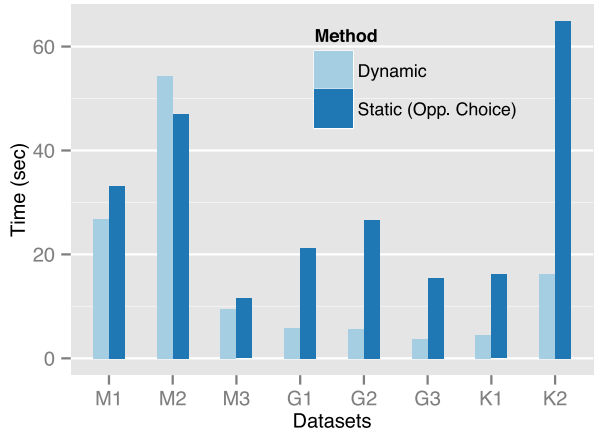
these observations, we set a threshold of 0.05 to predict the scale-free structure of the underlying graph topology and execute a BFS iteration for such cases.

To measure the relative improvement obtained by running BFS iteration based on the prediction, we compare the runtime of this dynamic approach against our implementation that does not compute K-S statistics and is hard-coded to make the opposite choice, i.e., executing BFS iteration only for the graphs M1-M4, G3. This experiment, using 2,025 processor cores, measures whether the prediction is correct and if correct, how much performance benefit do we gain against the opposite choice. As illustrated in Fig. 7a, we see positive speedups for all the graphs except M2. We see more than 3 × performance gains for all the small world graphs as well as G3. For M1 and M3, we gained approximately 25 percent improvement in the runtime. This experiment confirms that using BFS to identify and exclude the largest component is much more effective for small world graphs while running BFS on large diameter graph such as G3 is not optimal. Moreover, using the degree distribution statistics, we can choose an optimal strategy for most of the graphs.

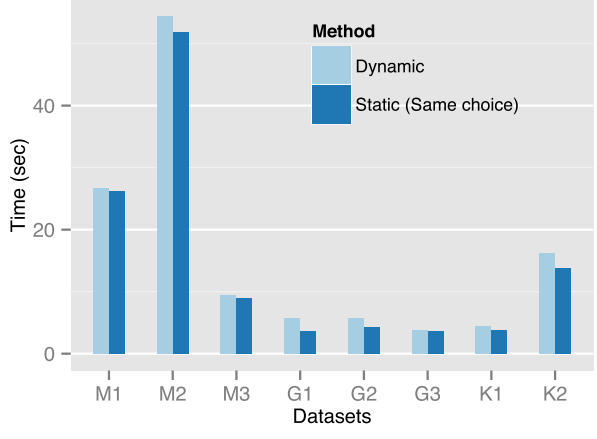
Note that computing the degree distribution of a graph and measuring K-S statistics adds an extra overhead to the overall runtime of the algorithm. We evaluate the additional overhead incurred by comparing the dynamic approach against the implementation which is hard-coded to make the same choice, i.e., execute BFS iteration only for G1-G2, K1-K2 (Fig. 7b) using 2,025 processor cores. The overhead varies from 60 percent for G1 to only 2 percent for M1. In general, we find this overhead to be relatively high for small-world graphs. Fitting the degree distribution curve against a power-law model is a sequential routine in our implementation, and it takes us about a second for scale-free graphs because they tend to have long-tailed degree distributions. We leave parallelizing and optimizing this routine as future work. Overall, we observe that the performance gains significantly outweigh the cost of computing the degree distribution and K-S test.

5.3 Strong Scaling

With the optimizations in place, we conducted strong scaling experiments on our algorithm. In this experiment, we use 256-4,096 cores for G1-G3, K1, and M1-M3. Results for M4, the largest graph are discussed separately as we could not process it with fewer than 4,096 cores. Graph K2 is



(a) Comparison of runtime of our algorithm making decision to run BFS dynamically versus the implementation which is hard-coded to make the opposite decision for all the graphs, using 2025 processor cores. For all the graphs but M2, our decision is correct (Table 2).



(b) Comparison of runtime of our algorithm making decision to run BFS dynamically versus the implementation hard-coded to make the same decision for all the graphs, using 2025 processor cores. The difference in the timings is the overhead of our prediction strategy.

Fig. 7. Evaluation of prediction heuristics in our algorithm.

ignored for this experiment because it has same topology as K1. In Fig. 8, we show the runtimes as well as speedups achieved by our algorithm. Most of these graphs cannot fit in the memory of a single node, therefore speedups are measured relative to the runtime on 256 cores. Ideal relative speedup on 4,096 cores is 16. We achieve maximum speedup of more than 8 × for the metagenomic graphs M1 and M2 and close to 6 × speedup for small world graphs G1, G2 and K1. G3 shows limited scalability due to its much smaller size relative to other graphs. We are able to compute connectivity for our largest graph M4 in 215 seconds using 32,761 cores (Table 3).

In Section 3.1.3 we discussed how each iteration of our parallel SV algorithm uses parallel sorting to update the partition ids of the edges. As a majority of time of this algorithm is spent in performing sorting, we also execute a micro benchmark that sorts 2 billion randomly generated 64 bit integers using 256 and 4,096 cores. Interestingly, we achieve speedup of 8.06 using our sample sorting method which is close to our scalability for M1 and M2. We anticipate that implementing more advanced

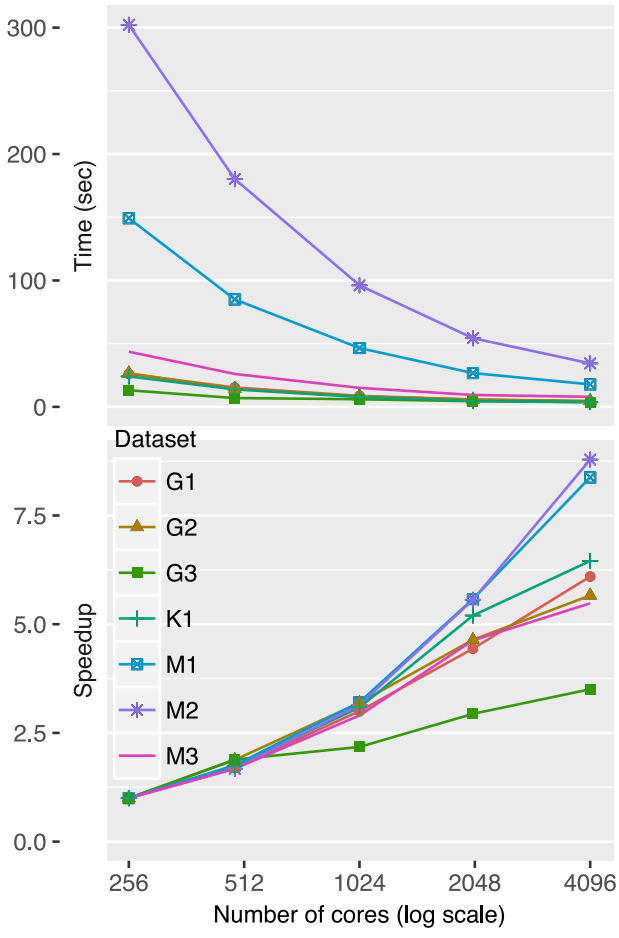


Fig. 8. Strong scalability results of our algorithm on different graphs using 4,096 cores. Speedups are computed relative to the runtime on 256 cores.

sorting algorithms [38] may further improve the efficiency of our parallel SV algorithm.

5.4 Performance Anatomy

We also report the percentage of total execution time on 2,025 cores that are attributable to each stage of our algorithm (Fig. 9). This figure is noteworthy especially for the graphs for which our algorithm chooses to execute BFS. For G1, G2, K1 and K2, more than 50 percent of the total percentage of time is devoted to predicting the graph structure and relabeling the vertices before running the parallel BFS and SV algorithm. This figure is not meant to convey the true overhead due to the relabel and prediction operations individually, as the time for relabeling is reduced after we sort the edges during the prediction stage (Section 3.2). Further, we measure the percentage time spent in the sorting operations in our parallel-SV algorithm for the graphs M1, M2, M3 and G3. As we expected, this measure is high and ranges from 91-94 percent for all the four graphs.

TABLE 3
Timings for the Largest Graph M4 with Increasing Processor Cores

Cores	8,281	16,384	32,761
Time for M4 (sec)	429.89	291.19	214.56

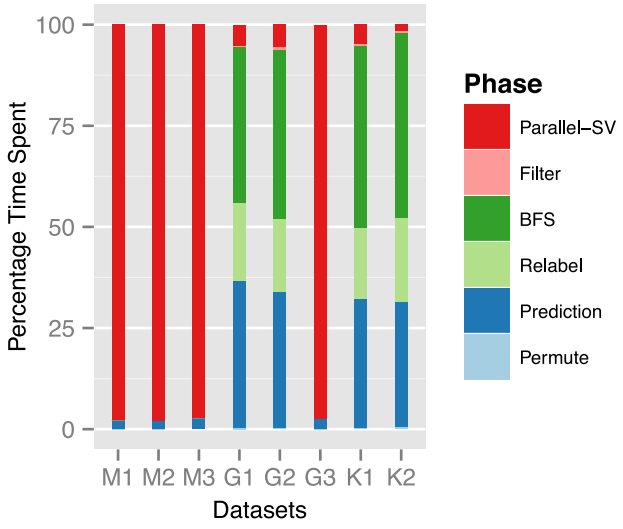


Fig. 9. Percentage time spend in different stages by the algorithm for different graphs using 2,025 cores. BFS is executed only for graphs G1, G2, K1 and K2.

5.5 Comparison with Previous Work

We achieve notable speedups when the performance of our algorithm is compared against the state-of-the-art Multistep algorithm [23] using 2,025 cores. As before, we begin counting the time once the graph edge list is read into the memory in both cases. We ran the Multistep algorithm with one process per physical core as we observed better performance doing so than using hybrid MPI-OpenMP mode. Also, because the Multistep method expects the vertex ids to be in the range 0 to $|V|-1$, we inserted our vertex relabeling routine in their implementation in order to run the software. Fig. 10 shows the comparison of our approach against the MultiStep method. We see >1 speedups for our method in all the graphs except G1. The speedup achieved ranges from 1.1 $\times$ for K2 to 24.5 $\times$ for G3. The speedup roughly correlates with the diameter of the graphs. The improvements achieved for the graphs M1, M2, M3 and G1 can be attributed to two shortcomings in the Multistep approach: 1) It executes BFS for computing the first component in all the graphs. BFS attains limited parallelism for large diameter graphs due to small frontier sizes. 2) It uses the label propagation technique to compute other components which in the worst case can take as many iterations as the diameter of the graph to reach the solution.

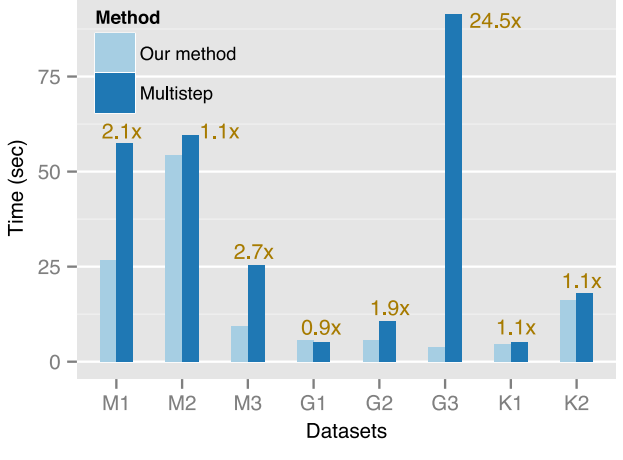


Fig. 10. Performance comparison of our algorithm against the Multistep method [23] using multiple graphs with 2,025 cores.

TABLE 4
Performance Comparison Against Rem's Sequential Connectivity Algorithm [40], [41] using 1,024 Cores

Dataset	Fastest Seq. Time (s)	Speedup		
		p = 64	256	1,024
Kronecker (25)	228.8	10.1	34.3	100.6
M3	406.2	2.5	9.3	27.0
G3	45.9	0.9	3.5	7.6

We could not compare our approach against the distributed-memory graph contraction algorithm [21] proposed to solve the connectivity problem, as the implementation is not open-source. Based on their experiment description, the graph contraction algorithm showed strong scalability only till 32 cores. Other distributed graph frameworks such as GraphX [17], and FlashGraph [39] based on in-memory Apache Spark and external-memory framework, respectively, can compute the connectivity of large-scale graphs as well. Slota et al. [23] show that their Multistep algorithm achieves superior performance against both of these methods. Because our algorithm performs better than Multistep, we skip a direct comparison against GraphX and FlashGraph.

5.6 Comparison with Sequential Implementation

We examine the performance of our algorithm against the best known sequential implementation for computing connectivity, for graph instances which can fit in the single node memory (64 GB)-these are relatively small. Previous works [13], [40] have shown that the Rem's method [41] based on the union-find approach achieves the best sequential performance. The sequential implementation we use in this algorithm was obtained from the authors of [13]. Again, because the disjoint-set structure used in the algorithm requires the vertices to be numbered from 0 to $n-1$, we placed our relabeling routine in the implementation. This experiment uses graphs M3 and G3, as all the other graphs require more than 64 GB memory. We also add a Kronecker graph of scale 25 ($m = 537$ M, $n = 17$ M) to include a short diameter graph instance. Results of this experiment are shown in Table 4. For these three graphs, our algorithm selects BFS iteration for Kronecker graph only. For the Kronecker graph, our algorithm achieves a 100 $\times$ speedup using 1,024 cores. For the other graphs, M3 and G3, where the SV algorithm is selected, the speedup decreases with respect to the sequential algorithm-which is partially due to the fact that the algorithm is not work optimal.

5.7 Comparison with Shared-Memory Implementations

The objective of the following comparative discussion between the distributed and shared-memory algorithms is not only to discuss the performance difference where shared-memory implementations tend to get good scaling per core, rather it is to highlight some of the constraints that shared-memory implementations have in contrast to their distributed counterparts.

Shared-memory parallel methods [13], [14] exhibit good speedups over the best sequential implementation. It is

therefore of no surprise to us that these algorithms can outperform our algorithm, especially for small to mid-range graphs. However, there are numerous problem scales that these shared memory algorithms cannot cope with due to the size of the graph; whereas our algorithm can easily deal with such networks. Our parallel algorithm utilizes bulk synchronous communication instead of the fast asynchronous communication found in shared-memory frameworks. While such communications are inherently slower, they do enable processing larger networks. Consider the largest network analyzed in this paper (Table 1): metagenomic graph M4 which has 53.6 billion edges and an equal number of vertices. Processing this graph in memory requires at least the following amount of memory: $2 \cdot (|V| + |E|) \times 8$ bytes. This assumes that the graph requires $|V|$ elements for the vertices and $2 \cdot |E|$ elements for the edges.[3] Also, $|V|$ integers are required for tracking the connected component labels. Given the size of the graph, 4 byte integers are not large enough to store all the unique keys and as such this requires using 8 byte integers. For the M4 network, a total of 1.7 TB DRAM is needed. As the sequencing cost continues to decline much faster than Moore's law [42], we envision the need to analyze even larger metagenomic graphs that require even more memory, in the near future. The problems of optimizing distributed-memory parallel algorithms while trying to attain peak performance continues to be an important challenge and one that deserves additional attention, especially the ability to reduce the overhead of communication.

Overall, we see that our proposed algorithm and the optimizations help us improve the state-of-the-art for distributed-memory parallel solution to the graph connectivity problem. Simple and fast heuristics to detect the graph structure enables our algorithm to choose the appropriate method dynamically for computing connectivity. This approach enabled us to compute connectivity for a graph with more than 50 billion edges and 300 million components in less than 4 minutes. The speedup we achieve over the state-of-the-art algorithm ranges from $1.1\times$ to $24.5\times$.

6 Conclusion

In this work, we presented an efficient distributed memory algorithm for parallel connectivity, based on the Shiloach-Vishkin PRAM algorithm. We proposed an edge-based adaptation of this classic algorithm and optimizations to improve its practical efficiency in distributed systems. Our algorithm is capable of finding connected components in large undirected graphs. We show that a dynamic approach that analyzes the graph and selectively uses the parallel BFS and SV algorithms achieves better performance than a static approach using one or both of these two methods. The dynamic approach prefers BFS execution only for a large short-diameter graph component.

Our method is efficient as well as generic, as demonstrated by the strong scalability of the algorithm on a variety of graph types. We also observed better performance when compared to a recent state-of-the-art algorithm. The measured speedup is significant, particularly in the case of large diameter graphs.

3. Recall that these are un-directed edges and it is customary in CSR, Compressed Sparse Row, format to store both directions of the edge.

Acknowledgments

We thank George Slota for sharing the implementation of Multistep method and helping us reproduce previous results. We also thank Aydin Buluç for making the Combinatorial BLAS library publicly accessible. This research was supported in part by the National Science Foundation under IIS-1416259. The cluster used for preliminary experiments in this work was supported by the National Science Foundation under CNS-1229081.

References

[1] A. Buluç and K. Madduri, "Parallel breadth-first search on distributed memory systems," in *Proc. Int. Conf. High Performance Comput. Netw. Storage Anal.*, 2011, Art. no. 65.

[2] S. Beamer, A. Buluc, K. Asanovic, and D. Patterson, "Distributed memory breadth-first search revisited: Enabling bottom-up search," in *Proc. IEEE 27th Int. Parallel Distrib. Process. Symp. Workshops PhD Forum*, 2013, pp. 1618–1627.

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

[4] P. E. Compeau, P. A. Pevzner, and G. Tesler, "How to apply de Bruijn graphs to genome assembly," *Nature Biotechnology*, vol. 29, no. 11, pp. 987–991, 2011.

[5] A. C. Howe, J. K. Jansson, S. A. Malfatti, S. G. Tringe, J. M. Tiedje, and C. T. Brown, "Tackling soil diversity with the assembly of large, complex metagenomes," *Proc. Nat. Academy Sci. United States America*, vol. 111, no. 13, pp. 4904–4909, 2014.

[6] R. C. Murphy, K. B. Wheeler, B. W. Barrett, and J. A. Ang, "Introducing the graph 500," *Cray User's Group*, 2010.

[7] D. S. Hirschberg, A. K. Chandra, and D. V. Sarwate, "Computing connected components on parallel computers," *Commun. ACM*, vol. 22, no. 8, pp. 461–464, 1979.

[8] A. Krishnamurthy, S. S. Lumetta, D. E. Culler, and K. Yelick, "Connected components on distributed memory machines," in *Proc. Parallel Algorithms: 3rd DIMACS Implementation Challenge*, 1994, pp. 1–21.

[9] L. Buš and P. Tvrdík, "A parallel algorithm for connected components on distributed memory machines," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface*. Berlin, Germany: Springer, 2001, pp. 280–287.

[10] S. Goddard, S. Kumar, and J. F. Prins, "Connected components algorithms for mesh-connected parallel computers," in *Proc. Parallel Algorithms: 3rd DIMACS Implementation Challenge*, 1994, vol. 30, pp. 43–58.

[11] D. Gregor and A. Lumsdaine, "The parallel BGL: A generic library for distributed graph computations," *Parallel Object-Oriented Sci. Comput.*, vol. 2, pp. 1–18, 2005.

[12] D. A. Bader and G. Cong, "A fast, parallel spanning tree algorithm for symmetric multiprocessors," in *Proc. 18th Int. Parallel Distrib. Process. Symp.*, 2004, Art. no. 38.

[13] M. M. A. Patwary, P. Refsnes, and F. Manne, "Multi-core spanning forest algorithms using the disjoint-set data structure," in *Proc. IEEE 26th Int. Parallel Distrib. Process. Symp.*, 2012, pp. 827–835.

[14] J. Shun, L. Dhulipala, and G. Blelloch, "A simple and practical linear-work parallel algorithm for connectivity," in *Proc. 26th ACM Symp. Parallelism Algorithms Archit.*, 2014, pp. 143–153.

[15] G. Cong and P. Muzio, "Fast parallel connected components algorithms on GPUs," in *Euro-Par 2014: Parallel Processing Workshops*. Berlin, Germany: Springer, 2014, pp. 153–164.

[16] K. Ueno and T. Suzumura, "Highly scalable graph search for the Graph500 benchmark," in *Proc. 21st Int. Symp. High-Performance Parallel Distrib. Comput.*, 2012, pp. 149–160.

[17] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica, "GraphX: Graph processing in a distributed dataflow framework," in *Proc. 11th USENIX Symp. Operating Syst. Des. Implementation*, 2014, pp. 599–613.

[18] R. Chen, J. Shi, Y. Chen, and H. Chen, "PowerLyra: Differentiated graph computation and partitioning on skewed graphs," in *Pro. 10th Eur. Conf. Comput. Syst.*, 2015, Art. no. 1.

[19] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "PowerGraph: Distributed graph-parallel computation on natural graphs," in *Proc. 10th USENIX Symp. Operating Syst. Des. Implementation*, 2012, pp. 17–30.

[20] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein, "Distributed GraphLab: A framework for machine learning and data mining in the cloud," *Proc. VLDB Endowment*, vol. 5, no. 8, pp. 716–727, 2012.
[21] J. Iverson, C. Kamath, and G. Karypis, "Evaluation of connected-component labeling algorithms for distributed-memory systems," *Parallel Comput.*, vol. 44, pp. 53–68, 2015.
[22] G. M. Slota, S. Rajamanickam, and K. Madduri, "BFS and coloring-based parallel algorithms for strongly connected components and related problems," in *Proc. IEEE 28th Int. Parallel Distrib. Process. Symp.*, 2014, pp. 550–559.
[23] G. M. Slota, S. Rajamanickam, and K. Madduri, "A case study of complex graph analysis in distributed memory: Implementation and optimization," in *Proc. IEEE 30th Int. Parallel Distrib. Process. Symp.*, 2016, pp. 293–302.
[24] P. Flick, C. Jain, T. Pan, and S. Aluru, "A parallel connectivity algorithm for de Bruijn graphs in metagenomic applications," in *Proc. Int. Conf. High Performance Comput. Netw. Storage Anal.*, 2015, Art. no. 15.
[25] G. E. Blelloch, C. E. Leiserson, B. M. Maggs, C. G. Plaxton, S. J. Smith, and M. Zagha, "A comparison of sorting algorithms for the connection machine CM-2," in *Proc. 3rd Annu. ACM Symp. Parallel Algorithms Archit.*, 1991, pp. 3–16.
[26] D. Easley and J. Kleinberg, *Networks, Crowds, and Markets: Reasoning About a Highly Connected World*. Cambridge, U.K.: Cambridge Univ. Press, 2010.
[27] A. Buluç and J. R. Gilbert, "The combinatorial BLAS: Design, implementation, and applications," *Int. J. High Performance Comput. Appl.*, vol. 25, pp. 496–509, 2011.
[28] A.-L. Barabási, R. Albert, and H. Jeong, "Scale-free characteristics of random networks: The topology of the world-wide Web," *Physica A: Statist. Mech. Appl.*, vol. 281, no. 1, pp. 69–77, 2000.
[29] A. Clauset, C. R. Shalizi, and M. E. Newman, "Power-law distributions in empirical data," *SIAM Rev.*, vol. 51, no. 4, pp. 661–703, 2009.
[30] M. Cha, H. Haddadi, F. Benevenuto, and P. K. Gummadi, "Measuring user influence in Twitter: The million follower fallacy," in *Proc. 4th Int. AAAI Conf. Weblogs Social Media*, 2010, vol. 10, no. 10–17, Art. no. 30.
[31] T. A. Davis and Y. Hu, "The University of Florida sparse matrix collection," *ACM Trans. Math. Softw.*, vol. 38, no. 1, 2011, Art. no. 1.
[32] J. Shun, "An evaluation of parallel eccentricity estimation algorithms on undirected real-world graphs," in *Proc. 21th ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining*, 2015, pp. 1095–1104.
[33] A. Gordon and G. Hannon, "Fastx toolkit," *Comput. Program Distrib. Author*, 2010. [Online]. Available: http://hannonlab.cshl.edu/fastx_toolkit/index. html, Accessed on: 2015–2016.
[34] T. Pan, P. Flick, C. Jain, Y. Liu, and S. Aluru, "Kmerind: A flexible parallel library for K-mer indexing of biological sequences on distributed memory systems," in *Proc. 7th ACM Int. Conf. Bioinf. Comput. Biology Health Inf.*, 2016, pp. 422–433.
[35] L. M. Rodriguez-R and K. T. Konstantinidis, "Estimating coverage in metagenomic data sets and why it matters," *ISME J.*, vol. 8, no. 11, pp. 2349–2351, 2014.
[36] B. Jenkins, "Hash functions," *Dr Dobbs J.*, vol. 22, no. 9, 1997, Art. no. 107.
[37] T. Nepusz, "Fitting power-law distributions to empirical data," 2016. [Online]. Available: https://github.com/ntamas/plfit
[38] E. Solomonik and L. Kale, "Highly scalable parallel sorting," in *Proc. IEEE Int. Symp. Parallel Distrib. Process.*, Apr. 2010, pp. 1–12.
[39] D. Zheng, D. Mhembere, R. Burns, J. Vogelstein, C. E. Priebe, and A. S. Szalay, "FlashGraph: Processing billion-node graphs on an array of commodity SSDs," in *Proc. 13th USENIX Conf. File Storage Technol.*, 2015, pp. 45–58.
[40] M. M. A. Patwary, J. Blair, and F. Manne, "Experiments on union-find algorithms for the disjoint-set data structure," in *Experimental Algorithms*. Berlin, Germany: Springer, 2010, pp. 411–423.
[41] E. W. Dijkstra, *A Discipline of Programming*, vol. 1. Englewood Cliffs, NJ, USA: Prentice-Hall, 1976.
[42] P. Muir, et al., "The real cost of sequencing: Scaling computation to keep pace with data generation," *Genome Biology*, vol. 17, no. 1, 2016, Art. no. 1.

Chirag Jain received the BTech degree in computer science from the Indian Institute of Technology Delhi. He is currently working toward the PhD degree in the School of Computational Science and Engineering, Georgia Institute of Technology, Atlanta. His research interests include bioinformatics, combinatorial algorithms, and high-performance computing.

Patrick Flick received the BSc and Msc degrees in computer science from the Karlsruhe Institute of Technology, Germany. Currently, he is working toward the PhD degree in computational science and engineering in the Georgia Institute of Technology, Atlanta. His research interests include high performance computing, string algorithms, and graph algorithms.

Tony Pan received the ScB degree in biophysics from Brown University and the MS degree in computer science from Rensselaer Polytechnic Institute. Previously, he held positions with General Electric, The Ohio State University, and Emory University. Currently, he is working toward the PhD degree in computational science and engineering in the Georgia Institute of Technology. His research interests include high performance computing, distributed information systems, bioinformatics, and biomedical and imaging informatics.

Oded Green received the BSc degree in computer engineering, the MSc degree in electrical engineering both from the Technion, Israel Institute of Technology, and the PhD degree from the School of Computational Science and Engineering, Georgia Institute of Technology. He is a research scientist in the School of Computational Science and Engineering, Georgia Institute of Technology. His research focuses on improving performance and increasing scalability for large-scale data analytics using a wide range of high performance computing platforms.

Srinivas Aluru is a professor in the School of Computational Science and Engineering, Georgia Institute of Technology. He co-directs the Georgia Tech Interdisciplinary Research Institute in Data Engineering and Science (IDEaS), and co-leads the NSF South Big Data Regional Innovation Hub which serves 16 Southern States in the U.S. and Washington D.C. Earlier, he held faculty positions with Iowa State University, Indian Institute of Technology Bombay, New Mexico State University, and Syracuse University. His conducts research in high performance computing, bioinformatics and systems biology, combinatorial scientific computing, and applied algorithms. He is currently serving as the chair of the ACM Special Interest Group on Bioinformatics, Computational Biology and Biomedical Informatics (SIGBIO). He received the the NSF Career award, IBM faculty award, Swarnajayanti Fellowship from the Government of India, and the Outstanding Senior Faculty Research award and the Deans award for faculty excellence at Georgia Tech. He received the IEEE Computer Society meritorious service award. He is a fellow of the AAAS and the IEEE.