© 2016 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.

Computing Connected Components with Linear Communication Cost in Pregel-like Systems

Xing Feng[†], Lijun Chang[†], Xuemin Lin[†], Lu Qin[‡], Wenjie Zhang[†]
[†]University of New South Wales, Australia
[‡]University of Technology Sydney, Australia
{xingfeng,ljchang,lxue,zhangw}@cse.unsw.edu.au, lu.qin@uts.edu.au

***Abstract*—The paper studies two fundamental problems in graph analytics: computing Connected Components (CCs) and computing BiConnected Components (BCCs) of a graph. With the recent advent of Big Data, developing efficient distributed algorithms for computing CCs and BCCs of a big graph has received increasing interests. As with the existing research efforts, in this paper we focus on the *Pregel* programming model, while the techniques may be extended to other programming models including *MapReduce* and *Spark*.**

The state-of-the-art techniques for computing CCs and BCCs in Pregel incur $O(m \times \#\text{supersteps})$ total costs for both data communication and computation, where m is the number of edges in a graph and #supersteps is the number of supersteps. Since the network communication speed is usually much slower than the computation speed, communication costs are the dominant costs of the total running time in the existing techniques. In this paper, we propose a new paradigm based on graph decomposition to reduce the total communication costs from $O(m \times \#\text{supersteps})$ to $O(m)$, for both computing CCs and computing BCCs. Moreover, the total computation costs of our techniques are smaller than that of the existing techniques in practice, though theoretically they are almost the same. Comprehensive empirical studies demonstrate that our approaches can outperform the existing techniques by one order of magnitude regarding the total running time.

I. INTRODUCTION

A graph $G = (V, E)$ is usually used to model data and their complex relationships in many real applications; for example, in social networks, information networks, and communication networks. Computing Connected Components (CCs) and computing BiConnected Components (BCCs) of a graph are two fundamental operations in graph analytics and are of great importance [5], [13], [20], [23], [36]. Given an undirected graph G, a CC of G is a maximal subgraph that is *connected* (*i.e.*, any pair of vertices are connected by a path). A BCC of G is a maximal subgraph such that it remains connected after removing any single vertex. For example, the graph in Figure 1 has two CCs: the subgraphs induced by vertices $\{v_1, \ldots, v_9\}$ and by $\{v_{10}, v_{11}\}$, respectively; the left CC is further divided into two BCCs: the subgraphs induced by vertices $\{v_4, v_5, v_8\}$ and by $\{v_1, v_2, v_3, v_6, v_7, v_8, v_9\}$, respectively.

Applications. Computing CCs is a key building block in processing large graphs. For example, CCs are basic structures for computing graph fractal dimensions when analyzing very large-scale web graphs [13]. Computing CCs also plays an important role in community detection in massive graphs by serving as a preprocessing step [8]. Computing BCCs is very important for analyzing large graphs. For example, BCCs can be used in the measurement study of topology patterns of large-scale community networks to measure their resilience

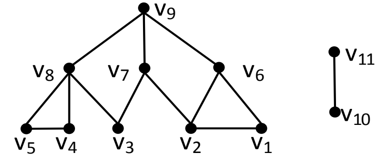


Fig. 1: A graph

to random failures [35]. Computing BCCs may also assist to identify the set of *articulation points* of a graph that typically belong to multiple BCCs. The articulation points are important to be identified in many applications; for example, in social networks [11], distributed networks [19], bioinformatics [20], and wireless sensor networks [33].

Distributed Computation. Driven by many recent applications involving large-scale graphs, there is a very strong demand to develop distributed computing techniques to process such large-scale graphs. For example, the topology of Facebook users is modeled as a graph with more than 1.4 billion vertices (*i.e.*, users) and 0.4 trillion edges (*i.e.*, relationships between users) in 2014,[1] and a snapshot of web graph in 2012 has 0.98 billion web pages and 42.6 billion hyperlinks.[2]

It is well known that regarding a single machine, CCs and BCCs of a graph can be computed by an in-memory algorithm in linear time (*i.e.*, $O(m)$) [5], [12] and by an external-memory algorithm with I/O cost $O(m \times \log n \times \log\log n)$ [18], where n and m are the number of vertices and the number of edges of the input graph, respectively. However, these techniques cannot be extended to the distributed computation due to their sequential computing nature. Thus, developing novel, efficient distributed techniques for computing CCs and BCCs of a large-scale graph has received increasing interests recently (*e.g.*, [21], [23], [24], [36]). Most of the existing techniques are based on open-source implementations of the *Pregel* system [16], including *Giraph* [3], *GPS* [24], and *Pregel+* [36]. In this paper, for ease of a comparison to the existing techniques, we also present our techniques based on the Pregel system. Nevertheless, our techniques may be extended to other distributed systems, such as *MapReduce* [6], *Spark* [37], *GraphLab* [15], and *GraphX* [10].

Cost Estimation of Pregel Algorithms. Pregel is designed based on the Bulk Synchronous Parallel (BSP) model [34], with computation performed in a serial of supersteps. Denote the total number of supersteps of a Pregel algorithm as #supersteps. The cost of one superstep of a BSP (also Pregel) algorithm on *p workers* (a.k.a cores) is $(\max_{i=1}^{p} w_i + \max_{i=1}^{p} h_i \times g + l)$ [34], where w_i is the cost for local computation at

[1]http://newsroom.fb.com/company-info/
[2]http://law.di.unimi.it/datasets.php

Algorithm	Total data communication
hash-min	$O(m \times \#\text{supersteps})$
single-pivot	$O(m \times \#\text{supersteps})$
S-V	$O(m \times \#\text{supersteps})$
our approach	***O(m)***

(a) CC Computation Algorithms

Algorithm	Total data communication
T-V(hash-min)	$O(m \times \#\text{supersteps})$
T-V(single-pivot)	$O(m \times \#\text{supersteps})$
T-V(S-V)	$O(m \times \#\text{supersteps})$
our approach	***O(m)***

(b) BCC Computation Algorithms

Fig. 2: Comparison of different Pregel algorithms for computing CCs and BCCs

worker i, h_i is the number of messages sent or received by worker i regarding data transmissions, g is the ability of a communication network to deliver data, and l is the cost of a barrier synchronization; here, g and l are system-dependent parameters. Thus, the total running time of a Pregel algorithm is expected to be determined by the total data communication cost (H), total computation cost (W), and #supersteps (*i.e.*, $\frac{W}{p} + \frac{H}{p} \times g + l \times \#\text{supersteps}$).

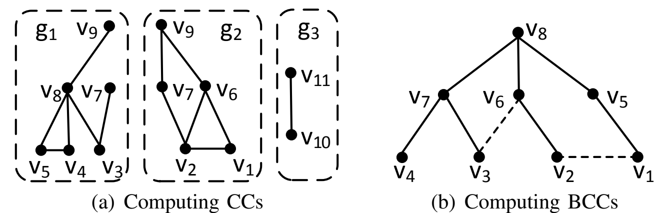
Existing Approaches. There are three existing algorithms for computing CCs in Pregel: hash-min, single-pivot, and S-V. The total costs for both data communication and computation of the three algorithms are $O(m \times \#\text{supersteps})$. hash-min [14], [23] and single-pivot [24] adopt the same strategy by coloring vertices such that all vertices in a CC end up with the same color. Each vertex is initialized with a different color, and then in each superstep, each vertex resets its color to be the smallest one among its own color and its neighbors' colors. While being initially developed against MapReduce, hash-min is adopted in [24] as a baseline algorithm to evaluate single-pivot in Pregel. Here, single-pivot proposes a heuristic to speed up the computation; it computes the first CC by conducting BFS starting from a randomly selected vertex while other CCs are computed by a follow-up hash-min.[3] The #supersteps of both hash-min and single-pivot are $O(\delta)$, where δ is the largest diameter among CCs.

The third approach, S-V [36], significantly extends the PRAM based algorithm in [27]. The main idea is to use a star to span all vertices of a CC by the two phases below. Initially, it constructs a rooted forest by making each vertex u point to its neighbor vertex v as its parent where $u.id < v.id$ and $v.id$ is maximized. Then, the first phase, *shortcutting*, is to connect every vertex u to its grand parent by an edge to replace the edge from u to its current parent. The second phase, *hooking*, is to attach the root of a star to a vertex v in another tree if there exists a vertex u in the star with v as its neighbor and $u.id < v.id$ — choose the vertex v with the largest id if there are several such vertices. S-V iteratively alternates the two phases till each tree is a star and no merges exist.[4] The #supersteps of S-V is $O(\log n)$.

For computing BCCs in Pregel, Yan et al. [36] presented an algorithm, T-V, to extend the PRAM algorithm in [31] to convert the problem of computing BCCs to computing CCs, and then apply the above techniques (*i.e.*, hash-min, single-pivot, or S-V) to compute CCs. The converting process requires an additional $O(\log n)$ #supersteps [36]. The total data communication costs and computation costs of T-V algorithms are $O(m \times \#\text{supersteps})$.

Our Approaches. Since the communication speed is much slower than the computation speed, communication costs are the dominant costs of the total running time in the existing techniques (see Figures 10(b) and 13(b) in Section V). In this paper, we aim to reduce the total data communication costs from $O(m \times \#\text{supersteps})$ of the existing techniques to $O(m)$ as depicted in Figure 2. Although theoretically we retain the other costs, our experiments show that the other costs are also reduced in practice. Essentially, we develop a new graph decomposition based paradigm to traverse the input graph only by a constant time, in contrast to traversing the entire graph every superstep of the existing techniques.

Computing CCs. We conduct graph decomposition by growing BFS trees from randomly selected seed vertices. Clearly, a pair of separately grown BFSs belong to one CC if they share a common vertex. To ensure that each edge is visited at most twice in our algorithm, *the unvisited neighbors of a common vertex in more than one BFSs are only extended once to the next level for a further extension in one BFS*. For example, the graph in Figure 1 is decomposed into three subgraphs as shown in Figure 3(a), which are obtained by conducting BFS searches from v_4, v_1, and v_{10}, respectively. g_1 and g_2 firstly overlap on $\{v_9\}$; assuming v_9 is extended in g_2, then (v_9, v_7) will not be visited in g_1. Finally, g_1 and g_2 are *combined* together to form a CC of G. At each superstep, we proceed BFS one level further and *randomly select new seed vertices whose cardinality increases exponentially along with supersteps*. Since the number of BFSs (*i.e.*, seed vertices) is only a small fraction of the number of vertices in a graph (*e.g.*, $\leq 0.4\%$, see Figure 11(d) in Section V), the last step (*i.e.*, combining) can be achieved through aggregator of Pregel systems at the master worker. We are able to show that the total data communication cost and #supersteps of our approach are $O(m)$ and $O(\log n)$, respectively; moreover, if the input graph is connected (or has a giant CC), then the #supersteps of our approach becomes $O(\min\{\delta, \log n\})$ (or with high probability).



(a) Computing CCs (b) Computing BCCs

Fig. 3: Our approaches

Computing BCCs. Suppose the input graph G is connected, the central idea is as follows. We first construct a BFS tree T of G. Then, based on T, G may be treated as being decomposed into $(m - n + 1)$ basic cycles $\{C_1, \ldots, C_{m-n+1}\}$ each of which is induced by a non-tree edge; that is, a basic cycle consists of a non-tree edge (u, v) and the paths from u and v to their nearest common ancestor in T. We can prove that *each basic cycle is biconnected and two biconnected subgraphs belong to the same* BCC *if they share a common edge*. Therefore, our algorithm runs iteratively from the bottom to the root in T to identify the vertices at the current layer and their neighbors in the upper layer to be in the same BCC.

[3] The total data communication cost of single-pivot becomes $O(m)$ if the input graph contains one giant CC and other small CCs.

[4] Note that we confirmed with the authors of [36] through private communication that the data communication cost of S-V per superstep is $O(m)$, which is misspelled as $O(n)$ in [36].

For example, given the BFS tree depicted by solid edges in Figure 3(b), there are two basic cycles, $C_1 = (v_8, v_7, v_3, v_6, v_8)$ and $C_2 = (v_8, v_6, v_2, v_1, v_5, v_8)$. Thus, at the bottom layer, the algorithm identifies $\{v_1, v_2, v_5, v_6\}$, $\{v_3, v_6, v_7\}$, and $\{v_4, v_7\}$ to be in the same BCCs, respectively. Then, the algorithm moves to the second bottom layer, and identifies that $\{v_1, v_2, v_5, v_6\}$ and $\{v_3, v_6, v_7\}$ should be in the same BCC including v_8 due to the common edge (v_6, v_8). To speed up the computation, we propose a vertex labeling approach. The total data communication cost of our approach is $O(m)$, and #supersteps is $O(\log n + \delta)$.

Contributions. Our main contributions are as follows.

- We develop a new paradigm for computing CCs and BCCs to reduce the total data communication cost.
- We propose a graph decomposition based approach for computing CCs of a graph with total data communication cost $O(m)$ in $O(\log n)$ supersteps.
- We propose a vertex labeling approach for computing BCCs with total data communication cost $O(m)$ in $O(\log n + \delta)$ supersteps.

We conduct extensive performance studies on large graphs, and show that our approaches have significantly smaller communication volume than the existing approaches and are one order of magnitude faster than the existing techniques.

Organization. A brief overview of related works immediately follows. In Section II, we give preliminaries and our problem statement. The graph decomposition based paradigm and our approach for computing CCs are presented in Section III, while our vertex labeling approach for computing BCCs is illustrated in Section IV. Section V presents our performance studies, and Section VI finally concludes the paper. Proofs are omitted due to space limits and can be found in the full version [7].

Related Works. Related works are categorized as below.

1) Computing CC*s.* Computing CCs by an in-memory algorithm over a single machine can be achieved in linear time regarding the input graph size by BFS or DFS [5]. PRAM algorithms were proposed in [1], [27]. Algorithms based on MapReduce include [4], [14], [21], [23], where the algorithm in [21] was independently developed from the Pregel version of S-V [36], and hash-min is used as a baseline to evaluate single-pivot in [24]. All these MapReduce and Pregel algorithms have total data communication costs $O(m\times$#supersteps). As stated earlier, we propose a new approach in Pregel to reduce the total data communication cost to $O(m)$.

2) Computing BCC*s.* Computing BCCs in the main memory of a single machine can be achieved in linear time based on DFS [12], or ear decomposition [26]. PRAM algorithms for computing BCCs were studied in [28], [31], and PRAM algorithm for testing graph biconnectivity was studied in [22]. The state-of-the-art algorithm T-V in Pregel [36] significantly extends the techniques in [31] to convert the problem of computing BCCs to computing CCs. In this paper, we develop a new approach with total data communication cost $O(m)$ instead of $O(m \times$ #supersteps) in [36].

3) Graph Decomposition. The paradigm of graph decomposition was studied in [2], [17], [29], [30] to decompose a graph into subgraphs with designated properties. It aims to minimize either the number of subgraphs, the maximum radius among subgraphs, or the number of cross partition/subgraph edges [2], [17], [29], [30]. While our approach is also based on the paradigm of graph decomposition, the existing techniques are irrelevant since we target at inherently different problems.

4) Other Distributed Graph Processing Systems. Besides the Pregel system [16] and its open-source implementations [24], [32], [36], other distributed graph processing systems include GraphLab [15], MapReduce [6], Spark [37], and GraphX [10]. Pregel and GraphLab are very similar to each other though GraphLab supports both synchronous and asynchronous models and does not allow graph mutations [15]. MapReduce [6] is a general purpose distributed data processing system, and has recently shown to be able to process graphs [21]. Spark [37] is an in-memory distributed data processing system which improves upon MapReduce by keeping data in main memory. GraphX [10] is a system built on Spark to bridge the gap between graph processing systems and general purpose data processing systems. While we will present our techniques based on the Pregel system, our techniques may be easily modified to the above distributed systems. This is because our techniques are based on the vertex-centric programming model that can be easily implemented in the other systems.

II. Preliminary

In this paper, we focus on *an unweighted undirected graph* $G = (V, E)$ [9], where V is the set of vertices and E is the set of edges. Denote the number of vertices, $|V|$, and the number of edges, $|E|$, in G by n and m, respectively. Each vertex $v \in V$ has a unique integer ID, denoted $v.id$. We denote an undirected edge between u and v by (u, v). Given a set $V' \subseteq V$ of vertices, the subgraph of G induced by V' is defined as $G[V'] = (V', \{(u, v) \in E \mid u, v \in V'\})$. In the following, for presentation simplicity we refer an unweighted undirected graph as a graph.

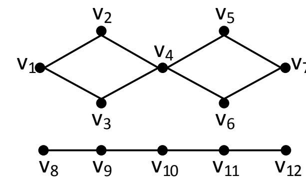


Fig. 4: An example graph

Definition 2.1: A **connected component** (CC) of a graph is a maximal subgraph in which any two vertices are connected to each other by a path. □

For example, for the graph in Figure 4, there are two connected components (CCs): the subgraphs induced by $\{v_1, v_2, \ldots, v_7\}$ (*i.e.*, the subgraph at the top), and by $\{v_8, v_9, \ldots, v_{12}\}$ (*i.e.*, the subgraph at the bottom), respectively.

Definition 2.2: A graph is **biconnected** if it is still connected after removing any *vertex* from it. A **biconnected component** (BCC) of a graph is a maximal biconnected subgraph. □

For example, the connected component in the top part of Figure 4 is not biconnected because removing v_4 will disconnect the subgraph. Instead, it consists of two biconnected components (BCCs): the subgraphs induced by $\{v_1, \ldots, v_4\}$ and by $\{v_4, \ldots, v_7\}$, respectively. In the following, we refer a BCC either by the set of vertices or by the set of edges in it. Note that, a vertex (*e.g.*, v_4) may belong to more than one BCCs, while each edge belongs to a unique BCC.

Problem Statement. Given a large-scale graph G, in this paper we study the problem of *distributed computing all connected components (*CC*s) of G* and the problem of *distributed computing all biconnected components (*BCC*s) of G.*

In this paper, for ease of a comparison to the existing techniques, we present our techniques based on the Pregel system which is introduced in below. Nevertheless, our techniques may be extended to other distributed systems, such as *MapReduce* [6], *Spark* [37], *GraphLab* [15], and *GraphX* [10].

A. The Pregel System

Pregel [16] is designed based on the *Bulk Synchronous Parallel (BSP)* model [34]. Initially, vertices of the input graph are distributed across a cluster of workers, where all adjacent edges of a vertex reside in the same worker. Then, computation tasks are performed in a serial of supersteps; let *#supersteps* denote the total number of supersteps in a Pregel algorithm.

In a superstep, each *active* vertex invokes a user-defined function, *compute()*. The *compute* function running on a vertex v, (1) performs computation based on v's current status and the messages v received from the previous superstep, (2) updates its status, (3) sends new messages to other vertices to be received in the next superstep, and (4) may (optionally) make v vote to halt. A halted vertex is reactivated if it receives messages. A Pregel program terminates when all vertices vote to halt and there is no message in transmit. Pregel is often regarded as a *vertex-centric* programming model since it performs computation for each vertex based on only the local information of the vertex itself and the messages it receives.

Combiner and Aggregator. Pregel also extends BSP in several ways. Firstly, Pregel supports combiner (*i.e.*, *combine()* function) to combine messages that are sent from vertices in one worker to the same vertex in another worker. Secondly, *aggregator* is also supported in Pregel. That is, in a superstep, every vertex can contribute some values to *aggregator* and a rule is specified to aggregate these values at the master worker; the result of aggregation is visible to all vertices in the next superstep. Thus, in Pregel, the master worker can act as a coordinator by conducting some computation for all workers. In this paper, we make use of the aggregator to design faster Pregel algorithms for computing CCs and BCCs.

III. Computing CCs

In order to reduce the communication cost of computing CCs, we develop a new paradigm in Section III-A, based on which we present a new approach in Section III-B while analyses are given in Section III-C.

A. A Graph Decomposition based Paradigm

Reducing Communication Cost. The existing approaches for computing CCs have total (data) communication costs and total computation costs $O(m \times \#\text{supersteps})$ (see Figure 2(a)). Since the communication speed is much slower than the computation speed, communication cost is the dominant cost of the total running time in the existing techniques (see Figures 10(b) and 13(b) in Section V), and thus the large communication costs of the existing techniques significantly degrade their performances. Motivated by this, in this paper, we aim to reduce the total communication cost of a Pregel algorithm.

One thing to notice is that the total communication cost of single-pivot becomes $O(m)$ (or with high probability) if the input graph is connected (or has one giant CC and other small CCs). For example, YH is such a graph in our experiments; the communication volume of single-pivot on YH is similar to our algorithm and is much smaller than hash-min and S-V (see Figure 10(c) in Section V). However, a Pregel algorithm has an inevitable computation cost of $O(n \times \#\text{supersteps})$ for checking at each superstep every vertex whether it is active. Consequently, the computation cost of single-pivot on YH, which has a large diameter, is very high (see Figure 10(b)) due to the large #supersteps (*i.e.*, δ); this results in high total running time (see Figure 10(a)). Therefore, #supersteps is also an important factor to be optimized for Pregel algorithms. In this paper, we retain the computation cost and #supersteps of our algorithm to match the best of existing algorithms.

Graph Decomposition based Paradigm. We develop a new graph decomposition based paradigm for computing CCs.

Definition 3.1: A **graph decomposition** of a graph $G = (V, E)$ is to decompose G into a set of subgraphs, $\{g_1, \ldots, g_i = (V_i, E_i), \ldots, g_l\}$, such that each subgraph g_i keeps a designated property (*e.g.*, be connected) and $\bigcup_{i=1}^{l} E_i = E$. □

Algorithm 1: CC-Overview

Input: A graph $G = (V, E)$ distributed across a set of workers
Output: The set of CCs of G

1 Distributed compute a graph decomposition of G;
2 Merge decomposed subgraphs into CCs;
3 **return** the set of CCs;

The framework is illustrated in Algorithm 1, which first decomposes an input graph into a set of connected subgraphs and then merges the decomposed subgraphs into CCs. During graph decomposition, we assign a unique color to all vertices in a decomposed subgraph which is connected. Since subgraphs sharing common vertices belong to the same CC, a vertex may receive different colors and the decomposed subgraphs having these colors belong to the same CC. We mark down, during graph decomposition, the subgraphs that should be merged, and dedicate the last superstep for the merge operation to obtain the correct CCs. Since the number of colors (*i.e.*, the number of decomposed subgraphs) usually is only a small fraction of the number of vertices in a graph (*e.g.*, $\leq 0.4\%$, see Figure 11(d)), the merge operation can be achieved through the aggregator of Pregel systems at the master worker.

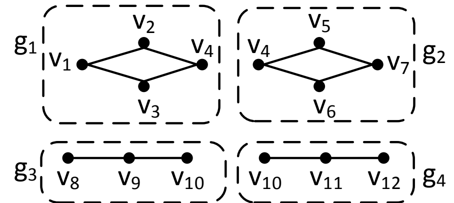


Fig. 5: Graph decomposition based paradigm

For example, assume the graph in Figure 4 is decomposed into four connected subgraphs $\{g_1, g_2, g_3, g_4\}$ as shown in Figure 5 and vertices in g_1 have color 1 and vertices in g_2 have color 7, then v_4 will receive colors 1 and 7, respectively, from subgraphs g_1 and g_2. Therefore, the subgraph having color 1 (*i.e.*, g_1) and the subgraph having color 7 (*i.e.*, g_2) belong to the same CC, and we merge g_1 and g_2 into a single CC. Similarly, we can obtain the other CC.

B. Our Algorithm

Following the paradigm in Algorithm 1, we propose a new approach for computing CCs, consisting of the following two

phases: *graph decomposition* and *subgraph merging*.

Graph Decomposition. We compute a graph decomposition by simultaneously conducting BFS searches starting from a set of seed vertices. When running a BFS, we label all visited vertices by the *color of the BFS*, which is the seed vertex id of the BFS. Thus, a vertex may receive multiple colors, one from each BFS visiting it; we store all the received colors of a vertex v into $v.cls$, and set v's color, $v.color$, to be the first received color. Once a vertex is assigned a color, it propagates the color to all its neighbors. Therefore, the unvisited neighbors of a vertex are only extended in one BFS because a vertex is assigned a color only once; this guarantees that each edge is visited at most twice during graph decomposition. When all edges of the graph have been visited (*i.e.*, all BFSs terminate), *each subgraph induced by vertices with the same color is a decomposed subgraph*; note that, here we say a vertex v have all colors it received (*i.e.*, all colors in $v.cls$).

Seed Vertex Selection. The largest diameter of the obtained subgraphs by graph decomposition, which is related to #supersteps, largely depends on the selection of seed vertices. For example, in Figure 4, assume the seed vertices are $\{v_1, v_7, v_8, v_{12}\}$, then we will obtain the graph decomposition in Figure 5; the largest diameter is 2. However, if the seed vertices are chosen as $\{v_1, v_2, v_8, v_9\}$, then the obtained subgraphs will be the subgraphs induced by $\{v_1, v_3, v_4\}$, $\{v_2, v_4, v_5, v_6, v_7\}$, $\{v_8, v_9\}$, $\{v_9, v_{10}, v_{11}, v_{12}\}$, respectively; the largest diameter will be 3.

We propose a randomized adaptive approach to seed vertex selection by iteratively selecting seed vertices and conducting BFSs. That is, in each superstep, we advance the existing BFSs by one level (*i.e.*, visit the neighbors of the currently visited vertices), and also start BFSs from the newly selected seed vertices. To make #supersteps bounded, we increase the number of seed vertices to be selected as the iteration proceeds; this is controlled by a parameter $\beta > 1$. Specifically, we randomly select $\beta^i - \beta^{i-1}$ new vertices to be potential seed vertices for superstep $i > 0$ (*i.e.*, 1 vertex for superstep 0). For each of the selected vertices, if it has already been visited by the existing BFSs, then we do nothing; otherwise, it is treated as a new seed vertex and we start a new BFS from it.

Algorithm 2: graph-decompose

```
Input: A graph G = (V, E), and a parameter β
1  Generate a random permutation P of the vertices in V;
2  for each vertex v ∈ V do
3      v.color ← nil; v.pos ← the position of v in P;
4  while either a vertex has no color or an edge is not visited do
5      for each vertex v ∈ V do
6          Let C be the set of colors v received in messages;
7          if v.color = nil then
8              if |C| > 0 then
9                  v.cls ← C;
10                 v.color ← the first color in C;
11             else if v.pos ≤ β^i then
12                 v.color ← vertex id of v;
13             Send v.color to v's neighbors if v.color ≠ nil;
14         else v.cls ← v.cls ∪ C;
```

Graph Decomposition Algorithm. The pseudocode of our graph decomposition algorithm is shown in Algorithm 2, denoted graph-decompose. We randomly permute all vertices in G by the approach in [25] (Line 1), such that the random vertex selection is achieved by sequentially selecting vertices according to the permutation order. For each vertex v, the position of v in the permutation, $v.pos$, and the color of v, $v.color$, are initialized at Line 3. Then, we iterate until every vertex has been assigned a color and all edges have been visited (Lines 5–14). In a superstep, we perform the following computations for each vertex $v \in V$. If v already has a color (*i.e.*, $v.color \neq nil$), we just add the set of colors v received in this superstep into $v.cls$ (Line 14). Otherwise, $v.color = nil$. In this case, if v receives colors in this superstep (*i.e.*, v is being visited by the existing BFSs), then we assign the first color v received to be its color (Lines 8–10); otherwise, if v is selected by the random selection process (*i.e.*, $v.pos \leq \beta^i$), then we start a new BFS from v (Lines 11–12). After v is assigned a color, we propagate its color to its neighbors (Line 13).

Subgraph Merging. In this phase, we merge the subgraphs that belong to the same CC into a single subgraph by merging colors, based on the facts that each subgraph is uniquely identified by a color and subgraphs with colors in $v.cls$ for a $v \in V$ belong to the same CC. To do so, we merge all received colors of a vertex into a single color. For example, if a vertex v receives two colors a and b (*i.e.*, $v.cls = \{a, b\}$), then a and b are merged into a single color. Note that, if another vertex u receives colors b and c, then b and c are also merged into a single color; this in turn merges a and c (*i.e.*, a, b, and c are merged into a single color).

Algorithm 3: merge-color Aggregator

```
   /* Master worker collects information                    */
1  for each vertex v ∈ V do
2      if |v.cls| > 1 then Send v.cls to the master worker;
   /* Master worker conducts the aggregation                */
3  Let C be the set of all colors in received messages;
4  Initialize a disjoint-set data structure for C;
5  for each set of received colors cls do
6      Let c be the first color in cls;
7      for each color c' ∈ cls do union(c, c');
8  for each color c ∈ C do find(c);
   /* Master worker sends aggregation to slaves             */
9  Broadcast the set of parents of colors in the disjoin-set data
   structure (i.e., {(c, parent(c)) | c ∈ C}) to all slave workers;
```

Merging color is achieved through the aggregator in Pregel systems, as shown in Algorithm 3. It first collects the sets of received colors of vertices to the master worker (Lines 1–2), then conducts the merging at the master worker (Lines 3–8), and finally sends the aggregated information back to slave workers (Line 9). To process these merging operations efficiently, we adopt the union-find algorithm based on the disjoint-set data structure [5]. We organize the colors into a set of trees by storing the parent parent(c) of each color c, such that the root of each tree is the representative color of all colors in the tree. Initially, each color corresponds to a singleton tree (*i.e.*, parent$(c) = c, \forall c \in C$) (Line 4). For each set of received colors, cls, we merge the first color c with every other color $c' \in cls$ (Lines 5–7); that is, we merge the tree containing c and the tree containing c'. Finally, we apply the find operation to all colors (Line 8), such that parent(c) stores the root/representative color of the tree containing c. Note that, union and find are two standard operations in the union-find algorithm [5].

Algorithm 4: update-color

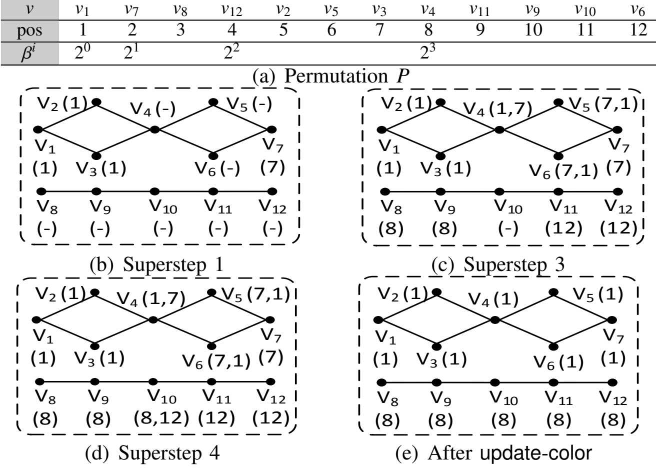
Input: The set of $(c, \mathsf{parent}(c))$ pairs received at slave workers

1 **for each** *vertex* $v \in V$ **do** $v.color \leftarrow \mathsf{parent}(v.color)$;

Now, we replace each color $v.color$ by its representative color $\mathsf{parent}(v.color)$. This guarantees that all vertices in the same CC are updated to have the same color. The pseudocode is given in Algorithm 4, denoted update-color.

v	v_1	v_7	v_8	v_{12}	v_2	v_5	v_3	v_4	v_{11}	v_9	v_{10}	v_6
pos	1	2	3	4	5	6	7	8	9	10	11	12
β^i	2^0	2^1		2^2				2^3				

(a) Permutation P



(b) Superstep 1 (c) Superstep 3 (d) Superstep 4 (e) After update-color

Fig. 6: Running example for computing CCs (*the numbers in brackets beside vertex v is v.cls, where the first color is v.color*)

Running Example. Consider the graph in Figure 4 with the random permutation P in Figure 6(a) and $\beta = 2$. At superstep 0, no vertex receives colors and v_1 is selected as a seed vertex; thus v_1 has color 1, and propagates its color to its neighbors. At superstep 1, both v_2 and v_3 receive color 1, and v_7 is selected as a new seed vertex; the result is shown in Figure 6(b). Here, we show *v.cls* in brackets, where the first color is *v.color*. The process continues. At superstep 3, v_4 has color 1 and also receives color 7; thus v_4 adds 7 to its set of received colors, as shown in Figure 6(c). Note that, in this superstep, neither v_4, v_5, nor v_6 propagate its newly received colors (*i.e.*, colors, 7, 1, and 1, respectively) to its neighbors.

At superstep 4, all edges have been visited by BFSs; thus, the graph decomposition ends and the result is shown in Figure 6(d). Vertices, v_4, v_5, v_6, and v_{10}, have received multiple colors; thus their sets of received colors are collected at the master worker for merging. That is, colors 1 and 7 are merged, and colors 8 and 12 are merged. Assume that the representative color of 1 and 7 is 1 and the representative color of 8 and 13 is 8, then after update-color the final colors of vertices are shown in Figure 6(e). We conclude that vertices $\{v_1, \ldots, v_7\}$ and $\{v_8, \ldots, v_{12}\}$ belong to two CCs, respectively.

C. Correctness and Complexity Analysis

We prove the correctness and give complexity analyses of our approach.

Correctness. For any color c, let V_c be the set of vertices with color c. Then, $V_c \cap V_{c'} = \emptyset, \forall c \neq c'$, because after update-color (Algorithm 4), every vertex has exactly one color. We prove the correctness of our approach by the following theorem.

Theorem 3.1: *For each assigned color c in Algorithm 4, the subgraph induced by V_c, $G[V_c]$, is a maximal connected subgraph (thus, it is a connected component).* □

Complexity Analyses. Now, we analyze the complexities of our approach regarding #supersteps, total communication cost, and total computation cost in the following.

Number of Supersteps. In the worst case, graph-decompose stops after at most $\log_\beta n$ (*i.e.*, $O(\log n)$) supersteps, because at that time all vertices would have been selected by the random seed vertex selection process. Moreover, if the input graph is connected (or has a giant CC), then graph-decompose terminates after at most $O(\delta)$ supersteps (or with high probability), because at that time all edges would have been visited by BFSs. In addition, merge-color and update-color take one superstep each. Thus, #supersteps is bounded by $O(\min\{\delta, \log n\})$ (or with high probability) if the input graph is connected (or has a giant CC), and it is bounded by $O(\log n)$ in the worst case.

Total Communication Cost. The total communication cost of our approach is $O(m)$, as follows. Firstly, graph-decompose traverses the input graph only once by visiting each undirected edge at most twice, and one message is generated for each visited edge. Secondly, the number of messages collected at the master worker in Algorithm 3 is at most $\sum_{v \in V} |v.cls| \leq 2 \times m$. Moreover, to reduce the communication cost, we also aggregate the messages locally at slave workers before sending to the master worker, by an algorithm similar to Algorithm 3. Thus, the communication cost of merge-color is $O(p \times \#\text{colors})$, where p is the number of workers and #colors is the total number of colors generated.

The #colors is the same as the number of selected seed vertices. To make it small, we implement a heuristic by choosing the vertex with the maximum degree as the first seed vertex at superstep 0. Let Δ be the maximum degree and d be the average degree. Then, the expected number n_i of seed vertices selected at superstep i is $n_i = (1 - \frac{\Delta \times d^{i-1}}{n}) \times (\beta^i - \beta^{i-1})$; intuitively, $\Delta \times d^{i-1}$ is the expected number of vertices already visited by the existing BFSs. Since Δ is large for real graphs due to the power-law graph model, n_i (thus #colors) usually is small; for example, in our experiments in Section V, #colors is only a small fraction (*i.e.*, $\leq 0.4\%$) of the number of vertices in a graph (see Figure 11(d)).

Total Computation Cost. The total computation cost of our approach is $O(m + n \times \#\text{supersteps})$. Firstly, $O(n \times \#\text{supersteps})$ is the inevitable cost due to the Pregel model. Besides, graph-decompose takes $O(m)$ time similar to the communication cost as discussed above, merge-color takes time linear to the size of received messages [5] (thus, bounded by $O(m)$), and update-color also takes linear time.

IV. Computing BCCs

In this section, we propose a new vertex labeling technique, based on graph decomposition, to reduce the total communication cost for computing BCCs from $O(m \times \#\text{supersteps})$ to $O(m)$. We first present the general idea in Section IV-A, and then illustrate our approach in Section IV-B, while analyses are given in Section IV-C. In the following, for ease of exposition we assume that the input graph is connected.

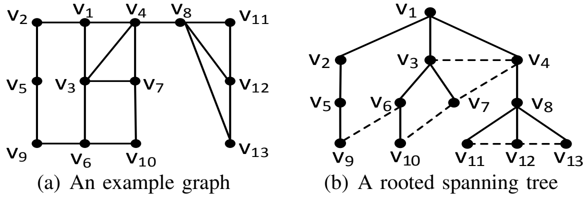
A. Vertex Labeling

Challenge. The main challenge of computing BCCs lies in the fact that a vertex may belong to several BCCs. Thus,

a similar idea to CC computation algorithms (*i.e.*, labeling each vertex by a color such that each CC corresponds to a subgraph induced by vertices with the same color) does not work for BCCs. Nevertheless, it is true that each edge can only participate in one BCC. Thus, the existing approaches compute BCCs by labeling edges instead [31], [36]. To do so, they construct an auxiliary graph G' for the input graph G by treating each edge in G as a vertex in G'; this reduces the problem of labeling edges in G to labeling vertices in G'. However, the above process incurs $O(m \times \#\text{supersteps})$ communication cost.

Intuition of Our Approach. We propose a new approach for computing BCCs with $O(m)$ communication cost by directly labeling vertices of the input graph. Our main idea is based on cycles in a graph.

Definition 4.1: A **cycle** consists of a sequence of vertices starting and ending at the same vertex, with each two consecutive vertices in it adjacent to each other in the graph. □



(a) An example graph (b) A rooted spanning tree

Fig. 7: An example graph and a spanning tree

For example, in Figure 7(a), (v_1, v_3, v_4, v_1) is a cycle. A cycle can also be represented by the sequence of edges formed by consecutive vertices. A cycle is *simple* if there is no repetition of vertices and edges, other than the repetition of the starting and ending vertices [9]. For example, in Figure 7(a), (v_1, v_3, v_1) and $(v_1, v_3, v_4, v_7, v_3, v_1)$ are not simple cycles, while (v_1, v_3, v_4, v_1) is a simple cycle. In the following, *we refer simple cycle as cycle for presentation simplicity*.

We have the lemma below for BCCs based on cycles.

Lemma 4.1: *Two edges belong to the same* BCC *if and only if there is a (simple) cycle containing both edges.* □

For example, in Figure 7(a), edges (v_2, v_5) and (v_3, v_6) belong to the same BCC because both appear in the cycle $(v_1, v_2, v_5, v_9, v_6, v_3, v_1)$; edges (v_4, v_7) and (v_8, v_{11}) belong to different BCCs because there is no cycle containing both edges.

Computing BCC*s.* Following Lemma 4.1, we can compute BCCs of a graph by enumerating cycles and building the relationship among edges, such that two edges have a relation if and only if they appear together in a cycle. Therefore, the transitive relationships between edges define the BCCs of the graph. However, there can be exponential number of cycles in a graph [5]. To make the above idea of computing BCCs by enumerating cycles work, we reduce the number of cycles to $m - n + 1$ by considering only basic cycles.

Definition 4.2: Given a spanning tree T of a graph G, we define a **basic cycle** for a non-tree edge $(u, v) \in G \backslash T$ as the cycle containing (u, v) and the path between u and v in T. □

For example, given the spanning tree in Figure 7(b) for the graph in Figure 7(a), the basic cycle corresponding to non-tree edge (v_6, v_9) is $(v_1, v_2, v_5, v_9, v_6, v_3, v_1)$; there are totally six basic cycles corresponding to non-tree edges $(v_6, v_9), (v_7, v_{10}), (v_3, v_4), (v_4, v_7), (v_{11}, v_{12}), (v_{12}, v_{13})$.

In the following, we assume there is a spanning tree T. Based on basic cycles, we have the following theorem.

Theorem 4.1: *Given any two edges e and e', they belong to the same* BCC *if and only if either 1) there is a basic cycle containing both e and e' or 2) there is a chain of basic cycles, $C_0, \ldots, C_l$, such that C_i and C_{i+1} overlap on edges for every $0 \le i < l$, and $e \in C_0$ and $e' \in C_l$.* □

Consider the graph in Figure 7(a) with the spanning tree depicted by solid edges in Figure 7(b), $e_1 = (v_{11}, v_{12})$ and $e_2 = (v_{12}, v_8)$ belong to the same BCC because they appear in the basic cycle $C_1 = (v_8, v_{11}, v_{12}, v_8)$. e_1 and $e_3 = (v_{12}, v_{13})$ belong to the same BCC, because there exists another basic cycle $C_2 = (v_8, v_{12}, v_{13}, v_8)$ such that $e_1 \in C_1$, $e_3 \in C_2$ and $C_1 \cap C_2 \neq \emptyset$.

Labeling Vertices. From Theorem 4.1, we only need to enumerate basic cycles, which is affordable. To tackle the challenge stated at the beginning of this subsection, we propose a vertex labeling technique based on the lemmas below.

Lemma 4.2: *Given a rooted spanning tree T of a graph G, for each vertex u in G, the set of non-tree edges associated with u and the tree edge $(u, p(u))$ belong to the same* BCC*, where $p(u)$ denotes the parent of u in T.* □

Lemma 4.3: *Given a rooted spanning tree T of a graph G, each* BCC *of G has a unique vertex that is closest to the root of T (denote it as the root of the* BCC*), and each vertex can be a non-root vertex in at most one* BCC. □

Therefore, we can label vertices by colors; each vertex has the color of the unique BCC in which the vertex is a non-root vertex. *The set of all vertices with the same color and their parents in the spanning tree corresponds to a* BCC. Alternatively, edges can infer their colors as follows: if it is a non-tree edge, then its color is the same as the color of either of its two end-points; otherwise, it is a tree edge and its color is the same as the child vertex of the edge. Consequently, each BCC of G corresponds to the set of edges with the same color.

Naive Vertex Labeling Algorithm. Given a rooted spanning tree of a graph G, a naive vertex labeling algorithm is to enumerate all basic cycles (*i.e.*, decompose G into a set of basic cycles). For each basic cycle C, we label all vertices in C, except the unique vertex that is closest to the root, with the same color. Note that, during labeling vertices in a basic cycle C, if there are vertices in C that have already been labeled (*i.e.*, their colors have been set by other basic cycles), then we collect all colors of such vertices (except the unique vertex that is closest to the root) and merge these colors, similar to merge-color and update-color in Section III. However, this is time-consuming considering the expensive cost of enumerating all basic cycles and of relabeling vertices. We present an efficient vertex labeling algorithm in the next subsection.

B. Our Algorithm

We propose an efficient approach for labelling all vertices in G by traversing the graph in a bottom-up fashion. We first make the input graph G a layered graph regarding a BFS tree.

Definition 4.3: Given a BFS tree T of a graph G, we make G a **layered graph** by assigning a level number to each vertex. The root vertex in T has a level number 0, and for every other vertex, its level number is one plus that of its parent in T. □

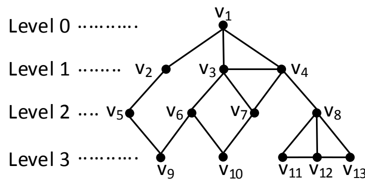


Fig. 8: A layered graph

For example, given the BFS tree in Figure 7(b) for the graph in Figure 7(a), the corresponding layered graph is shown in Figure 8. For each vertex in a layered graph, we categorize its neighbors into *upper-level neighbors* (*i.e.*, neighbors with smaller level numbers), *same-level neighbors*, and *lower-level neighbors* (*i.e.*, neighbors with larger level numbers). For a vertex at level i, all its upper-level neighbors are at level $(i-1)$, and all its lower-level neighbors are at level $(i+1)$. For example, for v_4 in Figure 8, its upper-level neighbor is v_1, its same-level neighbor is v_3, and its lower-level neighbors are v_7 and v_8. Note that, each vertex except the root vertex will have at least one upper-level neighbor including its parent vertex in T and possible other vertices corresponding to non-tree edges.

Given a layered graph, our algorithm labels vertices level-by-level in a bottom-up fashion based on the lemma below.

Lemma 4.4: *Given a layered graph, (*RULE-I*) each vertex has the same color as its same-level neighbors; (*RULE-II*) if a vertex has at least two upper-level neighbors, then the vertex and all its upper-level neighbors have the same color.* □

Consider the layered graph in Figure 8, v_3 and v_4 have the same color since they are same-level neighbors, v_9 have the same color as v_5 and v_6 since v_5 and v_6 are two upper-level neighbors of v_9. However, from Lemma 4.4, we do not know the relationship between colors of v_2 and v_3 since they are neither directly connected nor connected through a common lower-level neighbor. From Theorem 4.1, we know that v_2 and v_3 should have the same color. We define a merge operation below so that we can apply Lemma 4.4 to label all vertices.

Definition 4.4: Given a set S of vertices that are at the same level and have the same color, the **merge operation** is to merge all vertices in S into a single super-vertex. □

For example, in Figure 8, vertices v_5 and v_6 are at the same level and have been assigned the same color due to the common lower-level neighbor v_9. We merge them into a super-vertex, denoted $v_{5,6}$. Now, $v_{5,6}$ has two upper-level neighbors, v_2 and v_3; thus, according to Lemma 4.4, they have the same color as $v_{5,6}$ (*i.e.*, the color of v_5 and v_6). Therefore, we can continue this process to label all vertices.

The Algorithm. Armed with Lemma 4.4 and the merge operation, we present our efficient vertex labeling algorithm in Algorithm 5, denoted BCC-Labeling. Firstly, we construct a layered graph (Line 1); that is, we conduct a BFS of G starting from a random vertex r and assign each vertex in G a BFS level number.[5] Secondly, we compute CCs of the subgraph of G, consisting of only edges whose two end-points are at the same level, to label vertices according to RULE-I (Line 2). Note that, all vertices in the same CC (*i.e.*, with the same color and at the same level) need to be merged into a super-vertex to iteratively apply RULE-II level-by-level. Instead of physically

[5] Note that, this directly generalizes to disconnected graphs by first identifying CCs of the graph and then conducting BFS for each CC.

Algorithm 5: BCC-Labeling

Input: A graph $G = (V, E)$
Output: The BCCs of G

1 Assign each vertex in G a BFS level number by starting BFS from a random vertex r;
2 Run a CC computation algorithm on the subgraph consisting of only edges connecting same-level vertices, and let $v.sid$ denote the super-vertex id of the CC containing v;
3 $l \leftarrow$ the largest level number of vertices in G;
4 **for each** *vertex v at level l* **do** $v.color \leftarrow v.sid$;
5 **while** $l > 1$ **do**
  /* Send colors from level l to level $l-1$ */
6   **for each** *vertex v at level l* **do** **if** $v.sid \neq v.id$ **then**
7   Send two of v's upper-level neighbors to vertex $v.sid$;
8   **for each** *vertex v at level l* **do** **if** $v.sid = v.id$ **then**
9     $S \leftarrow$ {all vertices in v's received messages};
10     $S \leftarrow S \cup$ {v's upper-level neighbors};
11   **if** $|S| > 1$ **then** Notify all vertices at level l with color $v.color$ to send $v.color$ to their upper-level neighbors;
  /* Assign colors to vertices at level $l-1$ */
12   $l \leftarrow l-1$;
13   **for each** *vertex v at level l* **do**
14   Let C be the set of colors v received in messages;
15   **if** $v.sid = v.id$ **then** $v.cls \leftarrow C$;
16   **else if** $|C| > 0$ **then** Send C to the vertex $v.sid$;
17   **for each** *vertex v at level l* **do** **if** $v.sid = v.id$ **then**
18     $v.cls \leftarrow v.cls \cup$ {the colors v received in messages};
19   **if** $|v.cls| = 0$ **then** $v.color \leftarrow v.sid$;
20   **else** $v.color \leftarrow$ the first color in $v.cls$;
21   Send $v.color$ to all vertices whose super-vertex is v;
22   **for each** *vertex v at level l* **do** **if** $v.sid \neq v.id$ **then**
23     $v.color \leftarrow$ the color v received in messages;
24   Run merge-color aggregator with parameter l;
25   **for each** *vertex v at level l* **do**
26     $v.color \leftarrow$ parent($v.color$); $v.sid \leftarrow$ vertexID($v.color$);
27 update-color();

assigning all neighbors of vertices in a CC as the neighbors of the super-vertex, we store in $v.sid$ the id of the super-vertex containing v while neighbors are still kept at the individual vertices. The super-vertex id can be the id of any vertex in the super-vertex, and all vertices in the same super-vertex will have the same super-vertex id.

Then, we label vertices level-by-level in a bottom-up fashion. For vertices at the bottom level, we label all vertices in the same CC (indicated by $v.sid$) by a unique color (*i.e.*, $v.sid$) (Line 4). After that, we go to iterations to label other vertices level-by-level in two phases (Lines 6–26).

(Phase 1) Propagate colors from vertices at level l to vertices at level l − 1 (Lines 6–11). In order to apply RULE-II in Lemma 4.4, we need to check whether a super-vertex has at least two upper-level neighbors. Here, we ensure that *vertices at level l belong to the same super-vertex (i.e., with the same sid) if and only if they have the same color*; that is, $u.sid = v.sid$ if and only if $u.color = v.color$. Thus, we use the vertex with id $v.sid$ to check whether the super-vertex has at least two upper-level neighbors. However, instead of collecting all upper-level neighbors to the super-vertex, we only need to collect up to two neighbors from each vertex in the super-vertex (Line 7). We put these upper-level neighbors into a set S (Lines 9–10). If there are at least two upper-level neighbors

(*i.e.*, equivalently, $|S| > 1$), then we notify all vertices at level l with color $v.color$ (*i.e.*, in the super-vertex) to send their colors to their upper-level neighbors according to RULE-II in Lemma 4.4 (Line 11).

(Phase 2) Assign colors to vertices at level $l-1$ (Lines 12–26). Line 12 moves up one level higher (*i.e.*, $l \leftarrow l-1$); thus, in the following, we will talk about vertices at level l. A vertex at level l may receive several colors propagated from vertices at level $l+1$. We merge colors such that only one color is assigned to each super-vertex at level l. There are two cases for merging colors: 1) colors received by the same vertex should be merged; 2) colors received by vertices in the same CC (*i.e.*, the same super-vertex) should be merged. Therefore, we send all colors received by vertices in the same CC to the super-vertex of the CC for merging (Lines 13–16). For each super-vertex, we assign one of the received colors to be its color (Lines 17–21) and this color is also assigned to all vertices in the super-vertex (*i.e.*, CC) (Lines 22–23). If two super-vertices received the same color, then they should be merged into a single super-vertex according to the merge operation. This is achieved by the merge-color aggregator (Line 24), which will be discussed shortly. Finally, we update the colors and super-vertex ids of vertices at level l (Lines 25–26); here, vertexID(c) denotes the super-vertex id of vertices with color c.

Algorithm 6: update-color

1 Let C be the set of all colors in the disjoint-set data structure;
2 **for each** *color c in C* **do** Find(c);
3 **for each** *vertex $v \in V$* **do** $v.color \leftarrow$ parent($v.color$);

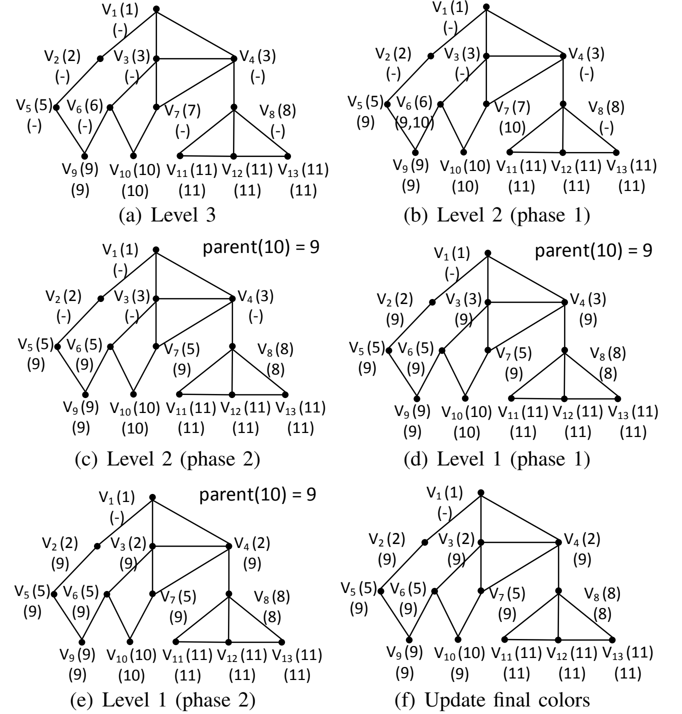
Note that, in the above bottom-up process, colors assigned to vertices are only tentative, because some colors assigned to vertices at lower levels may be merged at higher levels. For example, in Figure 8 after labeling vertices at level 3, v_9 and v_{10} have colors 9 and 10, respectively; the two colors are merged into a single color when labeling vertices at level 2 because v_6 receives both color 9 and color 10. However, we do not update colors for vertices at lower levels in this bottom-up process; instead, we update colors for all vertices in a final updating process (*i.e.*, Line 27), which is presented in Algorithm 6.

Algorithm 7: merge-color aggregator

Input: The current level number l

1 **for each** *vertex $v \in V$ at level l* **do**
2   **if** $|v.cls| > 0$ **then** Send $(v.id, v.cls)$ to the master worker;
3 **for each** *$(v.id, v.cls)$ received by the master worker* **do**
4   **for each** *color c' in $v.cls$* **do**
5   **if** *c' is not in the disjoint data structure* **then**
6   Add c' into the data structure with parent(c') = c' and vertexID(c') = $v.id$;
7   $c \leftarrow$ an arbitrary color in $v.cls$;
8   **for each** *color c' in $v.cls$* **do** union(c, c');
9 Let C be the set of colors in the received messages;
10 **for each** *color c' in C* **do** find(c');
11 Broadcast $\{(c, \text{parent}(c), \text{vertexID}(\text{parent}(c))) \mid c \in C\}$ to all slave workers;

merge-color *Aggregator.* The aggregator for merging colors and super-vertices is shown in Algorithm 7, denoted merge-color. It is similar to Algorithm 3. The only difference is that, we also merge super-vertices here; that is, two super-vertices are merged together if their sets of colors overlap. Thus, for each color c in the disjoint-set data structure, we also assign a super-vertex id to c, denoted vertexID(c), which is the super-vertex id of vertices with color c. After running the aggregator, parent(c) denotes the representative color of c (*i.e.*, color c should be replaced by parent(c)), and vertexID(parent(c)) denotes the super-vertex id of all vertices at the current level with color parent(c).



(a) Level 3 (b) Level 2 (phase 1) (c) Level 2 (phase 2) (d) Level 1 (phase 1) (e) Level 1 (phase 2) (f) Update final colors

Fig. 9: Running example for computing BCCs (*the number beside vertex v is $v.sid$, and that below is $v.color$ or $v.cls$*)

Running Example. Consider the layered graph in Figure 8. After computing CCs of the subgraph consisting of edges connecting vertices at the same level, the super-vertex ids of CCs are assigned as shown in brackets beside vertices in Figure 9(a); for example, v_{11} is the super-vertex of the CC consisting of $\{v_{11}, v_{12}, v_{13}\}$. Then, vertices at the bottom level (*i.e.*, $v_9, \ldots, v_{13}$) set their colors as $v.sid$ as shown in Figure 9(a), where colors are given in brackets below vertices.

In assigning colors to vertices at level 2, we first check whether a super-vertex at level 3 should propagate its color to its upper-level neighbors according to RULE-II in Lemma 4.4; if it is the case, we propagate its color to its neighbors at level 2 (*i.e.*, Phase 1). Here, the super-vertex of $\{v_{11}, v_{12}, v_{13}\}$ has only one upper-level neighbor, thus its color is not propagated. Both v_9 and v_{10} have two upper-level neighbors, thus their colors are propagated to their upper-level neighbors. The result is shown in Figure 9(b); v_6 receives colors, 9 and 10. Now, at Phase 2, every super-vertex sets its color either as the first received color (*e.g.*, $v_6.cls = \{9, 10\}$ and $v_6.color = 9$) or as its vertex id if it receives no color (*e.g.*, $v_8.cls = \emptyset$ and $v_8.color = 8$). Moreover, super-vertices having the same color need to be merged according to the merge operation, which is achieved by the merge-color aggregator. Here, the aggregator collects $(5, \{9\}), (6, \{9, 10\}), (7, \{10\})$ at the master worker and conducts the merging. Assume the result of the aggregator is parent(10) = 9 and vertexID(9) = v_5, then the colors and

super-vertex ids of vertices at level 2 are updated as shown in Figure 9(c). For example, the super-vertex ids of v_5, v_6 and v_7 now become 5, since they are merged into a super-vertex due to having the same color.

Similarly, we assign colors to vertices at level 1. At Phase 1, we propagate colors of vertices at level 2 to their neighbors at level 1. Since the super-vertex $\{v_5, v_6, v_7\}$ has three upper-level neighbors (*i.e.*, v_2, v_3, and v_4), its color is propagated to level 1; the result is shown in Figure 9(d). At Phase 2, super-vertices at level 1 collect their received colors and set their colors accordingly. Also the aggregator conducts the merging, and the result is shown in Figure 9(e).

Finally, we update colors of all vertices, and the result is shown in Figure 9(f). BCCs can be identified through the colors of vertices; that is, the set of vertices with the same color and their upper-level neighbors corresponds to a BCC. There are three BCCs, induced by vertices $\{v_1, \ldots, v_7, v_9, v_{10}\}$, $\{v_4, v_8\}$, and $\{v_8, v_{11}, v_{12}, v_{13}\}$, respectively.

C. Correctness and Complexity Analysis

We prove the correctness and give complexity analyses of of our approach.

Correctness. We prove the correctness of our approach by the following theorem.

Theorem 4.2: *Our vertex labeling approach labels all vertices, except the root vertex, of a* BCC *by a unique color.* □

Thus, Algorithm 5 correctly computes all BCCs of a graph.

Complexity Analyses. Now, we analyze the complexities of our approach regarding #supersteps, total communication cost, and total computation cost in the following.

<u>Number of Supersteps.</u> The #supersteps of Algorithm 5 is $O(\delta + \log n)$. Firstly, constructing the layered graph at Line 1 takes $O(\delta)$ supersteps. Secondly, running our CC computation algorithm at Line 2 takes at most $O(\log n)$ supersteps. Thirdly, the while loop at Lines 5–26 takes $O(\delta)$ supersteps. Finally, both Algorithm 6 and Algorithm 7 take constant supersteps.

<u>Total Communication Cost.</u> The total communication cost of Algorithm 5 is $O(m)$. Firstly, constructing the layered graph by conducting BFS at Line 1 has communication cost $O(m)$. Secondly, computing CCs at Line 2 by our approach in Section III has communication cost $O(m)$. Thirdly, the while loop at Lines 5–26 traverses the layered graph only once in a bottom-up fashion; thus its communication cost is $O(m)$. Finally, similar to Section III-C, Algorithm 6 and all invocations to Algorithm 7 have total communication cost $O(m)$.

<u>Total Computation Cost.</u> Similar to Section III-C, the total computation cost of our approach is $O(m + n \times \#\text{supersteps})$.

V. Experiments

We conduct extensive performance studies to evaluate the efficiency of our graph decomposition based approaches for computing CCs and for computing BCCs. Regarding computing CCs, we evaluate the following algorithms:

- S-V: the algorithm proposed in [36].
- hash-min: the algorithm proposed in [23].
- single-pivot: the algorithm proposed in [24].
- GD-CC: our algorithm in Section III.

Regarding computing BCCs, we evaluate algorithms below:

- T-V(S-V): the algorithm proposed in [36].
- T-V(hash-min): the algorithm proposed in [36].
- T-V(single-pivot): the combination of T-V algorithm in [36] with CC computation algorithm single-pivot.
- GD-BCC: our algorithm in Section IV.

Since most of the existing algorithms in our testings have been implemented in Pregel+,[6] an open source C++ implementation of the Pregel system, we also implement our algorithms in Pregel+. Specifically, we implement single-pivot, GD-CC, GD-BCC, and T-V(single-pivot), while other algorithms are already in Pregel+. Note that, although T-V(single-pivot) is not considered in [36], it is very natural to adjust the T-V algorithm in [36] to use single-pivot for computing CCs.

We compile all algorithms with GNU GCC with the -O3 optimization. Our experiments are conducted on a cluster of up to 25 Amazon EC2 r3.2xlarge instances (*i.e.*, machines) with enhanced networking. Each r3.2xlarge instance has 4 cores and 60GB RAM and runs 4 workers; thus, we have up to 100 total workers. We evaluate the performance of all algorithms on both real and synthetic graphs as follows.

TABLE I: Statistics of graphs

Graph	#Vertices	#Edges	#CCs	max |CC|
TW	41,652,230	1,202,513,046	1	41,652,230
T-S	92,288,289	3,012,576,376	32	50,634,118
PL1	264,464,510	2,005,938,490	4	66,123,323
PL2	529,005,705	4,027,878,138	8	66,142,392
YH	720,242,173	6,434,561,035	2×10^6	701,814,265

Graphs. We evaluate the algorithms on three real graphs and two synthetic graphs. The three real graphs are 1) TW[7] (a social network of Twitter users in 2010), 2) T-S (the combination of TW with a web graph of .sk domain crawled in 2005[7]), and 3) YH[8] (a web graph crawled by yahoo in 2002). The synthetic graphs PL1 and PL2 contain 4 and 8 CCs, respectively, each of which is generated by GTGraph[9] following a power-law degree distribution. The diameter of YH is large while that of other graphs are small. Statistics of these graphs are shown in Table I.

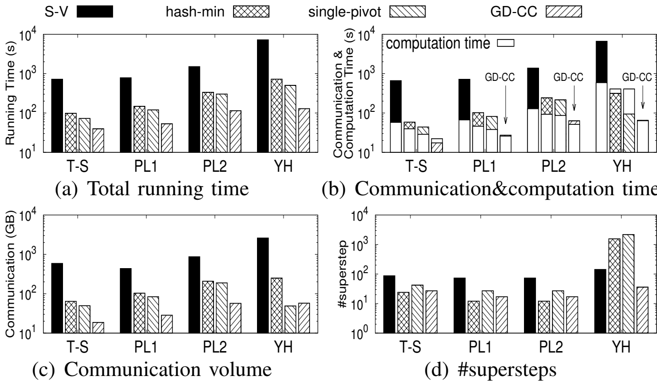
Evaluating Metric. We evaluate the algorithms in terms of total running time, computation time, communication time, communication volume (*i.e.*, the total data message sizes), and #supersteps; $\beta = 2$ by default. The communication time includes both the data communication time and the communication time related to the barrier synchronization (see cost estimation of Pregel algorithms in Section I). We vary the number of workers (#workers) from 40 to 60, 80, and 100, with #workers = 60 by default.

A. Experimental Results

Eval-I: Evaluating CC Computation Algorithms. The performance of the four CC computation algorithms on the four graphs is illustrated in Figure 10. Overall, our GD-CC approach performs the best and outperforms S-V, hash-min, single-pivot by up to 56, 5.6, 3.9 times, respectively (see Figure 10(a)). In Figure 10(b), we overlap the computation time with the communication time to compare the two costs. Note that, the entire bar (*i.e.*, the top part + the bottom part)

[6] http:www.cse.cuhk.edu.hk/pregelplus/
[7] http://law.di.unimi.it/datasets.php
[8] https://webscope.sandbox.yahoo.com/catalog.php?datatype=g
[9] http://www.cse.psu.edu/~kxm85/software/GTgraph/

represents the total length of the dominant cost; that is, if the communication time dominates, then the entire bar is the communication time, otherwise it is the computation time. For example, in the first bar, the entire bar (*i.e.*, the black part + the white part) represents the total communication time while the white bar shows the total computation time; in the fourth bar that represents GD-CC, the entire bar (*i.e.*, the white part + the bottom part) represents the total computation time while the bottom part shows the total communication time. For PL1 and YH, the communication time of our GD-CC algorithm is similar to the computation time, with the communication time being slightly larger. We can see that the communication time is larger than the computation time for all existing algorithms on all three small-diameter graphs, T-S, PL1, and PL2 (*e.g.*, communication time is one order of magnitude larger than computation time for S-V), while the computation time becomes the dominating factor for hash-min and single-pivot on the large-diameter graph (*i.e.*, YH) due to $O(n \times \#\text{supersteps})$ computation cost. This motives us to reduce the total communication volume (see Figure 10(c)) while also keeping #supersteps small (see Figure 10(d)). As a result, both the communication time and computation time of our approach are much smaller than the existing algorithms (see Figure 10(b)). Figure 10(d) confirms the $O(\min\{\delta, \log n\})$ #supersteps of our approach.

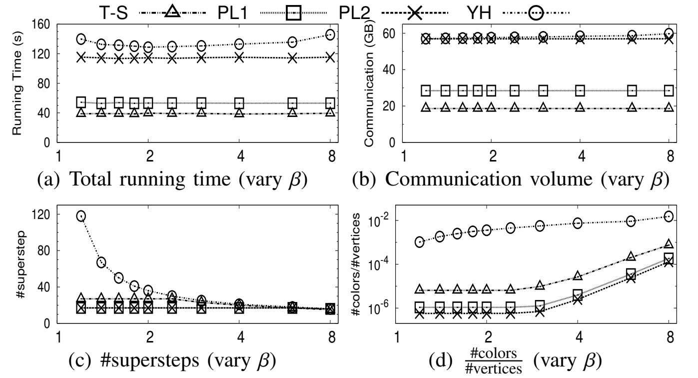


(a) Total running time (b) Communication&computation time

(c) Communication volume (d) #supersteps

Fig. 10: Evaluating CC computation algorithms

Eval-II: Influence of β on Our CC Computation Algorithm. It seems hard to give a theoretical result about the selection of β for our GD-CC algorithm. Thus, in this testing, we use experiments to select β, which decides how the number of selected seed vertices increases along with the supersteps. The results of running GD-CC with different β values are shown in Figure 11. In general, for small-diameter graphs, the total running time, communication volume, and #supersteps of GD-CC are not sensitive to β; this is because #supersteps on these graphs are $O(\delta)$ and they are small (see Figure 11(c)). On the large-diameter graph YH, the #supersteps of GD-CC, which is $O(\log_\beta n)$, decreases with larger β; however, the communication time increases with larger β due to the selection of more seed vertices. As a result, the total running time of GD-CC first decreases and then increases with $\beta = 2$ as the turning point. Moreover, the #colors aggregated at the master worker increases for all graphs when β becomes larger than 2 (see Figure 11(d)). Therefore, we set $\beta = 2$ for GD-CC. One thing to notice is that when $\beta = 2$, the #colors aggregated at the master worker is only a small fraction (*i.e.*, $\leq$ 0.4%) of the #vertices; thus, the merging of colors can be done through aggregator at the master worker.

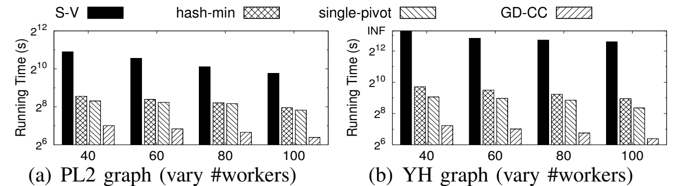


(a) Total running time (vary β) (b) Communication volume (vary β)

(c) #supersteps (vary β) (d) $\frac{\#\text{colors}}{\#\text{vertices}}$ (vary β)

Fig. 11: Evaluating the impact of β on our GD-CC algorithm

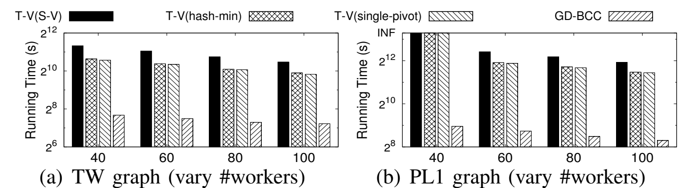
Eval-III: Scalability Testing of CC Computation Algorithms. To evaluate the scalability of CC computation algorithms, we vary the #workers from 40 to 100. The testing results are shown in Figure 12; S-V cannot finish in two hours on YH when running on 40 workers. Our GD-CC algorithm as well as other algorithms scale almost linearly with #workers. Still, GD-CC consistently outperforms other algorithms.



(a) PL2 graph (vary #workers) (b) YH graph (vary #workers)

Fig. 12: Scalability testing of CC computation algorithms

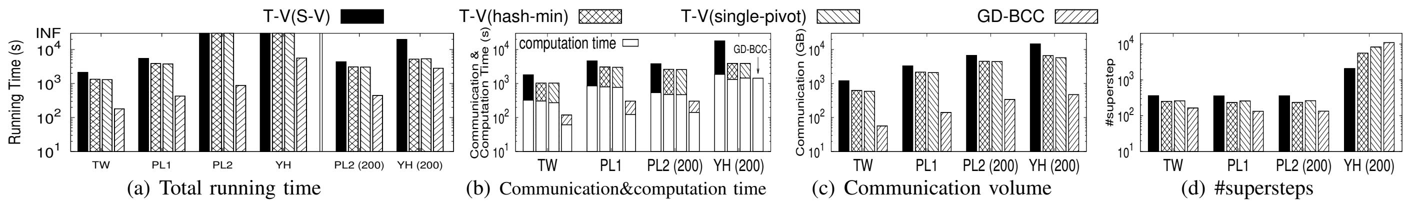
Eval-IV: Evaluating BCC Computation Algorithms. Similar to Figure 10, the testing results of running BCC algorithms are shown in Figure 13, where in Figure 13(b) we overlap the computation time with communication time to compare the two costs similar to that in Figure 10(b). Note that, since the existing algorithms run out-of-memory on PL2 and YH when running on 60 workers (*i.e.*, 15 machines), we increase the number of workers to 200 (*i.e.*, 50 machines) for processing PL2 and YH (denoted as PL2(200) and YH(200), respectively). Our GD-BCC algorithm consistently outperforms the existing algorithms (see Figure 13(a)). GD-BCC has significantly smaller communication volume than the existing algorithms (see Figure 13(c)); this confirms our $O(m)$ communication cost. Although GD-BCC has larger #supersteps on YH, GD-BCC still has similar computation time compared to existing algorithms (see Figure 13(b)), due to our computation cost of $O(m + n \times \#\text{supersteps})$. As a result, both the communication time and computation time of our GD-BCC algorithm are smaller than the existing algorithms (see Figure 13(b)). Note that, the running time of T-V(single-pivot) is almost the same as that of T-V(hash-min). This is because the converting process (*i.e.*, the T-V algorithm part) has the dominating cost.



(a) TW graph (vary #workers) (b) PL1 graph (vary #workers)

Fig. 14: Scalability testing of BCC computation algorithms

Eval-V: Scalability Testing of BCC Computation Algorithms. Similar to Figure 12, we vary the #workers from 40 to



(a) Total running time (b) Communication&computation time (c) Communication volume (d) #supersteps

Fig. 13: Evaluating BCC computation algorithms

100 to evaluate the scalability of BCC computation algorithms. The results are shown in Figure 14; the existing approaches run out-of-memory on PL1 when running on 40 workers (*i.e.*, 10 machines). GD-BCC as well as the existing approaches scale well with the increasing of #workers.

VI. Conclusion

In this paper, we proposed graph decomposition based approaches for distributed computing CCs and BCCs of a graph. Unlike existing approaches that have total data communication cost $O(m \times \#\text{supersteps})$, our new approaches have total data communication costs $O(m)$. Moreover, the computation costs and #supersteps of our techniques are similar to (or even smaller than) those of the existing techniques, respectively. Experiments show that our approaches outperform existing approaches by generating much smaller volumes of messages. Possible directions of future work are to control the #colors generated by our algorithms and to reduce #supersteps of our BCC computation algorithm for large-diameter graphs.

Acknowledgment

Lijun Chang is supported by ARC DE150100563 and ARC DP160101513. Xuemin Lin is supported by NSFC61232006, ARC DP140103578 and ARC DP150102728. Lu Qin is supported by ARC DE140100999 and ARC DP160101513. Wenjie Zhang is supported by ARC DP150103071 and ARC DP150102728.

References

[1] B. Awerbuch and Y. Shiloach. New connectivity and MSF algorithms for shuffle-exchange network and PRAM. *IEEE Trans. Computers*, 1987.

[2] M. Ceccarello, A. Pietracaprina, G. Pucci, and E. Upfal. Space and time efficient parallel graph decomposition, clustering and diameter approximation. In *Proc. of SPAA'15*, 2015.

[3] A. Ching and C. Kunz. Giraph: Large-scale graph processing infrastructure on hadoop. *Hadoop Summit*, 2011.

[4] J. Cohen. Graph twiddling in a mapreduce world. *Computing in Science and Engineering*, 2009.

[5] T. H. Cormen, C. Stein, R. L. Rivest, and C. E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2001.

[6] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *Proc. of OSDI'04*, 2004.

[7] X. Feng, L. Chang, X. Lin, L. Qin, and W. Zhang. Computing connected components with linear communication costs in pregel-like systems. In *UNSW-CSE-TR-201508, http://goo.gl/yCPS94*.

[8] S. Fortunato. Community detection in graphs. *Physics Reports*, 2010.

[9] A. Gibbons. *Algorithmic Graph Theory*. Cambridge University Press, 1985.

[10] 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. of OSDI'14*, 2014.

[11] N. Henry, A. Bezerianos, and J. Fekete. Improving the readability of clustered social networks using node duplication. *IEEE Trans. Vis. Comput. Graph.*, 2008.

[12] J. E. Hopcroft and R. E. Tarjan. Efficient algorithms for graph manipulation [H] (algorithm 447). *Commun. ACM*, 1973.

[13] U. Kang, M. McGlohon, L. Akoglu, and C. Faloutsos. Patterns on the connected components of terabyte-scale graphs. In *Proc. of ICDM'10*, 2010.

[14] U. Kang, C. E. Tsourakakis, and C. Faloutsos. PEGASUS: A peta-scale graph mining system. In *Proc. of ICDM'09*, 2009.

[15] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. Distributed graphlab: A framework for machine learning in the cloud. *PVLDB*, 2012.

[16] G. Malewicz, M. H. Austern, A. J. C. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *Proc. of SIGMOD'10*, 2010.

[17] G. L. Miller, R. Peng, and S. C. Xu. Parallel graph decompositions using random shifts. In *Proc. of SPAA'13*, 2013.

[18] K. Munagala and A. G. Ranade. I/o-complexity of graph algorithms. In *Proc. of SODA'99*, 1999.

[19] S. Nanda and D. Kotz. Localized bridging centrality for distributed network analysis. In *Proc. of ICCCN'08*, 2008.

[20] N. Przulj, D. Wigle, and I. Jurisica. Functional topology in a network of protein interactions. *Bioinformatics*, 2004.

[21] L. Qin, J. X. Yu, L. Chang, H. Cheng, C. Zhang, and X. Lin. Scalable big graph processing in mapreduce. In *Proc. of SIGMOD'14*, 2014.

[22] V. Ramachandran. *Parallel open ear decomposition with applications to graph biconnectivity and triconnectivity*. Citeseer, 1992.

[23] V. Rastogi, A. Machanavajjhala, L. Chitnis, and A. D. Sarma. Finding connected components in map-reduce in logarithmic rounds. In *Proc. of ICDE'13*, 2013.

[24] S. Salihoglu and J. Widom. Optimizing graph algorithms on pregel-like systems. *PVLDB*, 2014.

[25] P. Sanders. Random permutations on distributed, external and hierarchical memory. *Inf. Process. Lett.*, 1998.

[26] J. M. Schmidt. A simple test on 2-vertex- and 2-edge-connectivity. *Inf. Process. Lett.*, 2013.

[27] Y. Shiloach and U. Vishkin. An o(log n) parallel connectivity algorithm. *J. Algorithms*, 1982.

[28] G. M. Slota and K. Madduri. Simple parallel biconnectivity algorithms for multicore platforms. In *Proc. of HiPC'14*, 2014.

[29] I. Stanton. Streaming balanced graph partitioning algorithms for random graphs. In *Proc. of SODA'14*, 2014.

[30] I. Stanton and G. Kliot. Streaming graph partitioning for large distributed graphs. In *Proc. of KDD'12*, 2012.

[31] R. E. Tarjan and U. Vishkin. An efficient parallel biconnectivity algorithm. *SIAM J. Comput.*, 1985.

[32] Y. Tian, A. Balmin, S. A. Corsten, S. Tatikonda, and J. McPherson. From "think like a vertex" to "think like a graph". *PVLDB*, 2013.

[33] V. Turau. Computing bridges, articulations, and 2-connected components in wireless sensor networks. In *Proc. of ALGOSENSORS'06*, 2006.

[34] L. G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 1990.

[35] D. Vega, L. Cerda-Alabern, L. Navarro, and R. Meseguer. Topology patterns of a community network: Guifi.net. In *Proc. of WiMob'12*, 2012.

[36] D. Yan, J. Cheng, K. Xing, Y. Lu, W. Ng, and Y. Bu. Pregel algorithms for graph connectivity problems with performance guarantees. *PVLDB*, 2014.

[37] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proc. of NSDI'12*, 2012.