Pregel Algorithms for Graph Connectivity Problems with Performance Guarantees

Da Yan[#1], James Cheng[*2], Kai Xing[#3], Yi Lu[*4], Wilfred Ng[#5], Yingyi Bu[†6]

[#]*Department of Computer Science and Engineering, The Hong Kong University of Science and Technology*
{[1]yanda, [3]kxing, [5]wilfred}@cse.ust.hk

[*]*Department of Computer Science and Engineering, The Chinese University of Hong Kong*
{[2]jcheng, [4]ylu}@cse.cuhk.edu.hk

[†]*Department of Computer Science, University of California, Irvine*
[6]yingyib@ics.uci.edu

ABSTRACT

Graphs in real life applications are often huge, such as the Web graph and various social networks. These massive graphs are often stored and processed in distributed sites. In this paper, we study graph algorithms that adopt Google's Pregel, an iterative vertex-centric framework for graph processing in the Cloud. We first identify a set of desirable properties of an efficient Pregel algorithm, such as linear space, communication and computation cost per iteration, and logarithmic number of iterations. We define such an algorithm as a *practical Pregel algorithm* (PPA). We then propose PPAs for computing *connected components* (CCs), *biconnected components* (BCCs) and *strongly connected components* (SCCs). The PPAs for computing BCCs and SCCs use the PPAs of many fundamental graph problems as building blocks, which are of interest by themselves. Extensive experiments over large real graphs verified the efficiency of our algorithms.

1. INTRODUCTION

The popularity of online social networks, mobile communication networks and semantic web services, has stimulated a growing interest in conducting efficient and effective analysis on massive real-world graphs. To process such large graphs, Google's Pregel [15] proposed the vertex-centric computing paradigm, which allows programmers to think naturally like a vertex when designing distributed graph algorithms. A Pregel-like system runs on a shared-nothing distributed computing infrastructure which can be deployed easily on a cluster of low-cost commodity PCs. The system also removes from programmers the burden of handling fault recovery, which is important for programs running in a cloud environment.

Pregel was shown to be more suitable for iterative graph computation than the popular MapReduce model [7, 15, 16]. However, existing work on Pregel algorithms [16] is rather ad hoc, which looks more like a demonstration of how Pregel can be used to solve a number of graph problems, and lacks any analysis on the cost complexity.

In this paper, we study three fundamental graph connectivity problems and propose Pregel algorithms as solutions that have performance guarantees. The problems we study are *connected components* (**CC**s), *bi-connected components* (**BCC**s), and *strongly connected components* (**SCC**s). These problems have numerous real life applications and their solutions are essential building blocks for solving many other graph problems. For example, computing the BCCs of a telecommunications network can help detect the weaknesses in network design, while almost all reachability indices require SCC computation as a preprocessing step [5].

To avoid ad hoc algorithm design and ensure good performance, we define a class of Pregel algorithms that satisfy a set of rigid, but practical, constraints on various performance metrics: *(1)linear space usage, (2)linear computation cost per iteration*[1], *(3)linear communication cost per iteration, and (4)at most logarithmic number of iterations*. We call such algorithms as **Practical Pregel Algorithm**s (**PPA**s). A similar but stricter set of constraints was proposed for the MapReduce model recently [23]. In contrast to our requirement of logarithmic number of iterations, their work demands a constant number of iterations, which is too restrictive for most graph problems. In fact, even list ranking (i.e., ranking vertices in a directed graph consisting of only one simple path) requires $O(\log n)$ time using $O(n)$ processors under the *shared-memory* PRAM model [25], where n is the number of vertices. This bound also applies to many other basic graph problems such as connected components and spanning tree [21].

It is challenging to design Pregel algorithms for problems such as BCCs and SCCs. Although there are simple sequential algorithms for computing BCCs and SCCs based on depth-first search (DFS), DFS is $\mathcal{P}$-Complete [18] and hence it cannot be applied to design parallel algorithms for computing BCCs and SCCs.

We apply the principle of PPA to develop Pregel algorithms that satisfy strict performance guarantees. In particular, to compute BCCs and SCCs, we develop a set of useful building blocks that are the PPAs of fundamental graph problems such as *breadth-first search*, *list ranking*, *spanning tree*, *Euler tour*, and *pre/post-order traversal*. As fundamental graph problems, their PPA solutions can also be applied to numerous other graph problems besides BCCs and SCCs considered in this paper.

We evaluate the performance of our Pregel algorithms using large real-world graphs with up to hundreds of millions of vertices and billions of edges. Our results verify that our algorithms are efficient for computing CCs, BCCs and SCCs in massive graphs.

The rest of this paper is organized as follows. Section 2 reviews

[1]A Pregel algorithm proceeds in iterations.

Pregel and related work. We define PPA in Section 3. Sections 4-6 discuss algorithms for CCs, BCCs and SCCs. Then, we report the experimental results in Section 7 and conclude in Section 8.

2. RELATED WORK

Pregel [15]. Pregel is designed based on the bulk synchronous parallel (BSP) model. It distributes vertices to different machines in a cluster, where each vertex v is associated with its adjacency list (i.e., the set of v's neighbors). A program in Pregel implements a user-defined *compute*() function and proceeds in iterations (called *supersteps*). In each superstep, the program calls *compute*() for each active vertex. The *compute*() function performs the user-specified task for a vertex v, such as processing v's incoming messages (sent in the previous superstep), sending messages to other vertices (to be received in the next superstep), and making v vote to halt. A halted vertex is reactivated if it receives a message in a subsequent superstep. The program terminates when all vertices vote to halt and there is no pending message for the next superstep.

Pregel numbers the supersteps so that a user may use the current superstep number when implementing the algorithm logic in the *compute*() function. As a result, a Pregel algorithm can perform different operations in different supersteps by branching on the current superstep number.

Pregel allows users to implement a *combine*() function, which specifies how to combine messages that are sent from a machine M_i to the same vertex v in a machine M_j. These messages are combined into a single message, which is then sent from M_i to v in M_j. Combiner is applied only when commutative and associative operations are to be applied to the messages. For example, in Pregel's PageRank algorithm [15], messages from machine M_i that are to be sent to the same target vertex in machine M_j can be combined into a single message that equals their sum, since the target vertex is only interested in the sum of the messages. Pregel also supports aggregator, which is useful for global communication. Each vertex can provide a value to an aggregator in *compute*() in a superstep. The system aggregates those values and makes the aggregated result available to all vertices in the next superstep.

Pregel Algorithms. Besides this paper and Google's original paper on Pregel [15], we are only aware of two other papers studying Pregel algorithms, [16] and [20]. However, the algorithms are designed on a best-effort basis without any formal analysis on their complexity. Specifically, [16] aims at demonstrating that the Pregel model can be adopted to solve many graph problems in social network analysis, while [20] focuses on optimization techniques that overcome some performance bottlenecks caused by straightforward Pregel implementations.

MapReduce [8]. MapReduce, and its open-source implementation Hadoop, is another distributed system popularly used for large scale graph processing [11, 12, 13, 17, 22]. However, many graph algorithms are intrinsically iterative, in which case a Pregel program is much more efficient than its MapReduce counterpart. This is because each MapReduce job can only perform one iteration of graph computation, and it requires reading the entire graph from a distributed file system (DFS) and writing the processed graph back, which leads to excessive IO cost. Furthermore, a MapReduce job also needs to exchange the adjacency lists of vertices through the network, which results in heavy communication. In contrast, Pregel keeps vertices (along with their adjacency lists) in each local machine that processes their computation, and uses only message passing to exchange vertex states. The "think like a vertex" programming paradigm also makes it easier for programmers to design graph algorithms in Pregel than in MapReduce.

GraphLab [14] and PowerGraph [10]. GraphLab [14] is another vertex-centric distributed graph computing system but it follows a different design from Pregel. GraphLab supports both synchronous and asynchronous executions. However, asynchronous execution does not have the concept of superstep number and hence cannot support algorithms that branch to different operations at different supersteps, such as the S-V algorithm in Section 4.2. Moreover, since GraphLab is mainly designed for asynchronous execution, its synchronous mode is not as expressive as Pregel. For example, since GraphLab only allows a vertex to access the states of its adjacent vertices and edges, it cannot support algorithms where a vertex needs to communicate with a non-neighbor. Another limitation of GraphLab is that it does not support graph mutations.

GraphLab 2.2, i.e., PowerGraph [10], partitions a graph by edges rather than by vertices in order to address imbalanced workload caused by high-degree vertices. However, a more complicated edge-centric Gather-Apply-Scatter (GAS) computing model should be used, which compromises user-friendliness.

PRAM. The PRAM model assumes that there are many processors and a shared memory. PRAM algorithms have been proposed for computing CCs [21], BCCs [24], and SCCs [3, 4, 9]. However, the PRAM model is not suitable for Cloud environments that are built on shared-nothing architectures. Furthermore, unlike Pregel and MapReduce, PRAM algorithms are not fault tolerant.

However, the ideas of many PRAM algorithms can be applied to design efficient Pregel algorithms. In Section 4.2, we shall demonstrate how the PRAM algorithm of [21] for computing CCs can be translated into a PPA. In Section 5.1, we describe the idea of [24]'s PRAM algorithm for computing BCCs, which will be used to design a PPA for computing BCCs.

3. PRACTICAL PREGEL ALGORITHMS

We now define some frequently used notations and introduce the notion of practical Pregel algorithms.

Notations. Given a graph $G = (V, E)$, we denote the number of vertices $|V|$ by n, and the number of edges $|E|$ by m. We also denote the *diameter* of G by δ. For an undirected graph, we denote the set of *neighbors* of a vertex v by $\Gamma(v)$ and the *degree* of v by $d(v) = |\Gamma(v)|$. For a directed graph, we denote the set of *in-neighbors* and *out-neighbors* of v by $\Gamma_{in}(v)$ and $\Gamma_{out}(v)$, and the *in-degree* and *out-degree* of v by $d_{in}(v) = |\Gamma_{in}(v)|$ and $d_{out}(v) = |\Gamma_{out}(v)|$, respectively.

A Pregel algorithm is called a **balanced practical Pregel algorithm** (**BPPA**) if it satisfies the following constraints:

1. *Linear space usage:* each vertex v uses $O(d(v))$ (or $O(d_{in}(v)+d_{out}(v))$) space of storage.

2. *Linear computation cost:* the time complexity of the $compute()$ function for each vertex v is $O(d(v))$ (or $O(d_{in}(v)+d_{out}(v))$).

3. *Linear communication cost:* at each superstep, the size of the messages sent/received by each vertex v is $O(d(v))$ (or $O(d_{in}(v) + d_{out}(v))$).

4. *At most logarithmic number of rounds:* the algorithm terminates after $O(\log n)$ supersteps.

Constraints 1-3 offers good load balancing and linear cost at each superstep, while Constraint 4 controls the total running time. As we shall see in later sections, some algorithms satisfying Constraints 1-3 require $O(\delta)$ rounds. Since many large real graphs have a small diameter δ, especially for social networks due to the small world phenomenon, we consider algorithms requiring $O(\delta)$ rounds also satisfying Constraint 4 if $\delta \leq \log n$ for the input graph.

For some problems, the per-vertex requirements of BPPA can be too strict, and we can only achieve *overall linear space usage, computation and communication cost* (still in $O(\log n)$ rounds). We call a Pregel algorithm that satisfies these constraints simply as a **practical Pregel algorithm** (**PPA**).

Motivation. We define BPPA and PPA in order to characterize a set of Pregel algorithms that can run efficiently in practice. Apart from the algorithms proposed in this paper, other Pregel algorithms, e.g., the four demo algorithms in the Pregel paper [15], also have these characteristics (we can show that they are BPPAs): PageRank (constant supersteps), single-source shortest paths ($O(\delta)$ supersteps), bipartite matching ($O(\log n)$ supersteps), and semi-clustering (constant supersteps). However, these existing Pregel algorithms are designed on a best-effort basis and there is no formal performance requirement to be met or design rule to be followed. For example, while the Pregel algorithm developed in [16] for diameter estimation is an $O(\delta)$-superstep BPPA, the Pregel algorithm for triangle counting and clustering coefficient computation in [16] is not a PPA. Specifically, in superstep 1 of the triangle counting algorithm, a vertex sends a message for each pair of neighbors, leading to a quadratic number of messages to be buffered and sent. With the concept of PPA/BPPA in mind, users of the triangle counting algorithm can then be aware of the scalability limitation when applying this algorithm. The requirements of PPAs/BPPAs also serve as a guideline for programmers/researchers who want to develop efficient Pregel algorithms, and they may use existing PPAs as building blocks in their algorithms.

In the next three sections, we present PPAs/BPPAs for three fundamental graph connectivity problems. We also demonstrate how the PPAs/BPPAs of some fundamental graph problems can be used as building blocks to develop a more sophisticated PPA/BPPA for solving other graph problems such as computing BCCs.

4. CONNECTED COMPONENTS

In this section, we present two Pregel algorithms for computing CCs. Section 4.1 presents a BPPA that requires $O(\delta)$ supersteps. This algorithm works well on many real world graphs with a small diameter. However, it can be very slow on large-diameter graphs, such as spatial networks for which $\delta \approx O(\sqrt{n})$. We present an $O(\log n)$-superstep PPA in Section 4.2 to handle such graphs.

4.1 The Hash-Min PPA

Before presenting the $O(\delta)$-superstep BPPA for computing CCs, we first define some graph notations. We assume that all vertices in a graph G are assigned a unique ID. For convenience of discussion, we simply use v to refer to the ID of vertex v, and thus, the expression $u < v$ means that u's vertex ID is smaller than v's. We define the *color* of a (strongly) connected component in G to be the smallest vertex among all vertices in the component. The color of a vertex v, denoted by $color(v)$, is defined as the color of the component that contains v, and so, all vertices in G with the same color constitute a component.

A MapReduce algorithm, called Hash-Min, was proposed for computing CCs recently [17]. The idea of the algorithm is to broadcast the smallest vertex (ID) seen so far by each vertex v, denoted by $min(v)$; when the process converges, $min(v) = color(v)$ for all v. We now propose a BPPA counterpart as follows.

In Superstep 1, each vertex v initializes $min(v)$ as the smallest vertex in the set $(\{v\} \cup \Gamma(v))$, sends $min(v)$ to all v's neighbors and votes to halt. In each subsequent superstep, a vertex v obtains the smallest vertex from the incoming messages, denoted by u. If $u < v$, v sets $min(v) = u$ and sends $min(v)$ it to all its neighbors. Finally, v votes to halt.

We prove that the algorithm is a BPPA as follows. For any CC, it takes at most δ supersteps for the ID of the smallest vertex to reach all the vertices in the CC, and in each superstep, each vertex v takes at most $O(d(v))$ time to compute $min(v)$ and sends/receives $O(d(v))$ messages each using $O(1)$ space.

Three other MapReduce algorithms were also proposed in [17] for computing CCs. However, they require that each vertex maintain a set whose size can be as large as the size of its CC, and that the whole set be sent to some vertices. Thus, they cannot be translated into efficient Pregel implementations due to the highly skewed communication and computation, and the excessive space cost.

4.2 The S-V PPA

Our Contributions. We propose an $O(\log n)$-superstep PPA based on the S-V algorithm [21]. We note that the state-of-the-art distributed algorithms can only achieve $O(\log n)$ iterations in expectation on some types of graphs [17], while [17] also claims that the requirement of concurrent writes makes the S-V algorithm difficult to be translated to MapReduce (similarly to Pregel). We show that a direct translation of the S-V algorithm to Pregel is incorrect, and then change the algorithmic logic to obtain an $O(\log n)$-superstep PPA for CC computation.

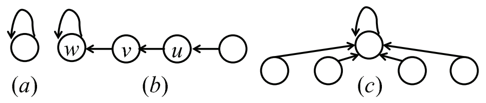


Figure 1: Forest structure of Shiloach-Vishkin's algorithm

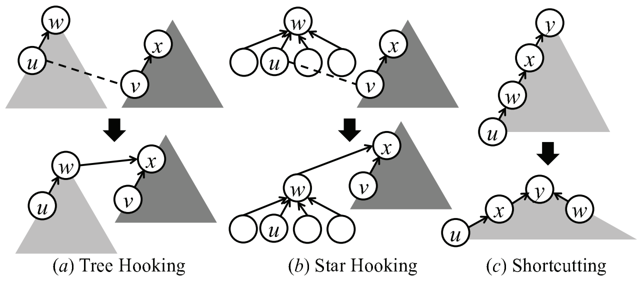


Figure 2: Tree hooking, star hooking, and shortcutting

Algorithm Overview. In the S-V algorithm, each vertex u maintains a pointer $D[u]$. Initially, $D[u] = u$, forming a self loop as shown Figure 1(a). Throughout the algorithm, vertices are organized by a forest such that all vertices in each tree in the forest belong to the same CC. The tree definition is relaxed a bit here to allow the tree root w to have a self-loop (see Figures 1(b) and 1(c)), i.e., $D[w] = w$; while $D[v]$ of any other vertex v in the tree points to v's parent.

The S-V algorithm proceeds in rounds, and in each round, the pointers are updated in three steps (illustrated in Figure 2): (1)*tree hooking*: for each edge (u, v), if u's parent $w = D[u]$ is a tree root, hook w as a child of v's parent $D[v]$ (i.e., merge the tree rooted at w into v's tree); (2)*star hooking*: for each edge (u, v), if u is in a star (see Figure 1(c) for an example of star), hook the star to v's tree as Step (1) does; (3)*shortcutting*: for each vertex v, move vertex v and its descendants closer to the tree root, by hooking v to the parent of v's parent, i.e., setting $D[v] = D[D[v]]$. The algorithm ends when every vertex is in a star.

We perform tree hooking in Step (1) only if $D[v] < D[u]$, so

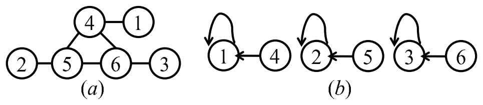


Figure 3: Illustration of the change to star hooking

that if u's tree is hooked to v's tree due to edge (u, v), then edge (v, u) will not hook v's tree to u's tree again.

The condition "$D[v] < D[u]$" is not required for star hooking, since if u's tree is hooked to v's tree, v's tree cannot be a star (e.g., in Figure 2(b), after hooking, u is two hops away from x). However, in Pregel's computing model, Step (2) cannot be processed. Consider the graph shown in Figure 3(a), and suppose that we obtain the three stars in Figure 3(b) right after Step (1). If the one-directional condition is not required, setting $D[D[u]]$ as $D[v]$ makes $D[1] = 2$, $D[2] = 3$ and $D[3] = 1$ through the edges $(4, 5)$, $(5, 6)$ and $(6, 4)$, thus forming a cycle and violating the tree formation required by the algorithm. Such a problem does not exist in the PRAM model since the values of $D[u]$ and $D[v]$ are immediately updated after each write operation, while in Pregel the values are those received from the previous superstep.

To address this problem, we examine the algorithm logic in Pregel execution and find that if we always require $v_a < v_b$ when setting $D[v_a] \leftarrow v_b$ during hooking, we can prove that the pointer values monotonically decrease, and thus $D[v] = color(v)$ for any vertex v when the algorithm terminates. Based on this result, we change the algorithm by requiring "$D[v] < D[u]$" for star hooking. Then, the S-V algorithm can be translated into a Pregel algorithm by exchanging messages to update the pointers $D[.]$ following the three steps in Figure 2, until every vertex v is in a star. Due to limited space, we present the details of the algorithm in Appendix A of our technical report [1].

The S-V based Pregel algorithm is an $O(\log n)$-superstep PPA, which can be proved as the original S-V algorithm computes CCs in $O(\log n)$ rounds [21], where each round is implemented in a constant number of supersteps. However, the algorithm is not a BPPA since a vertex v may become the parent of more than $d(v)$ vertices and hence receives/sends more than $d(v)$ messages in a superstep, though the overall number of messages in each superstep is always bounded by $O(n)$.

The S-V algorithm can also be extended to obtain an $O(\log n)$-superstep PPA for computing the spanning tree, the details of which can be found in Appendix A of [1].

5. BI-CONNECTED COMPONENTS

Our PPA for computing BCCs is based on the idea of the PRAM algorithm in [24], but we make the following new contributions.

Our Contributions. To our knowledge, the problem of computing BCCs in Pregel has never been studied before. Thus, it is important to show that a Pregel algorithm with strong performance guarantee exists for this problem, which we will establish by proposing a PPA for computing BCCs. Second, existing studies on designing Pregel algorithms [17, 16] often neglect the rich body of PRAM algorithms. Our PPA for computing BCCs demonstrates that some ideas from the PRAM algorithms can be applied to design Pregel algorithms. Third, though the main idea is based on [24], the design of our PPA for BCC computation is non-trivial. Specifically, our BCC algorithm is composed of a number of building blocks, and to ensure that our final algorithm is a PPA, we devise a PPA for each building block. Finally, these building blocks used in our BCC algorithm are themselves fundamental graph operations that are useful to the design of many other distributed graph algorithms.

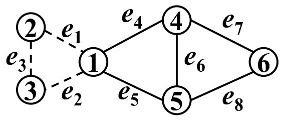


Figure 4: BCC illustration

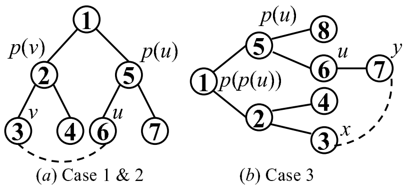


Figure 5: Illustration of construction of G^*

Therefore, we study them in greater depth and carefully design a PPA for each of them, which are often much simpler than the existing PRAM algorithms.

5.1 BCC and Its PRAM Algorithm

Bi-connected Component (BCC). A BCC of an undirected graph G is a maximal subgraph of G that remains connected after removing one arbitrary vertex. We illustrate the concept of BCC using the graph shown in Figure 4, where the dashed edges constitute one BCC, and the solid edges constitute another. Let R be the equivalence relation on the set of edges of G such that $e_1 R e_2$ iff $e_1 = e_2$ or e_1 and e_2 appear together in some simple cycle, then R defines the BCCs of G. For example, in Figure 4, edges $(4, 5)$ and $(5, 6)$ are in cycle $(4, 5, 6, 4)$, but there is no simple cycle containing both $(4, 5)$ and $(1, 2)$. A vertex is called an *articulation point* if it belongs to more than one BCC, such as vertex 1 in Figure 4. The removal of an articulation point disconnects the connected components containing it.

Tarjan-Vishkin's PRAM Algorithm. If we construct a new graph G' whose vertices correspond to the edges of G, and an edge (e_1, e_2) exists in G' iff $e_1 R e_2$, then the CCs of G' correspond to the BCCs of G. However, the number of edges in G' can be superlinear to m. Tarjan-Vishkin's (T-V) PRAM algorithm [24] constructs a concise graph G^* containing a small subset of the edges in G', whose size is bounded by $O(m)$. They prove that it is sufficient to compute the CCs of G^* to obtain the BCCs of G.

Since our PPA for computing BCCs is also based on this idea, we first present the definition of G^* below. Assume that $G = (V, E)$ is connected and T is a spanning tree of G. Also, assume that T is rooted and each vertex u in T is assigned a pre-order number $pre(u)$. Let $p(u)$ be the parent of a vertex u in T. We construct $G^* = (V^*, E^*)$ as follows. First, we set $V^* = E$. Then, we add an edge (e_1, e_2) to E^*, where $e_1, e_2 \in E$, iff e_1 and e_2 satisfy one of the following three cases:

- Case 1: $e_1 = (p(u), u)$ is a tree edge in T, and $e_2 = (v, u)$ is a non-tree edge (i.e., e_2 is not in T) with $pre(v) < pre(u)$.
- Case 2: $e_1 = (p(u), u)$ and $e_2 = (p(v), v)$ are two tree edges in T, u and v have no ascendant-descendant relationship in T, and $(u, v) \in E$.
- Case 3: $e_1 = (p(u), u)$ and $e_2 = (p(p(u)), p(u))$ are two tree edges in T, and $\exists(x, y) \in E$ s.t. x is a non-descendant of $p(u)$ in T and y is a descendant of u in T.

Figure 5 illustrates the three cases, where solid edges are tree edges in T, and dashed edges are non-tree edges in $(G - T)$. The vertices are labeled by their pre-order numbers. Case 1 is shown by $e_1 = (5, 6)$ and $e_2 = (3, 6)$ in Figure 5(a). Note that $e_1 R e_2$ since e_1 and e_2 are in a simple cycle $\langle 1, 2, 3, 6, 5, 1\rangle$. Case 2 is also shown in Figure 5(a) by $e_1 = (5, 6)$ and $e_2 = (2, 3)$, and also e_1 and e_2 are in the same simple cycle. Case 3 is shown in Figure 5(b) by $e_1 = (5, 6)$ and $e_2 = (1, 5)$, where e_1 and e_2 are in the simple cycle $\langle 1, 2, 3, 7, 6, 5, 1\rangle$.

Each non-tree edge (u, v) of G introduces at most one edge into G^* due to Case 1 (and Case 2), and each tree-edge $(p(u), u)$ introduces at most one edge due to Case 3. Therefore, $|E^*| = O(m)$.

5.2 PPA for Computing BCCs

Our PPA for computing BCCs is also based on the idea of Tarjan-Vishkin's algorithm, i.e., to construct the concise graph G^* and then compute the CCs of G^* to obtain the BCCs of G. Without loss of generality, we assume G is connected, as BCC computation in different CCs is independent and can be parallelized. To construct G^*, we first propose a set of building blocks in Sections 5.2.1-5.2.4, and then in Section 5.2.5 we put everything together to obtain our final PPA for computing BCCs.

5.2.1 Spanning Tree Computation

To construct G^*, we first need a spanning tree of G, denoted by T. We present an $O(\delta)$-superstep BPPA for spanning tree computation as follows.

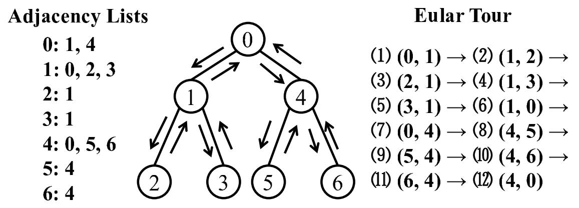
The algorithm performs *breadth-first search* (BFS) and computes a spanning tree over an unweighted graph G from a source vertex s. Each vertex v in G maintains two fields, the parent of v, denoted by $p(v)$; and the shortest-path distance (or BFS level) of v from s, denoted by $dist(v)$. Initially, only s is active with $p(s) = null$ and $dist(s) = 0$, and $dist(v) = \infty$ for all other v. In Superstep 1, s sends $\langle s, dist(s)\rangle$ to all its neighbors, and votes to halt. In each subsequent superstep, if a vertex v receives any message, it first checks whether v has been visited before (i.e., whether $dist(v) < \infty$): if not, it updates $dist(v) = dist(u) + 1$ and $p(v) = u$ with an arbitrary message $\langle u, dist(u)\rangle$ received, and sends $\langle v, dist(v)\rangle$ to all v's neighbors. Finally, v votes to halt.

When the algorithm terminates, we obtain a tree edge $(p(v), v)$ from each vertex $v \neq s$, which constitute a spanning tree rooted at s. It is easy to see that the algorithm is an $O(\delta)$-superstep BPPA. In the case if G is disconnected, we first compute $color(v)$ for each vertex v using the algorithm described in Section 4.1, and then pick the vertex s with $s = color(s)$ as the source for each CC. Since, multi-source BFS is done in parallel, the overall number of supersteps is still $O(\delta)$. For processing graphs with a large diameter δ, as mentioned in Section 4.2, the S-V algorithm can be extended to give an $O(\log n)$-superstep PPA to compute the spanning tree T.

5.2.2 Pre-order Numbering

Consider the three cases for constructing the edges of G^* in Section 5.1. In Case 1, we need the pre-order number of each vertex v (i.e., $pre(v)$) in the spanning tree T. We present an $O(\log n)$-superstep BPPA to compute the pre-order numbers for all vertices in T. We also propose a symmetric BPPA for computing post-order numbers.

To compute the pre-order numbering, we first compute a Euler tour of the spanning tree T. A Euler tour is a representation of a tree which is useful in many parallel graph algorithms. The tree is viewed as a directed graph, where each tree edge (u, v) is considered as two directed edges (u, v) and (v, u), and a Euler tour of the tree is simply a Eulerian circuit of the directed graph, i.e., a trail that visits every edge exactly once, and ends at the same vertex where it starts.


Adjacency Lists

0: 1, 4
1: 0, 2, 3
2: 1
3: 1
4: 0, 5, 6
5: 4
6: 4

Euler Tour

(1) (0, 1) → (2) (1, 2) →
(3) (2, 1) → (4) (1, 3) →
(5) (3, 1) → (6) (1, 0) →
(7) (0, 4) → (8) (4, 5) →
(9) (5, 4) → (10) (4, 6) →
(11) (6, 4) → (12) (4, 0)

Figure 6: Euler tour

We present an $O(\log n)$-superstep BPPA to compute the Euler tour of a tree T as follows.

BPPA for Computing Euler Tour. Assume that the neighbors of each vertex v are sorted according to their IDs, which is common for an adjacency list representation of a graph. For a vertex v, let $first(v)$ and $last(v)$ be the first and last neighbor of v in the sorted order; and for each neighbor u of v, if $u \neq last(v)$, let $next_v(u)$ be the neighbor of v next to u in the sorted adjacency list. We further define $next_v(last(v)) = first(v)$. Consider the example shown in Figure 6. For the adjacency list of vertex 4, we have $first(4) = 0$, $last(4) = 6$, $next_4(0) = 5$, $next_4(5) = 6$ and $next_4(6) = 0$.

If we translate $next_v(u) = w$ as specifying that the edge next to (u, v) is (v, w), we obtain a Euler tour of the tree. Referring to the example in Figure 6 again, where a Euler tour that starts and ends at vertex 0 is given. The next edge of $(2, 1)$ is $(1, 3)$, because $next_1(2) = 3$, while the next edge of $(6, 4)$ is $(4, 0)$ because $next_4(6) = 0$. In fact, starting from any vertex v and any neighbor u of v, $\langle(v, x = next_v(u)), (x, y = next_x(v)), (y, next_y(x)), \ldots, (u, v)\rangle$ defines a Euler tour.

We present a 2-superstep BPPA to construct the Euler tour as follows: In Superstep 1, each vertex v sends message $\langle u, next_v(u)\rangle$ to each neighbor u; in Supertep 2, each vertex u receives the message $\langle u, next_v(u)\rangle$ sent from each neighbor v, and stores $next_v(u)$ with v in u's adjacency list. When the algorithm finishes, for each vertex u and each neighbor v, the next edge of (u, v) is obtained as $(v, next_v(u))$.

The algorithm requires a constant number of supersteps, and in each superstep, each vertex v sends/receives $O(d(v))$ messages (each using $O(1)$ space). By implementing $next_v(.)$ as a hash table associated with v, we can obtain $next_v(u)$ in $O(1)$ expected time given u.

After obtaining the Euler tour of T, which is a cycle of edges, we break it at some edge to obtain a list of edges. We then compute the pre-order and post-order numbers of the vertices in T from the list, using the list ranking operation. Since our BPPAs for pre-order and post-order numbering are based on list ranking, we first introduce the concept of list ranking and present an $O(\log n)$-superstep BPPA for list ranking below.

BPPA for List Ranking. Consider a linked list $\mathcal{L}$ with n objects, where each object v is associated with a value $val(v)$ and a link to its predecessor $pred(v)$. The object v at the head of $\mathcal{L}$ has $pred(v) = null$. For each object v in $\mathcal{L}$, let us define $sum(v)$ to be the sum of the values of all the objects from v following the predecessor link to the head. The *list ranking* problem computes $sum(v)$ for each object v. If $val(v) = 1$ for each v in $\mathcal{L}$, then $sum(v)$ is simply the rank of v in the list, i.e., the number of objects preceding v plus 1.

In list ranking, the objects in $\mathcal{L}$ are given in arbitrary order. We may regard $\mathcal{L}$ simply as a directed graph consisting of a single simple path. Albeit simple, list ranking is an important problem in

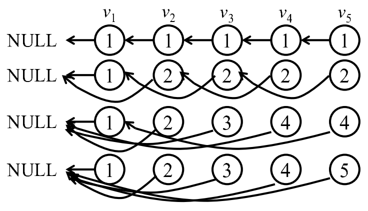


Figure 7: Illustration of BPPA for list ranking

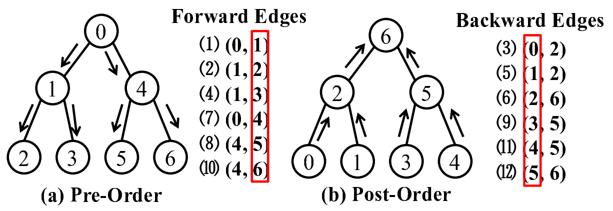


Figure 8: Pre-order & post-order

parallel computing because it serves as a building block to many other parallel algorithms.

We now describe our BPPA for list ranking. Initially, each vertex v assigns $sum(v) = val(v)$. Then in each round, each vertex v does the following: If $pred(v) \neq null$, v sets $sum(v) = sum(v) + sum(pred(v))$ and $pred(v) = pred(pred(v))$; otherwise, v votes to halt. The if-branch is accomplished in three supersteps: (1)v sends a message to $u = pred(v)$ requesting for the values of $sum(u)$ and $pred(u)$; (2)u sends back the requested values to v; and (3)v updates $sum(v)$ and $pred(v)$. This process repeats until $pred(v) = null$ for every vertex v, at which point all vertices vote to halt and we have $sum(v)$ as desired.

Figure 7 illustrates how the algorithm works. Initially, objects v_1–v_5 form a linked list with $sum(v_i) = val(v_i) = 1$ and $pred(v_i) = v_{i-1}$. Let us now focus on v_5. In Round 1, we have $pred(v_5) = v_4$ and so we set $sum(v_5) = sum(v_5) + sum(v_4) = 1 + 1 = 2$ and $pred(v_5) = pred(v_4) = v_3$. One can verify the states of the other vertices similarly. In Round 2, we have $pred(v_5) = v_3$ and so we set $sum(v_5) = sum(v_5) + sum(v_3) = 2 + 2 = 4$ and $pred(v_5) = pred(v_3) = v_1$. In Round 3, we have $pred(v_5) = v_1$ and so we set $sum(v_5) = sum(v_5) + sum(v_1)$=4 + 1=5 and $pred(v_5) = pred(v_1) = null$. We can prove by induction that in Round i, we set $sum(v_j) = \sum_{k=j-2^i+1}^{j} val(v_k)$ and $pred(v_j) = v_{j-2^i}$. Furthermore, each object v_j sends at most one message to $v_{j-2^{i-1}}$ and receives at most one message from $v_{j+2^{i-1}}$. The algorithm is a BPPA because it terminates in $\log n$ rounds, and each object sends/receives at most one message per round.

Let us assume that we already obtain the Euler tour $\mathcal{P}$ of the spanning tree starting from s, given by $\mathcal{P} = \langle(s, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k), (v_k, s)\rangle$. We break the cycle into a list by setting $pred(s, v_1) = null$. We now consider how to compute the pre-order and post-order numbers from the list using list ranking.

Pre-Order and Post-Order Numbering. Depth-first traversal of a tree T generates pre-order or post-order numbers for the vertices in a tree, and the numbers are useful in many applications, such as deciding the ancestor-descendant relationship of two tree vertices, which we discuss in Section 5.2.3. Let $pre(v)$ and $post(v)$ be the pre-order and post-order number of each vertex v in T, respectively. We present a BPPA for computing pre-order and post-order numbers from the Euler tour $\mathcal{P}$ of the tree T below.

We formulate a list ranking problem by treating each edge $e \in \mathcal{P}$ as a vertex and setting $val(e) = 1$. After obtaining $sum(e)$ for each $e \in \mathcal{P}$, we mark the edges in $\mathcal{P}$ as forward/backward edges using a two-superstep BPPA: in Superstep 1, each vertex $e = (u, v)$ sends $sum(e)$ to $e' = (v, u)$; in Superstep 2, each vertex $e' = (v, u)$ receives $sum(e)$ from $e = (u, v)$, sets e' itself as a forward edge if $sum(e') < sum(e)$, and a backward edge otherwise. In Figure 6, edge $(1, 2)$ is a forward edge because its rank (i.e., 2) is smaller than that of $(2, 1)$ (i.e., 3), while edge (4, 0) is a backward edge since its rank (i.e., 12) is larger than that of $(0, 4)$ (i.e., 7).

To compute $pre(v)$, we run a second round of list ranking by setting $val(e) = 1$ for each forward edge e in $\mathcal{P}$ and $val(e') = 0$ for each backward edge e'. Then, for each forward edge $e = (u, v)$, we get $pre(v) = sum(e)$ for vertex v. We set $pre(s) = 0$ for tree root s. For example, Figure 8(a) shows the forward edges (u, v) in the order in $\mathcal{P}$, where vertices are already labeled with pre-order numbers. Obviously, the rank of (u, v) gives $pre(v)$.

To compute $post(v)$, we run list ranking by setting $val(e) = 0$ for each forward edge e and $val(e') = 1$ for each backward edge e' in $\mathcal{P}$. Then, for each backward edge $e' = (v, u)$, we get $post(v) = sum(e')$ for vertex v. We set $post(s) = n - 1$ for tree root s, where n is the number of vertices in the tree. If n is not known, we can easily compute n using an aggregator in Pregel with each vertex providing a value of 1. (In the more general case where G is a forest, the aggregator counts the number of vertices for each tree/component). For example, Figure 8(b) shows the backward edges (v, u) in the order in $\mathcal{P}$, and vertices are relabeled with post-order numbers. Obviously, the rank of (v, u) gives $post(v)$.

The algorithm correctly computes $pre(v)/post(v)$ for all vertices v, because each vertex v in the tree (except root s) has exactly one parent u defined by the forward/backward edge $(u, v)/(v, u)$. Finally, the proof for BPPA follows directly from the fact that both Euler tour and list ranking can be computed by BPPAs.

5.2.3 *Ancestor-Descendant Query*

In Case 2 for constructing the edges of G^*, we need to decide whether two vertices u and v have ancestor-descendant relationship in the spanning tree T.

Let $pre(v)$ be the pre-order number of v and $nd(v)$ be the number of descendants of v in the tree. We show that if $pre(v)$ and $nd(v)$ is available for each vertex v in the tree, then an ancestor-descendant query can be answered in $O(1)$ time. Given u and v, following the definition of pre-order numbering, we have: u is an ancestor of v iff $pre(u) \leq pre(v) < pre(u)+nd(u)$. For vertex 1 in Figure 8(a), we have $pre(1) = 1$ and $nd(1) = 3$, and therefore any vertex v with $1 \leq pre(v)$<1+3 (i.e., vertices 1, 2 and 3) is a descendants of vertex 1.

We have presented a BPPA to compute $pre(v)$ in Section 5.2.2. We now show that $nd(v)$ can be obtained in the same process: for each forward edge $e = (u, v)$, we set $nd(v) = sum(e') - sum(e) + 1$ where e' is the backward edge (v, u). For tree root s, we set $nd(s) = n$. For example, we compute $nd(1) = sum(1, 0) - sum(0, 1) + 1 = 3 - 1 + 1 = 3$ for vertex 1 in Figure 8(a).

5.2.4 *Case 3 Condition Checking*

With the PPA/BPPA proposed in the previous subsections, Case 1 and Case 2 in Section 5.1 can now be checked in a constant number of supersteps by exchanging messages with neighbors. Case 3, however, is far more complicated to handle as vertices other than direct neighbors are involved.

We now develop a PPA for handling Case 3 as follows. We first need to compute two more fields for each vertex v, which are defined recursively as follows:

- $min(v)$: the minimum of (1)$pre(v)$, (2)$min(u)$ for all of v's children u, and (3)$pre(w)$ for all non-tree edges (v, w).

- $max(v)$: the maximum of (1)$pre(v)$, (2)$max(u)$ for all of v's children u, and (3)$pre(w)$ for all non-tree edges (v, w).

Let $desc(v)$ be the descendants of v (including v itself) and $\Gamma_{desc}(v)$ be the set of vertices connected to any vertex in $desc(v)$ by a non-tree edge. Intuitively, $min(v)$ (or $max(v)$) is the smallest (or largest) pre-order number among $desc(v) \cup \Gamma_{desc}(v)$.

In Case 3, (x, y) exists iff $x \in \Gamma_{desc}(u) - desc(v)$. Since x is not a descendant of $p(u)$, either $pre(x) < pre(p(u))$ which implies $min(u) < pre(p(u))$, or $pre(x) \geq pre(p(u)) + nd(p(u))$ which implies $max(u) \geq pre(p(u)) + nd(p(u))$. To summarize, Case 3 holds for u iff $min(u) < pre(p(u))$ or $max(u) \geq pre(p(u)) + nd(p(u))$.

When $pre(v)$, $nd(v)$, $min(v)$ and $max(v)$ are available for each vertex v, all the three cases can be handled using $O(1)$ supersteps. We now show how to compute $min(v)$ for each v by a PPA in $O(\log n)$ supersteps (computing $max(v)$ is symmetric).

For ease of presentation, let us simply use v to denote $pre(v)$. We further define $local(v)$ to be the minimum among v and all the neighbors connected to v by a non-tree edge. Note that $min(v)$ is just the minimum of $local(u)$ among all of v's descendants u.

We compute $min(v)$ in $O(\log n)$ rounds. At the beginning of the i-th round, each vertex $v = c \cdot 2^i$ (c is a natural number) maintains a field $global(c \cdot 2^i) = \min\{local(u) : c \cdot 2^i \leq u < (c+1)2^i\}$. Then in the $(i+1)$-th round, for each vertex $v = c \cdot 2^{i+1}$ we can simply update $global(v) = \min\{global(v), global(v + 2^i)\}$, i.e., merging the results from two consecutive segments of length 2^i. Here, each round can be done by a three-superstep PPA: (1) each v requests $global(v + 2^i)$ from $(v + 2^i)$; (2) $(v + 2^i)$ responds by sending $global(v+2^i)$ to v; (3) v receives $global(v+2^i)$ to update $global(v)$. Initially, $global(v) = local(v)$ for each v, and $local(v)$ can be computed similarly, by requesting $pre(u)$ from each neighbor u connected to v by a non-tree edge.

Given a vertex v, the descendants of v in T are $\{v, v+1, \ldots, v+nd(v) - 1\}$. At the beginning of the i-th round, we define $little(v)$ (and respectively, $big(v)$) to be the first (and respectively, the last) descendant that is a multiple of 2^i. Figure 9 illustrates the concept of $little(v)$ and $big(v)$.

We maintain the following invariant for each round:

$$min(v) = \min\{local(u) : u \in [v, little(v)) \cup [big(v), v + nd(v) - 1]\}. \quad (1)$$

Obviously, the correct value of $min(v)$ is computed when $little(v) = big(v)$, which must happen for some value of i as i goes from 0 to $\lfloor \log_2 n \rfloor$. We perform the following operations for each v in the i-th round, which maintains the invariant given by Equation (1):

- If $little(v) < big(v)$ and $little(v)$ is not a multiple of 2^{i+1}, set $min(v) = \min\{min(v), global(little(v))\}$, and then set $little(v) = little(v) + 2^i$.

- If $little(v) < big(v)$ and $big(v)$ is not a multiple of 2^{i+1}, set $min(v) = \min\{min(v), global(big(v) - 2^i)\}$, and then set $big(v) = big(v) - 2^i$.

We do not update $little(v)$ (or $big(v)$) if it is a multiple of 2^{i+1}, so that in the $(i + 1)$-th round it is aligned with $c \cdot 2^{i+1}$. We update $little(v)$, $big(v)$, and $min(v)$ by Pregel operations similar to those for updating $global(v)$.

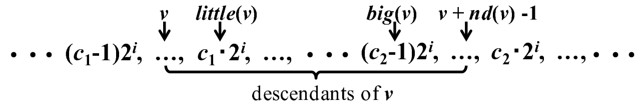
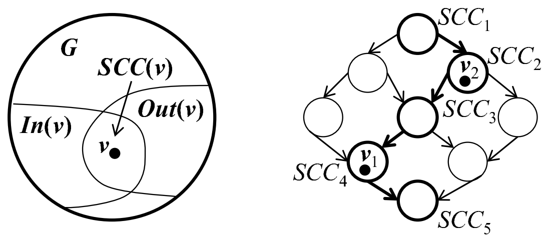

v
$little(v)$
$big(v)$
$v + nd(v)$ -1
$\cdots (c_1\text{-}1)2^i, \ldots, c_1 \cdot 2^i, \ldots, \cdots (c_2\text{-}1)2^i, \ldots, c_2 \cdot 2^i, \ldots, \cdots$
descendants of v

Figure 9: Illustration of computing $min(v)$



(a) $In(v)$, $Out(v)$ & $SCC(v)$ (b) DAG Path

Figure 10: Concepts in SCC computation

5.2.5 Integration of Building Blocks

Refer to Section 5.1 again, when computing the CCs of G^*, we only need to consider those vertices of G^* that correspond to the tree edges in T. This is because other vertices of G^* correspond to the non-tree edges e_2 of Case 1, and hence can be assigned to the CC of the corresponding e_1 later on.

We now integrate all the building blocks into a PPA for computing BCCs. The algorithm consists of a sequence of PPA tasks: (1)*HashMin*: to compute $color(v)$ for all $v \in V$ using the BPPA of Section 4.1; (2)*BFS* or *S-V*: to compute a spanning forest of G using the BPPA of Section 5.2.1 with sources $\{s \in V | color(s) = s\}$; alternatively, we may obtain the spanning forest using the S-V algorithm of Section 4.2, denoted by *S-V*; (3)*EulerTour*: to construct Euler tours from the spanning forest using the BPPA of Section 5.2.2; (4)*ListRank1*: to break each Euler tour into a list and mark each edge as forward/backward using list ranking (see Section 5.2.2); (5)*ListRank2*: using the edge forward/backward marks to compute $pre(v)$ and $nd(v)$ for each $v \in V$ using list ranking (see Section 5.2.2); (6)*MinMax*: to compute $min(v)$ and $max(v)$ for each $v \in V$ using the PPA of Section 5.2.4; (7)*AuxGraph*: to construct G^* using $pre(v)$, $nd(v)$, $min(v)$ and $max(v)$ information; this is a constant-superstep BPPA since all three cases for edge construction can be checked in a constant number of supersteps. (8)*HashMin2* or *S-V2*: to compute the CCs of G^* using *HashMin*, but only consider tree edges; alternatively, we may use the *S-V* algorithm for CC computation; (9)*Case1Mark*: to decide the BCCs of the non-tree edges in G^* using Case 1.

This algorithm is a PPA since each of its tasks is a BPPA/PPA.

6. STRONGLY CONNECTED COMPONENTS

In this section, we present two novel Pregel algorithms that compute *strongly connected component*s (**SCC**s) from a directed graph $G = (V, E)$. Let $SCC(v)$ be the SCC that contains v, and let $Out(v)$ (and $In(v)$) be the set of vertices that can be reached from v (and respectively, that can reach v) in G. Some PRAM algorithms [9, 4, 3] were designed based on the observation that $SCC(v) = Out(v) \cap In(v)$ (see Figure 10(a)). They compute $Out(v)$ and $In(v)$ by forward/backward BFS from source v that is randomly picked from G. This process then repeats on $G[Out(v) - SCC(v)]$, $G[In(v) - SCC(v)]$ and $G[V - (Out(v) \cup In(v))]$, where $G[X]$ denotes the subgraph of G induced by vertex set X. The correctness is guaranteed by the property that any remaining SCC must be in one of these subgraphs.

Our Contributions. It is difficult to translate the PRAM algorithms to work in Pregel. Thus, we design two Pregel algorithms based on label propagation. The first algorithm propagates the smallest vertex (ID) that every vertex has seen so far, while the second algorithm propagates multiple source vertices to speed up SCC computation. We also noticed a very recent algorithm that computes SCCs in Pregel [20], which shares a similar idea as our first algorithm. However, their algorithm performs label propagation for only one round, followed by a serial computation by the master machine, which is called FCS (Finishing Computations Serially). For processing large graphs, it is possible that the remaining graph after one round of computation is still too large to fit in the memory of a single machine. In contrast, we introduce graph decomposition, which allows us to run multiple rounds of label propagation. Moreover, our Optimization 2 described at the end of Section 6.1 performs a processing similar to FCS in [20], but it is fully distributed (by utilizing the property of recursive graph decomposition) instead of computation at the master machine. Finally, the idea used by our second SCC algorithm is new, which overcomes a weakness of our first algorithm.

Before describing our SCC algorithms, we first present a BPPA for graph decomposition which is used in our algorithms.

Graph Decomposition. Given a partition of V, denoted by V_1, V_2, $\ldots$, V_ℓ, we decompose G into $G[V_1]$, $G[V_2]$, $\ldots$, $G[V_\ell]$ in two supersteps (assume that each vertex v contains a label i indicating $v \in V_i$): (1)each vertex notifies all its in-neighbors and out-neighbors about its label i; (2)each vertex checks the incoming messages, removes the edges from/to the vertices whose label is different from its own label, and votes to halt.

6.1 Min-Label Algorithm

We first describe the Pregel operation for *min-label* propagation, where each vertex v maintains two labels $min_f(v)$ and $min_b(v)$.

Forward Min-Label Propagation. (1)Each vertex v initializes $min_f(v) = v$, propagates $min_f(v)$ to all v's out-neighbors, and votes to halt; (2)when a vertex v receives a set of messages from its in-neighbors $\Gamma_{in}(v)$, let min^* be the smallest message received, then if $min^* < min_f(v)$, v updates $min_f(v) = min^*$ and propagates $min_f(v)$ to all v's out-neighbors; v votes to halt at last. We repeat Step (2) until all vertices vote to halt.

Backward Min-Label Propagation. This operation is done after forward min-label propagation. The differences are that (1)initially, only vertices v satisfying $v = min_f(v)$ are active with $min_b(v) = v$, while for the other vertices u, $min_b(u) = \infty$; (2)each active vertex v propagates $min_b(v)$ towards all v's in-neighbors.

Both operations are BPPAs with $O(\delta)$ supersteps. After the forward and then backward min-label propagations, each vertex v obtains a label pair $(min_f(v), min_b(v))$. This labeling has the following property (the proof can be found in Appendix B of [1]):

LEMMA 1. *Let $V_{(i,j)} = \{v \in V : (min_f(v), min_b(v)) = (i, j)\}$. Then, (i)any SCC is a subset of some $V_{(i,j)}$, and (ii)$V_{(i,i)}$ is a SCC with color i.*

The **min-label algorithm** repeats the following operations: (1)forward min-label propagation; (2)backward min-label propagation; (3)an aggregator collects label pairs (i, j), and assigns a unique id $\mathcal{ID}$ to each $V_{(i,j)}$; then graph decomposition is performed to remove edges crossing different $G[V_{\mathcal{ID}}]$; finally, we mark each vertex v with label (i, i) to indicate that its SCC is found.

In each step, only unmarked vertices are active, and thus vertices do not participate in later rounds once its SCC is determined.

Each round of the algorithm refines the vertex partition of the previous round. Since all the three steps are BPPAs, each round of the min-label algorithm is a BPPA. The algorithm terminates once all vertices are marked.

The correctness of the algorithm follows directly from Lemma 1. We now analyze the number of rounds the min-label algorithm requires. Given a graph G, if we contract each SCC into a super-vertex, we obtain a DAG in which an edge directs from one super-vertex (representing a SCC, SCC_i) to another super-vertex (representing another SCC, SCC_j) iff there is an edge from some vertex in SCC_i to some vertex in SCC_j. Let $\mathcal{L}$ be the longest path length in the DAG, then we have the following bound (the proof can be found in Appendix B of [1]).

THEOREM 1. *The min-label algorithm runs for at most $\mathcal{L}$ rounds.*

The above bound is very loose, and often, more than one SCC is marked per DAG path in a round. We illustrate it using Figure 10(b), where there is a DAG path $P = \langle SCC_1, SCC_2, SCC_3, SCC_4, SCC_5\rangle$, and v_1 is the smallest vertex in G. Obviously, for any vertex v in SCC_4–SCC_5, $min_f(v) = v_1$. Since v_1 is picked as a source for backward propagation, for any vertex v in SCC_1–SCC_4, $min_b(v) = v_1$. Thus, any vertex $v \in SCC_4$ has label pair (v_1, v_1) and is marked. Now consider the sub-path before SCC_4, i.e., $\langle SCC_1, SCC_2, SCC_3\rangle$, and assume $v_2 \in SCC_2$ is the second smallest vertex in G, then a similar reasoning shows that SCC_2 is found as $V_{(v_2,v_2)}$. In this way, P is quickly broken into many subpaths, each can be processed in parallel, and hence in practice the number of rounds needed can be much less than $\mathcal{L}$.

In our implementation, we further perform two optimizations:

Optimization 1: Removing Trivial SCCs. If the in-degree or out-degree of a vertex v is 0, then v itself constitutes a trivial SCC and can be directly marked to avoid useless label propagation. We mark trivial SCCs before forward min-label propagation in each round.

We now describe a Pregel algorithm to mark all vertices with in-degree 0. Initially, each vertex v with in-degree 0 marks itself, sends itself to all out-neighbors and votes to halt. In subsequent supersteps, each vertex v removes its in-edges from the in-neighbors that appears in the incoming messages, and checks whether its in-degree is 0. If so, the vertex marks itself and sends itself to all out-neighbors. Finally, the vertex votes to halt. We also mark vertices with out-degree 0 in each superstep in a symmetric manner.

The algorithm takes only a small number of supersteps in practice (as shown by experiments in Section 7), since real world graphs (e.g., social networks) have a dense core, and the limited number of trivial SCCs only exist in the sparse boundary regions of the graphs. Furthermore, many vertices with zero in-degree/out-degree are marked as trivial SCCs in parallel in each superstep. On the other hand, removing trivial SCCs prevents them from participating in min-label propagation, which may otherwise degrade the algorithm effectiveness if some trivial SCC vertex has a small ID.

Optimization 2: Early Termination. We do not need to run the algorithm until all vertices are marked as a vertex of a SCC found. We also mark a vertex $v \in G[V_{\mathcal{ID}}]$ if $|V_{\mathcal{ID}}|$ is smaller than a threshold τ, so that all vertices in subgraph $G[V_{\mathcal{ID}}]$ remain inactive in later rounds. Here, $|V_{\mathcal{ID}}|$ is obtained using the aggregator of Step (3). We stop once all vertices are marked as a SCC/subgraph vertex. Then, we use one round of MapReduce to assign the subgraphs to different machines to compute the SCCs directly from each subgraph $G[V_{\mathcal{ID}}]$. Since each subgraph is small, its SCCs can be computed on a single machine using an efficient main memory algorithm, without inter-machine communication.

6.2 Multi-Label Algorithm

A real world graph usually has a giant SCC that contains the majority of its vertices, and it is desirable to find the giant SCC early (e.g., in the first round) so that we can terminate earlier by applying Optimization 2 mentioned above. However, the min-label algorithm may not find the giant SCC in the first round. For example, let the giant SCC be SCC_{max}, then if there is a vertex $v \notin SCC_{max}$ whose ID is smaller than all vertices in SCC_{max}, and if v links to a vertex in SCC_{max}, then SCC_{max} cannot be found in Round 1 by the min-label algorithm. On the other hand, the multi-label algorithm to be presented in this subsection almost always finds the giant SCC in the first round.

The multi-label algorithm aims to speed up SCC discovery and graph decomposition by propagating k source vertices in parallel, instead of just one randomly picked source as done by the min-label algorithm and existing algorithms [9, 4, 3, 20].

In this algorithm, each vertex v maintains two label sets $Src_f(u)$ and $Src_b(u)$. The algorithm is similar to the min-label algorithm, except that the min-label propagation operation is replaced with the *k-label* propagation operation described below:

Forward k-Label Propagation. Suppose that the current vertex partition is $V_1, V_2, \ldots, V_\ell$. (1)In Superstep 1, an aggregator randomly selects k vertex samples from each subgraph $G[V_i]$. (2)In Superstep 2, each source u initializes $Src_f(u) = \{u\}$ and propagates label u to all its out-neighbors, while each non-source vertex v initializes $Src_f(v) = \emptyset$. Finally, the vertex votes to halt. (3)In subsequent supersteps, if a vertex v receives a label $u \notin Src_f(v)$ from an in-neighbor, it updates $Src_f(v) = Src_f(v) \cup \{u\}$ and propagates u to all its out-neighbors, before voting to halt.

The backward k-label propagation is symmetric. Unlike the min-label algorithm where backward propagation is done after forward propagation, in the multi-label algorithm, we perform both forward and backward propagation in parallel.

The k-label propagation operation is also a BPPA with $O(\delta)$ supersteps, and when it terminates, each vertex v obtains a label pair $(Src_f(v), Src_b(v))$. This labeling has the following property (the proof can be found in Appendix B of [1]):

LEMMA 2. *Let $V_{(S_f,S_b)} = \{v \in V : (Src_f(v), Src_b(v)) = (S_f, S_b)\}$. Then, (i)any SCC is a subset of some $V_{(S_f,S_b)}$, and (ii)$V_{(S_f,S_b)}$ is a SCC if $S_f \cap S_b \neq \emptyset$.*

We now analyze the number of rounds required. In any round, we have ℓ subgraphs and thus around ℓk source vertices. Since we do not know ℓ, We only give a very loose analysis assuming $\ell = 1$ (i.e., there are only k sources). Furthermore, we assume that vertices are only marked because they form a SCC, while in practice Optimization 2 of Section 6.1 is applied to also mark vertices of sufficiently small subgraphs.

Suppose that we can mark $(1-\theta)n$ vertices as SCC vertices in each round, where $0 < \theta < 1$. Then, after i rounds the graph has $\theta^i n$ vertices, and in $O(\log_{1/\theta} n)$ rounds the graph is sufficiently small to allow efficient single-machine SCC computation. We now study the relationship between θ and k.

Assume that there are c SCCs in G: $SCC_1, SCC_2, \ldots, SCC_c$. Let n_i be the number of vertices in SCC_i and $p_i = n_i/n$. We analyze how many vertices are marked in expectation after one round. Note that if x sampled source vertices belong to the same SCC, then we actually waste $x-1$ samples. Our goal is to show that such waste is limited.

We define a random variable X that refers to the number of vertices marked. We also define an indicator variable X_i for each SCC SCC_i as follows: $X_i = 1$ if at least one sample belongs to SCC_i, and $X_i = 0$ otherwise. Let s_j be the j-th sample. We have

$$\begin{aligned} E[X_i] &= Pr\{X_i = 1\} = 1 - \prod_{j=1}^{k} Pr\{s_j \notin SCC_i\} \\ &= 1 - \prod_{j=1}^{k}(1-p_i) = 1-(1-p_i)^k. \end{aligned}$$

Note that $X = \sum_{i=1}^{c} n_i \cdot X_i$. According to the linearity of expectation, we have

$$\begin{aligned} E[X] &= \sum_{i=1}^{c} n_i \cdot E[X_i] = \sum_{i=1}^{c}[n_i - n_i(1-p_i)^k] \\ &= n - \sum_{i=1}^{c} n_i(1-p_i)^k = n - n\sum_{i=1}^{c} p_i(1-p_i)^k. \end{aligned}$$

In other words, $\theta = \sum_{i=1}^{c} p_i(1-p_i)^k$. Since the number of vertices remaining unmarked is θn, we want θ to be as small as possible. In fact, if the size of SCCs are biased, θ is small. This is because if there is a very large SCC, it is likely that some of its vertices are sampled as source vertices, and hence many vertices will be marked as being a vertex of the SCC.

The worst case happens when all the SCCs are of equal size, i.e., $p_i = 1/c$ for all i, in which case $\theta = (1-1/c)^k$. Since $(1-1/c) < 1$, θ decreases with k, but the rate of decrement depends on c. For example, when $c = 1000$, to get $\theta = 0.9$ we need to set $k = 100$. However, we note that real world graphs rarely have all SCCs with similar sizes, and the analysis is very loose. In practice, k can be much smaller even for very large c.

We now present a theorem that formalizes the above discussion (the proof can be found in Appendix B of [1]).

THEOREM 2. *If $p_i < \frac{2}{k+1}$ for all i, then $\theta \leq (1-1/c)^k$. Otherwise, $\theta = \sum_{i=1}^{c} p_i(1-p_i)^k < 1 - 1/k$.*

Finally, we emphasize that Theorem 2 is very loose: when there exists a SCC SCC_i with p_i much greater than $\frac{2}{k+1}$, θ is much smaller than $1 - 1/k$.

7. EXPERIMENTAL EVALUATION

We evaluate the performance of our algorithms over large real-world graphs. We ran all experiments on a cluster of 16 machines, each with 24 processors (two Intel Xeon E5-2620 CPU) and 48GB RAM. One machine is used as the master that runs only one working process (or simply, worker), while the other 15 machines act as slaves each running 10 workers. The connectivity between any pair of nodes in the cluster is 1Gbps.

All our algorithms were implemented in Pregel+[2], which is an open-source implementation of Pregel, though any Pregel-like system can be used to implement our algorithms. We remark that we did not use any optimization techniques in Pregel+, i.e., we used only the basic features of Pregel, as our aim is to test the performance of our algorithms in a general Pregel-like system. All the source codes of the algorithms discussed in this paper can be found at `http://www.cse.cuhk.edu.hk/pregelplus/download.html`.

Datasets. We used 10 real-world graph datasets, which are listed in Figure 11: (1)*BTC*[3]: a semantic graph converted from the Billion Triple Challenge 2009 RDF dataset [6]; (2)*LJ-UG*[4]: a network of

[2]http://www.cse.cuhk.edu.hk/pregelplus

[3]http://km.aifb.kit.edu/projects/btc-2009/

[4]http://konect.uni-koblenz.de/networks/livejournal-groupmemberships

Data	Type	$\|V\|$	$\|E\|$
BTC	undirected	164,732,473	772,822,094
LJ-UG	undirected	10,690,276	224,614,770
Facebook	undirected	59,216,214	185,044,032
USA	undirected	23,947,347	58,333,344
Euro	undirected	18,029,721	44,826,904
Twitter	directed	52,579,682	1,963,263,821
LJ-DG	directed	4,847,571	68,993,773
Pokec	directed	1,632,803	30,622,564
Flickr	directed	2,302,925	33,140,017
Patent	directed	3,774,768	16,518,948

Figure 11: Datasets

LiveJournal users and their group memberships; (3)*Facebook*[5]: a friendship network of the Facebook social network; (4)*USA*[6]: the USA road network; (5)*Euro*[7]: the European road network; (6)*Twitter*[8]: Twitter who-follows-who network based on a snapshot taken in 2009; (7)*LJ-DG*[9]: a friendship network of the LiveJournal blogging community; (8)*Pokec*[10]: a friendship network of the Pokec social network; (9)*Flickr*[11]: a friendship network of the Flickr social network; (10)*Patent*[12]: the US patent citation network.

7.1 Performance Comparison with GraphLab

As discussed in Section 2, GraphLab [14] (and PowerGraph [10]) only allows a vertex to access the states of its adjacent vertices and edges. As a result, it does not support algorithms in which a vertex needs to communicate with a non-neighbor, such as the S-V algorithm in Section 4.2, the list ranking algorithm in Section 5.2.2 and the algorithm presented in Section 5.2.4. Thus, we cannot implement our BCC algorithm in GraphLab. Moreover, we also cannot implement our SCC algorithms in GraphLab, because GraphLab does not support graph mutations, while the graph decomposition operation in our SCC algorithms involves edge deletion.

Due to the above-mentioned limitations, we only compare the performance of GraphLab with Pregel+ for the Hash-Min algorithm. We use GraphLab 2.2, which includes all the features of PowerGraph [10], and ran both GraphLab's asynchronous and synchronous modes. GraphLab's synchronous mode simulates Pregel, but due to the above-mentioned limitations, it is also difficult to implement the S-V, BCC and SCC algorithms in its synchronous mode.

Figure 12 reports the performance of Pregel+ and GraphLab when running Hash-Min over the small-diameter *BTC* graph (with skewed degree distribution) and the large-diameter *USA* road network (in which the degree of all vertices is small). The result shows that Pregel+ is significantly faster than GraphLab for processing the small-diameter *BTC* graph. For the large-diameter *USA* graph, Pregel+ is almost 3 times faster than the synchronous GraphLab, but is 1.6 times slower than the asynchronous GraphLab (for the reason given below).

For a large-diameter graph like *USA*, asynchronous execution is faster than its synchronous mode. This is because in asynchronous execution, the update to $min(v)$ is immediately visible to all other vertices, while in synchronous execution, the update is visible to

	BTC			USA		
	Pregel+	GraphLab		Pregel+	GraphLab	
		Sync	Async		Sync	Async
# of Supersteps	30	30	N/A	6262	6262	N/A
Computing Time (sec)	32.24	83.1	155	1011	2982	627

Figure 12: Pregel+ and GraphLab running Hash-Min

other vertices only at the next superstep, resulting in a slower convergence. However, for a small-diameter graph like *BTC*, the synchronous execution converges in only 30 supersteps; thus, even though the asynchronous mode can converge faster, the gain is not big enough to cover the overhead of data locking/unlocking in asynchronous execution.

Overall, the much superior performance of Pregel+ on small-diameter graphs with skewed degree distribution, and the reasonable performance of Pregel+ on large-diameter graphs, justify that Pregel+ is a good choice of distributed graph computing system for implementing our algorithms. In addition, the limitations of GraphLab discussed above make the implementation of certain categories of graph algorithms difficult using GraphLab, which further justifies the adoption of Pregel+ in our work.

Finally, we remark that this paper focuses on practical algorithms for Pregel-like systems rather than on the systems themselves, and we refer readers to our online report on a comprehensive comparison of existing systems including GraphLab, Giraph [2], GPS [19], and Pregel+: `http://www.cse.cuhk.edu.hk/pregelplus/papers/gSysComp.pdf`.

7.2 Performance of CC & BCC Algorithms

In Section 5 we proposed PPAs/BPPAs for a list of fundamental graph problems. Since they are used as building blocks in the PPA for computing BCCs, we also report their performance results as the steps of the BCC computation. Recall from Section 5.2.5 that the sequence of PPA tasks in the BCC computation include: (1)-(2)either *HashMin* + *BFS*, or *S-V*; (3)*EulerTour*; (4)*ListRank1*; (5)*ListRank2*; (6)*MinMax*; (7)*AuxGraph*; (8)either *HashMin2* or *S-V2*; (9)*Case1Mark*. Among them, Tasks (1) and (8) also report the performance of our two PPAs for computing CCs described in Section 4.

We report the per-task performance of BCC computation on the three *small-diameter* graphs *BTC*, *LJ-UG* and *Facebook* in Figure 13. Due to the small graph diameter, Hash-Min is much more efficient than S-V over these graphs. For example, Hash-Min finishes in 18 supersteps on *LJ-UG* and uses only 11.85 seconds. In contrast, S-V takes 58 supersteps and 142.24 seconds. Thus, the results verify that it is more efficient to compute CCs using Hash-Min when the graph diameter is small. We also give the total computational time of our BCC algorithm, and the results again show that using Hash-Min as a building block in the BCC computation achieves almost twice shorter total time than using S-V.

Next, we report the per-task performance of BCC computation on the two *large-diameter* road networks *USA* and *Euro* in Figure 14. Due to the large graph diameter, Hash-Min is very time-consuming. For example, it takes 1011.19 seconds and 6262 supersteps on *USA*. In contrast, S-V takes only 198 supersteps and 368.20 seconds. This again shows the difference between an $O(\delta)$-superstep PPA (e.g., Hash-Min) and an $O(\log n)$-superstep PPA (e.g., S-V). Similar behavior is also observed for CC computation over G^*, where HashMin2 uses 5437.72 seconds on *USA* while S-V2 on only 526.69 seconds. Overall, the S-V based BCC algorithm is 5.85 times faster than the Hash-Min based BCC algorithm on *USA*, and 4.17 times faster on *Euro*. This demonstrates the advantage of our S-V algorithm for processing large-diameter graphs.

[5] http://konect.uni-koblenz.de/networks/facebook-sg
[6] http://www.dis.uniroma1.it/challenge9/download.shtml
[7] http://www.dis.uniroma1.it/challenge9/download.shtml
[8] http://konect.uni-koblenz.de/networks/twitter_mpi
[9] http://snap.stanford.edu/data/soc-LiveJournal1.html
[10] http://snap.stanford.edu/data/soc-pokec.html
[11] http://konect.uni-koblenz.de/networks/flickr-growth
[12] http://snap.stanford.edu/data/cit-Patents.html

Task	BTC		LJ-UG		Facebook	
	# of Steps	Comp. Time	# of Steps	Comp. Time	# of Steps	Comp. Time
HashMin	30	32.24 s	18	11.85 s	16	37.10 s
BFS	31	20.56 s	19	8.61 s	17	10.14 s
S-V	86	449.97 s	58	142.24 s	72	337.02 s
EulerTour	3	14.26 s	3	2.26 s	3	7.56 s
ListRank1	49	544.71 s	53	97.98 s	57	408.79 s
ListRank2	49	541.86 s	53	98.59 s	57	411.17 s
MinMax	46	35.88 s	50	8.54 s	54	20.42 s
AuxGraph	4	58.05 s	4	21.94 s	4	22.52 s
HashMin2	34	42.91 s	11	21.04 s	16	43.31 s
S-V2	72	443.16 s	58	138.59 s	86	385.86 s
Case1Mark	4	35.62 s	4	20.83 s	4	13.27 s
Total Time (CC by HashMin)	1326.09 s		291.64 s		974.28 s	
Total Time (CC by S-V)	2123.51 s		530.97 s		1606.61 s	

Figure 13: CC/BCC performance on BTC, LJ-UG, & Facebook

Task	USA		Euro	
	# of Steps	Comp. Time	# of Steps	Comp. Time
HashMin	6262	1011.19 s	4896	692.39 s
BFS	6263	964.11 s	4897	639.78 s
S-V	198	368.20 s	212	340.19 s
EulerTour	3	3.04 s	3	2.42 s
ListRank1	55	203.05 s	55	165.89 s
ListRank2	55	197.78 s	55	160.49 s
MinMax	52	16.65 s	52	12.77 s
AuxGraph	4	12.29 s	4	9.89 s
HashMin2	7365	5437.72 s	2836	2935.28 s
S-V2	226	536.69 s	184	414.20 s
Case1Mark	4	2.90 s	4	1.94 s
Total Time (CC by HashMin)	7848.73 s		4620.85 s	
Total Time (CC by S-V)	1340.60 s		1107.79 s	

Figure 14: CC/BCC performance on USA & Euro

It might be argued that computing the CCs of a road network in Pregel is not important, as road networks are usually connected and not very large. However, some spatial networks are huge in size, such as the triangulated irregular network (TIN) that models terrain, where CC computation is useful when we want to compute the islands given a specific sea level. Also, CC computation is a critical building block in our PPA for computing BCCs, and finding BCCs of a spatial network is important for analyzing its weak connection points.

7.3 Performance of SCC Algorithms

We now report the performance of our min-label and multi-label algorithms for computing SCCs on the directed graphs.

7.3.1 Performance of Min-Label Algorithm

Before describing the results, we first review the sequence of tasks performed in each round of our min-label algorithm: (1)*Opt 1*: this task removes trivial SCCs as described in Optimization 1 of Section 6.1; (2)*MinLabel*: forward min-label propagation followed by backward min-label propagation; (3)*GDecom*: this task uses an aggregator to collect label pairs $(min_f(u), min_b(u))$ and assigns a new $\mathcal{ID}$ to each pair, sets the $\mathcal{ID}$ of each vertex u according to u's $(min_f(u), min_b(u))$, marks each vertex u with $min_f(u) = min_b(u)$ as being in a SCC, and performs graph decomposition using the algorithm described at the beginning of Section 6. Recall that we do not decompose a subgraph if its size (decided by number of vertices) is smaller than a user-defined threshold τ.

Round	Task	# of Steps	Comp. Time	Max Size
1	**Opt 1**	18	9.41 s	238,986
	MinLabel	15 + 14	75.80 s	
	GDecom	3	76.86 s	
2	**Opt 1**	5	0.81 s	22
	MinLabel	36 + 75	18.73 s	
	GDecom	3	1.74 s	
3	**Opt 1**	3	0.46 s	2
	MinLabel	6 + 5	2.02 s	
	GDecom	3	0.44 s	
4	**Opt 1**	1	0.21 s	0
	MinLabel	3 + 3	0.96 s	
	GDecom	3	0.50 s	
Total			187.94 s	

Figure 15: Min-label performance on Twitter ($\tau = 0$)

Round	Task	# of Steps	Comp. Time	Max Size
1	**Opt 1**	16	2.73 s	51,697
	MinLabel	14 + 16	14.91 s	
	GDecom	3	7.24 s	
2	**Opt 1**	4	0.34 s	81
	MinLabel	8 + 9	1.91 s	
	GDecom	3	0.72 s	
3	**Opt 1**	3	0.22 s	29
	MinLabel	6 + 7	1.08 s	
	GDecom	3	0.40 s	
4	**Opt 1**	1	0.14 s	12
	MinLabel	4 + 4	1.01 s	
	GDecom	3	0.32 s	
Total			31.02 s	

Figure 16: Min-label performance on LJ-DG ($\tau = 0$)

We first compute the SCCs of the largest graph, *Twitter*, where we only mark a vertex when its SCC is determined (i.e., τ=0). Figure 15 reports the number of supersteps and the computational time taken by each task. The last column "Max Size" shows the maximum $|V_{\mathcal{ID}}|$ among those subgraphs, $G[V_{\mathcal{ID}}]$, that are not marked as a SCC after each round, and all SCCs are found when *Max-Size* becomes 0. As Figure 15 shows, the min-label algorithm takes only 4 rounds to compute all the SCCs over *Twitter*, which demonstrates that in practice the min-label algorithm requires much less than $\mathcal{L}$ rounds given in Theorem 1. Besides, the min-label propagation operations take only a small number of supersteps due to the small graph diameter. For example, in Round 1, forward propagation takes only 15 supersteps, followed by a 14-superstep backward propagation. The total computational time is only 187.94 seconds for a graph with almost 2 billion edges, which is very efficient.

Note that after Round 2 in Figure 15, the largest unmarked subgraph has size merely 22. Therefore, another option is to distribute these small subgraphs to different machines for single-machine SCC computation using MapReduce. Thus, we also run our min-label algorithm over *Twitter* using $\tau = 50,000$, so that a subgraph is marked to avoid further decomposition once it contains less than 50,000 vertices. All vertices are marked after 2 rounds, and the performance is similar to those in Figure 15. We then run a MapReduce job to compute the SCCs of the marked subgraphs, which takes 199 seconds.

For the three relatively smaller graphs, *Pokec*, *Flickr* and *Patent*, the min-label algorithm with $\tau = 0$ finds all the SCCs in less than 4 rounds, and uses 18.09, 17.88, and 2.34 seconds, respectively. The detailed results are reported in Appendix C [1] due to limited space. However, we remark that the min-label algorithm with $\tau = 0$ is not always able to find all the SCCs for an arbitrary graph. One example is the *LJ-DG* datasets, the performance of which is shown in Figure 16 (only the results of the first 4 rounds are shown). In

Round	Task	# of Steps	Comp. Time	Max Size
1	**Opt 1**	18	8.07 s	238,986
	MultiLabel	17	423.85 s	
	GDecom	3	102.98 s	
2	**Opt 1**	5	0.81 s	206,319
	MultiLabel	5	0.55 s	
	GDecom	3	0.41 s	
3	**Opt 1**	1	0.16 s	206,292
	MultiLabel	6	0.94 s	
	GDecom	3	0.38 s	
MapReduce			181 s	

Figure 17: Multi-label performance on Twitter ($\tau = 50,000$)

Round	Task	# of Steps	Comp. Time	Max Size
1	**Opt 1**	16	2.66 s	51,697
	MultiLabel	19	27.02 s	
	GDecom	3	6.30 s	
2	**Opt 1**	5	0.74 s	50,706
	MultiLabel	6	0.71 s	
	GDecom	3	0.25 s	
3	**Opt 1**	1	0.13 s	50,629
	MultiLabel	8	0.80 s	
	GDecom	3	0.36 s	
MapReduce			26 s	

Figure 18: Multi-label performance on LJ-DG ($\tau = 50,000$)

fact, for the subsequent three rounds, “Max Size” decreases slowly as 11, 10 and 9. On the contrary, running min-label algorithm on *LJ-DG* using $\tau = 50,000$ takes only 2 rounds to mark all vertices, followed by a MapReduce job that computes the SCCs of the marked subgraphs in 27 seconds.

7.3.2 Performance of Multi-Label Algorithm

We now report the performance of our multi-label algorithm. For the parallel forward and backward k-label propagation, we fix $k = 10$. We also set $\tau = 50,000$ and subgraphs with less than $50,000$ vertices are not further decomposed. The performance of the multi-label algorithm on the two larger graphs, *Twitter* and *LJ-DG*, are shown in Figures 17 and 18. We can see that although Round 1 bounds the maximum unmarked subgraph size to a relatively small number, “Max Size” decreases slowly in the later rounds and we cannot afford to run till it gets smaller than $50,000$. However, the subgraphs are small enough to be assigned to different machines for local SCC computation using MapReduce, and the final round of MapReduce postprocessing is efficient for both graphs. We obtain similar results for the datasets *Pokec*, *Flickr* and *Patent*, and the results are presented in Appendix C [1].

Unlike the min-label algorithm for which we can afford to run until termination, the multi-label algorithm finds at most k SCCs in each round, and is only effective in earlier rounds when there are large SCCs. However, as discussed at the beginning of Section 6.2, the multi-label algorithm almost always finds the largest SCC in the first round, which is more desirable than the min-label algorithm. Thus, in applications where only the largest SCC (also called the giant SCC) is needed, the multi-label algorithm will be a better choice; in applications where all SCCs are needed, running multi-label algorithm for Round 1 followed by min-label algorithm for the subsequent rounds can be a good choice.

8. CONCLUSIONS

We proposed efficient distributed algorithms for computing three fundamental graph connectivity problems, namely CC, BCC, and SCC. Specifically, we defined the notion of PPA to design Pregel algorithms that have guaranteed performance, i.e., requiring only linear space, communication and computation per iteration, and only $O(\log n)$ or $O(\delta)$ iterations of computation. Experiments on large real-world graphs verified that our algorithms have good performance in shared-nothing parallel computing platforms.

9. REFERENCES

[1] *Online Appendix:* *`www.cse.cuhk.edu.hk/pregelplus/ppaApp.pdf`*.

[2] C. Avery. Giraph: Large-scale graph processing infrastructure on hadoop. *Proceedings of the Hadoop Summit. Santa Clara*, 2011.

[3] J. Barnat, J. Chaloupka, and J. van de Pol. Improved distributed algorithms for scc decomposition. *Electronic Notes in Theoretical Computer Science*, 198(1):63–77, 2008.

[4] J. Barnat and P. Moravec. Parallel algorithms for finding sccs in implicitly given graphs. In *Formal Methods: Applications and Technology*, pages 316–330. Springer, 2007.

[5] J. Cheng, S. Huang, H. Wu, and A. W.-C. Fu. Tf-label: a topological-folding labeling scheme for reachability querying in a large graph. In *SIGMOD Conference*, pages 193–204, 2013.

[6] J. Cheng, Y. Ke, S. Chu, and C. Cheng. Efficient processing of distance queries in large graphs: a vertex cover approach. In *inproceedingsSIGMOD*, pages 457–468. ACM, 2012.

[7] M. Dayarathna and T. Suzumura. A first view of exedra: a domain-specific language for large graph analytics workflows. In *WWW (Companion Volume)*, pages 509–516, 2013.

[8] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[9] L. K. Fleischer, B. Hendrickson, and A. Pınar. On identifying strongly connected components in parallel. In *Parallel and Distributed Processing*, pages 505–511. Springer, 2000.

[10] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *OSDI*, pages 17–30, 2012.

[11] U. Kang, C. E. Tsourakakis, and C. Faloutsos. Pegasus: A peta-scale graph mining system implementation and observations. In *ICDM*, pages 229–238. IEEE, 2009.

[12] H. J. Karloff, S. Suri, and S. Vassilvitskii. A model of computation for mapreduce. In *SODA*, pages 938–948, 2010.

[13] S. Lattanzi, B. Moseley, S. Suri, and S. Vassilvitskii. Filtering: a method for solving graph problems in mapreduce. In *SPAA*, pages 85–94, 2011.

[14] 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*, 5(8):716–727, 2012.

[15] 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 *SIGMOD Conference*, pages 135–146, 2010.

[16] L. Quick, P. Wilkinson, and D. Hardcastle. Using pregel-like large scale graph processing frameworks for social network analysis. In *ASONAM*, pages 457–463, 2012.

[17] V. Rastogi, A. Machanavajjhala, L. Chitnis, and A. D. Sarma. Finding connected components in map-reduce in logarithmic rounds. *ICDE*, 0:50–61, 2013.

[18] J. H. Reif. Depth-first search is inherently sequential. *Information Processing Letters*, 20(5):229–234, 1985.

[19] S. Salihoglu and J. Widom. Gps: a graph processing system. In *SSDBM*, page 22, 2013.

[20] S. Salihoglu and J. Widom. Optimizing graph algorithms on pregel-like systems. *PVLDB*, 7(7):577–588, 2014.

[21] Y. Shiloach and U. Vishkin. An o(log n) parallel connectivity algorithm. *J. Algorithms*, 3(1):57–67, 1982.

[22] S. Suri and S. Vassilvitskii. Counting triangles and the curse of the last reducer. In *WWW*, pages 607–614. ACM, 2011.

[23] Y. Tao, W. Lin, and X. Xiao. Minimal mapreduce algorithms. In *SIGMOD Conference*, pages 529–540, 2013.

[24] R. E. Tarjan and U. Vishkin. An efficient parallel biconnectivity algorithm. *SIAM Journal on Computing*, 14(4):862–874, 1985.

[25] J. C. Wyllie. The complexity of parallel computations. Technical report, Cornell University, 1979.