Practical Pregel Algorithms for Massive Graphs

Preliminary version: long version can be found in
https://www.dropbox.com/sh/m3rfyrslir81gvk/qhH2BNCjkg

September 13, 2013

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, a parallel computing model for graph processing in the Cloud. We first define a class of **practical Pregel algorithm**s (**PPA**s), which require only linear space, communication and computation per iteration, and logarithmic number of iterations. Next, we propose PPAs for a list of fundamental graph problems, including *connected components*, *spanning tree*, *Euler tour*, *list ranking*, and *pre/post-ordering*. Then, we demonstrate how these basic operations are used as building blocks to solve other graph problems. In particular, we develop novel PPAs for computing *biconnected components* and *strongly connected components*. We verified the efficiency of our algorithms on large real world graphs in a cluster with 60 machines.

1. INTRODUCTION

Massive graphs are becoming more and more common in recent years particularly thanks to the popularity of online social networks and the ubiquity of mobile communication networks. There has been a lot of interest in conducting effective analysis over these networks (modeled as graphs); however, efficient processing of massive graphs is challenging for at least the following two reasons. First, massive graphs are often too large to fit into the memory resource of a single computer. Second, due to the sheer volume of massive graphs, it is too expensive to run a merely linear-time algorithm in a single computer. In fact, it is shown that operations like BFS or t-step neighborhood over Facebook network and the Web graph cannot be finished in a reasonable amount of time [3].

To address these challenges, researchers have been focusing on developing efficient algorithms on *shared-nothing* parallel computing platforms, which is a vital step towards big data analysis. Such parallel computing solutions are attractive because they do not rely on any single powerful computer that may not be affordable to many researchers and even companies; instead, they use only commodity computers that are available in considerable numbers in many research institutes, or can be rented from Cloud service providers (e.g., Amazon AWS) with low costs.

Google proposed a vertex-oriented computational model for processing graphs in Cloud platforms, called Pregel [17], which was shown to be more suitable for solving graph problems than the popular MapReduce model [7, 10, 17, 18]. However, existing work on Pregel-based graph algorithms (e.g., diameter estimation, degrees of separations, triangle counting, k-core and k-truss) [18] is rather ad hoc, which looks more like a demonstration of how the Pregel model can be adopted to solve graph problems, but lacks an analysis on the complexity.

In this paper, we define a class of Pregel-based algorithms that satisfy a set of rigid, but practical, constraints on various performance metrics, namely **Practical Pregel Algorithm**s (**PPA**s): *(1) linear space usage, (2) linear computation cost per round, (3) linear communication cost per round, and (4) at most logarithmic number of rounds*. A similar but stricter set of constraints was proposed for the MapReduce model recently [25]. In contrast to our requirement of logarithmic number of rounds, their work demands a constant number of rounds only, 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 [28], where n is the number of vertices. This bound also applies to many other basic graph problems such as connected components and spanning tree [22].

Under the PPA framework, we develop algorithms for a number of fundamental graph problems, including *list ranking*, *connected components* (**CC**s), *breadth-first search* (**BFS**), *spanning tree*, *Euler tour*, *pre/post-order traversal*, *bi-connected components* (**BCC**s), and *strongly connected components* (**SCC**s). These problems themselves have numerous important applications, and their solutions are vital 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 [8].

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 [20] and hence it cannot be applied to design parallel algorithms for computing BCCs and SCCs. So far, we are not aware of any existing work on BCCs or SCCs based on a shared-nothing architecture. We first demonstrate how PPAs can be designed for the more basic problems (e.g., list ranking, spanning tree), and then show how these basic algorithms are applied as building blocks to design non-trivial Pregel algorithms for computing BCCs and SCCs.

Apart from the proposal of PPA and the development of a set of

fundamental graph algorithms under the PPA framework, another important contribution of our work is *the handling of skewed vertex degree distribution for workload balancing*. Most real world large graphs follow a power-law vertex degree distribution, which leads to a bottleneck in parallel computing due to the skewed workload caused by high-degree vertices. Though obvious, this problem has not been addressed in any existing Pregel graph algorithms that we are aware of, and also not addressed in many MapReduce graph algorithms [24, 15, 19].

We propose a simple and effective graph transformation technique for balancing the workload caused by high-degree vertices. The main idea is to transform the original degree-skewed graph into one whose largest degree is bounded. We show that the result of the original graph can be straightforwardly derived from the result computed over the new degree-bounded graph. This technique can be applied to any parallel computing model including Pregel and MapReduce.

We evaluate the performance of our Pregel algorithms using a number of real world graphs with up to hundreds of millions of vertices and billions of edges, as well as a number of other moderate-sized graphs popularly used by existing works on graph problems. The results show that our algorithms are efficient in processing massive graphs.

The rest of this paper is organized as follows. Section 2 reviews Pregel and other related work. Section 3 defines PPA and presents PPAs for a set of fundamental graph problems. Sections 4-6 discuss the design of Pregel algorithms for CCs, BCCs and SCCs. Then, Section 7 discusses our load balancing technique. Section 8 reports the experimental results and Section 9 concludes the paper.

2. PREGEL AND RELATED WORK

Pregel [17]. Pregel is designed based on the bulk synchronous parallel model. It partitions the vertex set among machines, where each vertex is stored with its edges. A Pregel algorithm proceeds in a sequence of iterations, called *supersteps*. During a superstep, Pregel invokes a user-defined $compute()$ function for each vertex, which processes the incoming messages sent by other vertices in the previous superstep, sends messages to other vertices (e.g., via outgoing edges), and probably deactivates the vertex (i.e., the vertex votes to halt). A deactivated vertex is reactivated if it receives any message in a subsequent superstep. A Pregel algorithm terminates when all vertices vote to halt.

A Pregel algorithm can proceed in *rounds* of k different operations $op_1, op_2, \ldots, op_k$. This is achieved by branching on $t \bmod k$ for superstep t: if $t \bmod k = i$, $compute()$ performs op_{i+1}.

Besides the $compute()$ function, Pregel also supports aggregators. Aggregators are a mechanism for global communication. Each active vertex can provide a value to an aggregator in superstep t. The system combines those values so that the resulting value is made available to all vertices in superstep $(t + 1)$.

MapReduce [12]. MapReduce has been adopted to solve many graph problems [24, 27, 11, 4, 15, 16, 14, 19, 29, 2], most of them use the divide-and-conquer (D&C) technique: they decompose the computation over a large input graph into smaller problem instances, each of which can be solved on a single machine. Examples include algorithms for triangle counting [24], maximum clique computation [29] and minimum spanning tree [16]. Recently, four MapReduce algorithms were proposed for finding CCs based on message passing rather than D&C [19]. However, MapReduce requires passing the entire state of a graph from one stage to the next. On the contrary, Pregel keeps vertices and edges on the machine that performs the computation, and uses network transfers only for messages. As a result, Pregel can incur much less communication cost for solving graph problems. In addition, many of the problems studied in this paper, such as BCCs and SCCs, are non-trivial and have not been studied under the MapReduce model.

PRAM. The PRAM model assumes that there are many processors and a shared memory. PRAM algorithms have been proposed for computing CCs [22], BCCs [26], and SCCs [13, 6, 5]. 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.

External-Memory (EM). When the input graph cannot fit in main memory, EM algorithms are used to reduce random I/O cost. EM algorithms were proposed for computing CCs and BCCs [9], and also SCCs [30, 23]. However, these EM algorithms are sequential algorithms, which can be too costly for processing massive graphs and are not suitable for Cloud environments.

3. PRACTICAL PREGEL ALGORITHMS

We first 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 define the set of *neighbors* of a vertex v as $\Gamma(v) = \{u : (v, u) \in E\}$ and the *degree* of v as $d(v) = |\Gamma(v)|$. For a directed graph, we define the set of *in-neighbors* and *out-neighbors* of v as $\Gamma_{in}(v) = \{u : (u, v) \in E\}$ and $\Gamma_{out}(v) = \{u : (v, u) \in E\}$, and the *in-degree* and *out-degree* of v as $d_{in}(v) = |\Gamma_{in}(v)|$ and $d_{out}(v) = |\Gamma_{out}(v)|$, respectively.

Assume that all vertices in G are assigned a unique ID. For convenience of discussion, we simply use v to refer to the ID of vertex v in this paper, and thus expression $u < v$ means that u's vertex ID is smaller than v's. We define the *color* of a component (CC, BCC or SCC) 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 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. For most real world large graphs, they have a diameter δ smaller than $\log n$, especially for social networks due to the small world phenomenon. Thus, we assume $O(\delta) = O(\log n)$ in this paper and consider algorithms requiring $O(\delta)$ rounds also satisfying Constraint 4.

For some problems the per-vertex requirements of BPPA can be too strict, and we can only achieve *overall linear space usage*,

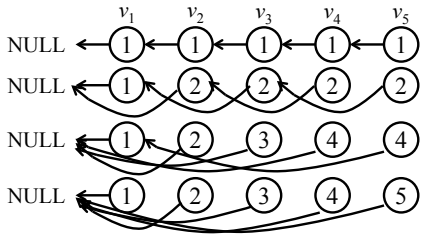


Figure 1: Illustration of BPPA for list ranking

computation and communication cost (still in $O(\log n)$ rounds). We call a Pregel algorithm satisfying these constraints simply as a **practical Pregel algorithm** (**PPA**).

We now demonstrate how some fundamental graph problems can be solved by BPPAs.

3.1 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 $\mathcal{L}$, the *list ranking* problem computes the summed value of v, denoted by $sum(v)$, as the sum of the values of all the objects from v following the predecessor link to the head. 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 in the list plus 1 (for v itself).

In list ranking, the objects in $\mathcal{L}$ are given in arbitrary order. A logical view of $\mathcal{L}$ is simply a directed graph consisting of a single simple path. Albeit simple, list ranking is well-recognized as an important problem in parallel computing because it serves as a building block to many parallel algorithms. We propose a BPPA for list ranking as follows.

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 steps: (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 1 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 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 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 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.

3.2 Connected Components

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. A MapReduce algorithm, called Hash-Min, was also proposed to implement this idea recently [19]. Here, we propose a BPPA counterpart as follows.

In Superstep 1, each vertex v initializes $min(v)$ as the smallest vertex in $(\{v\} \cup \Gamma(v))$, sends $min(v)$ to all v's neighbors and votes to halt. In subsequent supersteps, each 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 are also proposed in [19] for computing CCs, all of which can be translated into Pregel. 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, both the communication and computation are highly skewed.

3.3 BFS and Spanning Tree

We now present an $O(\delta)$-superstep BPPA that performs BFS and computes a spanning tree over an unweighted graph G from a source vertex s. We assume that the input graph G is connected; otherwise, we first compute $color(v)$ for each vertex v using the algorithm described in Section 3.2, and then pick the vertex s with $s = color(s)$ as the source for each CC. The overall number of supersteps is still $O(\delta)$.

Each vertex v maintains two fields, the parent of v, denoted by $p(v)$; and the shortest-path distance of v from s, denoted by $dist(v)$. We initialize $p(s) = null$ and $dist(s) = 0$, and $dist(v) = \infty$ for all other v. In Superstep 1, we send $\langle s, dist(s)\rangle$ to all s's neighbors, and s volts to halt. In each subsequent superstep, if a vertex v receives any message $\langle u, dist(u)\rangle$, it first checks whether v has been visited before (i.e., whether $dist(v) < \infty$): if so, it votes to halt; otherwise, it updates $dist(v) = dist(u) + 1$ and $p(v) = u$ with an arbitrary message $\langle u, dist(u)\rangle$ received, sends $\langle v, dist(v)\rangle$ to all v's neighbors, and then votes to halt.

When the algorithm terminates, the tree edges $(p(v), v)$ are obtained, which constitute a spanning tree rooted at s. It is easy to see that the algorithm is a BPPA using $O(\delta)$ supersteps.

3.4 Euler Tour

A Euler tour is a representation of a tree and a useful building block in many parallel graph algorithms [26]. 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.

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. Consider the example shown in Figure 2. For the adjacency list of vertex 4, we have $first(4) = 0$, $last(4) = 6$, $next_4(0) = 5$ and $next_4(5) = 6$.

We present a BPPA to construct the Euler tour as follows. For each vertex v and for each neighbor u of v, we define a field $f_v(u)$: $f_v(u) = w$ specifies that the edge next to (u, v) is (v, w). We compute $f_v(u)$ as follows: if $v \neq last(u)$, then $f_v(u) = next_u(v)$; otherwise, $f_v(u) = first(u)$. Then, starting from any vertex v and any neighbor u of v, $\langle (u, x = f_v(u)), (x, y = f_u(x)), (y, f_x(y)),$

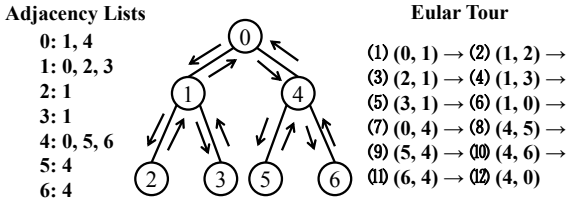


Figure 2: Euler tour

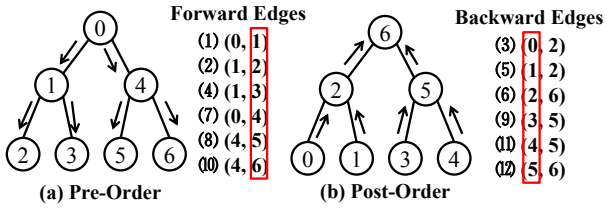


Figure 3: Pre-order & post-order

$\ldots, (v, u)\rangle$ defines a Euler tour. Referring to the example in Figure 2 again, where a Euler tour that starts and ends at vertex 0 is given. The next edge of $(2, 1)$ is $(1, 3)$, because $f_1(2) = 3$, while the next edge of $(6, 4)$ is $(4, 0)$ because $f_4(6) = 0$.

The above operation can be done in three supersteps: (1) each vertex v sends the message "v" to each neighbor u; (2) each vertex u checks whether $v = last(u)$ for each incoming message "v", and sends $next_u(v)$ back to v if $v \neq last(u)$ or sends $first(u)$ if $v = last(u)$; (3) each vertex v sets $f_v(u)$ to be the value in the message sent from u in Superstep (2), and votes to halt.

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. In Section 7, we will present a graph transformation technique that eliminates high-degree vertices and bounds the degree of each vertex by a small constant (e.g. 100), and thus getting $next_v(u)$ for all v's neighbors u can be viewed as an $O(1)$-time operation. Therefore, the algorithm can be counted as a BPPA.

3.5 Pre-Order and Post-Order Traversal

Graph traversals (e.g., BFS and DFS) often define a tree, and assigning pre-order/post-order numbers to vertices in the tree is useful in many applications and graph solutions. Let $pre(v)$ and $post(v)$ be the pre-order and post-order number of each vertex v in the tree, respectively. We present a BPPA for traversing a tree starting from a source vertex s in pre-order/post-order as follows.

We first compute the Euler tour $\mathcal{P}$ of the tree starting from s, given by $\mathcal{P} = \langle (s, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k), (v_k, s)\rangle$. Next, 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: (1) each vertex $e = (u, v)$ sends $sum(e)$ to $e' = (v, u)$; (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 2, 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 3(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 3(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.

3.6 Ancestor-Descendant Query

In a tree structure rooted at a source vertex s (e.g., the BFS/DFS tree of a graph starting from s), we often need to know whether a vertex u is an ancestor of another vertex v. The ancestor-descendant query needs to be answered in real time in many situations (e.g., reachability querying).

Let $pre(v)$ be the pre-order number of v and $nd(v)$ be the number of descendants of v in the tree. We describe a BPPA to compute $pre(v)$ and $nd(v)$ for each vertex v in a tree rooted at s, so that 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 3(a), we have $pre(1) = 1$ and $nd(1) = 3$, and therefore the vertices v with $1 \leq pre(v) < 1 + 3$ (i.e., vertices 1, 2 and 3) are the descendants of 1.

We have discussed how to compute $pre(v)$ in Section 3.5. 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 3(a).

4. CONNECTED COMPONENTS

There is a rich literature of parallel graph algorithms proposed for the PRAM model. In this section and the next, we demonstrate that some of the PRAM algorithms can be translated into PPAs either directly or with some efforts. Then in Section 6, we show that when there is no obvious translation, we can still apply some general concepts used in a PRAM algorithm to design novel PPAs to solve the problem.

In Section 3, we already proposed a BPPA for computing CCs, which requires $O(\delta)$ supersteps. However, the algorithm can be very slow for large spatial networks such as a road network where $\delta \approx O(\sqrt{n})$. Now we present a PPA requiring $O(\log n)$ supersteps by adapting Shiloach-Vishkin's (S-V) algorithm over the PRAM model [22]. It is claimed in [19] that the requirement of concurrent writes makes the S-V algorithm difficult to be translated to MapReduce; we however show that a translation into a PPA is possible by altering a condition check in the S-V algorithm.

Shiloach-Vishkin's Algorithm. Algorithm 1 presents the S-V algorithm, where "par$(o \in S)$ $\{op(o)\}$" means that the operation $op(o)$ runs in parallel for all objects o in set S. In this algorithm,

Algorithm 1 The Shiloach-Vishkin Algorithm

```
1: par(u ∈ V) {D[u] ← u}
2: par((u, v) ∈ E) {if(v < u) D[u] ← v}
3: while(∃u : star[u] = false) {
4:    par((u, v) ∈ E)
5:       if(D[D[u]] = D[u] and D[v] < D[u]) D[D[u]] ← D[v]
6:    par((u, v) ∈ E)
7:       if(star[u] && D[u] ≠ D[v]) D[D[u]] ← D[v]
8:    par(u ∈ V) D[u] ← D[D[u]]
9: }
```

Algorithm 2 Computing $star[u]$

```
1: par(u ∈ V) {
2:    star[u] ← true
3:    if(D[u] ≠ D[D[u]]){    star[u] ← false;
4:       star[D[u]] ← false; star[D[D[u]]] ← false}
5:    star[u] ← star[D[u]]
6: }
```

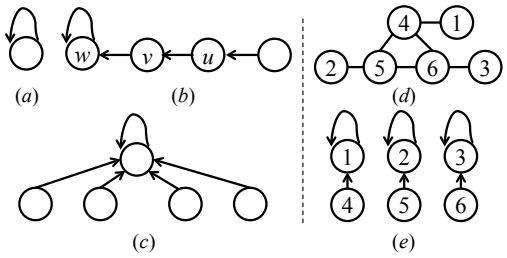


Figure 4: Illustration of Shiloach-Vishkin's algorithm

each vertex u maintains a pointer $D[u]$, initialized as u (Line 1), forming a self loop as shown Figure 4(a). Throughout the algorithm, vertices are organized by a forest such that all vertices in a tree belong to the same CC. We relax the tree definition a bit here to allow the tree root w to have a self-loop (see Figures 4(b) and 4(c)), i.e., $D[w] = w$; while the pointer $D[v]$ of any other vertex v in the tree indicates v's parent.

Pointers are updated by examining edge links in parallel. Initially, for each vertex u, $D[u]$ is set as a neighbor $v < u$ if such a v exists (Line 2). Then, in each round, the pointers are updated by two kinds of operations: (1) *hooking* (Lines 5 & 7): given an edge (u, v), if u's parent $w = D[u]$ is a tree root, hook it as a child of v's parent $D[v]$ (i.e., merge the tree rooted at w into v's tree); (2) *shortcutting* (Line 8): move vertex v and its descendants closer to the tree root, by pointing v to the parent of v's parent (i.e., $D[D[v]]$).

Each vertex u is also associated with a flag $star[u]$ indicating whether it is in a star (see Figure 4(c)). Note that a star is a special tree. The algorithm terminates when all trees become stars, and each star corresponds to a CC. The value of $star[u]$ is required in two places in the while-loop: Line 3 and Line 7. Algorithm 2 shows the PRAM algorithm for computing $star[u]$ for all $u \in V$ according to the current pointer setting, based on the fact that a vertex u with $D[u] \neq D[D[u]]$ invalidates a tree from being a star (see vertex u in Figure 4(b)).

We now illustrate how to translate the steps of Algorithm 1 into Pregel.

Tree Hooking in Pregel. We compute Lines 4–5 of Algorithm 1 in four supersteps: (1) each vertex u sends request to $w = D[u]$ for $D[w]$; (2) each vertex w responds to u by sending $D[w]$; and meanwhile, each vertex v sends $D[v]$ to all v's neighbors; (3) each vertex u obtains $D[w]$ and $D[v]$ from incoming messages and evaluates the if-condition, and then an arbitrary v that satisfies the condition (if it exists) is chosen and $D[v]$ is sent to w; (4) each vertex w that receives $D[v]$ sets $D[w] \leftarrow D[v]$.

Star Hooking in Pregel. We compute Lines 6–7 of Algorithm 1 similarly. However, the condition "$D[u] \neq D[v]$" in Line 7 prevents the correct translation of the S-V algorithm into a PPA. For example, consider the graph in Figure 4(d), and suppose that we obtain the three stars in Figure 4(e) right before Line 6. If we only require "$D[u] \neq D[v]$", setting $D[D[u]] \leftarrow D[v]$ makes $D[1] = 2$, $D[2] = 3$ and $D[3] = 1$ through the edge $(4, 5)$, $(5, 6)$ and $(6, 4)$, thus forming a cycle and violating the tree formation required by the algorithm. Such 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. This problem, however, can be fixed by requiring "$D[v] < D[u]$" instead of "$D[u] \neq D[v]$" in Line 7.

Computing $star[u]$ in Pregel. We compute Algorithm 2 in six supersteps: (1) each vertex u sets $star[u] \leftarrow true$, and sends request to $w = D[u]$ for $D[w]$; (2) each vertex w responds by sending back $D[w]$; (3) each vertex v checks whether $D[u] = D[w]$; if not, it sets $star[u] \leftarrow false$ and notifies w; (4) each vertex w that gets notified sets $star[w] \leftarrow false$ and notifies $v = D[w]$; (5) each vertex v that gets notified sets $D[v] \leftarrow false$. To process Line 5, in Superstep (4), we also make each vertex u send request to $w = D[u]$ for $star[w]$; in Superstep (5), we also make w respond by sending $star[w]$ (note that if w get notified, it must set $star[w]$ to be false first). Finally, in Superstep (6), each vertex u gets $star[w]$ and uses it to update $star[u]$.

Analysis. The other steps can be translated to Pregel similarly. To check the condition in Line 3 of Algorithm 1, we use an aggregator that computes the AND of $star[u]$ for all vertices u. The algorithm terminates if its value equals $true$.

The correctness of this PPA can be justified as follows. Since we always require $v_a < v_b$ when setting $D[v_a] \leftarrow v_b$ during hooking (Lines 2, 4 and 6), the pointer values monotonically decrease, and $D[v] = color(v)$ for any vertex v when the algorithm terminates.

Since each step requires a constant number of supersteps, each round of the S-V algorithm uses a constant number (16 in our implementation) of supersteps. Following an analysis similar to [22], the algorithm computes CCs in $O(\log n)$ rounds, and therefore we obtain a PPA for computing CCs. However, the algorithm is not a BPPA since a vertex w may become the parent of more than $d(w)$ vertices and hence receives/sends more than $d(w)$ messages at a superstep, though the overall number of messages is always bounded by $O(n)$ at each superstep.

Extension for Computing Spanning Tree. In Section 3.3, we presented an $O(\delta)$-superstep BPPA for spanning tree construction based on BFS. We now illustrate how to modify the S-V algorithm to obtain an $O(\log n)$-superstep PPA that computes the spanning tree, which is faster for large-diameter graphs. The main idea is to mark an edge (u, v) as a tree-edge if a hooking operation is performed due to (u, v), which is straightforward for Line 2 of Algorithm 1. For Lines 4 and 6 that perform $D[D[u]] \leftarrow D[v]$, the last superstep is for $w = D[u]$ to pick an arbitrary message $D[v]$ sent by some v, and set $D[w] \leftarrow D[v]$. In this case, w needs to notify both u and v to mark their edge (u, v) as a tree-edge.

5. BI-CONNECTED COMPONENTS

A *bi-connected component* (**BCC**) of an undirected graph G is a

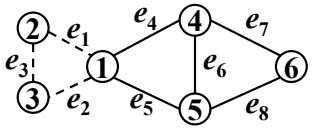


Figure 5: BCC illustration

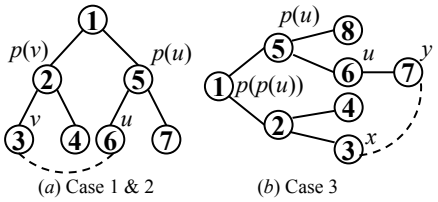


Figure 6: Illustration of construction of G^*

maximal subgraph of G that remains connected after removing one arbitrary vertex. We illustrate the concept of BCC using the graph shown in Figure 5, 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 5, vertices 4 and 5 are in cycle $(4, 5, 6, 4)$, but there is no simple cycle containing both 2 and 4. A vertex is called an articulation point if it belongs to more than one BCC, such as vertex 1 in Figure 5. The removal of an articulation point disconnects the connected component containing it.

Obviously, 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 edges in G' can be superlinear to m. Tarjan-Vishkin's (T-V) PRAM algorithm [26] 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. In the following, we first define G^*, and then propose a non-trivial PPA to compute G^*.

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 6 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 6(a). Note that $e_1 R e_2$ as 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 6(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 6(c) 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, $|E^*| = O(m)$. Also, to compute the CCs of G^*, we only need to consider those vertices of G^* that correspond to the tree edges in T, because other vertices of G^* correspond to the non-tree edges e_2 of Case 1, which can be assigned to the CC of the corresponding e_1 later on. We already have a PPA to compute CCs from G^* in $O(\delta)$ or $O(\log n)$ supersteps (see Sections 3.2 and 4). To design a PPA for computing BCCs, we still need a PPA for constructing G^* from G.

We first apply the BPPA in Section 3.3 or the PPA in Section 4 to compute a spanning tree T of G. Then, we use the BPPA in Section 3.5 to compute $pre(v)$ for each vertex v in T and the BPPA in Section 3.6 to compute $nd(v)$ for answering ancestor-descendant queries, after which both Case 1 and Case 2 can 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 show how to check the condition of Case 3 in a constant number of supersteps. We 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).

In the remainder of this section, we simply use v to denote $pre(v)$ for ease of presentation. We further define $local(v)$ to be the minimum among v and all the neighbors connected to v by a non-tree edge. Then, $min(v)$ 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 7 illustrates the concept of $little(v)$ and $big(v)$.

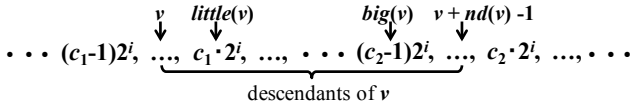
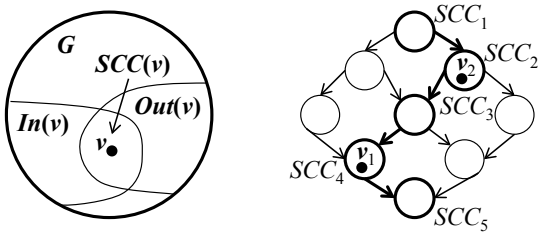


Figure 7: Illustration of computing $min(v)$



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

Figure 8: Concepts in SCC computation

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 $global(v)$.

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. In the PRAM model, existing algorithms [13, 6, 5] were designed based on the observation that $SCC(v) = Out(v) \cap In(v)$ (see Figure 8(a)). Their approach computes $Out(v)$ 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.

We also apply the relationship between $SCC(v)$, $Out(v)$ and $In(v)$ and 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. Before describing our SCC algorithms, we first present a BPPA for the graph decomposition operation that is employed in our algorithms.

Graph decomposition. Given a partition of V, denoted by V_1, V_2, …, V_ℓ, we decompose G into $G[V_1]$, $G[V_2]$, …, $G[V_\ell]$ 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 $min_f[\Gamma_{in}(v)] = \min\{min_f(u) : u \in \Gamma_{in}(v)\}$, then if $min_f[\Gamma_{in}(v)] < min_f(v)$, v updates $min_f(v) = min_f[\Gamma_{in}(v)]$ 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 $min_f(v) = v$, while for the other vertices u, $min_f(u) = \infty$; (2) each active vertex v propagates $min_b(v)$ to 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 $(min_f(v), min_b(v))$. The labeling has the following property:

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.*

PROOF. We first prove (i). Given a SCC and a vertex v in the SCC, suppose that v has a label pair (i, j), we show that for any other vertex u in the SCC, u also has the same label pair (i, j). We only prove $min_f(u) = i$, and the proof of $min_b(u) = j$ is symmetric. Let us denote $v_1 \to v_2$ iff v_1 can reach v_2. Since $i \to v \to u$, $min_f(u) > i$ is impossible. Also, since $min_f(u) \to u \to v$, $min_f(u) < i$ is impossible. Therefore, we have $min_f(u) = i$.

We now prove (ii). First, $\forall v \in V_{(i,i)}$, $v \to i$ and $i \to v$, and thus $V_{(i,i)} \subseteq SCC(i)$. Second, if $V_{(i,i)}$ exists, then $i \in V_{(i,i)}$, and by (i), we have $SCC(i) \subseteq V_{(i,i)}$. Therefore, $V_{(i,i)} = SCC(i)$. □

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.

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

PROOF. We show the correctness of Theorem 1 by proving the following invariant: at the end of Round i, the length of any maximal path in the DAG is at most $\mathcal{L} - i$. We only need to show that in each round, a SCC is determined for any maximal path in the DAG. This is because, after the SCC is marked and the exterior edges are removed, (1) if the SCC is an endpoint of the DAG path (let us denote its length by ℓ), the path length becomes $\ell - 1$; (2) otherwise, the path is broken into two paths with length $< \ell - 1$. In either case, the new paths have length at most $\ell - 1$.

We now show that for an arbitrary maximal DAG path p, at least one SCC is found as $V_{(i,i)}$. First, let us assume i_1 is the smallest vertex in G, then $V_{(i_1,i_1)}$ is found as a SCC for any maximal path p_1 that contains $SCC(i_1)$. Now consider those maximal paths that does not contain $SCC(i_1)$, and let i_2 be the smallest vertex in the SCCs of these paths. Then $V_{(i_2,i_2)}$ is found as a SCC for any such path p_2 that contains $SCC(i_2)$. The reasoning continues for those maximal paths that do not contain $SCC(i_1)$ and $SCC(i_2)$ until all maximal paths are covered. □

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 8(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. In practice, the algorithm takes only a small number of supersteps despite the worst case bound of $O(n)$ supersteps.

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

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.

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, ..., V_ℓ. (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 $(Src_f(v), Src_b(v))$. The labeling has the following property:

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 $S_f \cap S_b \neq \emptyset$.*

PROOF. We first prove (i). Given a SCC and a vertex v in the SCC, suppose that v has a label pair (S_f, S_b), we show that for any other u in the SCC, u also has the same label pair (S_f, S_b). We only prove $Src_f(u) = S_f$, and the proof of $Src_b(u) = S_b$ is symmetric. (1) For any vertex $s \in S_f$, we have $s \to v \to u$ and thus $s \in Src_f(u)$; therefore, $S_f \subseteq Src_f(u)$. (2) For any vertex $s \in Src_f(u)$, we have $s \to u \to v$ and thus $s \in Src_f(v) = S_f$; therefore, $Src_f(u) \subseteq S_f$. As a result, $Src_f(u) = S_f$.

We now prove (ii), Since $S_f \cap S_b \neq \emptyset$, $\exists u \in S_f \cap S_b$. First, $\forall v \in V_{(S_f,S_b)}$, $v \to u$ and $u \to v$, and thus $V_{(S_f,S_b)} \subseteq SCC(u)$. Second, if $V_{(S_f,S_b)}$ exists, then $u \in V_{(S_f,S_b)}$ and by (i), we have $SCC(u) \subseteq V_{(S_f,S_b)}$. Therefore, $V_{(S_f,S_b)} = SCC(u)$. □

We now analyze the number of rounds required. In any round, we may have ℓ subgraphs and thus 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, and 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

$$E[X_i] = Pr\{X_i = 1\} = 1 - \prod_{j=1}^{k} Pr\{s_j \notin SCC_i\}$$

$$= 1 - \prod_{j=1}^{k}(1 - p_i) = 1 - (1 - p_i)^k.$$

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

$$E[X] = \sum_{i=1}^{c} n_i \cdot E[X_i] = \sum_{i=1}^{c}[n_i - n_i(1 - p_i)^k]$$

$$= n - \sum_{i=1}^{c} n_i(1 - p_i)^k = n - n\sum_{i=1}^{c} p_i(1 - p_i)^k.$$

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 practise, 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 the technical report of this paper [1] but is omitted here due to space limit).

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. LOAD BALANCING

Large real world graphs usually have highly skewed degree distributions [24, 21]. High-degree vertices cause unbalanced workload for parallel graph algorithms not only for the Pregel model [17, 18] but also for the MapReduce model [15, 19]. To the best of our knowledge, the problem of skewed workload due to high-degree vertices has not been addressed by any previous work for any of the graph problems studied in this paper. Here we propose a simple and effective solution for this problem, which applies to both Pregel and MapReduce models.

The main idea of our solution is to transform the input graph G into a graph with bounded maximum degree d_{max}, denoted by G^*. We illustrate the idea using a simple example.

Consider the graph shown in Figure 9(a), where vertex u has in-degree 8 and out-degree 2. Suppose that the maximum degree desired is 2, i.e., $d_{max} = 2$, then we can add dummy vertices (the hollow ones in Figure 9(b)) to organize the in-neighbors of u by a tree with fan-out no more than 2, so that the transformed graph satisfies the maximum degree requirement. In reality, d_{max} can be

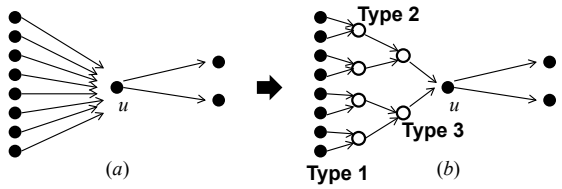


Figure 9: High-degree to low-degree transformation

set to a larger value such as 100 to make the tree height small (so as to enable faster message propagation).

We first consider directed graphs. We define three types of edges in Figure 9(b): (1) Type 1: edges between a vertex in $\Gamma_{in}(u)$ and a dummy vertex; (2) Type 2: edges between dummy vertices (there can be more than one layer of such edges); (3) Type 3: edges between a dummy vertex and u.

We eliminate in-degree overflow in two supersteps. (1) For each vertex u, if $d_{in}(u) > d_{max}$, we construct the tree, add the dummy vertices, and send message to each vertex $v \in \Gamma_{in}(u)$ to notify v to replace its out-edge (v, u) with $(v, p(v))$, where $p(v)$ is the dummy vertex that is the father of v in the tree (u is the root); then, u votes to halt. (2) When a vertex v receives a message $\langle u, p(v)\rangle$ from u, it replaces edge (v, u) with $(v, p(v))$ and votes to halt.

After this operation, G is transformed into a graph G_0^* having no vertex with in-degree overflow. The transformation does not change the out-degree of any vertex in G, and all dummy vertices have out-degree 1. We then eliminate out-degree overflow of G_0^* using a symmetric two-superstep Pregel algorithm, which transforms G_0^* further into a graph G^*. Thus, we obtain a graph G^* without degree overflow in totally four supersteps.

We now consider the case when G is undirected. We treat each edge as bi-directed and use the four-superstep transformation to obtain a directed graph G_1^*. Then, we transform G_1^* into an undirected (or bi-directed) graph G^* in two more supersteps: (1) each vertex u broadcasts itself to all neighbors in $\Gamma_{in}(u)$ and $\Gamma_{out}(u)$, and votes to halt; (2) for each vertex u, and for each vertex v received by u, add out-edge (u, v) (and in-edge (v, u)) if it does not already exist.

In G_1^*, each non-dummy vertex has in-degree at most d_{max}, and thus G^* will not increase its out-degree by more than d_{max}. Similarly, the in-degree will not be increased by more than d_{max}. Therefore, the total degree is at most $4 \cdot d_{max}$ with bi-directed edges, or $2 \cdot d_{max}$ with undirected edges. To ensure maximum degree of σ_{max}, we run the four-superstep approach with $\sigma_{max}/2$.

Now we show a list of fundamental graph problems whose results over graph G can be obtained by solving them over the degree-bounded graph G^*.

Graph connectivity. Since this transformation preserves the vertex-wise reachability, we obtain the CCs/SCCs of G as follows: (1) compute the CCs/SCCs of G^*, and (2) remove the dummy vertices from each CC/SCC to obtain the CCs/SCCs of G. Furthermore, since this transformation preserves articulation points, we can obtain the BCCs of G similarly.

Shortest-path computation. This transformation is also applicable for single-source shortest path computation. For the new edges introduced due to in-degree overflow, we set the weight of each Type 1 edge $(v, p(v))$ to be that of (v, u) (see Figure 9(b)), and the weight of each Type 2 or Type 3 edge to 0. Symmetrically, we assign weight to new edges introduced due to out-degree overflow. We can then obtain the shortest path distance in G by computing the shortest path distance from G^*, while the shortest paths in G can also be obtained by simply ignoring the dummy vertices from

the shortest paths computed from G^*.

8. EXPERIMENTS

In this section, we evaluate the performance of our algorithms over large real graphs using a cluster of 60 machines, each with 2 quad-core Intel Xeon E5420 processors and 16GB RAM. The machines are connected by Cisco 4948E switch, and the connectivity between any pair of nodes in the cluster is 1Gbps. All our algorithms were implemented in Pregelix[1], an open-source counterpart to Pregel built on top of Hadoop Distributed File System (HDFS).

Datasets. The experiments were conducted on five real graph datasets: *USA*, *BTC*, *Twitter*, *LJ* and *Patents*. *USA*[2] is an undirected graph that models the full USA road network. *BTC*[3] is a semantic graph converted from an RDF dataset. *Twitter*[4] is a directed graph where vertices are Twitter users and edge (u, v) exists iff v is a follower of u. *LJ*[5] is a friendship network of a LiveJournal blogging community. *Patents*[6] is a citation graph for U.S. patents.

Table 1 shows some statistics of the graphs, where $|V|$ is the number of vertices and $|E|$ is the number of edges in each graph. We explain $|V_{in}|$ and $|V_{out}|$ as follows. In *USA*, most vertices have degree no more than 4, and the maximum degree is only 8. However, the other four datasets all have skewed degree distribution, though *Patents* has mostly small degree vertices and the highest degree is less than 1000. In Table 2, we summarize the degree distribution of *BTC*, *LJ* and *Twitter*, which are all highly skewed. For example, in *Twitter*, while most vertices have out-degree less than 100, there are 71 vertices whose out-degree is at least 10^6. We preprocess *BTC*, *Twitter*, *LJ* and *Patents* using the graph transformation approach described in Section 7, with $d_{max} = 100$. In Table 1, $|V_{in}|$ refers to the number of vertices (in G_0^*) after handling in-degree overflow, and $|V_{out}|$ refers to the number of vertices (in G^*) after handling out-degree overflow.

Table 1: Datasets (U: un-directed; D: directed)

	Type	$\|V\|$	$\|E\|$	$\|V_{in}\|$	$\|V_{out}\|$
USA	U	23,947,347	58,333,344	–	–
BTC	U	164,732,473	772,822,094	168,693,976	172,655,479
Twitter	D	41,652,230	1,468,365,182	52,840,962	63,808,250
LJ	D	4,847,571	68,993,773	5,093,348	5,324,590
Patents	D	3,774,768	16,518,948	3,776,736	3,781,318

Per-Superstep Performance. In Pregel, the per-superstep computation/communication cost depends on the number of active vertices in the superstep, which changes as the job proceeds. Let us denote the BPPA of Section 3.2 for CC computation by *HashMin*, and the BPPA of Section 3.3 for spanning tree and BFS computation by *BFS*. Figure 10 shows the time spent by *HashMin* and *BFS* in each superstep when running over *BTC*. For *HashMin*, initially all vertices are active, and as the algorithm proceeds, more and more vertices votes to halt, and thus the running time per superstep decreases. For *BFS*, initially only the source vertices are active; as the wavefront propagates, more and more vertices become active and thus the running time per superstep increases. However, as the wavefront propagates to the boundary, the number of active vertices decreases eventually. These behaviors can be well observed in Figure 10. The results for other datasets show very similar trends and hence omitted due to space limit.

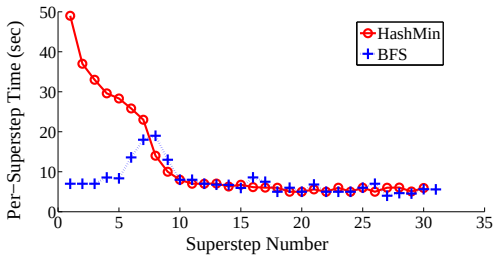


Figure 10: Running time per-superstep of HashMin and BFS

CC & BCC Results. In Section 3 we proposed PPAs for a list of fundamental graph problems. Due to space limit we cannot report their performance one by one individually. Instead, we report their performance results as steps of the BCC computation. To do that, we first list the sequence of PPA tasks in the BCC computation: (1) *HashMin*: to compute $color(v)$ for all $v \in V$ using the BPPA of Section 3.2; (2) *BFS*: to compute the spanning forest of G using the BPPA of Section 3.3 with sources $\{s \in V | color(s) = s\}$; alternatively, we may obtain the spanning forest using the S-V algorithm of Section 4, denoted by *S-V*; (3) *EulerTour*: to construct Euler tours from the spanning forest using the BPPA of Section 3.4; (4) *ListRank1*: to break each Euler tour into a list and mark each edge as forward/backward using list ranking (see Section 3.5); (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 3.5); (6) *MinMax*: to compute $min(v)$ and $max(v)$ for each $v \in V$ using the PPA of Section 5; (7) *CreateGraph*: to construct G^* described in Section 5 using $pre(v)$, $nd(v)$, $min(v)$ and $max(v)$ information; (8) *HashMin2*: to compute the CCs of G^*, but only consider tree edges; alternatively, we may use the S-V algorithm for CC computation, denoted by *S-V2*; (9) *Non-tree-edge*: to consider non-tree edges of G^* using Case 1 of Section 5.

We report the results of BCC computation over the three larger datasets *USA*, *BTC* and *Twitter*. For *Twitter*, we constructed an undirected version where we make each original edge bi-directed. The runtime of the tasks is given in Table 3, where we show four metrics: (1) *Load*: the time for loading the vertices from HDFS into different machines; (2) *#Supersteps*: the number of supersteps; (3) *Compute*: the total time spent by all supersteps; (4) *Write*: the time for writing the results to HDFS.

For BCC computation, we use CC and spanning forest computation as a building block. Since we can choose to use either (i) *HashMin* and *BFS*, or (ii) *S-V* to compute CCs and/or spanning forest, the dotted line in the two rows in Table 3 indicate that we can choose either (i) or (ii). As a result, the last two rows of Table 3 report the total running time of BCC computation using either (i) or (ii) as the building block.

One observation obtained from Table 3 is that, when computing CCs, *HashMin* is more efficient for small-diameter graphs like *BTC* and *Twitter*, while *S-V* is quite stable even for graphs with very large diameter like *USA*. For example, it takes *HashMin* only 375.9 seconds (30 supersteps) over *BTC*, but *S-V* requires 4974.9 seconds (226 supersteps). On the other hand, in *USA*, while *S-V* spends 2185.7 seconds (242 supersteps), the runtime of *HashMin* is prohibitive: 37056.8 seconds (6262 supersteps). This observation matches our analysis: *HashMin* requires $O(\delta)$ supersteps, while *S-V* requires $(16 \log n + 2)$ supersteps; when δ is small, the constant

[1] http://hyracks.org/projects/pregelix/
[2] http://www.dis.uniroma1.it/challenge9/download.shtml
[3] http://vmlion25.deri.ie/
[4] http://law.di.unimi.it/webdata/twitter-2010
[5] http://snap.stanford.edu/data/soc-LiveJournal1.html
[6] http://snap.stanford.edu/data/cit-Patents.html

Table 2: Degree distribution

Degree	BTC	LJ		Twitter		Degree	BTC	LJ		Twitter	
	Count	Count_{in}	Count_{out}	Count_{in}	Count_{out}		Count	Count_{in}	Count_{out}	Count_{in}	Count_{out}
0	71,476	358,331	539,119	5,963,082	1,548,949	$[10^3, 10^4)$	15,770	529,454	263,364	220,586	139,535
$[1, 10)$	160,698,421	2,950,147	2,718,911	18,652,645	24,406,938	$[10^4, 10^5)$	3,234	773	61	3,720	6,205
$[10, 10^2)$	3,840,972	1,443,565	1,494,884	15,060,437	14,364,021	$[10^5, 10^6)$	615	5	3	26	359
$[10^2, 10^3)$	101,958	94,750	94,593	1,751,734	1,186,152	$[10^6, 10^7)$	27	–	–	–	71

Table 3: Runtime on BCC computation (in seconds)

Task	BTC				USA				Twitter			
	Load	#Superstep	Compute	Write	Load	#Superstep	Compute	Write	Load	#Superstep	Compute	Write
HashMin	51.6	30	375.9	16	47.8	6262	37056.8	4.287	25.9	17	428.7	23.1
BFS	16.6	31	232.6	17.9	11.1	6263	27391.3	7.5	18.1	18	209.5	22.9
S-V	44.1	226	4974.9	16.1	43.6	242	2185.7	3.3	36.8	98	4765.3	27.1
EulerTour	12.6	4	1378.4	16.8	7.9	4	143.5	3.5	14.4	4	39.8	5
ListRank1	24.7	50	1366.7	16.9	12.4	56	902.5	5	11.1	58	813.9	167.5
ListRank2	26.5	50	310.4	6.2	12.6	56	734.7	2.8	13.8	58	883.8	3.3
MinMax	27.6	46	63	15.7	7.5	52	291.9	4.4	25.6	54	367.6	27.6
CreateGraph	30.6	4	195.7	34.4	9.5	4	32.4	9.6	37.9	4	440.1	104.1
HashMin2	18.8	147	1167.2	14.6	14.6	6199	36746.6	4.1	103.6	18	577	25.8
S-V2	21.4	114	2265.3	16.5	8	258	1777.7	2.7	24.4	82	3971.5	23.3
Non-tree-edge	42.7	4	166.4	30.2	9.7	4	29	3	78.1	4	1275.1	212.4
Total time (CC by HashMin)	5676.7				103506				5955.7			
Total time (CC by S-V)	11103.8				6242.9				13369.5			

Table 4: Min-label algorithm over Twitter (in seconds)

Round	Task	Load	#Superstep	Compute	Write	MaxSize
1	Opt 1	27.5	18	116.8	24	40262
	Forward	20.9	47	491.4	26.3	
	Backward	19.1	30	238.8	23.6	
	GDecomp	22.3	3	158.2	26.4	
2	Opt 1	38.2	5	52.2	27.8	203
	Forward	25.3	47	297.3	28.7	
	Backward	43.9	47	274.8	50	
	GDecomp	52.7	3	59.8	50.8	
3	Opt 1	49.6	5	54.7	29.1	3
	Forward	43.8	5	43.7	26.2	
	Backward	35.9	6	63.4	25.1	
	GDecomp	28.2	3	51	28	
4	Opt 1	28.3	2	31.8	32.1	0
	Forward	26.1	3	38.6	23.3	
	Backward	27.5	3	28	27.1	
	GDecomp	30.1	3	55	25.9	
Total time		3049.3				

Table 5: Multi-label algorithm over Twitter (in seconds)

Round	Task	Load	#Superstep	Compute	Write	MaxSize
1	Opt 1	23.8	18	167	29.1	218319
	FB	27.8	35	1220.7	47.8	
	GDecomp	40.1	3	281.2	23.7	
2	Opt 1	39.4	5	47.2	44	186129
	FB	24.9	5	50.6	30.1	
	GDecomp	51.7	3	68.8	40.9	
3	Opt 1	54.3	1	5.7	67.6	186100
	FB	21.7	5	74	25.4	
	GDecomp	30.3	3	39.1	27.7	
4	Opt 2	852				
Total time		3456.6				

number 16 makes *S-V* inferior to *HashMin*; but when δ is large, as in *USA* where $\delta \approx O(\sqrt{n})$, $(16 \log n + 2) \ll \delta$.

It might be argued that computing the CCs of a road network in Pregel is not important, as road network graphs 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.

Another observation from Table 3 is that, the diameter of the concise graph G^* which is the input to *HashMin2*, can be quite different from the diameter of G. For example, in *BTC*, while *HashMin* over G takes only 30 supersteps, *HashMin2* over G^* spends 147 supersteps, even more than the 114 supersteps *S-V2* requires. But in general, the CC algorithm that is more efficient over G likely remains to be more efficient over G^*.

SCC Results. We first compute the SCCs using the min-label algorithm, until all SCCs are found. Table 4 reports the results for the largest (and also densest) dataset *Twitter* (the results of other datasets can be found in [1] but omitted here due to space limit). The computation takes only 4 rounds in total, which demonstrates that in practice the min-label algorithm requires much less than $\mathcal{L}$ rounds.

In Table 4, each round consists of 4 tasks: (1) *Opt 1*: this task removes trivial SCCs by Optimization 1. (2) *Forward*: forward min-label propagation. (3) *Backward*: backward min-label propagation. (4) *GDecomp*: 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 $(min_f(u), min_b(u))$, marks each vertex u with $min_f(u) = min_b(u)$ as being in a SCC, and performs graph decomposition.

Column *MaxSize* in Table 4 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 *MaxSize* becomes 0. As shown by Column *#Supersteps* in Table 4, the min-label algorithm requires only a small number of supersteps for label propagation over small-diameter social graphs. In total, it takes only 3049.3 seconds to compute all the SCCs. In fact, after one round *MaxSize* becomes 40262 already, and all subgraphs can be assigned to different machines to for SCC computation locally. Thus, if we apply Optimization 2 here, i.e., using one round of MapReduce to implement this divide-and-conquer task, then the job takes only 295 seconds. In this way, along with Round 1, we can compute all the SCCs in 1490.3 seconds, which is much faster than running the min-label algorithm until termination.

We next report the results of running the multi-label algorithm over *Twitter* in Table 5, where we fix the number of sources $k = 10$. In each round, Row *FB* refers to the parallel forward and backward k-label propagation. In the experiments, we stop decomposing a subgraph $G[V_{\mathcal{ID}}]$ if $|V_{\mathcal{ID}}| \leq 50000$, by marking all vertices in $V_{\mathcal{ID}}$. As shown in Table 5, *MaxSize* decreases slowly, and is much larger than that of the min-label algorithm. Fortunately, the subgraphs are small enough to be assigned to different machines for local SCC computation, which takes 852 seconds.

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. In fact, the multi-label algorithm almost always finds the largest SCC in the first round which usually contains the majority of the vertices in a graph, while the min-label algorithm may not find it in the first round. Thus, in applications where only the largest SCC (also called the giant SCC) is needed, the multi-label algorithm can be a better choice.

To test the effect of k, we also run the multi-label algorithm over *Twitter* with $k = 20$ and 50. The results show that large k does not significantly speed up the decrement of *MaxSize* although it consumes more space. We thus omit the results here but they are reported in [1].

We also run our SCC algorithms over the other two directed graphs, *LJ* and *Patents*. The results follow similar trends to those of *Twitter*, and both SCC algorithms are efficient. In fact, for *Patents*, after a 17-superstep trivial SCC marking task, our main algorithm finishes all SCC computation in just one round. The details of the results are reported in [1].

9. CONCLUSIONS

As existing Pregel algorithms have rather ad hoc design criteria, this paper addresses the problem by defining a class of Pregel algorithms, called PPAs, which require only linear space, communication and computation per round, and logarithmic number of rounds. We showed that many fundamental graph problems can be solved by PPAs efficiently, and these basic operations can be used as building blocks to solve other graph problems. In particular, we developed non-trivial PPAs for computing BCCs and SCCs. Experiments on large real world graphs demonstrated that our algorithms have good performance in shared-nothing parallel computing platforms.

10. REFERENCES

[1] Practical pregel algorithms for massive graphs. *Technical Report (https://www.dropbox.com/sh/m3rfyrslir81gvk/qhH2BNCjkg)*, 2013.

[2] F. N. Afrati, D. Fotakis, and J. D. Ullman. Enumerating subgraph instances using map-reduce. *2013 IEEE 29th International Conference on Data Engineering (ICDE)*, 0:62–73, 2013.

[3] L. Backstrom, P. Boldi, M. Rosa, J. Ugander, and S. Vigna. Four degrees of separation. In *WebSci*, pages 33–42, 2012.

[4] B. Bahmani, R. Kumar, and S. Vassilvitskii. Densest subgraph in streaming and mapreduce. *Proceedings of the VLDB Endowment*, 5(5):454–465, 2012.

[5] 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.

[6] 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.

[7] Y. Bu, V. R. Borkar, M. J. Carey, J. Rosen, N. Polyzotis, T. Condie, M. Weimer, and R. Ramakrishnan. Scaling datalog for machine learning on big data. *CoRR*, abs/1203.0160, 2012.

[8] 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.

[9] Y.-J. Chiang, M. T. Goodrich, E. F. Grove, R. Tamassia, D. E. Vengroff, and J. S. Vitter. External-memory graph algorithms. In *Proceedings of the sixth annual ACM-SIAM symposium on Discrete algorithms*, pages 139–149. Society for Industrial and Applied Mathematics, 1995.

[10] 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.

[11] G. De Francisci Morales, A. Gionis, and M. Sozio. Social content matching in mapreduce. *Proceedings of the VLDB Endowment*, 4(7):460–469, 2011.

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

[13] 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.

[14] U. Kang, C. E. Tsourakakis, and C. Faloutsos. Pegasus: A peta-scale graph mining system implementation and observations. In *Data Mining, 2009. ICDM'09. Ninth IEEE International Conference on*, pages 229–238. IEEE, 2009.

[15] H. Karloff, S. Suri, and S. Vassilvitskii. A model of computation for mapreduce. In *Proceedings of the Twenty-First Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 938–948. Society for Industrial and Applied Mathematics, 2010.

[16] S. Lattanzi, B. Moseley, S. Suri, and S. Vassilvitskii. Filtering: a method for solving graph problems in mapreduce. In *Proceedings of the 23rd ACM symposium on Parallelism in algorithms and architectures*, pages 85–94. ACM, 2011.

[17] 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.

[18] 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.

[19] V. Rastogi, A. Machanavajjhala, L. Chitnis, and A. D. Sarma. Finding connected components in map-reduce in logarithmic rounds. *2013 IEEE 29th International Conference on Data Engineering (ICDE)*, 0:50–61, 2013.

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

[21] H. Shang and M. Kitsuregawa. Efficient breadth-first search on large graphs with skewed degree distributions. In *Proceedings of the 16th International Conference on Extending Database Technology*, pages 311–322. ACM, 2013.

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

[23] J. F. Sibeyn, J. Abello, and U. Meyer. Heuristics for semi-external depth first search on directed graphs. In *Proceedings of the fourteenth annual ACM symposium on Parallel algorithms and architectures*, pages 282–292. ACM, 2002.

[24] S. Suri and S. Vassilvitskii. Counting triangles and the curse of the last reducer. In *Proceedings of the 20th international conference on World wide web*, pages 607–614. ACM, 2011.

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

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

[27] C. E. Tsourakakis, U. Kang, G. L. Miller, and C. Faloutsos. Doulion: counting triangles in massive graphs with a coin. In *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 837–846. ACM, 2009.

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

[29] J. Xiang, C. Guo, and A. Aboulnaga. Scalable maximum clique computation using mapreduce. *2013 IEEE 29th International Conference on Data Engineering (ICDE)*, 0:74–85, 2013.

[30] Z. Zhang, J. X. Yu, L. Qin, L. Chang, and X. Lin. I/o efficient: computing sccs in massive graphs. In *Proceedings of the 2013 international conference on Management of data*, pages 181–192. ACM, 2013.