Partition Aware Connected Component Computation in Distributed Systems

Ha-Myung Park
KAIST
hamyung.park@kaist.ac.kr

Namyong Park
Seoul National University
namyong.park@snu.ac.kr

Sung-Hyon Myaeng
KAIST
myaeng@kaist.ac.kr

U Kang
Seoul National University
ukang@snu.ac.kr

***Abstract*—How can we find all connected components in an enormous graph with billions of nodes and edges? Finding connected components is a fundamental operation for various graph computation tasks such as pattern recognition, reachability, graph compression, etc. Many algorithms have been proposed for decades, but most of them are not scalable enough to process recent web scale graphs. Recently, a MapReduce algorithm was proposed to handle such large graphs. However, the algorithm repeatedly reads and writes numerous intermediate data that cause network overload and prolong the running time. In this paper, we propose PACC (Partition-Aware Connected Components), a new distributed algorithm based on graph partitioning for load-balancing and edge-filtering. Experimental results show that PACC significantly reduces the intermediate data, and provides up to 10 times faster performance than the current state-of-the-art MapReduce algorithm on real world graphs.**

I. Introduction

How can we find all connected components in an enormous graph with billions of nodes and edges? A connected component of a graph is a set of nodes where any two nodes are connected by a path of edges. Finding connected components is a fundamental operation for various graph computation tasks such as pattern recognition [1], [2], reachability [3], [4], graph compression [5], [6], graph partition [7], [8], random walk [9], etc.

In order to compute connected components in billion-scale graphs, many algorithms have been proposed in different ways: I/O efficient [10], [11], distributed memory [12]–[14], and MapReduce algorithms [15]–[17]. Most of the algorithms have limited scalability. The I/O efficient algorithms utilize only a single machine, and thus they cannot process a graph whose size exceeds the external memory space of the machine. The distributed memory algorithms increase the computation speed by utilizing multiple machines; however, they cannot handle a graph whose intermediate data size is larger than the distributed memory space. The MapReduce algorithms, which are disk-based and distributed, increase the size of processable graphs theoretically; however, they show significantly slower performance than the I/O efficient or distributed memory algorithms on medium size graphs because they write and read massive intermediate data repeatedly during computation [11].

In this paper, we propose a new distributed algorithm PACC (Partition-Aware Connected Components) for computing connected components in a graph, which is more scalable than I/O efficient or distributed memory algorithms, and is much faster than the MapReduce algorithms. PACC is inspired by the *alternating* algorithm proposed by Kiveris et al. [17], which is the state-of-the-art MapReduce algorithm. We observe that the alternating algorithm suffers from ‘the curse of the last reducer’. PACC, on the other hand, distributes workloads evenly among machines by partitioning the graph. Also, we note that the alternating algorithm reads and writes a lot of edges unnecessarily and repeatedly, even when only few edges change during a round. PACC achieves speedup by filtering unnecessary edges out, and by using a single machine algorithm when a few edges remain. The main contributions of this paper are as follows:

- **Algorithm**. We propose PACC, a new distributed algorithm for computing connected components in a large graph, which is designed to balance workloads, and to minimize input/shuffled data size and required rounds.
- **Theory**. We prove the correctness and various characteristics of PACC. PACC guarantees the input size does not increase in each round.
- **Experiment**. The proposed algorithms are experimentally evaluated with both real world and synthetic graphs. The results show that the proposed algorithm (PACC-opt with a parameter τ=20M) outperforms the previous algorithm (Alt-opt; the optimized alternating algorithm) by up to 10$\times$ (see Figure 1).

The binary code of our method and the datasets used in the paper are available at http://datalab.snu.ac.kr/pacc. The remaining part of this paper is organized as follows. In

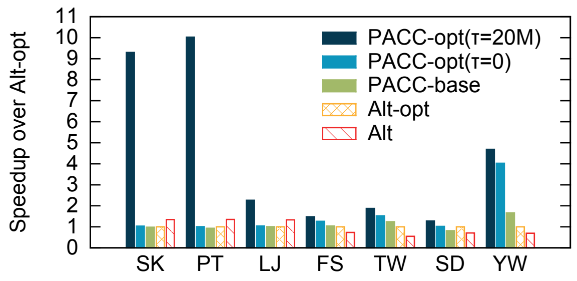

Fig. 1: Speedup over the optimized alternating algorithm. PACC-opt with τ=20M shows the best performance on all the graphs, and it shows up to 10 times faster performance than the optimized alternating algorithm on the Patent graph. Details on the dataset are given in Table II.

TABLE I: Table of symbols.

Symbol	Definition
$G = (V, E)$	Simple graph with the set V of vertices and the set E of edges.
u, v, n	Vertices.
(u, v)	Edge between u and v.
$\Gamma(u)$	Set of neighbors of u: $\{v \mid (u, v) \in E\}$.
$\Gamma^{+}(u)$	Set of large neighbors of u: $\{v \mid v \in \Gamma(u), v > u\}$.
$\Gamma^{-}(u)$	Set of small neighbors of u: $\{v \mid v \in \Gamma(u), v < u\}$.
ρ	Number of partitions.
ξ	Hash function $V \rightarrow \{0, \cdots, \rho - 1\}$. $\xi(u)$ is the partition containing u.
$[S]_i$	i-th partition of a set S: $\{v \mid v \in S, \xi(v) = i\}$.
$m(u)$	Minimum node in $\Gamma(u) \cup \{u\}$: $\min(\Gamma(u) \cup \{u\})$.
$m_i(u)$	Minimum node in $[\Gamma(u) \cup \{u\}]_i$: $\min([\Gamma(u) \cup \{u\}]_i)$.
τ	Threshold for the number of input edges.

Section II, we review previous researches on the connected component computation. In Section III, we formally define the problem of connected component computation and introduce the alternating algorithm which is the baseline of our work. The details of the proposed algorithm and an analysis on the algorithm are presented in Section IV. We give experimental results and evaluations of the proposed algorithm in Section V. Finally, we conclude in Section VI. Frequently used symbols in this paper are listed in Table I.

II. Related Work

In this section, we introduce several methods for computing connected components. We first show single-machine algorithms which can be used as a module of the proposed algorithm. Then, we show distributed algorithms including distributed-memory and MapReduce algorithms.

A. Single-machine algorithms

A well known and effective way to compute connected components is to conduct a traditional graph traversal algorithm, the breadth-first search or the depth-first search; they are linear-time algorithms. For fast computation, a multi-core algorithm is also proposed by Patwary et al. [18]. This algorithm is based on the Union-Find algorithm which maintains disjoint sets of nodes where each set represents connected nodes, and if an edge links two different sets, the algorithm unifies the two sets. As these algorithms are memory based, however, they cannot handle very large graphs exceeding the size of main memory.

Disk-based algorithms, GraphChi [10] and DSP-CC [11], increase the size of data which can be processed in a single machine. GraphChi is a disk-based graph mining platform running on a single machine. This platform provides an implementation for computing connected components based on label propagation; each node receives labels from its neighbors, chooses the minimum label, and propagates the label to the neighbors, iteratively. DSP-CC is another disk-based algorithm based on the Union-Find algorithm; this algorithm shows impressive performance on billion scale graph by fully utilizing solid-state drives (SSDs). However, both GraphChi and DSP-CC do not scale to graphs with hundreds of billions of nodes and edges.

Note that the above single-machine algorithms can be used as a module of the proposed algorithm (see Section IV-A); that is, the proposed algorithm is a tool for extending such single machine algorithms to be able to handle very large graphs.

B. Distributed-memory algorithms

Parallel Random-Access Machine (PRAM) is a classical model for analyzing the performance of parallel algorithms. A lot of PRAM algorithms for computing connected components have been proposed. As the PRAM model is just theoretical, Bader and Cong [19] survey the practical algorithms implemented on symmetric multiprocessors (SMPs), and propose another algorithm for SMPs based on a parallel depth-first search; the algorithm first finds a shallow spanning tree on a single machine and starts the depth-first search on different nodes of the spanning tree.

Recently, several distributed-memory graph mining platforms based on a vertex-centric programming model have been proposed: Pregel [12], GraphLab [13], PowerGraph [14], etc. These platforms provide implementations for connected component computation, which are based on a label propagation like GraphChi.

Even though the above distributed-memory algorithms achieve faster performance than single machine algorithms, all the distributed-memory algorithms are limited in handling very large graphs with hundreds of billions of nodes and edges which exceed the shared memory space.

C. MapReduce algorithms

MapReduce [20] is a disk-based programming framework supporting parallel and distributed computation for processing enormous data. Thanks to its fault-tolerance, high scalablility, and ease of use, MapReduce has been used for various graph analysis tasks such as radius/diameter calculation [15], triangle counting [21], [22], and graph visualization [23], [24].

In order to compute connected components in very large graphs, several algorithms have been proposed in MapReduce. One obvious way to compute connected components in a distributed environment is to do the breadth-first search repeatedly until all nodes are visited. This simple algorithm requires as many rounds as the sum of the diameter of each connected component. Conducting the breadth-first search from every node concurrently, Pegasus [15] and Zones [25], [26] reduce the number of required rounds to $O(d)$ where d is the diameter of the largest connected component in a graph. These algorithms, however, still require a lot of rounds, which is critical when the data is very large; for example, the diameter of YahooWeb, a graph used in our experiments, is more than 30. Hash-Greater-to-Min [16] requires only logarithmic rounds on the number of nodes (i.e., $O(\log n)$) to compute connected components. Hash-to-Min, proposed in the same paper, shows better performance than Hash-Greater-to-Min, although Hash-to-Min does not guarantee the logarithmic round number. However, both Hash-to-Min and Hash-greater-to-Min generate large amount of intermediate data, more than double the number of edges in the initial graph, which becomes a performance bottleneck.

The work most related to ours is presented in [17], which proposes two MapReduce algorithms: *two-phase* and *alternating*. These algorithms resolve the problem of massive intermediate data of the Hash-to-Min algorithm; during the execution of the algorithms, the number of edges never increases. However, the algorithms hit another performance bottleneck because of the load-balancing problem. We introduce the alternating algorithm with more details in Section III-B, due to its relevance to our work. The authors of [17] also propose the *two-phase-DHT* algorithm which augments MapReduce with a distributed hashtable (DHT) to reduce the number of rounds. The two-phase-DHT algorithm shows better performance than the two-phase and alternating algorithms. The algorithm's performance, however, depends heavily on the performance of the DHT as it requires massive random accesses to the DHT; the two-phase-DHT algorithm is outperformed by the optimized version of the alternating algorithm on billion-scale graphs.

III. Preliminaries

In this section, we define the connected component and connected component computation, and introduce the alternating algorithm which is the baseline of our method.

A. Problem Definition

We first define the connected component.

Definition 1. *(Connected component) A connected component of an undirected graph is a set of nodes such that any two nodes are connected by paths and no node is connected to a node that is not in the set.*

In Figure 2, for example, the graph has three connected components: $\{1, 2, 4, 7, 8, 9, 10\}$, $\{5, 11\}$, and $\{3, 6, 12\}$. The goal of connected component computation is to find all connected components in a given graph. This task can be achieved by finding the minimum node reachable from each node. For instance in Figure 2, we can recognize that the nodes 1, 2, 4, 7, 8, 9, and 10 are in the same connected component by noting that these nodes share the same minimum reachable node 1; each minimum reachable node (1, 3, and 5 in the example) becomes the unique identification number of each connected component. Next, we define the problem of connected component computation as follows:

Definition 2. *(Connected Component Computation) Given an undirected graph $G = (V, E)$, the problem of connected component computation is to map each node to the identification number of the connected component containing the node where the identification number can be the minimum number of nodes in the connected component.*

For a node u, we denote by $\Gamma(u) = \{v|(u, v) \in E\}$ the neighbors of u. Additionally, we use $\Gamma^+(u) = \{v|v \in \Gamma(u), u < v\}$ and $\Gamma^-(u) = \Gamma(u) \setminus \Gamma^+(u)$ to denote the large and small neighbors of a node u, respectively. We let $m(u) = \min(\Gamma(u) \cup \{u\})$ be the minimum neighbor of u. For example in Figure 2, $\Gamma(7)$, $\Gamma^+(7)$, $\Gamma^-(7)$, and $m(7)$ are $\{1, 2, 4, 8, 9, 10\}$, $\{8, 9, 10\}$, $\{1, 2, 4\}$, and 1, respectively.

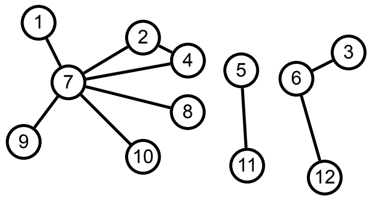

Fig. 2: An example graph with 3 connected components.

B. The Alternating Algorithm

Kiveris et al. [17] proposed two MapReduce algorithms, namely *two-phase* and *alternating*, for connected component computation. Among the two algorithms, we briefly introduce the alternating algorithm as it is the baseline of our work. Note that in practice, the alternating algorithm requires a smaller number of rounds than the two-phase algorithm, although the two-phase algorithm has a theoretical bound $O(\log^2(n))$ on the number of required rounds while the alternating algorithm does not.

The alternating algorithm conducts two core operations, the large-star and the small-star, alternately until no more edges are added or deleted. For each u, the large-star operation replaces for every $v \in \Gamma^+(u)$ the edge (v, u) with an edge $(v, m(u))$. For each u, similarly, the small-star operation replaces for every $v \in \Gamma^-(u) \setminus \{m(u)\}$ the edge (v, u) with an edge $(v, m(u))$.

A graph is transformed into a star-like graph after several rounds of the alternating algorithm. It implies that while most nodes will have few neighbors, some nodes will have a massive amount of neighbors. This phenomenon causes 'the curse of the last reducer' problem [27] which means that most computations are concentrated on few reducers, while other reducers do nothing and just wait for them.

This load-balancing problem is addressed partially in the same paper; for every node u that has more neighbors than a threshold ρ, it makes ρ copies of u, links the copies to u, and evenly distributes the neighbors of u to the copies. The copied nodes are cleaned up by one additional finalization round. This optimized alternating algorithm (shortly Alt-opt), however, is not a perfect remedy; it has the following two problems. One is that the method requires additional information about the degree of nodes in each iteration. The other is that the number of edges can increase in each round due to the edges created for the copied nodes, while the original alternating algorithm guarantees that the number of edges never increases. Note that our method does not require such an additional information, and guarantees that the number of edges does not increase (see Lemma 7).

IV. Proposed Method: Partition-Aware Connected Components

We propose PACC (Partition-Aware Connected Component), an algorithm for connected component. There are several challenges to efficiently compute connected components in a distributed environment.

1) **Load Balancing.** The alternating algorithm suffers from 'the curse of the last reducer' in the large-star operation. How can we balance the workload effectively?
2) **Minimize input/shuffled data.** The size of input and shuffled data is closely related to the performance of a distributed algorithm. How can we minimize the size of input/shuffled data of PACC?
3) **Minimize the number of rounds.** The number of rounds taken by an algorithm significantly affects the performance. How can we minimize the number of rounds of PACC?

We handle the above challenges with the following main ideas, the details of which are introduced in later subsections.

1) **Partitioning the nodes** of a graph prevents edges from getting concentrated on a few nodes; and thus, the workloads are balanced among machines. (Section IV-A)
2) **Filtering out edges that do not cross partitions or do not change in the future** significantly reduces the number of input edges in each round. Accordingly, the size of shuffled data is also decreased. (Section IV-B)
3) **Replacing several rounds of distributed computation with a single machine computation**, when the number of remaining edges becomes lower than a threshold τ by the edge-filtering, further reduces the running time. (Section IV-B)

We first describe PACC-base which partitions the nodes for load-balancing, and explain how the partitioning contributes to the load-balancing (Section IV-A). After that, we introduce our desired algorithm PACC-opt which adopts an edge-filtering method for reducing the input/shuffled data size, and we show how it enables reducing the number of rounds (Section IV-B). Lastly, we give the theoretical analysis of PACC-opt (Section IV-C).

A. *Partitioning for load-balancing*

PACC-base consists of two steps: partitioning (lines 2-5 in Algorithm 1), and computation (line 6 in Algorithm 1). The partitioning step is to partition the input graph into ρ overlapping subgraphs so that the connected components are computed independently in each subgraph in the computation step. Each partition (or subgraph) can be processed in a different or in the same machine because a partition is a logical division of data.

1) The partitioning step: The alternating algorithm has a load-balancing problem as mentioned in Section III-B, and the main cause of the problem is that edges congregate around few nodes in each round. Figure 3b shows the result of a round of the alternating algorithm given the input graph in Figure 3a. In this example, the alternating algorithm gathers all the edges to the minimum node 1; consequently, computations are concentrated on the node 1 in the next round. PACC-base resolves this problem by partitioning nodes; each node is linked to the minimum node within the same partition instead of the 'global' minimum node. A random hash function $\xi : V \rightarrow \{0, \cdots, \rho - 1\}$ is used for partitioning nodes where ρ is the number of partitions, and we denote by $\xi(v)$ the partition of a node v. The partition id of each node is randomly selected. The partitioning prevents edges from getting concentrated on a few nodes. Thus, the workloads are distributed into partitions. For example in Figure 3c, PACC-base divides the nodes into two partitions, and the edges are distributed into the partitions.

The partitioning step consists of several rounds, and each round conducts two distributed operations: PA-large-star and PA-small-star. The two operations are similar to the large-star and the small-star operations of the alternating algorithm, respectively; additionally, PA-large-star and PA-small-star consider the partitions of nodes. Let $[S]_i = \{v | v \in S, \xi(v) = i\}$ be the i-th partition of a set S. Let $m(u) = \min(\Gamma(u) \cup \{u\})$ be the minimum neighbor of u, and $m_i(u) = \min([\Gamma(u) \cup \{u\}]_i)$ be the 'local' minimum neighbor of u in the i-th partition. For each node $u \in V$ and for some neighbors v of u, PA-large-star and PA-small-star replace the edge (v, u) with an edge $(v, m_{\xi(v)}(u))$ if $v \neq m_{\xi(v)}(u)$, or $(v, m(u))$ if $v = m_{\xi(v)}(u)$ and $v \neq m(u)$. PA-large-star is responsible for large neighbors $\Gamma^+(u)$ of u, and PA-small-star is responsible

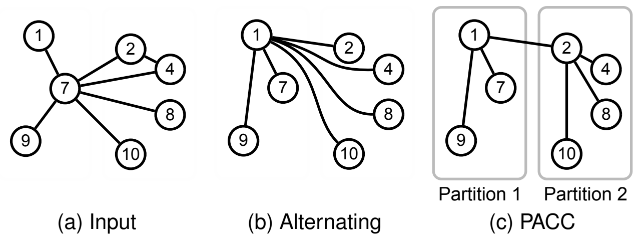

Fig. 3: The results of (b) a round of the alternating algorithm, and (c) a round of the proposed algorithm PACC-base, given the input graph in (a). While the alternating algorithm gathers all the edges to node 1, PACC-base distributes the edges into several partitions (2 partitions in this example).

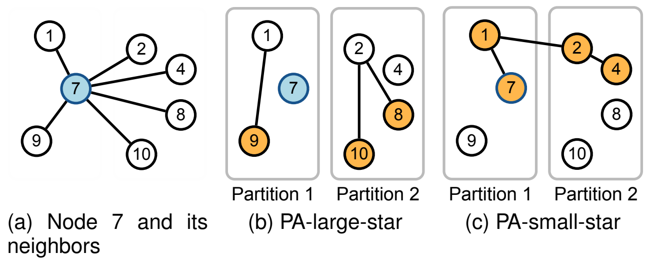

Fig. 4: An illustration of PA-large-star and PA-small-star operations at node 7. The nodes with an orange color are linked to the local minimum node in each partition. The local minimum node 2 in the partition 2 is linked to the global minimum node 1.

Algorithm 1: PACC-base

```
Input: Edges $(u, v)$ as a set $E$ of key-value pairs $\langle u; v \rangle$
Output: A unique connected component id for every node $v \in V$
1 $out \leftarrow E$
  // Partitioning step: lines 2 through 5
2 repeat
3     $out \leftarrow$ PA-large-star($out$)
4     $out \leftarrow$ PA-small-star($out$)
5 until Convergence;
6 return CC-Computation(out) ;                // Computation step
```

Algorithm 2: PA-large-star

```
  Map     : input $\langle u; v \rangle$
1 emit $\langle u; v \rangle$ and $\langle v; u \rangle$

  Reduce  : input $\langle u; \Gamma(u) \rangle$
2 foreach $v \in \Gamma^{+}(u)$ do
3     if $v \neq m_{\xi(v)}(u)$ then
4         emit $\langle v; m_{\xi(v)}(u) \rangle$
5     else if $v \neq m(u)$ then
6         emit $\langle v; m(u) \rangle$
```

Algorithm 3: PA-small-star

```
  Map     : input $\langle u; v \rangle$
1 emit $\langle \max(u, v); \min(u, v) \rangle$

  Reduce  : input $\langle u; \Gamma^{-}(u) \rangle$
2 foreach $v \in \Gamma^{-}(u) \cup \{u\}$ do
3     if $v \neq m_{\xi(v)}(u)$ then
4         emit $\langle v; m_{\xi(v)}(u) \rangle$
5     else if $v \neq m(u)$ then
6         emit $\langle v; m(u) \rangle$
```

Algorithm 4: CC-Computation

```
  Map     : input $\langle u; v \rangle$
1 emit $\langle \xi(\max(u, v)); (u, v) \rangle$

  Reduce  : input $\langle p; E_p \rangle$
2 LocalCC($E_p$)
```

for small neighbors and u, i.e., $\Gamma^{-}(u) \cup \{u\}$. Figure 4 shows an illustration of PA-large-star and PA-small-star operations at node $u = 7$. The local minimum nodes in the two partitions are the node 1 and the node 2. PA-large-star links the large neighbors of the node 7 (i.e., $\Gamma^{+}(7) = \{8, 9, 10\}$) to the local minimum node in each partition; the node 9 is connected to the node $m_1(7) = 1$, and the node 8 and 10 are connected to the node $m_2(7) = 2$. PA-small-star is in charge of the node $u = 7$ and its small neighbors $\Gamma^{-}(7) = \{1, 2, 4\}$; the node 7 is connected to the node $m_1(7) = 1$, and the node 4 is connected to the node $m_2(7) = 2$. The node 2 is connected to the global minimum neighbor $m(7) = 1$ of the node 7 because the node 2 is a local minimum node of the node 7.

Both PA-large-star and PA-small-star can be easily implemented in a distributed manner. The MapReduce version pseudo codes are in Algorithm 2 and Algorithm 3.

Lemma 1. *PA-large-star and PA-small-star keep the connectivity of the original input graph.*

Proof. Let us consider a node u and its neighbors $\Gamma(u)$. After PA-large-star on u, every node $v \in \Gamma^{+}(u)$ has a path to the global minimum node $m(u)$ of u: v is directly connected to $m(u)$ if $v = m_{\xi(v)}(u)$, or via $m_{\xi(v)}(u)$ in the other case. Similarly, performing PA-small-star on u connects $v \in \Gamma^{-}(u) \cup \{u\}$ to $m(u)$ directly if $v = m_{\xi(v)}$, or via $m_{\xi(v)}(u)$ in the other case. Accordingly, PA-large-star and PA-small-star share the same connectivity with the large-star and the small star operations which preserve the connectivity of the graph as proved in Lemma 1 and 3 of [17]. $\square$

2) The computation step: After several rounds, a graph is divided into ρ overlapping subgraphs. Each subgraph is an induced subgraph on the nodes in a partition and their small neighbors, if any, that are in different partitions. In Figure 3c, the induced subgraphs on the nodes $\{1, 7, 9\}$ and $\{1, 2, 4, 8, 10\}$ are the examples. The CC-Computation operation in the computation step finds connected components in each subgraph independently: it first groups all edges according to the subgraph they belong to, and then computes the connected components in each subgraph using a single machine algorithm. The CC-Computation operation can be implemented as a single round MapReduce algorithm as in Algorithm 4. Each map operation sends an edge (u, v) to a partition $\xi(\max(u, v))$. Then, each reduce operation receives a set of edges E_p corresponding to the partition p, and finds connected components in the edge-induced subgraph on E_p by delegating the task to LocalCC, a single machine algorithm. Any single machine algorithm that finds the minimum node reachable from each node in the given edges can be used for LocalCC; we use a Union-Find algorithm with the path compression [28] for our experiments.

Note that, even though each subgraph contains a part of a graph, the CC-Computation operation finds the 'global' minimum nodes reachable from each node because local minimum nodes are connected to the global minimum nodes, and all the other nodes are connected to the local minimum nodes. We explicitly claim the following lemma.

Lemma 2. *After the partitioning step of PACC-base, each subgraph is a set of star graphs where the center nodes are the local minimum nodes in the corresponding partition, and each star graph contains the global minimum node of the corresponding connected component.*

3) Putting it together: We present the PACC-base algorithm in Algorithm 1. PACC-base partitions the input graph by alternately performing PA-large-star and PA-small-star operations on it until convergence. The algorithm converges when there is no change in the graph structure, that is, when the sum of the number of edges modified by both operations in one iteration becomes zero. From the resulting, overlapping subgraphs, PACC-base then finds the connected components using the CC-Computation operation.

B. Edge-filtering

In this section, we present our proposed algorithm, PACC-opt, which optimizes PACC-base with the edge-filtering process. We start from the observation that the number of edges

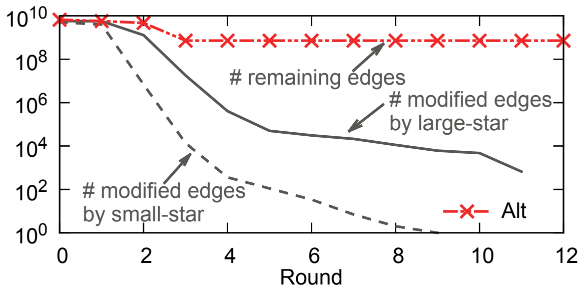

Fig. 5: The number of remaining edges and the number of modified edges by large-star and small-star operations in each round of the alternating algorithm on YahooWeb. While the number of modified edges decreases rapidly, that of the remaining edges does not decrease below a point, that is, the number of nodes. This implies that most edges just come and go with no change.

affected by the large-star and the small-star operations drastically decreases in each round, while the number of remaining edges does not decrease below the number of nodes in a graph (see Figure 5). This implies that most edges just come and go with no changes.

Lemma 3. *The number of input edges at each round of the alternating algorithm does not decrease under $|V|-|C|$ where $|V|$ is the number of nodes and $|C|$ is the number of connected components.*

Proof. The final output of the alternating algorithm is a set of star graphs where each node is connected to the minimum node in its connected component. That is, the final output contains $|V|-|C|$ edges. Meanwhile, by the Lemma 2 and 3 in the paper [17], the number of output edges of each round does not increase. It implies that every round gets no less than $|V|-|C|$ edges as inputs. □

The partitioning step of PACC-opt is similar to that of the alternating algorithm, but the goal of the step is partitioning rather than computing connected components. Concentrating on partitioning enables PACC-opt to exclude a lot of edges, which leads to a significant decrease in the size of input and shuffled data. There are two cases that PACC-opt filters an edge (u, v) out, where we assume $u < v$ without loss of generality:

Case 1. The two nodes are in the same partition, and v has no neighbor except u. That is, $\xi(u) = \xi(v)$ and $\Gamma(v) = \{u\}$.

Case 2. The node u has no small neighbor, and all the neighbors of u have no neighbor except u. That is, $\Gamma^{-}(u) = \emptyset$, and $\Gamma(n) = \{u\}$ for every $n \in \Gamma(u)$.

The key idea of the edge-filtering in case 1 is from the fact that CC-Computation of PACC-opt finds connected components in each subgraph independently. The only requirement for correctly finding connected components is that the global minimum nodes of each connected component should be reachable from every node within each partition. For this purpose, filtering an edge in case 1 makes the edge to stay in a partition, and does not let the edge cross partitions. As a result, the edges filtered in case 1 form a set of tree graphs where the edges do not cross partitions.

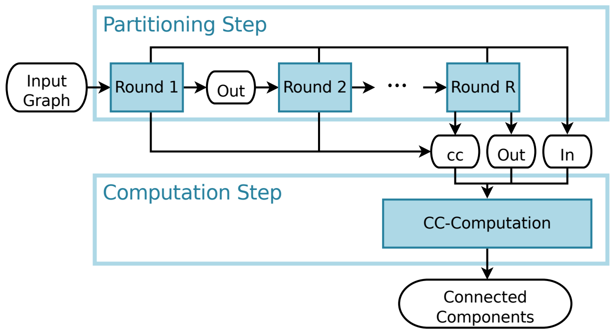

Fig. 6: Data-flow of PACC-opt. Each round of the partitioning step emits three different sets: ‘out’, ‘in’, and ‘cc’. The ‘out’ set of each round is the input of the next round, and some edges are accumulated in the ‘in’ set or the ‘cc’ set. The computation step gets the ‘out’ set of the final round, the ‘in’ set, and the ‘cc’ set as inputs, and finalizes connected components.

Lemma 4. *The edges filtered in case 1 together form a set of tree graphs.*

Proof. By the definition of the case 1, an edge (v, u), assuming $u < v$, is filtered out when u and v are in the same partition, and v has no neighbor except u. Thus, the node v has no small neighbors except u and no large neighbors as well. However, v could have had large neighbors v_d at one time, which had been filtered out later by the case 1; in that circumstance, by the same reason as above, the node v was the only small neighbor of the nodes v_d, meaning that there is no cycle. Thus the claim follows. □

Edges of case 2 together constitute a star graph where the center node is the minimum. By the definition of PA-large-star and PA-small-star operations, these edges do not change in the following rounds. Thus, we can simply exclude such edges from the input of the next round.

Each round in the partitioning step filters such edges out and excludes them in the next round by emitting three different edge sets: ‘out’, ‘in’, and ‘cc’; only the ‘out’ set becomes the input of the next round, while the ‘in’ and the ‘cc’ sets are not used as input. Edges of the case 1 and the case 2 are emitted to the ‘in’ set and the ‘cc’ set, respectively, whose elements are accumulated from all the rounds. Edges in other cases are emitted to the ‘out’ set, which becomes the input of the next round. The union of the ‘in’ set, the ‘cc’ set, and the ‘out’ set of the final round becomes the input of the CC-Computation operation which computes connected components in each subgraph. The data-flow of PACC-opt is depicted in Figure 6.

The MapReduce version pseudo code of PA-large-star and PA-small-star with the edge-filtering (namely PA-large-star-opt and PA-small-star-opt) are listed in Algorithms 5 and 6, respectively. Edges of the case 1 are filtered in PA-small-star-opt. Given a node u, PA-small-star-opt links each small

Algorithm 5: PA-large-star-opt

```
Map    : input $\langle u; v \rangle$
emit $\langle u; v \rangle$ and $\langle v; u \rangle$

Reduce  : input $\langle u; \Gamma(u) \rangle$
if $u = m(u)$ and $\Gamma(v) = \{u\}\ \forall v \in \Gamma(u)$ then
    foreach $v \in \Gamma^+(u)$ do
        emit $\langle v; u \rangle$ to $lcc$
else
    foreach $v \in \Gamma^+(u)$ do
        if $v \neq m_{\xi(v)}(u)$ then
            emit $\langle v; m_{\xi(v)}(u) \rangle$ to $lout$
        else if $v \neq m(u)$ then
            emit $\langle v; m(u) \rangle$ to $lout$
```

Algorithm 6: PA-small-star-opt

```
Map    : input $\langle u; v \rangle$
emit $\langle u; v \rangle$ and $\langle v; u \rangle$

Reduce  : input $\langle u; \Gamma(u) \rangle$
foreach $v \in \Gamma^-(u) \cup \{u\}$ do
    if $v \neq m_{\xi(v)}(u)$ then
        if $v = u$ and $\Gamma^+(u) = \emptyset$ then
            emit $\langle v; m_{\xi(v)}(u) \rangle$ to $sin$
        else
            emit $\langle v; m_{\xi(v)}(u) \rangle$ to $sout$
    else if $v \neq m(u)$ then
        add a tag to $v$ if $v = u$ and $\Gamma^+(u) = \emptyset$
        emit $\langle v; m(u) \rangle$ to $sout$
```

neighbor $v \in \Gamma^-(u)$ of u and u itself to the local minimum node $m_{\xi(v)}(u)$. Then, if $u \neq m_{\xi(u)}(u)$ and u has no large neighbors, the neighbor set of u after PA-small-star-opt becomes $\{m_{\xi(u)}(u)\}$. In this case (i.e., $u \neq m_{\xi(u)}(u)$ and $\Gamma^+(u) = \emptyset$), PA-small-star-opt outputs $(u, m_{\xi(u)}(u))$ separately (to sin at line 5 of Algorithm 6), and excludes this edge from the input of the next round. Edges of the case 2 are filtered in PA-large-star-opt. Given a node u, PA-large-star-opt gathers all the neighbors $\Gamma(u)$ of u, and thus we can easily know whether u has a small neighbor or not by comparing u and $m(u)$: $u = m(u)$ means u has no small neighbor. For each neighbor v of u, PA-large-star-opt sees whether $\Gamma(v) = \{u\}$ by checking the existence of a tag of the edge (v, u) which we attach in PA-small-star-opt of the previous round if v has no large neighbor (line 9 of Algorithm 6).

The edge-filtering not only decreases the amount of data to read and write, but also drastically reduces the number of edges to the point where the input edges can be processed by a single machine. This enables an additional optimization: PACC-opt replaces several MapReduce rounds with one round of a single machine algorithm (LocalCC) when the input size is small enough, that is, smaller than a threshold τ. This optimization saves preparation time for multiple rounds. The pseudo code for PACC-opt is listed in Algorithm 7. As in CC-Computation, we use a Union-Find algorithm for LocalCC in our experiments.

Algorithm 7: PACC-opt

```
Input: Edges $(u, v)$ as a set $E$ of key-value pairs $\langle u; v \rangle$
Output: A unique connected component id for every node $v \in V$
$out \leftarrow E$
$in \leftarrow \emptyset$
$cc \leftarrow \emptyset$
repeat
    if # edges in out $> \tau$ then
        $(lout, lcc) \leftarrow$ PA-large-star-opt($out$)
        $cc \leftarrow cc \cup lcc$
        $(sout, sin) \leftarrow$ PA-small-star-opt($lout$)
        $out \leftarrow sout$
        $in \leftarrow in \cup sin$
    else
        $out \leftarrow$ LocalCC($out$)
until Convergence;
return CC-Computation($out \cup in \cup cc$)
```

C. Analysis

We first prove the correctness of PACC-opt.

Lemma 5. *PACC-opt correctly finds all connected components.*

Proof. The input of the CC-Computation operation is the union of the 'in' set, the 'cc' set, and the 'out' set of the final round. As shown in Lemma 4, the 'in' set contains only tree graphs where the root of each tree graph is a local minimum node of the corresponding connected component. In the 'cc' set and the 'out' set of the final round, the local minimum nodes of the connected components are connected to the global minimum nodes as proved in Lemma 2. Thus, as the union of the 'in' set and the 'out' set of the final round forms tree graphs whose root nodes are the global minimum nodes, each node is reachable from the global minimum node in the same connected component. Note that, every edge in the 'in' set does not cross partitions by the definition of case 1. Each subgraph processed by CC-Computation is an induced subgraph on the nodes in a partition and their small neighbors. It implies that each subgraph contains the global minimum nodes, which are linked to the local minimum nodes, and thus CC-Computation correctly finds connected components in each subgraph independently. ☐

We now prove an upper bound on the input size of the CC-Computation.

Lemma 6. *The number of input edges of CC-Computation in PACC-opt is not larger than $|V| - 1$ where $|V|$ is the number of nodes.*

Proof. As shown in the proof of Lemma 5, the input of CC-Computation forms a set of tree graphs. Thus, the number of edges is not larger than $|V| - 1$. Accordingly, the expected number of edges in each subgraph is $(|V| - 1)/\rho$ when the hash function ξ evenly distributes the nodes into partitions. ☐

Lemma 7. *PA-large-star and PA-small-star operations do not increase the number of edges.*

Proof. Without loss of generality, let us assume $u < v$ in every edge $(v, u) \in E$. Then, it is trivial to show that PA-large-star and PA-small-star do not duplicate any edge because

TABLE II: The summary of datasets.

Dataset	$\lvert V \rvert$	$\lvert E \rvert$	Source
Skitter (SK)	1,696,415	11,095,298	SNAP[1]
Patent (PT)	3,774,768	16,518,948	SNAP
LiveJournal (LJ)	4,847,571	68,993,773	SNAP
Friendster (FS)	65,608,366	1,806,067,135	SNAP
Twitter (TW)	41,652,230	1,468,365,182	Kwak et al.[2] [29]
SubDomain (SD)	89,247,739	2,043,203,933	Webscope[3]
YahooWeb (YW)	720,242,173	6,636,600,779	Webscope
RMAT 21	1,115,331	31,457,280	N/A (Synthetic graphs)
RMAT 23	4,118,887	125,829,120	
RMAT 25	15,215,025	503,316,480	
RMAT 27	56,101,382	2,013,265,920	
RMAT 29	207,015,915	8,053,063,680	

they just replace each edge with another edge. PA-large-star replaces an edge (v, u) with $(v, m(u))$ or $(v, m_{\xi(v)}(u))$. Similarly, PA-small-star replaces an edge (v, u) with $(u, m(v))$, $(u, m_{\xi(u)}(v))$, or (v, u). □

V. Experiments

In this section, we experimentally evaluate PACC. We aim to answer the following questions.

Q1 Effect of Partitioning (Section V-B1). How well does the node-partitioning (in Section IV-A) evenly distribute workloads?

Q2 Effect of Edge-Filtering (Section V-B2). How much does the edge-filtering (in Section IV-B) reduce the number of input edges?

Q3 Scalability (Section V-B3). How does PACC scale up in terms of the number of machines and the data size?

We first introduce datasets and experimental environments in Section V-A. Then, we answer the questions in Section V-B presenting the experimental results.

A. Setup

1) Datsets: We use both real world and synthetic graphs to evaluate the proposed algorithm. We bring the real world graphs from various sources. The datasets and the sources are listed in Table II. Skitter is an internet topology graph. Patent is a citation network among US patents. LiveJournal and Friendster are friendship networks in online communities of the same names. Twitter is a follower-followee network in a social network service Twitter. SubDomain and YahooWeb are hyperlink networks of domain level and page level, respectively. Also, we generate synthetic graphs with different number of nodes and edges using RMAT [30], which is a widely-used model for generating graphs that match the characteristics of real-world networks, such as power-law degree distribution and community structure, with a recursive and random process. We set the four RMAT parameters (a, b, c, d) to $(0.57, 0.19, 0.19, 0.05)$, and use TeGViz [24], a distributed graph generator, to generate large-scale RMAT graphs that exceed the capacity of a single machine. Note that we assume all the graphs are undirected even if they are originally directed; if there are two edges (u, v) and (v, u), we consider that they are the same and remove one of them at the beginning of each algorithm.

[1]http://snap.stanford.edu/data/
[2]http://an.kaist.ac.kr/traces/WWW2010.html
[3]http://webscope.sandbox.yahoo.com

2) Environment: We implement PACC-base (in Section IV-A), PACC-opt (in Section IV-B), the alternating algorithm (Alt), and its optimized version (Alt-opt), on Hadoop which is the de facto standard implementation of MapReduce. For the threshold parameter τ of PACC-opt, 20M and 0 are used to show the effect of the LocalCC optimization. The number ρ of partitions for PACC-base, PACC-opt, and the optimized alternating algorithms is set to the number of used reducers; 100 unless otherwise stated. We use a cluster server with 41 machines for the experiments. Each machine is equipped with an Intel Xeon E3-1230v3 CPU (quad-core at 3.30GHz), and 32GB RAM. Hadoop v1.2.1 is installed on the cluster with 1 master machine and 40 slave machines. Each slave machine can run 3 mappers and 3 reducers (120 mappers and 120 reducers in total) concurrently. The available memory size for each mapper and reducer is set to 4GB.

B. Results

1) Effect of Partitioning: In order to show the effect of partitioning, we present a box-and-whisker plot of the running time of reducers in each round of PACC-base, PACC-opt ($\tau = 0$), the alternating, and the optimized alternating algorithms in Figure 7. We only show the result on the YahooWeb graph because results on other graphs have similar trends. The last round of PACC-base and PACC-opt is for CC-Computation, and that of Alt-opt is for the finalization. We exclude the running time of mappers because they are load-balanced well in any cases. The bottom and the top of each box are the first and the third quartile, respectively, and the ends of whiskers represent the maximum and the minimum. A long whisker implies a bad load-balancing as a few reducers take much longer time than others. In the figure, PACC-base and PACC-opt show the best load-balancing with very short whiskers regardless of the rounds. The optimized alternating algorithm also reduces the lengths of whiskers from the alternating algorithm, but they are longer than those of PACC-base and PACC-opt. Note that, the running time of PACC-opt dramatically decreases in each round, thanks to the reduced input size by the edge-filtering.

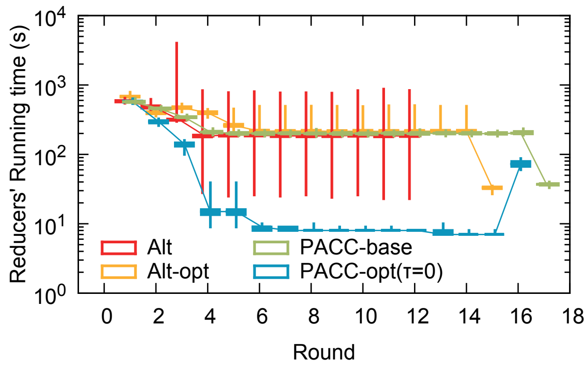

Fig. 7: A box-and-whisker plot of the running time of reducers in each round on YahooWeb. In PACC-base and PACC-opt, reducers take similar time regardless of the rounds.

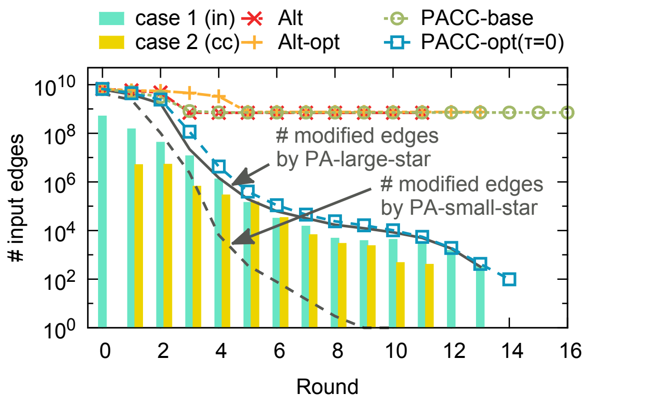

Fig. 8: The number of input edges of PACC-base, PACC-opt, the alternating, and the optimized alternating on the YahooWeb graph at each round. While the number of input edges of the other algorithms does not decrease below the number of nodes, that of PACC-opt decreases, and almost hits the lower bound, i.e. the number of modified edges.

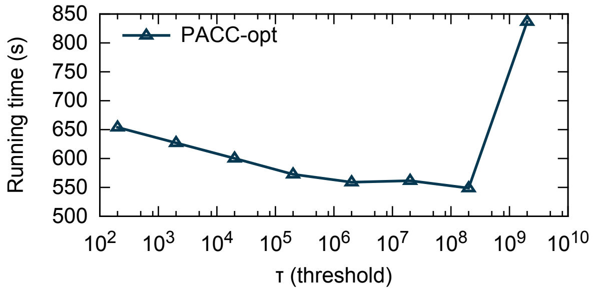

Fig. 9: The running time vs. τ. PACC-opt shows the best performance with τ = 200M.

2) *Effect of Edge-Filtering:* Figure 8 shows the number of input edges in each round of PACC-base, PACC-opt, the alternating, and the optimized alternating algorithms on the YahooWeb graph. While the input size of the other algorithms does not decrease under a point (the number of non-root nodes), that of PACC-opt rapidly decreases. The solid and dashed lines represent the number of edges modified by PA-large-star and PA-small-star, respectively; the upper solid line (PA-large-star) is the lower-bound of the input size as an edge to be modified should be in the input at that round. The input size of PACC-opt is very close to the lower-bound. Vertical bars with the two colors represent the number of edges excluded by the cases 1 and 2 in Section IV-B; that is, the number of edges output to the 'in' set and the 'cc' set, respectively, at that round. Edges are mainly filtered by the case 1, but the case 2 is not negligible. The reduced number of input edges effectively reduces the running time as shown in Figure 7.

To show the effect of the threshold τ, Figure 9 presents the running time of PACC-opt with the threshold τ varying from 200 to 2 billion on Twitter. PACC-opt shows the best performance with τ = 200M. When τ = 2 billion, LocalCC gets the entire Twitter graph as the input; then the total running time is dominated by that of the single machine algorithm, LocalCC.

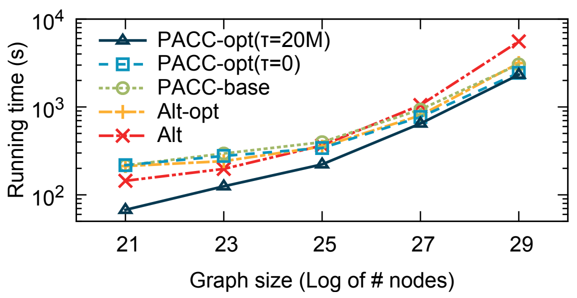

Fig. 10: The running time of PACC-base, PACC-opt, the alternating, and the optimized alternating on RMAT graphs with various sizes. PACC-opt with $\tau = 0$ exhibits the best scalability with the increase of graph size while PACC-opt with τ = 20M requires the shortest running time in all cases.

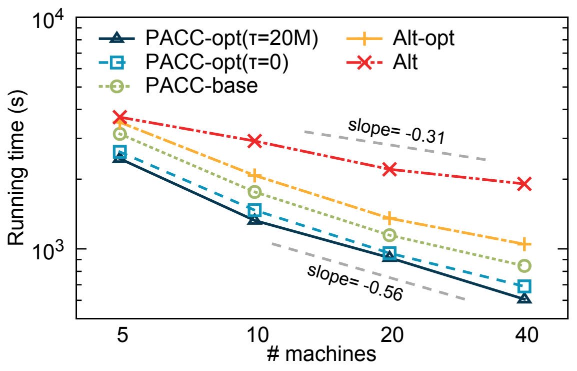

Fig. 11: Machine-scalability. The running time of PACC-base, PACC-opt, the alternating, and the optimized alternating on Twitter with varying number of machines. All three PACC variants scale up better with the increase of the number of machines than the alternating algorithms.

3) *Scalability:* Figure 10 shows the running time of PACC-base, PACC-opt, the alternating, and the optimized alternating algorithms on RMAT graphs with various sizes. Among PACC variants, PACC-opt with τ=0 exhibits the best scalability: while in general the performance of all three PACC variants scales up near linearly with the increase of graph size, the rate of increase in the running time of PACC-opt with τ=0 is the smallest among all tested algorithms. However, the running time of the alternating algorithm increases more rapidly than that of PACC-opt with τ=0 due to poor load-balancing, and is the greatest on RMAT 29 graph.

Figure 11 presents the machine scalability of the tested algorithms, showing the running time of each algorithm on Twitter with various numbers of machines. Both X and Y axes are in a log scale. The slope of each line represents the machine scalability of each algorithm: the lower the slope is, the better the machine scalability is. PACC-opt shows the best machine scalability (slope = -0.56) followed by PACC-base and Alt-opt. The alternating algorithm has the worst machine scalability (slope ≥ -0.31), and the main reason is the bad load-balancing; workloads are focused on few machines and the amount of work those machines take does not change much even if the number of machines increases.

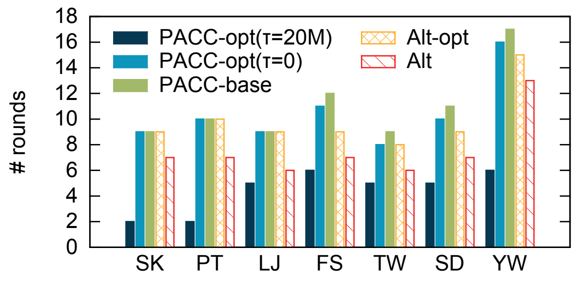

Fig. 12: The number of rounds required by PACC-base, PACC-opt, the alternating, and the optimized alternating algorithm on various real world graphs in Table II. PACC-opt with $\tau = 20$M requires the smallest number of rounds in all cases.

4) Results on Real-World Graphs: We test PACC-base, PACC-opt, the alternating, and its optimized version on various real world graphs. Figure 1 shows the speedup over the optimized alternating algorithm on seven real world graphs. The proposed algorithm PACC-opt with τ=20M shows the best performance on all graphs, and it shows up to 10 times faster performance than the optimized alternating algorithm on the Patent graph. The number of rounds taken by each algorithm is presented in Figure 12. PACC-opt with τ=20M takes the least number of rounds among the used algorithms, and this is due to the edge-filtering and the thresholding. PACC-opt reduces the number of input edges dramatically in each round, which becomes smaller than 20M before the 6-th round on every graph. Even though PACC-opt with τ=0 takes the most rounds, it shows similar performance as PACC-opt with τ=20M on large graphs thanks to the reduced number of edges in each round; as shown in Figure 8, each round of PACC-opt takes much shorter time than the other algorithms.

VI. Conclusion

In this paper, we propose PACC, a scalable distributed algorithm for computing connected components in an enormous graph. PACC gets performance improvement from evenly distributing workloads by partitioning nodes, and minimizing the intermediate data size and the number of rounds by filtering unnecessary edges. PACC shows the best performance on real world graphs: it provides up to 10$\times$ faster performance than the current state-of-the-art MapReduce algorithm. Future work includes extending the method for other graph algorithms including subgraph enumeration.

Acknowledgments

This work was supported by Research Resettlement Fund for the new faculty of Seoul National University. This work was also supported by the National Research Foundation of Korea (NRF) funded by the Ministry of Science, ICT and Future Planning(NRF-2015K1A3A1A14021055), and by Institute for Information & communications Technology Promotion (IITP) grant funded by the Korea government (MSIP) (No.R0190-15-2012, High Performance Big Data Analytics Platform Performance Acceleration Technologies Development).

References

[1] L. He, Y. Chao, K. Suzuki, and K. Wu, “Fast connected-component labeling,” *Pattern Recognition*, vol. 42, no. 9, pp. 1977–1987, 2009.
[2] U. Kang, M. McGlohon, L. Akoglu, and C. Faloutsos, “Patterns on the connected components of terabyte-scale graphs,” in *ICDM*, 2010, pp. 875–880.
[3] D. Medini, A. Covacci, and C. Donati, “Protein homology network families reveal step-wise diversification of type III and type IV secretion systems,” *PLoS Computational Biology*, vol. 2, no. 12, 2006.
[4] R. Albert, “Scale-free networks in cell biology,” *Journal of cell science*, vol. 118, no. 21, pp. 4947–4957, 2005.
[5] U. Kang and C. Faloutsos, “Beyond ’caveman communities’: Hubs and spokes for graph compression and mining,” in *ICDM*, 2011, pp. 300–309.
[6] Y. Lim, U. Kang, and C. Faloutsos, “Slashburn: Graph compression and mining beyond caveman communities,” *TKDE*, vol. 26, no. 12, pp. 3077–3089, 2014.
[7] Y. Lim, W. Lee, H. Choi, and U. Kang, “Discovering large subsets with high quality partitions in real world graphs,” in *BIGCOMP*, 2015, pp. 186–193.
[8] ——, “Mtp: discovering high quality partitions in real world graphs,” *WWW*, pp. 1–24, 2016.
[9] J. Jung, K. Shin, L. Sael, and U. Kang, “Random walk with restart on large graphs using block elimination,” *ACM Trans. Database Syst.*, vol. 41, no. 2, p. 12, 2016.
[10] A. Kyrola, G. E. Blelloch, and C. Guestrin, “Graphchi: Large-scale graph computation on just a PC,” in *OSDI*, 2012, pp. 31–46.
[11] M. Kim, S. Lee, W. Han, H. Park, and J. Lee, “DSP-CC-: I/O efficient parallel computation of connected components in billion-scale networks,” *TKDE*, vol. 27, no. 10, pp. 2658–2671, 2015.
[12] 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*, 2010, pp. 135–146.
[13] 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*, vol. 5, no. 8, pp. 716–727, 2012.
[14] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, “Powergraph: Distributed graph-parallel computation on natural graphs,” in *OSDI*, 2012, pp. 17–30.
[15] U. Kang, C. E. Tsourakakis, and C. Faloutsos, “PEGASUS: A peta-scale graph mining system,” in *ICDM*, 2009, pp. 229–238.
[16] V. Rastogi, A. Machanavajjhala, L. Chitnis, and A. D. Sarma, “Finding connected components in map-reduce in logarithmic rounds,” in *ICDE*, 2013, pp. 50–61.
[17] R. Kiveris, S. Lattanzi, V. S. Mirrokni, V. Rastogi, and S. Vassilvitskii, “Connected components in mapreduce and beyond,” in *SoCC*, 2014, pp. 18:1–18:13.
[18] M. M. A. Patwary, P. Refsnes, and F. Manne, “Multi-core spanning forest algorithms using the disjoint-set data structure,” in *IPDPS*, 2012, pp. 827–835.
[19] D. A. Bader and G. Cong, “A fast, parallel spanning tree algorithm for symmetric multiprocessors (smps),” *JPDC*, vol. 65, no. 9, pp. 994–1006, 2005.
[20] J. Dean and S. Ghemawat, “Mapreduce: simplified data processing on large clusters,” *Commun. ACM*, vol. 51, no. 1, pp. 107–113, 2008.
[21] H. Park and C. Chung, “An efficient mapreduce algorithm for counting triangles in a very large graph,” in *CIKM*, 2013, pp. 539–548.
[22] H. Park, F. Silvestri, U. Kang, and R. Pagh, “Mapreduce triangle enumeration with guarantees,” in *CIKM*, 2014, pp. 1739–1748.
[23] U. Kang, J. Y. Lee, D. Koutra, and C. Faloutsos, “Net-ray: Visualizing and mining billion-scale graphs,” in *PAKDD*, 2014, pp. 348–361.
[24] B. Jeon, I. Jeon, and U. Kang, “Tegviz: Distributed tera-scale graph generation and visualization,” in *ICDM*, 2015, pp. 1620–1623.
[25] J. Cohen, “Graph twiddling in a mapreduce world,” *CS&E*, vol. 11, no. 4, pp. 29–41, 2009.
[26] S. J. Plimpton and K. D. Devine, “Mapreduce in MPI for large-scale graph algorithms,” *Parallel Comp.*, vol. 37, no. 9, pp. 610–632, 2011.
[27] S. Suri and S. Vassilvitskii, “Counting triangles and the curse of the last reducer,” in *WWW*, 2011, pp. 607–614.
[28] T. H. Cormen, *Introduction to algorithms*. MIT press, 2009.
[29] H. Kwak, C. Lee, H. Park, and S. Moon, “What is Twitter, a social network or a news media?” in *WWW*, 2010, pp. 591–600.
[30] D. Chakrabarti, Y. Zhan, and C. Faloutsos, “R-MAT: A recursive model for graph mining,” in *SDM*, 2004, pp. 442–446.