PTE: Enumerating Trillion Triangles On Distributed Systems

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

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

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

ABSTRACT

How can we enumerate triangles from an enormous graph with billions of vertices and edges? Triangle enumeration is an important task for graph data analysis with many applications including identifying suspicious users in social networks, detecting web spams, finding communities, etc. However, recent networks are so large that most of the previous algorithms fail to process them. Recently, several MapReduce algorithms have been proposed to address such large networks; however, they suffer from the massive shuffled data resulting in a very long processing time.

In this paper, we propose PTE (*Pre-partitioned Triangle Enumeration*), a new distributed algorithm for enumerating triangles in enormous graphs by resolving the structural inefficiency of the previous MapReduce algorithms. PTE enumerates trillions of triangles in a billion scale graph by decreasing three factors: the amount of shuffled data, total work, and network read. Experimental results show that PTE provides up to 47× faster performance than recent distributed algorithms on real world graphs, and succeeds in enumerating more than 3 *trillion* triangles on the ClueWeb12 graph with 6.3 billion vertices and 72 billion edges, which any previous triangle computation algorithm fail to process.

CCS Concepts

•**Information systems** → Data mining; •**Theory of computation** → **Parallel algorithms; Distributed algorithms;** *MapReduce algorithms; Graph algorithms analysis;*

Keywords

triangle enumeration; big data; graph algorithm; scalable algorithm; distributed algorithm; network analysis

*Corresponding Author

KDD '16, August 13-17, 2016, San Francisco, CA, USA
© 2016 ACM. ISBN 978-1-4503-4232-2/16/08...$15.00
DOI: `http://dx.doi.org/10.1145/2939672.2939757`

1. INTRODUCTION

How can we enumerate trillion triangles from an enormous graph with billions of vertices and edges? The problem of *triangle enumeration* is to discover every triangle in a graph one by one where a triangle is a set of three vertices connected to each other. The problem has numerous graph mining applications including detecting suspicious accounts like advertisers or fake users in social networks [29, 16], uncovering hidden thematic layers on the web [8], discovering roles [5], detecting web spams [3], finding communities [4, 24], etc. A challenge in triangle enumeration is handling big real world networks, such as social networks and WWW, which have millions or billions of vertices and edges. For example, Facebook and Twitter have 1.39 billion [9] and 300 million active users [27], respectively, and at least 1 trillion unique URLs are on the web [1].

Even recently proposed algorithms, however, fail to enumerate triangles from such large graphs. The algorithms have been proposed in different ways: I/O efficient [13, 21, 19], distributed memory [2, 11], and MapReduce algorithms [6, 22, 23, 26]. These algorithms have a limited scalability. The I/O efficient algorithms use only a single machine, and thus they cannot process a graph exceeding the external memory space of the machine. The distributed memory algorithms use multiple machines, but they also cannot process a graph whose intermediate data exceed the capacity of distributed-memory. The state of the art MapReduce algorithm [23], named CTTP, significantly increases the size of processable dataset by dividing the entire task into several sub-tasks and processing them in separate MapReduce rounds. Even CTTP, however, takes a very long time to process an enormous graph because, in every round, CTTP reads the entire dataset and shuffles a lot of edges. Indeed, shuffling a large amount of data in a short time interval causes network congestion and heavy I/O to disks which decrease the scalability and the fault tolerance, and prolong the running time significantly. Thus, it is desirable to shrink the amount of shuffled data.

In this paper, we propose PTE (*Pre-partitioned Triangle Enumeration*), a new distributed algorithm for enumerating triangles in an enormous graph by resolving the structural inefficiency of the previous MapReduce algorithms. We show that PTE successfully enumerates trillions of triangles in a billion scale graph by decreasing three factors: the amount of shuffled data, total work, and network read. The main contributions of this paper are summarized as follows:

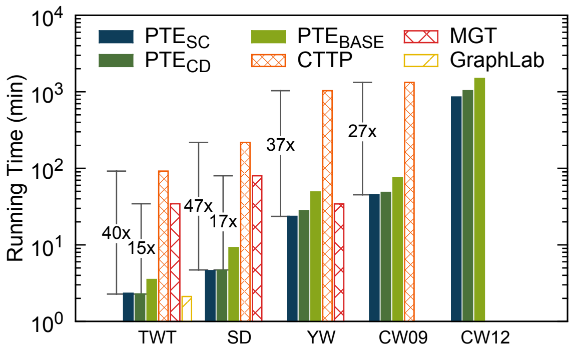


Figure 1: The running time of proposed methods ($\mathbf{PTE_{SC}}$, $\mathbf{PTE_{CD}}$, $\mathbf{PTE_{BASE}}$) and competitors (CTTP, MGT, GraphLab) on real world datasets (log scale). GraphX is not shown since it failed to process any of the datasets. Missing methods for some datasets mean they failed to run on the datasets. $\mathbf{PTE_{SC}}$ shows the best performances outperforming CTTP and MGT by up to 47× and 17×, respectively. Only the proposed algorithms succeed in processing the ClueWeb12 graph containing 6.3 billion vertices and 72 billion edges.

- We propose PTE, a new distributed algorithm for enumerating triangles in an enormous graph, which is designed to minimize the amount of shuffled data, total work, and network read.
- We prove the efficiency of the proposed algorithm: the algorithm operates in $O(|E|)$ shuffled data, $O(|E|^{3/2}/\sqrt{M})$ network read, and $O(|E|^{3/2})$ total work, the worst case optimal, where $|E|$ is the number of edges of a graph and M is the available memory size of a machine.
- Our algorithm is experimentally evaluated using large real world networks. The results demonstrate that our algorithm outperforms the best previous distributed algorithms by up to 47× (see Figure 1). Moreover, PTE successfully enumerates more than 3 trillion triangles in the ClueWeb12 graph containing 6.3 billion vertices and 72 billion edges. Any previous algorithms, including GraphLab, GraphX, and CTTP, fail to process the graph because of massive intermediate data.

The codes and datasets used in this paper are provided in `http://datalab.snu.ac.kr/pte`. The remaining part of the paper is organized as follows. In Section 2, we review the previous researches related to the triangle enumeration. In Section 3, we formally define the problem and introduce important concepts and notations used in this paper. We introduce the details of our algorithm in Section 4. The experimental results are given in Section 5. Finally, we conclude in Section 6. The symbols frequently used in this paper are summarized in Table 1.

2. RELATED WORK

In order to handle enormous graphs, several triangle enumeration algorithms have been proposed recently. In this section, we introduce the distinct approaches of the algorithms, including recent MapReduce algorithms related to our work. We also outline the MapReduce model and emphasize the importance of reducing the amount of shuffled data in improving the performance.

Table 1: 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.
i, j, k	Vertex colors.
(u, v)	Edge between u and v where $u \prec v$.
(u, v, n)	Triangle with vertices u, v, and n where $u \prec v \prec n$.
$(i, j, k), (i, j)$	Subproblems.
$d(u)$	Degree (number of neighbors) of u.
$id(u)$	Vertex number of u, a unique identifier.
$\prec$	Total order on V. $u \prec v$ means u precedes v.
ρ	Number of vertex colors.
ξ	Coloring function: $V \rightarrow \{0, \cdots, \rho-1\}$. $\xi(u)$ is the color of vertex u.
E_{ij}	Set of edges (u, v) where $(\xi(u), \xi(v)) = (i, j)$ or (j, i).
$E^{\star}_{ij}$	Set of edges (u, v) where $(\xi(u), \xi(v)) = (i, j)$.
M	Available memory size of a machine.
P	Number of processors in a distributed system.

2.1 I/O Efficient Triangle Algorithms

Recently, several triangle enumeration algorithms have been proposed in I/O efficient ways to handle graphs that do not fit into the main memory [13, 21, 19]. Hu et al. [13] propose Massive Graph Triangulation (MGT) which buffers a certain number of edges on the memory and finds all triangles containing one of these edges by traversing every vertex. Pagh and Silvestri [21] propose a cache oblivious algorithm which colors the vertices of a graph hierarchically so that it does not need to know the cache structure of a system. Kim et al. [19] present a parallel external-memory algorithm exploiting the features of a solid-state drive (SSD).

These algorithms, however, cannot process a graph exceeding the external memory space of a single machine. Moreover, these algorithms cannot output all the triangles in a large graph containing numerous triangles; for example, the ClueWeb12 graph (see Section 5) has 3 trillion triangles requiring 70 Terabytes of storage.

2.2 Distributed-Memory Triangle Algorithms

The triangle enumeration problem has been recently targeted in the distributed-memory model which assumes a multi-processor system where each processor has its own memory. We call a processor with a memory *a machine*. Arifuzzaman et al. [2] propose a distributed-memory algorithm based on *Message Passing Interface* (MPI). The algorithm divides a graph into overlapping subgraphs and finds triangles in each subgraph in parallel. GraphLab-PowerGraph (shortly GraphLab) [11], which is an MPI-based distributed graph computation framework, provides an implementation for triangle enumeration. GraphLab copies each vertex and its outgoing edges γ times on average to multiple machines where γ is determined by the characteristic of the input graph and the number of machines. Thus, $\gamma|E|$ data are replicated in total, and GraphLab fails when $\gamma|E|/P \geq M$ where P is the number of processors and M is the available memory size of a machine. GraphX, a graph computation library for Spark, also provides an implementation of the same algorithm as in GraphLab; thus it has the same limitation in scalability. PDTL [10] is a parallel and distributed extension of MGT. The experiment of the paper shows impressive speed of PDTL, but it has limited scalability: 1) every machine must hold a copy of an entire graph, 2) a part of PDTL runs on a single machine, which can be a performance bottleneck, and 3) it stores entire triangles in a single machine. In summary, all the previous distributed memory algorithms are limited in handling large graphs.

2.3 MapReduce

MapReduce [7] is a programming model supporting parallel and distributed computation to process large data. MapReduce is highly scalable and easy to use, and thus it has been used for various important graph mining and data mining tasks such as radius [18], graph queries [17], triangle [16, 22, 23], visualization [15], and tensors [14]. A MapReduce round transforms an input set of key-value pairs to an output set of key-value pairs by three steps: it first transforms each pair of the input to a set of new pairs (*map step*), groups the pairs by key so that all values with the same key are aggregated together (*shuffle step*), and processes the values by each key separately and outputs a new set of key-value pairs (*reduce step*).

The amount of shuffled data significantly affects the performance of a MapReduce task because shuffling includes heavy tasks of writing, sorting, and reading the data [12]. In detail, each map worker buffers the pairs from the map step in memory (collect). The buffered pairs are partitioned into R regions and written to local disk periodically where R is the number of reduce workers (spill). Each reduce worker remotely reads the buffered data from the local disks of the map workers via a network (shuffle). When a reduce worker has read all the pairs for its partition, the reduce worker sorts the pairs by keys so that all values with the same key are grouped together (merge and sort). Because of such heavy I/O and network traffic, a large amount of shuffled data decreases the performance significantly. Thus, it is desirable to shrink the amount of shuffled data as much as possible.

2.4 MapReduce Triangle Algorithms

Several triangle computation algorithms have been designed in MapReduce. We review the algorithms in terms of the amount of shuffled data. The first MapReduce algorithm is proposed by Cohen [6]. The algorithm is a variant of node-iterator [25], a well-known sequential algorithm. It shuffles $O(|E|^{3/2})$ length-2 paths (also known as wedges) in a graph. Suri and Vassilvitskii [26] reduce the amount of shuffled data to $O(|E|^{3/2}/\sqrt{M})$ by proposing a graph partitioning based algorithm *Graph Partition* (GP). Considering types of triangles, Park and Chung [22] improve GP by a constant factor in their algorithm, *Triangle Type Partition* (TTP). That is, TTP also shuffles $O(|E|^{3/2}/\sqrt{M})$ data during the process. Aforementioned algorithms cause an out of space error when the size of shuffled data is larger than the total available space. Park et al. [23] avoid the out of space error by introducing a multi-round algorithm, namely Colored TTP (CTTP). CTTP limits the shuffled data size of a round, and thus significantly increases the size of processable data. However, CTTP still shuffles the same amount of data as TTP does, that is, $O(|E|^{3/2}/\sqrt{M})$. Note that our proposed algorithm in this paper reduces the amount of shuffled data to $O(|E|)$, improving the performance significantly.

3. PRELIMINARIES

In this section, we define the problem that we are going to solve, introducing several terms and notations formally. We also describe two previously introduced major algorithms for distributed triangle enumeration.

3.1 Problem Definition

We first define the problem of triangle enumeration.

Definition 1. (Triangle enumeration) Given a simple graph $G = (V, E)$, the problem of *triangle enumeration* is to discover every triangle in the graph G one by one, where a simple graph is an undirected graph containing no loops or no duplicate edges, and a triangle is a set of three vertices fully connected to each other.

Note that we do not require the algorithm to retain or emit each triangle (u, v, n) into any memory system, but to call a local function `enum(·)` with the triangle as the parameter.

On a vertex set V, we define a total order to uniquely express an edge or a triangle.

Definition 2. (Total order on V) The order of two vertices u and v is determined as follows:

- $u \prec v$ if $d(u) < d(v)$ or $(d(u) = d(v)$ and $id(u) < id(v))$

where $d(u)$ is the degree, and $id(u)$ is the unique identifier of a vertex u.

We denote by (u, v) an edge between two vertices u and v, and by (u, v, n) a triangle consisting of three vertices u, v, and n. Unless otherwise noted, the vertices in an edge (u, v) have the order of $u \prec v$, and we presume it has a *direction*, from u to v, even though the graph is undirected. Similarly, the vertices in a triangle (u, v, n) also has the order of $u \prec v \prec n$, and we give each edge a name to simplify the description as follows (see Figure 2):

Definition 3. For a triangle (u, v, n) where the vertices are in the order of $u \prec v \prec n$, we call (u, v) *pivot edge*, (u, n) *port edge*, and (v, n) *starboard edge*.

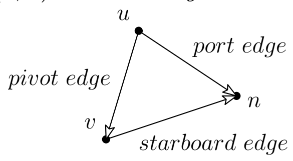


Figure 2: A triangle with directions by the total order on the three vertices.

3.2 Triangle Enumeration in TTP and CTTP

Park and Chung [22] propose a MapReduce algorithm, named *Triangle Type Partition* (TTP), for enumerating triangles. We introduce TTP briefly because of its relevance to our work, and we show that it shuffles $O(|E|^{3/2}/\sqrt{M})$ data.

The first step of TTP is to color the vertices with $\rho = O(\sqrt{|E|/M})$ colors by a hash function $\xi : V \to \{0, \cdots, \rho - 1\}$. Let E_{ij} with $i, j \in \{0, \cdots, \rho - 1\}$ and $i \le j$ be the set $\{(u, v) \in E \mid i = \min(\xi(u), \xi(v))$ and $j = \max(\xi(u), \xi(v))\}$. A triangle is classified as *type-1* if all the vertices in the triangle have the same color, *type-2* if there are exactly two vertices with the same color, and *type-3* if no vertices have the same color. TTP divides the entire problem into $\binom{\rho}{2} + \binom{\rho}{3}$ subproblems of two types:

(i, j) **subproblem**, with $i, j \in \{0, \cdots, \rho - 1\}$ and $i < j$, is to enumerate triangles in an edge-induced subgraph on $E'_{ij} = E_{ij} \cup E_{ii} \cup E_{jj}$. It finds every triangle of type-1 and type-2 where the vertices are colored with i and j. There are $\binom{\rho}{2}$ subproblems of this type.

(i, j, k) **subproblem**, with $i, j, k \in \{0, \cdots, \rho - 1\}$ and $i < j < k$, is to enumerate triangles in an edge-induced subgraph on $E'_{ijk} = E_{ij} \cup E_{ik} \cup E_{jk}$. It finds every triangle of type-3 where the vertices are colored with i, j and k. There are $\binom{\rho}{3}$ subproblems of this type.

Algorithm 1: Graph Partitioning

```
   /* $\psi$ is a meaningless dummy key */
   Map : input $\langle\psi; (u, v) \in E\rangle$
1  emit $\langle(\xi(u), \xi(v)); (u, v)\rangle$
   Reduce : input $\langle(i, j); E^\star_{ij}\rangle$
2  emit $E^\star_{ij}$ to a distributed storage
```

Each map task of TTP gets an edge $e \in E$ and emits $\langle(i, j); e\rangle$ and $\langle(i, j, k); e\rangle$ for every E'_{ij} and E'_{ijk} containing e, respectively. Thus, each reduce task gets a pair $\langle(i, j); E'_{ij}\rangle$ or $\langle(i, j, k); E'_{ijk}\rangle$, and finds all triangle in the edge-induced subgraph. For each edge, a map task emits $\rho - 1$ key-value pairs (Lemma 2 in [22]); that is, $O(|E|\rho) = O(|E|^{3/2}/\sqrt{M})$ pairs are shuffled in total. TTP fails to process a graph when the shuffled data size is larger than the total available space. CTTP [23] avoids the failure by dividing the tasks into multiple rounds and limiting the shuffled data size of a round. However, CTTP still shuffles exactly the same pairs as TTP; hence CTTP also suffers from the massive shuffled data resulting in a very long running time.

4. PROPOSED METHOD

In this section, we propose PTE (*Pre-partitioned Triangle Enumeration*), a distributed algorithm for enumerating triangles in an enormous graph. There are several challenges in designing an efficient and scalable distributed algorithm for triangle enumeration.

1. **Minimize shuffled data.** Massive data are shuffled for generating subgraphs by the previous algorithms. How can we minimize the amount of shuffled data?
2. **Minimize computations.** The previous algorithms contain several kinds of redundant operations (details in Section 4.2). How can we remove the redundancy?
3. **Minimize network read.** In previous algorithms, each subproblem reads necessary sets of edges via network, and the amount of network read is determined by the number of vertex colors. How can we decrease the number of vertex colors to minimize network read?

We have the following main ideas to address the above challenges, which are described in detail in later subsections.

1. **Separating graph partitioning from generating subgraphs** decreases the amount of shuffled data to $O(|E|)$ from $O(|E|^{3/2}/\sqrt{M})$ of the previous MapReduce algorithms (Section 4.1).
2. **Considering the color-direction of edges** removes redundant operations and minimizes computations (Section 4.2).
3. **Carefully scheduling triangle computations** in subproblems shrinks the amount of network read by decreasing the number of vertex colors (Section 4.3).

In the following we first describe PTE$_{\text{BASE}}$ which exploits pre-partitioning to decrease the shuffled data (Section 4.1). Then, we describe PTE$_{\text{CD}}$ which improves on PTE$_{\text{BASE}}$ to remove redundant operations (Section 4.2). After that, we explain our desired method PTE$_{\text{SC}}$ which further improves on PTE$_{\text{CD}}$ to shrink the amount of network read (Section 4.3). The theoretical analysis of the methods and implementation issues are discussed in the end (Sections 4.4 and 4.5). Note that although we describe PTE using MapReduce primitives for simplicity, PTE is general enough to be implemented in any distributed framework (discussions in Section 4.5 and experimental comparisons in Section 5.2.3).

Algorithm 2: Triangle Enumeration (PTE$_{\text{BASE}}$)

```
    /* $\psi$ is a meaningless dummy key */
    Map : input $\langle\psi; problem = (i, j)$ or $(i, j, k)\rangle$
1   initialize $E'$
2   if problem is of type $(i, j)$ then
        /* $E_{ij} = E^\star_{ij} \cup E^\star_{ji}$, and $E_{ii} = E^\star_{ii}$ */
3       read $E_{ij}$, $E_{ii}$, $E_{jj}$
4       $E' \leftarrow E_{ij} \cup E_{ii} \cup E_{jj}$
5   else if problem is of type $(i, j, k)$ then
        /* $E_{ij} = E^\star_{ij} \cup E^\star_{ji}$, $E_{ik} = E^\star_{ik} \cup E^\star_{ki}$, and
           $E_{jk} = E^\star_{jk} \cup E^\star_{kj}$ */
6       read $E_{ij}$, $E_{ik}$, $E_{jk}$
7       $E' \leftarrow E_{ij} \cup E_{ik} \cup E_{jk}$
8   enumerateTriangles($E'$)

    /* enumerate triangles in the edge-induced subgraph on $E$ */
9   Function enumerateTriangles($E$)
10      foreach $(u, v) \in E$ do
11          foreach $n \in \{n_u|(u, n_u) \in E\} \cap \{n_v|(v, n_v) \in E\}$ do
12              if $\xi(u) = \xi(v) = \xi(n)$ then
13                  if $(\xi(u) = i$ and $i + 1 \equiv j \mod \rho)$ or
                       $(\xi(u) = j$ and $j + 1 \equiv i \mod \rho)$ then
14                      enum($(u, v, n)$)
15              else
16                  enum($(u, v, n)$)
```

4.1 PTE$_{\text{BASE}}$: Pre-partitioned Triangle Enumeration

In this section we propose PTE$_{\text{BASE}}$ which rectifies the massive shuffled data problem of previous MapReduce algorithms. The main idea is partitioning an input graph into sets of edges before enumerating triangles, and storing the sets in a distributed storage like *Hadoop Distributed File System* (HDFS) of Hadoop, or *Resilient Distributed Dataset* (RDD) of Spark. We observe that the subproblems of TTP require each set E_{ij} of edges as a unit. It implies that if each edge set E_{ij} is directly accessible from a distributed storage, we need not shuffle the edges like TTP does. Consequently, we partition the input graph into $\rho + \binom{\rho}{2} = \frac{\rho(\rho+1)}{2}$ sets of edges according to the vertex colors in each edge; ρ and $\binom{\rho}{2}$ are for E_{ij} when $i = j$ and $i < j$, respectively. Each edge $(u, v) \in E_{ij}$ keeps the order $u \prec v$. Each vertex is colored by a coloring function ξ which is randomly chosen from a pairwise independent family of functions [28]. The pairwise independence of ξ guarantees that edges are evenly distributed. PTE$_{\text{BASE}}$ sets ρ to $\lceil\sqrt{6|E|/M}\rceil$ to fit the three edge sets for an (i, j, k) subproblem into the memory of size M in a processor: the expected size of an edge set is $2|E|/\rho^2$, and the sum $6|E|/\rho^2$ of the size of the three edge sets should be less than or equal to the memory size M.

After the graph partitioning, PTE$_{\text{BASE}}$ reads edge sets and finds triangles in each subproblem. In each (i, j) subproblem, it reads E_{ij}, E_{ii}, and E_{jj}, and enumerates triangles in the union of the edge sets. In each (i, j, k) subproblem, similarly, it reads E_{ij}, E_{ik}, and E_{jk}, and enumerates triangles in the union of the edge sets. The edge sets are read from a distributed storage via a network, and the total amount of network read is $O(|E|\rho)$ (see Section 3.2). Note that the network read is different from the data shuffle; the data shuffle is a much heavier task since it requires data collecting and writing in senders, data transfer via a network, and data merging and sorting in receivers (see Section 2.3). However, the network read contains data transfer via a network only.

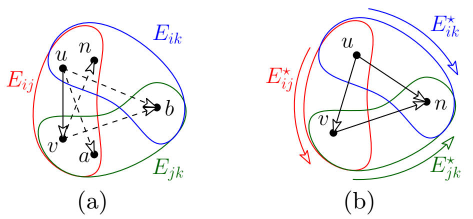


Figure 3: (a) An example of finding type-3 triangles containing an edge $(u,v) \in E_{ij}$ in an (i,j,k) subproblem. PTE$_{\text{BASE}}$ finds the triangle (u,v,b) by intersecting u's neighbor set $\{v,a,b\}$ and v's neighbor set $\{n,b\}$. However, it is unnecessary to consider the edges in E_{ij} since the other two edges of a type-3 triangle containing (u,v) must be in E_{ik} and E_{jk}, respectively. (b) `enumerateTrianglesCD(`$E^\star_{ij}$, $E^\star_{ik}$, $E^\star_{jk}$`)` in PTE$_{\text{CD}}$ finds every triangle whose pivot edge, port edge, and starboard edge have the same color-directions as those of $E^\star_{ij}$, $E^\star_{ik}$, and $E^\star_{jk}$, respectively. The arrows denote the color-directions.

Pseudo codes for PTE$_{\text{BASE}}$ are listed in Algorithm 1 and Algorithm 2. The graph partitioning is done by a pair of map and reduce steps (Algorithm 1). In the map step, PTE$_{\text{BASE}}$ transforms each edge (u,v) into a pair $\langle(\xi(u),\xi(v));(u,v)\rangle$ (line 1). The edges of the pairs are aggregated by the keys; and for each key (i,j), a reduce task receives $E^\star_{ij}$ and emits it to a separate file in a distributed storage (line 2), where $E^\star_{ij}$ is $\{(u,v) \in E | (\xi(u),\xi(v)) = (i,j)\}$. Note that the union of $E^\star_{ij}$ and $E^\star_{ji}$ is E_{ij}, that is, $E^\star_{ij} \cup E^\star_{ji} = E_{ij}$.

Thanks to the pre-partitioned edge sets, the triangle enumeration is done by a single map step (see Algorithm 2). Each map task reads edge sets needed to solve a subproblem (i,j) or (i,j,k) (lines 3, 6), makes the union of the edge sets (lines 4, 7), and enumerates triangles with a sequential algorithm `enumerateTriangles` (line 8). Although any sequential algorithm for triangle enumeration can be used for `enumerateTriangles`, we use CompactForward [20], one of the best sequential algorithms, with a modification (lines 9-16): we skip the vertex sorting procedure of CompactForward because the edges are already ordered by the degrees of its vertices. Given a set E' of edges, `enumerateTriangles` runs in $O(|E'|^{3/2})$ total work, the same as that of CompactForward. Note that we propose a specialized algorithm to reduce the total work in Section 4.2. Note also that although every type-1 triangle appears $\rho - 1$ times, PTE$_{\text{BASE}}$ emits the triangle only once: for each type-1 triangle of color i, PTE$_{\text{BASE}}$ emits the triangle if and only if $i + 1 \equiv j \mod \rho$ given a subproblem (i,j) or (j,i) (lines 12-14). Still, a type-1 triangle is computed $\rho - 1$ times which is unnecessary. We completely resolve the issue in Section 4.2.

4.2 PTE$_{\text{CD}}$: Reducing the Total Work

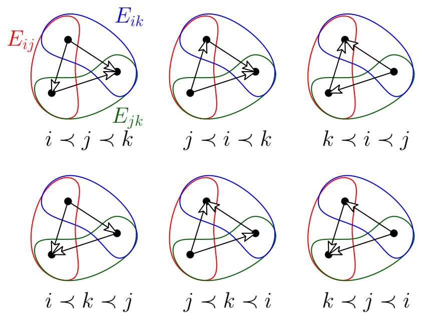
PTE$_{\text{CD}}$ improves on PTE$_{\text{BASE}}$ to minimize the amount of computations by exploiting *color-direction*. We first give an example (Figure 3(a)) to show that the function `enumerateTriangles` in Algorithm 2 performs redundant operations. Let us consider finding type-3 triangles containing an edge $(u,v) \in E_{ij}$ in an (i,j,k) subproblem. `enumerateTriangles` finds such triangles by intersecting the two outgoing neighbor sets of u and v. In the figure, the neighbor sets are $\{v,a,b\}$ and $\{n,b\}$; and we find the triangle (u,v,b). However, it is unnecessary to consider edges in E_{ij} (that is, (u,v), (u,a), (v,n)) since the other two edges of a type-3 triangle containing (u,v) must be in E_{ik} and E_{jk}, respectively. The redundant operations can be removed by intersecting u's neighbors only in E_{ik} and v's neighbors only in E_{jk} instead of looking at all the neighbors of u and v. In the figure, the two neighbor sets are both $\{b\}$; and we find the same triangle (u,v,b). PTE$_{\text{CD}}$ removes the redundant operations by adopting a new function `enumerateTrianglesCD` (lines 11-14 in Algorithm 3). We define the *color-direction* of an edge (u,v) to be from $\xi(u)$ to $\xi(v)$; and we also define the *color-direction* of $E^\star_{ij}$ to be from i to j. Then, `enumerateTrianglesCD(`$E^\star_{ij}$, $E^\star_{ik}$, $E^\star_{jk}$`)` finds every triangle whose pivot edge, port edge, and starboard edge have the same color-directions as those of $E^\star_{ij}$, $E^\star_{ik}$, and $E^\star_{jk}$, respectively (see Figure 3(b)). Note that the algorithm does not look at any edges in E_{ij} for the intersection in the example of Figure 3(a) by separating the input edge sets.

Redundant operations of another type appear in (i,j) subproblems; PTE$_{\text{BASE}}$ outputs a type-1 triangle exactly once but still computes it multiple times: *type*-1 triangles with a color i appears $\rho - 1$ times in (i,j) or (j,i) subproblems for $j \in \{0,\cdots,\rho-1\} \setminus \{i\}$. PTE$_{\text{CD}}$ resolves the duplicate computation by performing `enumerateTrianglesCD(`$E^\star_{ii}$, $E^\star_{ii}$, $E^\star_{ii}$`)` exactly once for each vertex color i; and thus, every type-1 triangle appears only once.

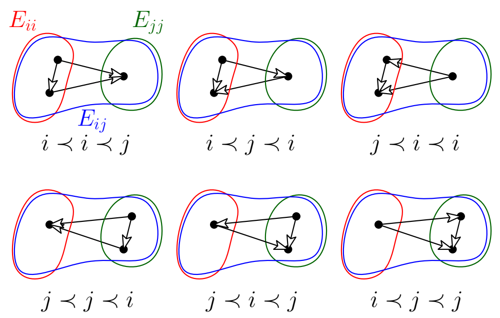
Algorithm 3 shows PTE$_{\text{CD}}$ using the new function `enumerateTrianglesCD`. To find type-3 triangles in each (i,j,k) subproblem, PTE$_{\text{CD}}$ calls the function `enumerateTrianglesCD` 6 times for every possible color-direction (lines 10, 15-18) (see Figure 4(a)). To find type-2 triangles in each (i,j) subproblem, similarly, PTE$_{\text{CD}}$ calls `enumerateTrianglesCD` 6 times (lines 7-8) (see Figure 4(b)). For type-1 triangles whose vertices have a color i, PTE$_{\text{CD}}$ performs `enumer-`

Algorithm 3: Triangle Enumeration (PTE$_{\text{CD}}$)

```
   /* ψ is a meaningless dummy key                                    */
   Map        : input ⟨$\psi$; $problem = (i,j)$ or $(i,j,k)$⟩
 1 if $problem$ is of type $(i,j)$ then
 2     read $E^\star_{ij}$, $E^\star_{ji}$, $E^\star_{ii}$, $E^\star_{jj}$
       /* enumerate Type-1 triangles                                    */
 3     if $i + 1 = j$ then
 4         enumerateTrianglesCD($E^\star_{ii}$, $E^\star_{ii}$, $E^\star_{ii}$)
 5     else if $j = \rho - 1$ and $i = 0$ then
 6         enumerateTrianglesCD($E^\star_{jj}$, $E^\star_{jj}$, $E^\star_{jj}$)

       /* enumerate Type-2 triangles                                    */
 7     foreach $(x,y,z) \in$
         $\{(i,i,j),(i,j,i),(j,i,i),(i,j,j),(j,i,j),(j,j,i)\}$ do
 8         enumerateTrianglesCD($E^\star_{xy}$, $E^\star_{xz}$, $E^\star_{yz}$)
 9 else if $problem$ is of type $(i,j,k)$ then
10     enumerateType3Triangles($(i,j,k)$)

   /* enumerate every triangle $(u,v,n)$ such that $\xi(u) = i$,
      $\xi(v) = j$ and $\xi(n) = k$                                      */
11 Function enumerateTrianglesCD($E^\star_{ij}$, $E^\star_{ik}$, $E^\star_{jk}$)
12     foreach $(u,v) \in E^\star_{ij}$ do
13         foreach
             $n \in \{n_u|(u,n_u) \in E^\star_{ik}\} \cap \{n_v|(v,n_v) \in E^\star_{jk}\}$ do
14             enum($(u,v,n)$)

15 Function enumerateType3Triangles($i$, $j$, $k$)
16     read $E^\star_{ij}$, $E^\star_{ik}$, $E^\star_{ji}$, $E^\star_{jk}$, $E^\star_{ki}$, $E^\star_{kj}$
17     foreach $(x,y,z) \in$
         $\{(i,j,k),(i,k,j),(j,i,k),(j,k,i),(k,i,j),(k,j,i)\}$ do
18         enumerateTrianglesCD($E^\star_{xy}$, $E^\star_{xz}$, $E^\star_{yz}$)
```



(a) The six color-directions of a type-3 triangle in an (i, j, k) subproblem.



(b) The six color-directions of a type-2 triangle in an (i, j) subproblem.

Figure 4: The color-directions of a triangle according to its type. The function `enumerateTrianglesCD` is called for each color-direction; that is, PTE$_{\text{CD}}$ calls it 6 times for type-3 triangles and type-2 triangles, respectively.

`ateTrianglesCD(`$E^\star_{ii}$, $E^\star_{ii}$, $E^\star_{ii}$`)` only if $i + 1 \equiv j \mod \rho$ given a subproblem (i, j) or (j, i) so that `enumerateTrianglesCD(`$E^\star_{ii}$, $E^\star_{ii}$, $E^\star_{ii}$`)` operates exactly once (lines 3-6). As a result, the algorithm emits every triangle exactly once.

Removing the two types of redundant operations, PTE$_{\text{CD}}$ decreases the number of operations for intersecting neighbor sets by more than $2 - \frac{2}{\rho}$ times from PTE$_{\text{BASE}}$ in expectation. As we will see in Section 5.2.1, PTE$_{\text{CD}}$ decreases the operations by up to 6.83× than PTE$_{\text{BASE}}$ on real world graphs.

THEOREM 1. *PTE$_{CD}$ decreases the number of operations for intersecting neighbor sets by more than $2 - \frac{2}{\rho}$ times compared to PTE$_{BASE}$ in expectation.*

PROOF. To intersect the sets of neighbors of u and v in an edge (u, v) such that $\xi(u) = i$, $\xi(v) = j$ and $i \neq j$, the function `enumerateTriangles` in PTE$_{\text{BASE}}$ performs $d^\star_{ij}(u) + d^\star_{ik}(u) + d^\star_{ji}(v) + d^\star_{jk}(v)$ operations while `enumerateTrianglesCD` in PTE$_{\text{CD}}$ performs $d^\star_{ik}(u) + d^\star_{jk}(v)$ operations for each color $k \in \{0, \cdots, \rho - 1\} \setminus \{i, j\}$ where $d^\star_{ij}(u)$ is the number of u's neighbors in $E^\star_{ij}$. Thus, PTE$_{\text{BASE}}$ performs $(\rho - 2) \times (d^\star_{\xi(u)\xi(v)}(u) + d^\star_{\xi(v)\xi(u)}(v))$ additional operations compared to PTE$_{\text{CD}}$ for each $(u, v) \in E^{out}$ where E^{out} is the set of edges $(u, v) \in E$ such that $\xi(u) \neq \xi(v)$; that is,

$$(\rho - 2) \times \sum_{(u,v) \in E^{out}} \left(d^\star_{\xi(u)\xi(v)}(u) + d^\star_{\xi(v)\xi(u)}(v) \right) \tag{1}$$

Given an edge (u, v) such that $\xi(u) = \xi(v) = i$, PTE$_{\text{BASE}}$ performs $d^\star_{ii}(u) + d^\star_{ij}(u) + d^\star_{ii}(v) + d^\star_{ij}(v)$ operations for each color $j \in \{0, \cdots, \rho - 1\} \setminus \{i\}$; meanwhile, PTE$_{\text{CD}}$ performs $d^\star_{ij}(u) + d^\star_{ij}(v)$ operations for each color $j \in \{0, \cdots, \rho - 1\}$. Thus, PTE$_{\text{BASE}}$ performs $(\rho - 2) \times (d^\star_{\xi(u)\xi(v)}(u) + d^\star_{\xi(v)\xi(u)}(v))$ more operations than PTE$_{\text{CD}}$ for each $(u, v) \in E^{in}$ where E^{in} is $E \setminus E^{out}$; that is,

$$(\rho - 2) \times \sum_{(u,v) \in E^{in}} \left(d^\star_{\xi(u)\xi(v)}(u) + d^\star_{\xi(v)\xi(u)}(v) \right) \tag{2}$$

Then, the total number of *additional* operations performed by PTE$_{\text{BASE}}$, compared to PTE$_{\text{CD}}$, is the sum of (1) and (2):

$$(\rho - 2) \times \sum_{(u,v) \in E} \left(d^\star_{\xi(u)\xi(v)}(u) + d^\star_{\xi(v)\xi(u)}(v) \right) \tag{3}$$

The expected value of $d^\star_{\xi(u)\xi(v)}(u)$ is $d^\star(u)/\rho$ where $d^\star(u)$ is the number of neighbors v of u such that $u \prec v$, since the coloring function ξ is randomly chosen from a pairwise independent family of functions. Thus, (3) becomes as follows:

$$\frac{(\rho - 2)}{\rho} \times \sum_{(u,v) \in E} \left(d^\star(u) + d^\star(v) + O(\rho) \right) \tag{4}$$

We add $O(\rho)$ because $(d^\star(u) + d^\star(v))/\rho$ can be less than 1 but $d^\star_{\xi(u)\xi(v)}(u) + d^\star_{\xi(v)\xi(u)}(v)$ is always larger than or equal to 1. Meanwhile, the number of operations by PTE$_{\text{CD}}$ for intersecting neighbor sets is $\sum_{(u,v) \in E} (d^\star(u) + d^\star(v))$ as we will see in Theorem 5. Thus, PTE$_{\text{CD}}$ reduces the number of operations by more than $2 - \frac{2}{\rho}$ times from PTE$_{\text{BASE}}$ in expectation. ☐

4.3 PTE$_{\text{SC}}$: Reducing the Network Read

PTE$_{\text{SC}}$ further improves on PTE$_{\text{CD}}$ to shrink the amount of network read by *scheduling* calls of the function `enumerateTrianglesCD`. Reading each $E^\star_{ij}$ in $\rho - 1$ subproblems, PTE$_{\text{CD}}$ (as well as PTE$_{\text{BASE}}$) reads $O(|E|\rho)$ data via a network in total. For example, $E^\star_{01}$ is read in every $(0, 1, k)$ subproblem for $2 \leq k < \rho$, and the $(0, 1)$ subproblem. It implies that the amount of network read depends on ρ, the number of vertex colors. PTE$_{\text{BASE}}$ and PTE$_{\text{CD}}$ set ρ to $\lceil \sqrt{6|E|/M} \rceil$ as mentioned in Section 4.1. In PTE$_{\text{SC}}$, we reduce it to $\lceil \sqrt{5|E|/M} \rceil$ by setting the sequence of triangle computation as in Figure 5 which represents the schedule of data loading for an (i, j, k) subproblem. We denote by Δ_{ijk} the set of triangles enumerated by `enumerateTrianglesCD(`$E^\star_{ij}, E^\star_{ik}, E^\star_{jk}$`)`. PTE$_{\text{CD}}$ handles the triangle sets one by one from left to right in the figure. The check-marks (✓) show the relations between edge sets and triangle sets. For example, $E^\star_{ij}$, $E^\star_{ik}$, and $E^\star_{jk}$ should be retained in the memory together to enumerate Δ_{ijk}. When we restrict to read an edge set only once in a subproblem, the shaded areas represent when the edge sets are in the memory. For example, $E^\star_{ij}$ is read before Δ_{ijk}, and is released after Δ_{kij}. Then, we can easily see that the maximum number of edge sets which are retained in the memory together at a time is 5, and it leads to setting ρ to $\lceil \sqrt{5|E|/M} \rceil$. The pro-

	Δ_{ijk}	Δ_{ikj}	Δ_{jik}	Δ_{jki}	Δ_{kij}	Δ_{kji}
$E^\star_{ij}$	✓	✓	▨	▨	✓	
$E^\star_{ik}$	✓	✓	✓			
$E^\star_{ji}$			✓	✓	▨	✓
$E^\star_{jk}$	✓	▨	✓	✓		
$E^\star_{ki}$				✓	✓	✓
$E^\star_{kj}$		✓	▨	▨	✓	✓

Figure 5: The schedule of data loading for an (i, j, k) subproblem. PTE$_{\text{SC}}$ enumerates triangles in columns one by one from left to right. Each checkmark (✓) shows the relations between edge sets and triangle types. Each shaded area denotes when the edge set is in the memory, when we restrict to read an edge set only once in a subproblem.

Algorithm 4: Type-3 triangle enumeration in PTE$_{\mathrm{SC}}$

```
1 Function enumerateType3Triangles(i, j, k)
2     read $E^\star_{ij}$, $E^\star_{ik}$, $E^\star_{ji}$, $E^\star_{jk}$, $E^\star_{kj}$
3     foreach $(x, y, z) \in \{(i,j,k), (i,k,j), (j,i,k)\}$ do
4         enumerateTrianglesCD($E^\star_{xy}$, $E^\star_{xz}$, $E^\star_{yz}$)
5     release $E^\star_{ik}$
6     read $E^\star_{ki}$
7     foreach $(x, y, z) \in \{(j,k,i), (k,i,j), (k,j,i)\}$ do
8         enumerateTrianglesCD($E^\star_{xy}$, $E^\star_{xz}$, $E^\star_{yz}$)
```

cedure of type-3 triangle enumeration with the scheduling method is described in Algorithm 4 which replaces the function `enumerateType3Triangles` in Algorithm 3. Note that the number 5 of edge sets loaded in the memory at a time is optimal as shown in the following theorem.

THEOREM 2. *Given an (i, j, k) subproblem, the maximum number of edge sets retained in the memory at a time cannot be smaller than 5, if each edge set can be read only once.*

PROOF. Suppose that there is a schedule to make the maximum number of edge sets retained in the memory at a time less than 5. Then, we first read 4 edge sets in the memory. Then, 1) any edge set in the memory cannot be released until all triangles containing an edge in the set have been enumerated, and 2) we cannot read another edge set until we release one in the memory. Thus, for at least one edge set in the memory, it should be able to process all triangles containing an edge in the edge set without reading an additional edge set. However, it is impossible because enumerating all triangles containing an edge in an edge set requires 5 edge sets but we have only 4 edge sets. For example, the triangles in Δ_{ijk}, Δ_{ikj}, and Δ_{kij}, which are related to an edge set $E^\star_{ij}$, require $E^\star_{ij}$, $E^\star_{ik}$, $E^\star_{jk}$, $E^\star_{kj}$, and $E^\star_{ki}$. Thus, there is no schedule to make the maximum number of edge sets retained in the memory at a time less than 5. □

4.4 Analysis

In this section we analyze the proposed algorithm in terms of the amount of shuffled data, network read, and total work. We first prove the claimed amount of shuffled data generated by the graph partitioning in Algorithm 1.

THEOREM 3. *The amount of shuffled data for partitioning a graph is $O(|E|)$ where $|E|$ is the number of edges in the graph.*

PROOF. The pairs emitted from the map operation is exactly the data to be shuffled. For each edge, a map task emits one pair; accordingly, the amount of shuffled data is the number $|E|$ of edges in the graph. □

We emphasize that while the previous MapReduce algorithms shuffle $O(|E|^{3/2}/\sqrt{M})$ data, we reduce it to be $O(|E|)$. Instead of data shuffle requiring heavy disk I/O, network read, and massive intermediate data, we only require the same amount of network read, that is $O(|E|^{3/2}/\sqrt{M})$.

THEOREM 4. *PTE requires $O(|E|^{3/2}/\sqrt{M})$ network read.*

PROOF. We first show that every $E^\star_{ij}$ for $(i, j) \in \{0, \cdots, \rho-1\}^2$ are read $\rho - 1$ times. It is clear that $E^\star_{ij}$ such that $i = j$ is read $\rho-1$ times in (i, k) or (k, i) subproblems for $k \in \{0, \cdots, \rho-1\} \setminus \{i\}$. We now consider $E^\star_{ij}$ for $i \neq j$. Without loss of generality, we assume $i < j$. Then, $E^\star_{ij}$ is read $\rho - 2$ times in (i, j, k) or (i, k, j) or (k, i, j) subproblems where $k \in \{0, \cdots, \rho-1\} \setminus \{i, j\}$, and once in an (i, j) subproblem; $\rho - 1$ times in total. The total amount of data read by PTE is as follows:

$$(\rho-1)\sum_{i=0}^{\rho-1}\sum_{j=0}^{\rho-1}|E^\star_{ij}| = |E|(\rho-1) = |E|\left(\sqrt{\frac{5|E|}{M}} - 1\right) = O\left(\frac{|E|^{3/2}}{\sqrt{M}}\right)$$

where $|E^\star_{ij}|$ is the number of edges in $E^\star_{ij}$. □

Finally, we prove the claimed total work of the proposed algorithm.

THEOREM 5. *PTE requires $O(|E|^{3/2})$ total work.*

PROOF. Intersecting two sets requires comparisons as many times as the number of elements in the two sets. Accordingly, the number of operations performed by `enumerateTrianglesCD`$(E^\star_{ij}, E^\star_{ik}, E^\star_{jk})$ is

$$\sum_{(u,v)\in E^\star_{ij}} \left(d^\star_{ik}(u) + d^\star_{jk}(v) + O(1)\right)$$

where $d^\star_{ik}(u)$ is the number of u's neighbors in $E^\star_{ik}$. We put $O(1)$ since $d^\star_{ik}(u) + d^\star_{jk}(v)$ can be smaller than 1. PTE$_{\mathrm{SC}}$ (as well as PTE$_{\mathrm{CD}}$) calls `enumerateTrianglesCD` for every possible triple $(i, j, k) \in \{0, \cdots, \rho-1\}^3$; thus the total number of operations is as follows:

$$\begin{aligned}&\sum_{i=0}^{\rho-1}\sum_{j=0}^{\rho-1}\sum_{k=0}^{\rho-1}\sum_{(u,v)\in E^\star_{ij}} \left(d^\star_{ik}(u) + d^\star_{jk}(v) + O(1)\right)\\ =&O(E\rho) + \sum_{(u,v)\in E}\left(d^\star(u) + d^\star(v)\right)\end{aligned}$$

The left term $O(|E|\rho)$ is $O(|E|^{3/2}/\sqrt{M})$ for checking all edges in each subproblem, which occurs also in PTE$_{\mathrm{BASE}}$. The right summation is the number of operations for intersecting neighbors, and it is $O(|E|^{3/2})$ when the vertices are ordered by Definition 2 because the maximum value of $d^\star(u)$ for every $u \in V$ is $2\sqrt{|E|}$ as proved in [25]. □

Note that it is the worst case optimal and the same as one of the best sequential algorithms [20].

4.5 Implementation

In this section, we discuss practical implementation issues of PTEs. We focus on the most famous distributed computation frameworks, Hadoop and Spark. Note that PTEs can be implemented for any distributed framework which supports map and reduce functionalities.

PTE on Hadoop. We describe how to implement PTE on Hadoop which is the de facto standard of MapReduce framework. The graph partitioning method (Algorithm 1) of PTE is implemented as a single MapReduce round. The result of the graph partitioning method has a custom output format that stores each edge set as a separate file in *Hadoop Distributed File System* (HDFS); and thus each edge set is accessible by the path. The triangle enumeration method (Algorithms 2 or 3) of PTE is implemented as a single map step where each map task processes an (i, j) or (i, j, k) subproblem. For this purpose, we generate a text file where each line is (i, j) or (i, j, k), and make each map task read a line and solve the subproblem.

PTE on Spark. We describe how to implement PTE on Spark, which is another popular distributed computing

Table 2: The summary of datasets.

Dataset	Vertices	Edges	Triangles
Twitter (TWT)[1]	42M	1.2B	34824916864
SubDomain (SD)[2]	101M	1.9B	417761664336
YahooWeb (YW)[3]	1.4B	6.4B	85782928684
ClueWeb09 (CW09)[4]	4.8B	7.9B	31012381940
ClueWeb12 (CW12)[5]	6.3B	72B	3064499157291
ClueWeb12/256	287M	260M	184494
ClueWeb12/128	457M	520M	1458207
ClueWeb12/64	679M	1.0B	11793453
ClueWeb12/32	947M	2.1B	94016252
ClueWeb12/16	1.3B	4.2B	750724625
ClueWeb12/8	1.8B	8.2B	6015772801
ClueWeb12/4	2.6B	17B	48040237257
ClueWeb12/2	4.0B	35B	384568217900

framework. The reduce operation of Algorithm 1 is replaced by a pair of *partitionBy* and *mapPartitions* operations of general *Resilient Distributed Dataset* (RDD). The partitionBy operation uses a custom partitioner that partitions edges according to their vertex colors. The mapPartition operation outputs an RDD (namely edgeRDD) where each partition contains an edge set. A special RDD where each partition is in charge of a subproblem (i, j) or (i, j, k) wraps the edgeRDD; the edge sets used in each partition of the special RDD are loaded directly from the edgeRDD.

5. EXPERIMENTS

In this section, we experimentally evaluate our algorithms and compare them to recent single machine and distributed algorithms. We aim to answer the following questions.

Q1 How much do the three methods of PTE contribute to the performance improvement? (Section 5.2.1)

Q2 How does PTE scale up in terms of the number of machines? (Section 5.2.2)

Q3 How does the performance of PTE change depending on the underlying distributed framework (MapReduce or Spark)? (Section 5.2.3)

We first introduce datasets and experimental environment in Section 5.1. After that, we answer the questions in Section 5.2 presenting the result of the experiments.

5.1 Setup

5.1.1 Datasets

We use real world datasets to evaluate the proposed algorithms. The datasets are summarized in Table 2. *Twitter* is a followee-follower network in the social network service Twitter. *SubDomain* is a hyperlink network among domains where an edge exists if there is at least one hyperlink between two subdomains. *YahooWeb*, *ClueWeb09*, and *ClueWeb12* are page level hyperlink networks on the Web. *ClueWeb12/k* is a subgraph of ClueWeb12. We use random edge sampling to make ClueWeb12/k, with the sampling probability $1/k$ where k varies from 2 to 256.

Each dataset is preprocessed to be a simple graph. We reorder the vertices in each edge (u, v) to be $u \prec v$ using an algorithm in [6]. These tasks are done in $O(E)$.

[1]http://an.kaist.ac.kr/traces/WWW2010.html

[2]http://webdatacommons.org/hyperlinkgraph

[3]http://webscope.sandbox.yahoo.com

[4]http://boston.lti.cs.cmu.edu/clueweb09/wiki/tiki-index.php?page=Web+Graph

[5]http://www.lemurproject.org/clueweb12/webgraph.php

5.1.2 Experimental Environment

We implement PTE on Hadoop (open source version of MapReduce) and Spark. Results described in Sections 5.2.1 and 5.2.2 are from Hadoop implementation of PTE; we also describe the Spark results in Section 5.2.3. We compare PTEs with previous algorithms: CTTP [23], MGT [13] and the triangle counting implementations on GraphLab and GraphX. CTTP is the state of the art MapReduce algorithm. MGT is an I/O efficient external memory algorithm. GraphX is a graph processing API on Spark, a distributed computing framework. GraphLab is another distributed graph processing framework using MPI.

All experiments were conducted on a cluster with 41 machines where each machine is equipped with an Intel Xeon E3-1230v3 CPU (quad-core at 3.30GHz), and 32GB RAM. The cluster runs Hadoop v1.2.1, and consists of a master node, and 40 slave nodes. Each slave node can run 3 mappers and 3 reducers (120 mappers and 120 reducers in total) concurrently. The memory size for each mapper and reducer is set to 4GB. Spark v1.5.1 is also installed at the cluster where each slave is set to use 30GB memory and 3 cores of CPU; one core is for system maintenance. We operate GraphLab PowerGraph v2.2 on OpenRTE v1.5.4 at the same cluster servers.

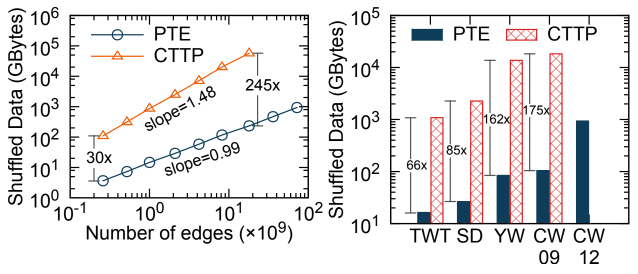
5.2 Experimental Results

In this section, we present experimental results to answer the questions listed at the beginning of Section 5.

5.2.1 Effect of PTE's three methods

Effect of pre-partitioning. We compare the amount of shuffled data between PTE and CTTP to show the effect of pre-partitioning (Section 4.1). Figure 6(a) shows the results on ClueWeb12 with various numbers of edges. It shows that PTE shuffles far fewer data than CTTP, and the difference gets larger as the data size increases; as the number of edges varies from 0.26 billions to 36 billions, the difference increases from 30× to 245×. The slopes for PTE and CTTP are 0.99 and 1.48, respectively. They reflect the claimed complexity of shuffled data size, $O(|E|)$ of PTE and $O(|E|^{1.5}/\sqrt{M})$ of CTTP. Figure 6(b) shows the results on real world graphs; PTE shuffles far fewer data by up to 175× than CTTP does.

Effect of color-direction. To show the effect of color-direction (Section 4.2), we count the number of all oper-



(a) (b)

Figure 6: The shuffled data size of PTE and CTTP (a) on ClueWeb12 with various numbers of edges, and (b) on real world graphs. PTE shuffles up to 245× fewer data than CTTP on ClueWeb12; the gap grows when the data size increases. On real world graphs, PTE shuffles up to 175× fewer data than CTTP on ClueWeb09.

Table 3: **The number of operations by PTE$_{CD}$ and PTE$_{BASE}$ on various graphs. PTE$_{CD}$ decreases the number of operations by up to 6.85× from PTE$_{BASE}$.**

Dataset	PTE$_{BASE}$	PTE$_{CD}$	$\frac{PTE_{BASE}}{PTE_{CD}}$
CW12/256	1.5×10^8	2.1×10^7	6.85
CW12/128	6.2×10^8	1.0×10^8	6.22
CW12/64	2.7×10^9	4.6×10^8	5.83
CW12/32	1.2×10^{10}	2.2×10^9	5.10
CW12/16	4.9×10^{10}	1.1×10^{10}	4.44
CW12/8	2.1×10^{11}	5.4×10^{10}	3.90
CW12/4	8.9×10^{11}	2.6×10^{11}	3.47
CW12/2	3.7×10^{12}	1.2×10^{12}	3.18
CW12	1.6×10^{13}	4.9×10^{12}	3.14
TWT	1.1×10^{12}	5.4×10^{11}	2.03
SD	8.3×10^{12}	4.0×10^{12}	2.06
YW	7.4×10^{11}	2.9×10^{11}	2.55
CW09	2.7×10^{11}	6.9×10^{10}	3.94

Table 4: **The amount of data read via a network on various graphs. The ratio ($\frac{PTE_{CD}}{PTE_{SC}}$) is about $1.10 \approx \sqrt{6/5}$ for all datasets, as expected.**

Dataset	PTE$_{CD}$	PTE$_{SC}$	$\frac{PTE_{CD}}{PTE_{SC}}$
CW12/256	22.5GB	18.7GB	1.20
CW12/128	58.8GB	51.5GB	1.14
CW12/64	174GB	162GB	1.08
CW12/32	485GB	426GB	1.14
CW12/16	1.3TB	1.2TB	1.09
CW12/8	3.7TB	3.3TB	1.10
CW12/4	9.9TB	9.0TB	1.10
CW12/2	26.9TB	24.1TB	1.12
CW12	72.9TB	64.3TB	1.13
TWT	102GB	83GB	1.23
SD	206GB	191GB	1.08
TW	1.9TB	1.7TB	1.13
CW09	2.9TB	2.6TB	1.11

ations in intersecting two neighbor sets (line 13 in Algorithm 3) in Table 3. PTE$_{CD}$ reduces the number of comparisons by up to 6.85× from PTE$_{BASE}$ by the color-direction.

Effect of scheduling computation. We also show the effect of scheduling calls of the function `enuerateTrianglesCD` (Section 4.3) by comparing the amount of data read via a network by PTE$_{CD}$ and PTE$_{SC}$ in Table 4. As expected, for every dataset, the ratio ($\frac{PTE_{CD}}{PTE_{SC}}$) is about $1.10 \approx \sqrt{6/5}$.

Running time comparison. We now compare the running time of PTEs, and competitors (CTTP, MGT, GraphLab, GraphX) in Figure 7. PTE$_{SC}$ shows the best performance and PTE$_{CD}$ follows it very closely. CTTP fails to run when edges are more than 20 billions within 5 days. GraphLab, GraphX and MGT also fail when the number of edges is larger than 600 million, 3 billion and 10 billion, respectively, because of out of memory or out of range error. The out of range error occurs when a vertex id exceeds the range of 32-bit integer limit. Note that, even when MGT can treat vertex ids exceeding the integer range, the performance of MGT would worsen as the graph size increases since MGT performs massive I/O ($O(|E|^2/M)$) when the input data size is large. The slope of the PTEs is 1.26, which is smaller than 1.5. It means that the practical time complexity is smaller than the worst case $O(|E|^{3/2})$.

Figure 1 shows the running time of various algorithms on real world datasets. PTE$_{SC}$ shows the best performances outperforming CTTP and MGT by up to 47× and 17×, respectively. Only the proposed algorithms succeed in processing ClueWeb12 with 6.3 billion vertices and 72 billion edges while all other algorithms fail to process the graph.

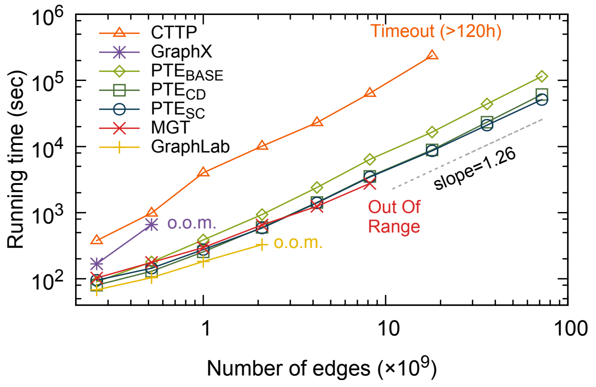


Figure 7: **The running time on ClueWeb12 with various numbers of edges. o.o.m.: out of memory. PTE$_{SC}$ shows the best data scalability; only PTEs succeed in processing the subgraphs containing more than 20B edges. The pre-partitioning (CTTP vs PTE$_{BASE}$) significantly reduces the running time while the effect of the scheduling function (PTE$_{CD}$ vs PTE$_{SC}$) is relatively insignificant.**

5.2.2 *Machine Scalability*

We evaluate the machine scalability of PTEs by measuring the running time of them and CTTP on YahooWeb varying the number of machines from 5 to 40 in Figure 8. Note that GraphX and GraphLab are omitted because they fail to process YahooWeb on our cluster. Every PTE shows strong scalability: the slope -0.94 of the PTEs is very close to the ideal value -1. It means that the running time decreases $2^{0.94} = 1.92$ times as the number of machines is doubled. On the other hand, CTTP shows weak scalability when the number of machine increases from 20 to 40.

5.2.3 *PTE on Spark*

We implemented PTE$_{SC}$ on Spark as well as on Hadoop to show that PTEs are general enough to be implemented in any distributed system supporting map and reduce functionalities. We compare the running time of the two implementations in Figure 9. The result indicates that the Spark implementation does not show a better performance than the Hadoop implementation even though the Spark implementation uses a distributed memory as well as disks. The reason is that PTE is not an iterative algorithm; thus, there are little data to reuse from memory. Even reading data

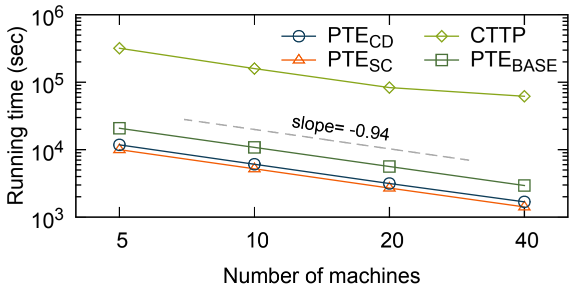


Figure 8: **Machine scalability of PTEs and CTTP on YahooWeb. GraphLab and GraphX are excluded because they failed to process YahooWeb. PTEs show very strong scalability with exponent -0.94 which is very close to -1, the ideal.**

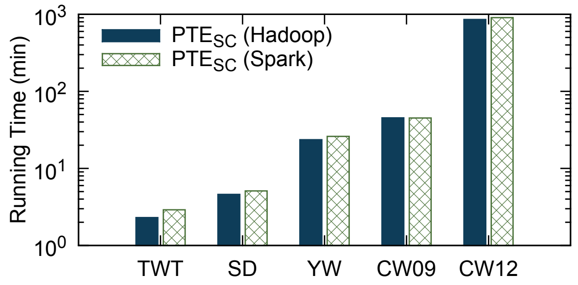


Figure 9: The running time of PTE_{SC} on Hadoop and Spark. There is no significant difference between them.

from distributed memory, not from remote disk, does not contribute to performance improvement of PTE; they take similar time since disk reading and network transmission occur simultaneously.

6. CONCLUSION

In this paper, we propose PTE, a scalable distributed algorithm for enumerating triangles in very large graphs. We carefully design PTE so that it minimizes the amount of shuffled data, total work, and network read. PTE operates in $O(|E|)$ shuffled data, $O(|E|^{3/2}/\sqrt{M})$ network read, and $O(|E|^{3/2})$ total work, the worst case optimal. PTE shows the best performances in real world data: it outperforms the state-of-the-art scalable distributed algorithm by up to 47×. Also, PTE is the only algorithm that successfully enumerates more than 3 trillion triangles in ClueWeb 12 graph with 72 billion edges while all other algorithms including GraphLab, GraphX, MGT, and CTTP fail. Future research directions include extending the work to support general subgraphs.

Acknowledgments

This work was supported by IT R&D program of MSIP/IITP [R0101-15-0176, Development of Core Technology for Human-like Self-taught Learning based on Symbolic Approach]. This work was also funded 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"). The ICT at Seoul National University provides research facilities for this study.

7. REFERENCES

[1] Jesse Alpert and Nissan Hajaj. `http://googleblog.blogspot.kr/2008/07/we-knew-web-was-big.html`, 2008.

[2] Shaikh Arifuzzaman, Maleq Khan, and Madhav V. Marathe. PATRIC: a parallel algorithm for counting triangles in massive networks. In *CIKM*, 2013.

[3] Luca Becchetti, Paolo Boldi, Carlos Castillo, and Aristides Gionis. Efficient algorithms for large-scale local triangle counting. *TKDD*, 2010.

[4] Jonathan W Berry, Bruce Hendrickson, Randall A LaViolette, and Cynthia A Phillips. Tolerating the community detection resolution limit with edge weighting. *Phys. Rev. E*, 83(5):056119, 2011.

[5] Bin-Hui Chou and Einoshin Suzuki. Discovering community-oriented roles of nodes in a social network. In *DaWaK*, pages 52–64, 2010.

[6] Jonathan Cohen. Graph twiddling in a mapreduce world. *CiSE*, 11(4):29–41, 2009.

[7] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI*, pages 137–150, 2004.

[8] Jean-Pierre Eckmann and Elisha Moses. Curvature of co-links uncovers hidden thematic layers in the world wide web. *PNAS*, 99(9):5825–5829, 2002.

[9] Facebook. `http://newsroom.fb.com/company-info`, 2015.

[10] Ilias Giechaskiel, George Panagopoulos, and Eiko Yoneki. PDTL: parallel and distributed triangle listing for massive graphs. In *ICPP*, 2015.

[11] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *OSDI*, pages 17–30, 2012.

[12] Herodotos Herodotou. Hadoop performance models. *arXiv*, 2011.

[13] Xiaocheng Hu, Yufei Tao, and Chin-Wan Chung. Massive graph triangulation. In *SIGMOD*, pages 325–336, 2013.

[14] ByungSoo Jeon, Inah Jeon, Lee Sael, and U Kang. Scout: Scalable coupled matrix-tensor factorization - algorithm and discoveries. In *ICDE*, 2016.

[15] U Kang, Jay-Yoon Lee, Danai Koutra, and Christos Faloutsos. Net-ray: Visualizing and mining billion-scale graphs. In *PAKDD*, 2014.

[16] U Kang, Brendan Meeder, Evangelos E. Papalexakis, and Christos Faloutsos. Heigen: Spectral analysis for billion-scale graphs. *TKDE*, pages 350–362, 2014.

[17] U Kang, Hanghang Tong, Jimeng Sun, Ching-Yung Lin, and Christos Faloutsos. Gbase: an efficient analysis platform for large graphs. *VLDB J.*, 21(5):637–650, 2012.

[18] U Kang, Charalampos E. Tsourakakis, and Faloutsos Faloutsos. Pegasus: A peta-scale graph mining system - implementation and observations. *ICDM*, 2009.

[19] Jinha Kim, Wook-Shin Han, Sangyeon Lee, Kyungyeol Park, and Hwanjo Yu. OPT: A new framework for overlapped and parallel triangulation in large-scale graphs. In *SIGMOD*, pages 637–648, 2014.

[20] Matthieu Latapy. Main-memory triangle computations for very large (sparse (power-law)) graphs. *Theor. Comput. Sci.*, pages 458–473, 2008.

[21] Rasmus Pagh and Francesco Silvestri. The input/output complexity of triangle enumeration. In *PODS*, pages 224–233, 2014.

[22] Ha-Myung Park and Chin-Wan Chung. An efficient mapreduce algorithm for counting triangles in a very large graph. In *CIKM*, pages 539–548, 2013.

[23] Ha-Myung Park, Francesco Silvestri, U Kang, and Rasmus Pagh. Mapreduce triangle enumeration with guarantees. In *CIKM*, pages 1739–1748, 2014.

[24] Filippo Radicchi, Claudio Castellano, Federico Cecconi, Vittorio Loreto, and Domenico Parisi. Defining and identifying communities in networks. *PNAS*, 101(9):2658–2663, 2004.

[25] Thomas Schank. Algorithmic aspects of triangle-based network analysis. *Phd thesis, University Karlsruhe*, 2007.

[26] Siddharth Suri and Sergei Vassilvitskii. Counting triangles and the curse of the last reducer. In *WWW*, pages 607–614, 2011.

[27] Twitter. `https://about.twitter.com/company`, 2015.

[28] Mark N. Wegman and Larry Carter. New hash functions and their use in authentication and set equality. *J. Comput. Syst. Sci.*, 22(3):265–279, 1981.

[29] Zhi Yang, Christo Wilson, Xiao Wang, Tingting Gao, Ben Y. Zhao, and Yafei Dai. Uncovering social network sybils in the wild. *TKDD*, 2014.