Bermuda: An Efficient MapReduce Triangle Listing Algorithm for Web-Scale Graphs

Dongqing Xiao
Worcester Polytechnic Institute
100 Institute Road
Worcester, MA, U.S.A
dxiao@wpi.edu

Mohamed Eltabakh
Worcester Polytechnic Institute
100 Institute Road
Worcester, MA, U.S.A
meltabakh@wpi.edu

Xiangnan Kong
Worcester Polytechnic Institute
100 Institute Road
Worcester, MA, U.S.A
xkong@wpi.edu

ABSTRACT

Triangle listing plays an important role in graph analysis and has numerous graph mining applications. With the rapid growth of graph data, distributed methods for listing triangles over massive graphs are urgently needed. Therefore, the triangle listing problem has been studied in several distributed infrastructures including MapReduce. However, existing algorithms suffer from generating and shuffling huge amounts of intermediate data, where interestingly, a large percentage of this data is redundant. Inspired by this observation, we present the *"Bermuda"* method, an efficient MapReduce-based triangle listing technique for massive graphs.

Different from existing approaches, Bermuda effectively reduces the size of the intermediate data via redundancy elimination and sharing of messages whenever possible. As a result, Bermuda achieves orders-of-magnitudes of speedup and enables processing larger graphs that other techniques fail to process under the same resources. Bermuda exploits the locality of processing, i.e., in which reduce instance each graph vertex will be processed, to avoid the redundancy of generating messages from mappers to reducers. Bermuda also proposes novel message sharing techniques within each reduce instance to increase the usability of the received messages. We present and analyze several reduce-side caching strategies that dynamically learn the expected access patterns of the shared messages, and adaptively deploy the appropriate technique for better sharing. Extensive experiments conducted on real-world large-scale graphs show that Bermuda speeds up the triangle listing computations by factors up to 10x. Moreover, with a relatively small cluster, Bermuda can scale up to large datasets, e.g., ClueWeb graph dataset (688GB), while other techniques fail to finish.

CCS Concepts

•Computing methodologies → MapReduce algorithms;

SSDBM '16 Budapest, Hungary
© 2016 ACM. ISBN 978-1-4503-4215-5…$15.00
DOI:

Keywords

Distributed Triangle Listing; MapReduce; Graph Analytics.

1. INTRODUCTION

Graphs arise naturally in many real-world applications such as social networks, bio-medical networks, and communication networks. In these applications, the graph can often be massive involving billions of vertices and edges. For example, Facebook's social network involves more than 1.23 billion users (vertices), and more than 208 billion friendships (edges). Such massive graphs can easily exceed the available memory of a single commodity computer. That is why distributed analysis on massive graphs has become an important research area in recent years [12, 21].

Triangle listing—which involves listing all triangles in a given graph—is well identified as a building-block operation in many graph analysis and mining techniques [10, 17]. First, several graph metrics can be directly obtained from triangle listing, e.g., *clustering coefficient* and *transitivity*. Such graph metrics have wide applications including quantifying graph density, detecting spam pages in web graphs, and measuring content quality in social networks [5]. Moreover, triangle listing has a broad range of applications including the discovery of dense sub-graphs [17], study of motif occurrences [22], and uncovering of hidden thematic relations in the web [10]. There is another well-known and closely-related problem to triangle listing, which is the *triangle counting* problem. Clearly, solving the triangle listing problem would automatically solve triangle counting, but not vice versa. Compared to triangle counting, triangle listing serves a broader range of applications. For example, Motif identification [22], community detection [6], and dense subgraphs [17] are all dependent on the more complex *triangle listing* problem.

Several techniques have been proposed for processing web-scale graphs including streaming algorithms [5, 7], external-memory algorithms [14, 18, 19], and distributed parallel algorithms [2, 31]. The streaming algorithms are limited to the *approximate triangle counting* problem. External-memory algorithms exploit asynchronous I/O and multi-core parallelism for efficient triangle listing [13, 14, 19]. In spite of achieving an impressive performance, external-memory approaches assume that the input graphs are in a centralized storage, which is not the case for many emerging applications that generate graphs distributed in nature. Even more seriously, external-memory approaches cannot easily scale up in terms of computing resources and parallelization degree. Algorithm [31] presents a parallel algorithm for exact triangle counting using the MapReduce framework. The algorithm proposes a partitioning scheme that improves the memory requirements to some extent, yet it still suffers from a huge communication cost. Algorithm [2] presents an efficient MPI-based distributed memory algorithm on the basis of [31] with load balancing techniques. However, as a memory-based algorithm, it suffers from memory limitations.

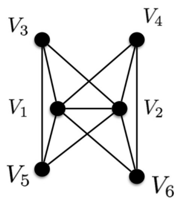
In addition to these techniques, several distributed and specialized graph frameworks have been recently proposed as general-purpose graph processing engines [11, 12, 21]. However, most of these frameworks are customized for iterative graph processing where distributed computations can be kept in-memory for faster subsequent iterations. However, the triangle listing algorithms are not iterative and would not make use of these optimizations.

In this paper, we propose *"Bermuda"*, a new scalable and distributed technique for the triangle listing problem on the cloud-based MapReduce infrastructure and its open-source implementation Hadoop [30]. We opt for MapReduce because of two main reasons, which are: (1) MapReduce has several desirable characteristics including scalability to TBs of data, efficient fault tolerance, and flexible data model that can handle graphs seamlessly, and (2) Triangle listing is usually a one step in bigger analysis tasks and workflows, and its output may feed other analytical tasks over non-graph data. MapReduce is a perfect fit for such workflows as it can handle all types of data ranging from graphs to structured and un-structured datasets in the same infrastructure.

Compared to the existing techniques, Bermuda has several key contributions, which can be summarized as follows:

- ***Independence of Graph Partitioning:*** Bermuda does not require any special partitioning of the graph, which suites current applications in which graph structures are very complex and dynamically changing.
- ***Awareness of Processing Order and Locality in the reduce phase:*** Bermuda's efficiency and optimizations are driven by minimizing the communication overhead and the number of message passing over the network. Bermuda achieves these goals by dynamically keeping track of where and when vertices will be processed in the reduce phase, and then maximizing the re-usability of information among the vertices that will be processed together. We propose several reduce-side caching strategies for enabling such re-usability and sharing of information.
- ***Portability of Optimization:*** We implemented Bermuda over the MapReduce infrastructure. However, the proposed optimizations can be deployed over other platforms such as Pregel [21], PowerGraph [12], and Spark-GraphX [11], and can be integrated with techniques that apply a graph pre-partitioning step.
- ***Scalability to Large Graphs even with Limited Compute Clusters:*** As our experiments show, Bermuda's optimizations—especially the reduction in communication overheads—enable the scalability to very large graphs, while the state-of-art technique fail to finish the job given the same resources.

The rest of the paper is organized as follows. We introduce the preliminaries in Section 2. The core of Bermuda and its optimizations are presented in Section 3. The experimental evaluation, and related work are presented in Sections 4, and 5, respectively. Finally, the conclusion remarks are included in Section 6.



Node	N_v
V_1	V_2, V_3, V_4, V_5, V_6
V_2	V_1, V_3, V_4, V_5, V_6
V_3	V_1, V_2, V_5
V_4	V_1, V_2, V_6
V_5	V_1, V_2, V_3
V_6	V_1, V_2, V_4

Figure 1: Adjacency List Example.

Symbol	Definition
$G(V, E)$	A simple graph
N_v	Adjacent nodes of v in G
N_v^H	Adjacent nodes of v with higher degree
d_v	Degree of v in G
$\hat{d_v}$	Effective Degree of v in G
$\triangle_{vuw}$	A triangle formed by u, v and w
$\triangle(v)$	The set of triangles that contains v
$\triangle(G)$	The set of all triangles in G

Table 1: Summary of Notations.

2. PRELIMINARIES

In this section, we first introduce several preliminary concepts and notations, and formally define the triangle listing problem. We then overview existing sequential algorithms for triangle listing, and highlight the key components of the MapReduce computing paradigm. Finally, we present naive parallel algorithms using MapReduce and discuss the open optimization opportunities, which will form the core of the proposed Bermuda technique.

2.1 Triangle Listing Problem

Suppose we have a simple undirected graph $G(V, E)$, where V is the set of vertices (nodes), and E is the set of edges. Let $n = |V|$ and $m = |E|$. Let $N_v = \{u|(u, v) \in E\}$ denote the set of *adjacent nodes* of node v, and $d_v = |N_v|$ denote the degree of node v. We assume that G is stored in the most popular format for graph data, *i.e.*, the adjacency list representation (as shown in Figure 1). Given any three distinct vertices $u, v, w \in V$, they form a triangle $\triangle_{uvw}$, iif $(u, v), (u, w), (v, w) \in E$. We define the set of all triangles that involve node v as $\triangle(v) = \{\triangle_{uvw}|\ (v, u), (v, w), (u, w) \in E\}$. Similarly, we define $\triangle(G) = \bigcup_{v \in V} \triangle(v)$ as the set of all triangles in G. For convenience, Table 1 summarizes the graph notations that are frequently used in the paper.

DEFINITION 1. **Triangle Listing Problem**: *Given a large-scale distributed graph $G(V, E)$, our goal is to report all triangles in G, i.e., $\triangle(G)$, in a highly distributed way.*

2.2 Sequential Triangle Listing

In this section, we present a sequential triangle listing algorithm which is widely used as the basis of parallel approaches [2, 24, 31]. In this work, we also use it as the basis of our distributed approach.

A naive algorithm for listing triangles is as follows. For each node $v \in V$, find the set of edges among its neighbors, i.e., pairs of neighbors that complete a triangle with node v. Given this simple method, each triangle (u, v, w) is listed

Algorithm 1 NodeIterator++

```
   Preprocessing step
1: for each $(u, v) \in E$ do
2:     if $u \succ v$, store $u$ in $N_v^H$
3:     else store $v$ in $N_u^H$
   Triangle Listing
4: $\triangle(G) \leftarrow \emptyset$
5: for each $v \in V$ do
6:     for each $u \in N_v^H$ do
7:         for each $w \in N_v^H \bigcap N_u^H$ do
8:             $\triangle(G) \leftarrow \triangle(G) \bigcup \{\triangle_{vuw}\}$
```

Algorithm 2 MR-Baseline

```
   Map: Input: $\langle v; N_v^H \rangle$
1: emit $\langle v; (v, N_v^H) \rangle$
2: for each $u \in N_v^H$ do
3:     emit $\langle u; (v, N_v^H) \rangle$

   Reduce:Input:$[\langle u; (v, N_v^H) \rangle]$
4: initiate $N_u^H$
5: for each $\langle u; (v, N_v^H) \rangle$ do
6:     for each $w \in N_u^H \cap N_v^H$ do
7:         emit $\triangle_{vuw}$
```

six times—all six permutations of u, v and w. Several other algorithms have been proposed to improve on and eliminate the redundancy of this basic method, e.g., [5, 28]. One of the algorithms, known as *NodeIterator++* [28], uses a total ordering over the nodes to avoid duplicate listing of the same triangle. By following a specific ordering, it guarantees that each triangle is counted only once among the six permutations. Moreover, the NodeIterator++ algorithm adopts an interesting node ordering based on the nodes' degrees, with ties broken by node IDs, as defined blow:

$$u \succ v \iff d_u > d_v \text{ or } (d_u = d_v \text{ and } u > v) \qquad (1)$$

This degree-based ordering improves the running time by reducing the diversity of the effective degree $\hat{d_v}$. The running time of NodeIterator++ algorithm is $O(m^{3/2})$. A comprehensive analysis can be found in [28].

The standard *NodeIterator++* algorithm performs the degree-based ordering comparison during the final phase, i.e., the triangle listing phase. The work in [2] and [31] further improves on that by performing the comparison $u \succ v$ for each edge $(u, v) \in E$ in the preprocessing step (Lines 1-3, Algorithm 1). For each node v and edge (u, v), node u is stored in the effective list of v (N_v^H) if and only if $u \succ v$, and hence $N_v^H = \{u : u \succ v \text{ and } (u, v) \in E\}$. The preprocessing step cuts the storage and memory requirement by half since each edge is stored only once. After the preprocessing step, the effective degree of nodes in G is $O(\sqrt{m})$ [28]. Its correctness proof can be found in [2]. The modified NodeIterator++ algorithm is presented in Algorithm 1.

2.3 MapReduce Overview

MapReduce is a popular distributed programming framework for processing large datasets [9]. MapReduce, and its open-source implementation Hadoop [33], have been used for many important graph mining tasks [24, 31]. In this paper, our algorithms are designed and analyzed in the MapReduce framework.

Computation Model. An analytical job in MapReduce executes in two rigid phases, called the *map* and *reduce* phases. Each phase consumes/produces records in the form of *key-value* pairs—We will use the keywords *pair*, *record*, or *message* interchangeably to refer to these key-value pairs. A pair is denoted as $\langle k; val \rangle$, where k is the key and val is the value. The *map* phase takes one key-value pair as input at a time, and produces zero or more output pairs. The *reduce* phase receives multiple key-listOfValues pairs and produces zero or more output pairs. Between the two phases, there is an implicit phase, called *shuffling/sorting*, in which the mappers' output pairs are shuffled and sorted to group the pairs of the same key together as input for reducers.

Bermuda will leverage and extend some of the basic functionality of MapReduce, which are:

- **Key Partitioning:** Mappers employ a *key partitioning function* over their outputs to partition and route the records across the reducers. By default, it is a hash-based function, but can be replaced by any other user-defined logic.
- **Multi-Key Reducers:** Typically, the number of distinct keys in an application is much larger than the number of reducers in the system. This implies that a single reducer will sequentially process multiple keys—along with their associated groups of values—in the same reduce instance. Moreover, the processing order is defined by *key sorting function* used in shuffling/sorting phase. By default, a single reduce instance processes each of its input groups in total isolation from the other groups with no sharing or communication.

2.4 Triangle Listing in MapReduce

Both [31] and [2] use the NodeIterator++ algorithm as the basis of their distributed algorithms. [31] identifies the triangles by checking the existence of pivot edges, while [2] uses set intersection of effective adjacency list (Line 7, Algorithm 1). In this section, we present the MapReduce version of the NodeIterator++ algorithm similar to the one presented in [2], referred to as *MR-Baseline* (Algorithm 2).

The general approach is the same as in the NodeIterator++ algorithm. In the map phase, each node v needs to emit two types of messages. The first type is used for the initiation its own effective adjacency list in the reduce side, referred to as *a core message* (Line 1, Algorithm 2). The second type is used for identifying triangles, referred to as *pivot messages* (Lines 2-3, Algorithm 2). All pivot messages from v to its effective adjacent nodes are identical. In the reduce phase, each node u will receive a core message from itself, and a pivot message from adjacent nodes with the lower degree. Then, each node identifies the triangles by performing a set intersection operation (Lines 5-6, Algorithm 2).

We omit the code of the pre-processing procedure since its implementation is straightforward in MapReduce. In addition, we will exclude the pre-processing cost for any further consideration since it is typically dominated by the actual running time of the triangle listing algorithm, plus it is the same overhead for all algorithms.

2.4.1 Analysis and Optimization Opportunities

The algorithm correctness and overall computational complexity follow the sequential case. Our analysis will thus focus on the space usage of the intermediate data and the execution efficiency captured in terms of the wall-clock execution time. For the convenience of analysis, we assume that each edge (u, v) requires one memory word.

Intermediate Data Size. As presented in [31], the total number of intermediate records generated by MR-Baseline can be $O(m^{\frac{3}{2}})$ in the worst case, where m is the number of edges. The size of this intermediate data can be much larger than the original graph size. Thus, issues related network congestion and job failure may arise with massive input graphs. Indeed, the network congestion resulting from transmitting a large amount of data during the shuffle phase can be a bottleneck, degrading the performance, and limiting the scalability of the algorithm.

Execution Time. It is far from trivial to list the factors contributing to the execution time of a map-reduce job. In this work, we consider the following two dominating factors of the triangle list algorithm. The first one is the total size of the intermediate data generated and shuffled between the map and reduce phases. And the second factor is the variance and imbalance among the mappers' workloads. We refer to the imbalanced workload among mappers as *"map skew"*. Map skew leads to the straggler problem, *i.e.*, a few mappers take significantly longer time to complete than the rest, thus they delay the progress of the entire job [20, 27]. We use the variance of the map output size to measure the imbalance among mappers. More specifically, the bigger the variance of the mappers' output sizes, the greater the imbalance and the more serious the straggler problem. The map output variance is defined as in the following theorem.

THEOREM 1. *For a given graph $G(V, E)$, let a random variable x denotes the effective degree for any vertex in G and the variance of x is denotes as Var(x). Then, the expectation of x ($E(x)$) equals the average degree computed as $E(x) = \frac{m}{n}$. For typical graphs, $Var(x) \neq 0$ and $E(x) \neq 0$ always hold. Since each mapper starts with approximately the same input size (say receives c graph nodes), the variance of the output size among mappers is close to $4cE(X)^2Var(x)$.*

PROOF. Let function $g(x)$ be the map output size generated by single node with the effective degree x, then $g(x) = x^2$ (Line 2-3, Algorithm 2). Thus, the total size of map output generated by c nodes in a single mapper $T_i(X) = \sum_{i=1}^{c} g(x_i)$. Since $x_1, x_2, ..x_c$ are independent and identically distributed random variables, $Var(T(x)) = c * Var(g(x))$. Apply delta method [23] to estimate Var(g(x)) as follows:

$$Var(g(x)) \approx g'(x)^2 Var(x) \approx (2x)^2 Var(x)$$

The approximate variance of $g(x)$ is then

$$Var(g(x)) \approx 4E(x)^2 Var(x)$$

Here, the variance of the total map output size among mappers is close to $4cE(X)^2Var(x)$. $\square$

Opportunities for Optimization: In the plain Hadoop, each reduce instance processes its different keys (nodes) independent from each others. Generally, this is good for parallelization. However, in triangle listing, it involves significant redundancy in communication. In the map phase, each node sends the **identical** pivot message to each of its effective neighbors in N_v^H (Lines 2-3, Algorithm 2) even though many of them may reside in and get processed by the same reduce instance. For example, if a node v has 1,000 *effective* neighbors on reduce worker j, then v sends the same message 1,000 times to reduce worker j. In web-scale graphs, such redundancy in intermediate data can severely degrade the performance, drastically consume resources, and in some cases causes job failures.

3. Bermuda TECHNIQUE

With the MR-Baseline Algorithm, one node needs to send the same pivot message to multiple nodes residing in the same reducer. Therefore, the network traffic can be reduced by sending only one message to the destination reducer, and either have main-memory cache or distributing the message to actual graph nodes within each reducer. Although this strategy seems quite simple, and other systems such as GPS and X-Pregel [3, 26] have implemented it, the trick lies on how to efficiently perform the caching and sharing. In this section, we propose new effective caching strategies to maximize the sharing benefit while encountering little overhead. We also present novel theoretical analysis for the proposed techniques.

In the frameworks of GPS and X-Pregel, adjacency lists of high degree nodes are used for identifying distinct destination reducer and distributing the message to target nodes in the reduce side. This method requires extensive memory and computations for message sharing. In contrast, in Bermuda, each node uses the *universal key partition function* to group its destination nodes. Thus, each node would only send the same pivot message to each reduce instance only once. At the same time, reduce instances will adopt different message-sharing strategies to guarantee the correctness of algorithm. As a result, Bermuda achieves a trade off between reducing the network communication—which is known to be a big bottleneck for map-reduce jobs—and increasing the processing cost and memory utilization.

In the following, we present two modified algorithms with different message-sharing strategies.

3.1 Bermuda Edge-Centric Node++

A straightforward (and intuitive) approach for sharing the pivot messages within each reduce instance is to organize either the pivot or core messages in main-memory for efficient random access. We propose the Bermuda Edge-Centric Node++ (Bermuda-EC) algorithm, which is based on the observation that for a given input graph, it is common to have the number of core messages smaller than the number of pivot messages. Therefore, the main idea of Bermuda-EC algorithm is to first read the core messages, cache them in memory, and then stream the pivot messages, and on-the-fly intersect the pivot messages with the needed core messages (See Figure 2). The MapReduce code of the Bermuda-EC algorithm is presented in Algorithm 3.

In order to avoid pivot message redundancy, a universal key partitioning function is utilized by mappers. The corresponding modification in the map side is as follows. First, each node v employs a universal key partitioning function $h()$ to group its destination nodes (Line 3, Algorithm 3). This grouping captures the graph nodes that will be processed by the same reduce instance. Then, each node v sends a pivot message including the information of N_v^H to each non-empty group (Lines 4-6, Algorithm 3). Following this strategy, each reduce instance receives each pivot message exactly once even if it will be referenced multiple times.

Moreover, we use tags to distinguish core and pivot messages, which are not listed in the algorithm for simplicity. Combined with the MapReduce internal sorting function,

Algorithm 3 Bermuda-EC

```
    Map: Input:($\langle v; N_v^H \rangle$)
    Let $h(.)$ be a key partitioning function into [0,k-1]
 1: $j \leftarrow h(v)$
 2: emit $\langle j; (v, N_v^H) \rangle$
 3: Group the set of nodes in $N_v^H$ by $h(.)$
 4: for each $i \in [0, k-1]$ do
 5:     if $gp_i \neq \emptyset$ then
 6:         emit $\langle i; (v, N_v^H) \rangle$

    Reduce:Input:[$\langle i; (v, N_v^H) \rangle$]
 7: initiate all the core nodes' $N_u^H$ in main memory
 8: for each pivot message $\langle i; (v, N_v^H) \rangle$ do
 9:     for each $u \in N_v^H$ and $h(u) = i$ do
10:         for each $w \in N_v^H \cap N_u^H$ do
11:             emit $\triangle_{vuw}$
```

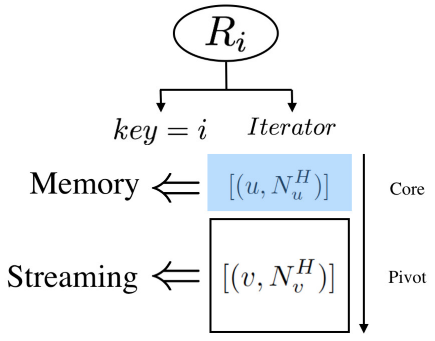


Figure 2: Bermuda-EC Execution.

Bermuda-EC guarantees that all core messages are received by the reduce function before any of the pivot messages as illustrated in Figure 2. Therefore, it becomes feasible to cache only the core messages in memory, and then perform the intersection as the pivot messages are received.

The corresponding modification in the reduce side is as follows. For a given reduce instance R_i, it first reads all the core message into main-memory (Line 7, Algorithm 3). Then, it iterates over all pivot message. Each pivot message is intersected with the cached core messages for identifying the triangles. As presented in the MR-Baseline algorithm (Algorithm 2), each pivot message (v, N_v^H) needs to be processed in reduce instance R_i only for nodes $u : u \in N_v^H$ *where* $h(u) = i$. Interestingly, this information is encoded within the pivot message. Thus, each pivot message is processed for all its requested core nodes once received (Lines 9-11, Algorithm 3).

3.1.1 Analysis of Bermuda-EC

Extending the analysis in Section 2.4, we demonstrate that Bermuda-EC achieves improvement over MR-Baseline w.r.t both space usage and execution efficiency. Furthermore, we discuss the effect of the number of reducers k on the algorithm performance.

THEOREM 2. *For a given number of reducers k, we have:*

- *The expected total size of the map output is $O(km)$.*
- *The expected size of core messages to any reduce instance is $O(m/k)$.*

PROOF. As shown in Algorithm 3, the size of the map output generated by node v is at most $k * \hat{d_v}$. Thus, the total size of the map output T is as follows:

$$T < \sum_{v \in V} k\hat{d_v} = k \sum_{v \in V} \hat{d_v} = km$$

For the second bound, observe that a random edge is present in a reduce instance R_i and represented as a core message with probability $1/k$. By following the *Linearity of Expectation*, the expected number of the core messages to any reduce instance is $O(m * \frac{1}{k})$. □

Space Usage. Theorem 2 shows that when $k \ll \sqrt{m}$ (the usual case for massive graphs), then the total size of the map output generated by Bermuda-EC algorithm is significantly less than that generated by the MR-Baseline algorithm. In other words, Bermuda-EC is able to handle even larger graphs with limited compute clusters.

Execution Time. A positive consequence of having a smaller intermediate result is that it requires less time for generating and shuffling/sorting the data. Moreover, the imbalance of the map outputs is also reduced significantly by limiting the replication factor of the pivot messages up to k. The next theorem shows the approximate variance of the number of the intermediate result from mappers. When $k < E(x)$, it implies smaller variance among the mappers than that of the MR-Baseline algorithm. Together, Bermuda-EC achieves better performance and scales to larger graphs compared to the MR-Baseline algorithm.

THEOREM 3. *For a given graph $G(V, E)$, let a random variable x denotes the effective degree of any node in G and the variance of x is denotes as $Var(x)$. Then the expectation of x ($E(x)$) equals the average degree and computed as $E(x) = \frac{m}{n}$. For typical graphs, $Var(x) \neq 0$ and $E(x) \neq 0$ always hold. Since each mapper starts with approximately the same input size (say receives c graph nodes), the variance of the map output's size under the Bermuda-EC Algorithm is $O(2ck^2Var(x))$, where k represents the number of reducers.*

PROOF. Assume the number of reducers is k. Given a graph node v, where its effective degree $\hat{d_v} = x$. Let random variable $y(x)$ be the number of distinct reducers processing the effective neighbors of v, and thus $y(x) \leq k$. Then, the size of the map output generated by a single node u would be xy, denoted as $g(x)$(Lines 3-4, Algorithm 3). Thus, the total size of the map output generated by c nodes in a single mapper $T(X) = \sum_{i=1}^{c} g(x_i)$. Since $x_1, x_2, ..x_c$ are independent and identically distributed random variables, then $Var(T(x)) = c * Var(g(x))$. The approximate variance of $g(x)$ is as follows

$$\begin{aligned} Var(xy) &= E(x^2y^2) - E(xy)^2 \\ &< E(x^2y^2) \\ &< k^2E(x^2) \\ &< k^2(E(x)^2 + Var(x)) \\ &< 2k^2Var(x) \end{aligned}$$

As presented in [28], $E(x^2) \approx \frac{m^{\frac{3}{2}}}{n}$ and $E(x) = \frac{m}{n}$. Thus $\frac{E(x^2)}{E(x)^2} \approx \frac{n}{\sqrt{m}}$. In many real graphs where $n^2 > m$ it implies

$\frac{n}{\sqrt{m}} > \sqrt{m} > 2$. It implies $E(x^2) > 2E(x)^2$, thus $Var(x) = E(x^2) - E(x)^2 > E(x)^2$. □

We now study in more details the effect of parameter k (the number of reducers) on the space and time complexity for the Bermuda-EC algorithm.

Effect on Space Usage. The reducers number k trades off the memory used by a single reduce instance and the size of the intermediate data generated during the MapReduce job. The memory used by a single reducer should not exceed the available memory of a single machine, *i.e.*, $O(m/k)$ should be sub-linear to the size of main memory in a single machine. In addition, the total space used by the intermediate data must also remain bounded, *i.e.*, $O(km)$ should be no larger than the total storage. Given a cluster of machines, these two constraints define the bounds of k for a given input graph $G(V, E)$.

Effect on Execution Time. The reducers number k trades off the reduce computation time and the time for shuffling and sorting. As the parallelization degree k increases, it reduces the computational time in the reduce phase. At the same time, the size of the intermediate data, *i.e.*, $O(km)$ increases significantly as k increases (notice that m is very large), and thus the communication cost becomes a bottleneck in the job's execution. Moreover, the increasing variance among mappers $O(2ck^2Var(x))$ implies a more significant straggler problem which slows down the execution progress.

In general, Bermuda-EC algorithm favors the smaller setting of k for higher efficiency while subjects to memory bound that the expected size of core message $O(m/k)$ should not exceed the available memory of a single reduce instance.

Unfortunately, for processing web-scale graphs such as *ClueWeb* with more than 80 billion edges (and total size of approximately 700GBs)—which as we will show the state-of-art techniques cannot actually process—the number of reducers needed for Bermuda-EC for acceptable performance is in the order of 100s. Although, this number is very reasonable for most mid-size clusters, the intermediate results $O(km)$ will be huge, which leads to significant network congestion.

Disk-Based Bermuda-EC: A generalization to the proposed Bermuda-EC algorithm that guarantees no failure even under the case where the core messages cannot fit in a reducer's memory is the *Disk-Based Bermuda-EC* variation. The idea is straightforward and relies on the usage of the local disk of each reducer. The main idea is as follows: (1) Partition the core messages such that each partition fits into main memory, and (2) Buffer a group of pivot messages, and then iterate over the core messages one partition at a time, and for each partition, identify the triangles as in the standard Bermuda-EC algorithm. Obviously, such method trades off between disk I/O (pivot message scanning) and main-memory requirement. For a setting of reduce number k, the expected size of core messages in a single reduce instance is $O(m/k)$, thus the expected number of rounds is $O(\frac{m}{kM})$ where M represents the size of available main-memory for single reducer. The expected size of pivot message reaches $O(m)$. Therefore, the total disk I/O reaches $O(\frac{m^2}{kM})$. In the case of massive graph, it implies longer time.

3.2 Bermuda Vertex-Centric Node++

As discussed in Section 3.1, the Bermuda-EC algorithm assumes that the core messages can fit in the memory of a single reducer. However, it is not always guaranteed to be the case, especially in web-scale graphs.

One crucial observation is that the access pattern of the pivot messages can be learned and leveraged for better reusability. In MapReduce, a single reduce instance processes many keys (graph nodes) in a specific sequential order. This order is defined based on the key comparator function. For example, let $h()$ be the key partitioning function and $l()$ be key comparator function within the MapReduce framework, then $h(u) = h(w) = i$ and $l(u, w) < 0$ implies that the reduce instance R_i is responsible for the computations over nodes u, w, and also the computations of node u precede that of node w. Given these known functions, the relative order among the keys in the same reduce instance becomes known, and the access pattern of the pivot message can be predicted. The knowledge of the access pattern of the pivot messages holds a great promise for proving better caching and better memory utilization.

Inspired by these facts, we propose the Bermuda-VC algorithm which supports random access over the pivot messages by caching them in main-memory while streaming in the core messages. More specifically, Bermuda-VC will reverse the assumption of Bermuda-EC, where we now try to make the pivot messages arrive first to reducers, get them cached and organized in memory, and then the core messages are received and processed against the pivot messages. Although the size of the pivot messages is usually larger than that of the core messages, their access pattern is more predictable which will enable better caching strategies as we will present in this section. The Bermuda-VC algorithm is presented in Algorithm 4.

The Bermuda-VC algorithm uses a shared buffer for caching the pivot messages. And then, for the reduce-side computations over a core node u, the reducer compares u's core message with all related pivot messages—some are associated with u's core message, while the rest should be

Algorithm 4 Bermuda-VC

```
    Map: Input:($\langle v; (N_v^L, N_v^H)\rangle$)
    Let $h(.)$ be a key partitioning function into [0,k-1]
    Let $l(.)$ be a key comparator function
 1: emit $\langle v; (v, N_v^L, N_v^H)\rangle$
 2: Group the set of nodes in $N_v^H$ by $h(.)$
 3: for each $i \in [0, k-1]$ do
 4:   if $gp_i \neq \emptyset$ then
 5:     $gp_i \Leftarrow sort(gp_i) basedon l(.)$
 6:     $u \Leftarrow gp_i.first$
 7:     $AP_{v,i} \Leftarrow accessPattern(gp_i)$
 8:     emit $\langle u; (v, AP_{v,i}, N_v^H)\rangle$

    Reduce:Input:[$\langle u; (v, AP_{v,i}, N_v^H)\rangle$]
 9: initiate the core node u' $N_u^L, N_u^H$ in main memory
10: for each pivot message $\langle u; (v, AP_{v,i}, N_v^H)\rangle$ do
11:   for each $w \in N_v^H \bigcap N_u^H$ do
12:     emit $\triangle_{vuw}$
13:   Put $(v, AP_{v,j}, N_v^H)$ into shared buffer
14:   $N_u^L \leftarrow N_u^L - \{v\}$
15: for each $r \in N_u^L$ do
16:   Fetch $(r, AP_{r,i}, N_r^H)$ from shared buffer
17:   for each $w \in N_r^H \bigcap N_u^H$ do
18:     emit $\triangle_{ruw}$
```

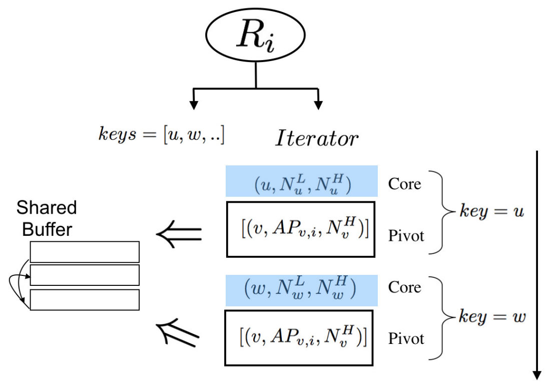


Figure 3: Bermuda-VC Execution.

residing in the shared buffer. Bermuda-VC algorithm applies the same scheme to avoid generating redundant pivot messages. It utilizes a universal key partitioning function to group effective neighbors N_v^H of each node v. In order to guarantee the availability of the pivot messages, a universal key comparator function is utilized to sort the destination nodes in each group (Line 5, Algorithm 4). As a result, destination nodes are sorted based on their processing order. The first node in group gp_i indicates the earliest request of a pivot message. Hence, each node v sends a pivot message to the first node of each non-empty group by emitting key value pairs where key equals the first node ID (Lines 6-8, Algorithm 4).

Combined with the sorting phase of the MapReduce framework, Bermuda-VC guarantees the availability of all needed pivot messages of any node u when u's core message is received by a reducer, i.e., the needed pivot messages are either associated with u itself or associated with another nodes processed before u.

The reducers' processing mechanism is similar to that of the MR-Baseline algorithm. Each node u reads its core message for initiating N_u^H and N_v^L (Line 9), and then it iterates over every pivot message associated with key u against its effective adjacency list N_v^H to enumerate the triangles (Lines 10-12). As discussed before, not all expected pivot messages are carried with key u. The rest of the related pivot messages reside in the shared buffer. Here, N_v^L is used for fetching the rest of these pivot messages (Line 14, Algorithm 4), and enumerating the triangles (Lines 15-18, Algorithm 4). Moreover, the new coming pivot messages associated with node u are pushed into the shared buffer for further access by other nodes (Line 13). Figure 3 illustrates the reduce-side processing flow of Bermuda-VC. In the following sections, we will discuss in more details the management of the pivot messages in the shared buffer.

3.3 Message Sharing Management

It is obvious that the best scenario is to have the shared buffer fits into the main memory of each reduce instance. However, that cannot be guaranteed. In general, there are two types of operations over the shared buffer inside a reduce instance, which are: *"Put"* for adding new incoming pivot messages into the shared buffer (Line 13), and *"Get"* for retrieving the needed pivot messages (Lines 15-18). For massive graphs, the main memory may not hold all the

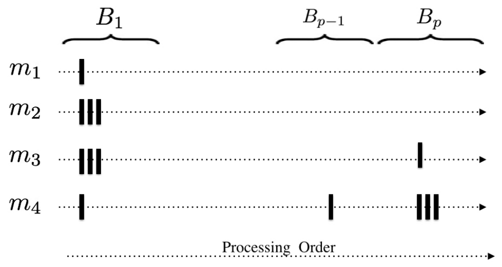


Figure 4: Various access Patterns for Pivot Messages.

pivot messages. This problem is similar to the classical caching problem studied in [16, 25], where a *reuse-distance* factor is used to estimate the distances between consecutive references of a given cached element, and based on that effective replacement policies can be deployed. We adopt the same idea in Bermuda-VC.

Interestingly, in addition to the reuse distance, all access patterns of each pivot message can be easily estimated in our context. The access pattern AP of a pivot message is defined as the sequence of graph nodes (keys) that will reference this message. In particular, the access pattern of a pivot message from node v to reduce instance R_i can be computed based on the sorted effective nodes gp_i received by R_i. Several interesting metrics can be derived from this access pattern. For example, the first node in gp_i indicates the occurrence of the first reference, the size of gp_i equals the cumulative reference frequency. Such access pattern information is encoded within each pivot message (Lines 7-8, Algorithm 4). With the availability of this access pattern, effective message sharing strategies can be deployed under limited memory.

As an illustrative example, Figure 4 depicts different access patterns for four pivot messages $\{m_1, m_2, m_3, m_4\}$. The black bars indicate requests to the corresponding pivot message, while the gaps represent the re-use distances (which are idle periods for this message). Pivot messages may exhibit entirely different access patterns, e.g., pivot message m_1 is referenced only once, while others are utilized more than once, and some pivot messages are used in dense consecutive pattern in a short interval, *e.g.*, m_2 and m_3. Inspired by these observations, we propose two heuristic-based replacement policies, namely *usage-based tracking*, and *bucket-based tracking*. They trade off the tracking overhead with memory hits as will be described next.

3.3.1 Usage-Based Tracking

Given a pivot message originated from node v, the total use frequency is limited to $\sqrt{m}$, referring to the number of its effective neighbors, which is much smaller than the expected number of nodes processed in a single reducer, which is estimated to n/k. This implies that each pivot message may become useless (and can be discard) as a reducer progresses, and it is always desirable to detect the earliest time at which a pivot message can be discarded to maximize the memory's utilization.

The main idea of the usage-based tracking is to use a usage counter per pivot message in the shared buffer. And then, the tracking is performed as follows. Each *Put* operation sets the counter as the total use frequency. And, only the

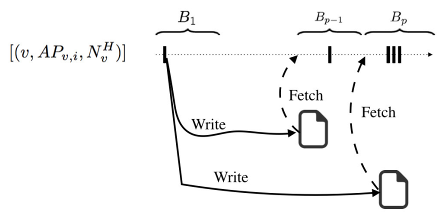


Figure 5: Read and Write Back-up files

pivot messages whose usage counter is larger than zero are added to the shared buffer. Each *Get* operation decrements the counter of the target pivot message by one. Once the counter reached zero, the corresponding pivot message is evicted from the shared buffer.

The usage-based scheme may fall short in optimizing sparse and scattered access patterns. For example, as shown in Figure 4, the reuse distance of message m_4 is large. Therefore, the usage-based tracking strategy has to keep m_4 in the shared buffer although it will not be referenced for a long time. What's worse, such scattered access is common in massive graphs. Therefore, pivot messages may unnecessarily overwhelm the available memory of each single reduce instance.

3.3.2 Bucket-Based Tracking

We introduce the *Bucket-based* tracking strategy to optimize message sharing over scattered access patterns. The main idea is to manage the access patterns of each pivot message at a smaller granularity, called a bucket. The processing sequence of keys/nodesis sliced into buckets as illustrated in Figure 4. In this work, we use the range partitioning method for balancing workload among buckets. Correspondingly, the usage counter of one pivot message is defined per bucket, i.e., each message will have an array of usage counters of a length equals to the number of its buckets. For example, the usage count of m_4 in the first bucket is 1 while in the second bucket is 0. Therefore, for a pivot message that will remain idle (with no reference) for a long time, its counter array will have a long sequence of adjacent zeros. Such access pattern information can be computed in the map function, encoded in access pattern (Line 7 in Algorithm 4), and passed to the reduce side.

The corresponding modification of the *Put* operation is as follows. Each new pivot message will be pushed into the shared buffer (in memory) and backed up by local files (in disk) based on its access pattern. Figure 5 illustrates this procedure. For the arrival of a pivot message with the access pattern $[1, 0, .., 1, 3]$, the reduce instance actively adds this message into back-up files for buckets B_{p-1} (next-to-last bucket) and B_p (last bucket). And then, at the end of each bucket processing and before the start of processing the next bucket, all pivot messages in the shared buffer are discarded, and a new set of pivot messages is fetched from the corresponding back-up file into memory(See Figure 5).

The Bucket-based tracking strategy provides better memory utilization since it prevents the long retention of unnecessary pivot messages. In addition, usage-based tracking can be applied to each bucket to combine both benefits, which is referred to as the *bucket-usage* tracking strategy.

3.4 Analysis of Bermuda-VC

In this section, we show the benefits of the Bermuda-VC algorithm over the Bermuda-EC algorithm. Furthermore, we discuss the effect of parameter p, which is the the number of buckets, on the performance.

Under the same settings of the number of reducers k, the Bermuda-VC algorithm generates more intermediate message and takes longer execution time. Firstly, the Bermuda-VC algorithm generates the same number of pivot messages while generating more core messages (*i.e.*, additional N_v^L for reference in the reduce side). Thus, the total size of the extra N_v^L core message is $\sum_{v \in V} N_v^L = m$. Such noticeable size of extra core messages requires additional time for generating and shuffling. Moreover, an additional computational overhead (Lines 13-14) is required for the message sharing management.

However, because of the proposed sharing strategies, the Bermuda-VC algorithm can work under smaller settings for k—which are settings under which the Bermuda-EC algorithm will probably fail. In this case, the benefits brought by having a smaller k will exceed the corresponding cost. In such cases, Bermuda-VC algorithm will outperform Bermuda-EC algorithm.

Moreover, compared to the disk-based Bermuda-EC algorithm, the Bermuda-VC algorithm has a relatively smaller disk I/O cost because the predictability of the access pattern of the pivot messages, which enable purging them early, while that is not applicable to the core messages. Notice that, for any given reduce instance, the expected usage count of pivot message from u is d_u^H/k. Thus, the expected usage count for any pivot message is $E(d_u^H)/k$, equals m/nk. Therefore, the total disk I/O with pivot messages is at most m^2/nk, smaller than disk I/O cost of Bermuda-EC algorithm m^2/Mk, where M stands for the size of the available memory in a single machine.

Effect of the number of buckets p: At a high level, p trades off the space used by the shared buffer with the I/O cost for writing and reading the back-up files. Bermuda-VC algorithm favors smaller settings of p in the capacity of main-memory. As p decreases, the expected number of reading and writing decreases, however the total size of the pivot messages in the shared buffer may exceed the capacity of the main-memory. For a setting of p, the expected size of the pivot messages for any bucket is $O(m/kp)$. Therefore, a visible solution for $O(m/kp) \leq M$ is $\geq O(m/kM)$. In this work, p is set as $O(m/kM)$ where m is the size of the input graph, and M is the size of the available memory in a single machine.

4. EXPERIMENTS

In this section, we present an experimental evaluation of the *MR-Baseline*, *Bermuda-EC*, and *Bermuda-VC* algorithms. We also compare Bermuda algorithms against GP (Graph Partitioning algorithm for triangle listing) [31]. The objective of our experimental evaluation is to show that the proposed Bermuda method improves both time and space complexity compared to the MR-Baseline algorithm. Moreover, compared to Bermuda-EC, Bermuda-VC is able to get better performance under the proposed message caching strategies.

All experiments are performed on a shared-nothing computer cluster of 30 nodes. Each node consists of one quad-core

	Nodes	Undirected Edges	Avg Degree	Size
Twitter	$4.2 * 10^7$	$2.4 * 10^9$	57	24GB
Yahoo	$1.9 * 10^8$	$9.0 * 10^9$	47	67GB
ClueWeb12	$9.6 * 10^8$	$8.2 * 10^{10}$	85	688GB

Table 2: Basic statistics on the datasets.

	MR Baseline	Bermuda EC and VC	Reduction Factor (RF)
Twitter	$3.0 * 10^{11}$	$1.2 * 10^{10}$	30
Yahoo	$1.4 * 10^{11}$	$1.9 * 10^{10}$	7.5
ClueWeb12	$3.0 * 10^{12\#}$	$2.6 * 10^{11}$	11.5

Table 3: The size of pivot messages generated (and shuffled) by MR-Baseline and Bermuda algorithms along with the reduction factor. (#: Indicate a counted number without actual generation).

Intel Core Duo 2.6GHZ processors, 8GB RAM, 400GB disk, and interconnected by 1Gb Internet. Each node runs Linux OS and Hadoop version 2.4.1. Each node is configured to run up to 4 map and 2 reduce tasks concurrently. The replication factor is set to 3 unless otherwise is stated.

4.1 Datasets

We use three large real-world graph datasets for our evaluation. *Twitter* is one representative social network which captures current biggest micro-blogging community. Edges represent the friendship among users [1]. *Yahoo* is one of the largest real-world web graphs with over one billion vertices [2], where edges represent the link among web pages. And *ClueWeb12* is one subset of real-world web with six billion vertices [3].

In our experiments, we consider each edge of the input to be undirected. Thus, if an edge (u,v) appears in the input, we also add edge (v, u) if it does not already exist. The graph sizes varies from $4.2 * 10^7$ of *Twitter* , $1.9 * 10^8$ of *Yahoo*, to $9.6 * 10^8$ of *ClueWeb12*, with different densities; *ClueWeb12* is the largest but also the sparest dataset. The statistics on the three datasets are presented in Table 2.

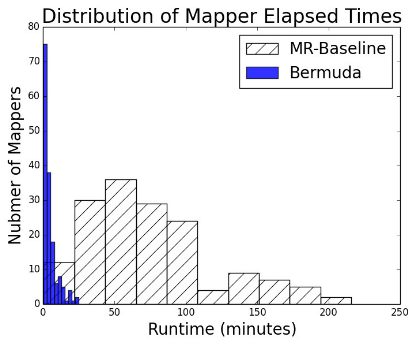
4.2 Experiment Result

4.2.1 Bermuda Technique

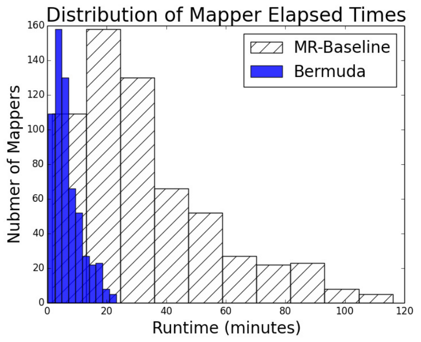
Bermuda directly reduces the size of intermediate records by removing redundancy. We experimentally verify the reduction of the pivot messages as reported in Table 3. In the case of the Twitter dataset, Bermuda's output is around 30x less than that generated by the MR-Baseline algorithm. Furthermore, in the case of ClueWeb, the size of the intermediate result generated by the MR-Baseline algorithm exceeds the available disk capability of the cluster. The reported number in Table 3 is obtained through a counter without actual record generation. The drastic difference in the size of the pivot messages has a large impact on the running time. In the case of Twitter, MR-Baseline takes more than 4 hours to generate and transform 300 billion records. Whereas, the Bermuda-EC and Bermuda-VC algorithms only generate and transform 12 billion records under the settings of $k = 20$, which takes 9 minutes on average.

Moreover, Bermuda methods handle the map-side imbalance more effectively. As discussed in Section 2.4, the size of the intermediate records generated by the MR-Baseline algorithm heavily depends on the degree distribution of the input nodes. Whereas Bermuda mitigates the effect of skewness by limiting the replication factor of the pivot messages up to k. Figure 6 shows the distribution of the mappers' elapsed times on the Twitter and Yahoo dataset, respectively.

Figure 6a illustrates the map-side imbalance problem of the MR-Baseline Algorithm as indicated by the heavy-tailed distribution of the elapsed time (the x-axis). The majority of the map tasks finish in less than 100 minutes, but there are a handful of map tasks that take more than 200 minutes. The mappers that have the longest completion time received high degree nodes to pivot on. This is because for a node of effective degree $\hat{d}$, the MR-Baseline algorithm generates $O(\hat{d}^2)$ pivot messages. Figure 6a shows a significantly more balanced workload distribution under the Bermuda algorithms. This is indicated by a smaller spread out between the fastest and slowest mappers , which is around 10 minutes. This is because for a node of effective degree $\hat{d}$, Bermuda would generate only $O(min(k, \hat{d})\hat{d})$ pivot messages. Therefore, the variance of mappers' outputs is significantly reduced. Figure 6b manifests the same behavior over the Yahoo dataset. This empirical observation is in accordance with our theoretical



(a) Twitter



(b) Yahoo

Figure 6: The distribution of mappers elapsed times for MR-Baseline and Bermuda-EC algorithms. All runs were measured on Twitter and Yahoo datasets using 158 and 600 mappers, respectively. The distribution of Bermuda-VC is similar to Bermuda-EC.

[1]http://an.kaist.ac.kr/traces/WWW2010.html
[2]http://webscope.sandbox.yahoo.com.2015
[3]http://www.lemurproject.org /clueweb12/webgraph.php/.

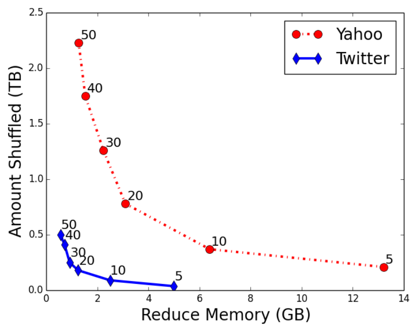


Figure 7: The total shuffle messages vs. memory for each reduce worker. All metrics were measured on Twitter and Yahoo datasets where k values range from 5 to 50.

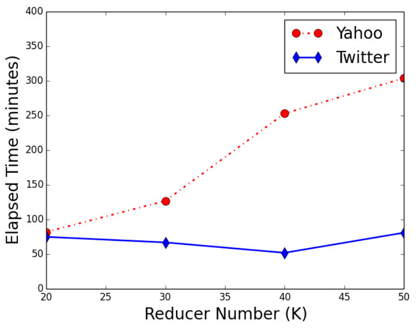


Figure 8: The running of Bermuda-EC algorithm on Twitter and Yahoo dataset. The Yahoo graph can not fit into memory when $k < 20$. The ClueWeb dataset are not presented in here because the Bermuda -EC algorithm failed on the dataset.

analysis and Theorems 1 and 3. Thus, Bermuda methods outperform the MR-Baseline Algorithm.

4.2.2 Effect of the number of reducers

In Bermuda-EC, the number of reducers k trades off between the memory used by each reducer and the used disk space. Figure 7 illustrates such trade off on the Twitter and Yahoo datasets. Initially, as k increases, the increase of the storage overhead is small while the reduction of memory is drastic. In the case of the Yahoo dataset, as k increases, the size of the core messages decreases, and can fit in the available main-memory. As k increases further, the decrease in the memory requirements gets smaller, while the increase of the disk storage grows faster. For a given graph $G(V, E)$ and a given cluster of machines, the range of k is bounded by two factors, the total disk space available and the memory available on each individual machine. In the case of Yahoo, k should be no smaller than 20, otherwise the core messages cannot fit into the available main memory.

Figure 8 illustrates the runtime of the Bermuda-EC algorithm under different settings of k over the Twitter and Yahoo datasets. In the case of Twitter, the elapsed time reduces as k initially increases. We attribute this fact to the increase of parallel computations. As k continues to increase, this benefit disappears and the total runtime slowly increases. We attribute the increase of the execution time to the following two factors: (1) The increasing size of intermediate

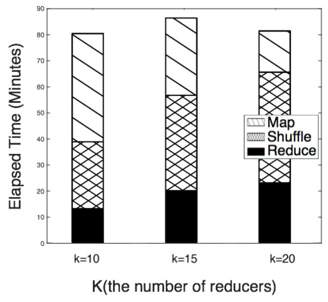


Figure 9: The running of Disk-Based Bermuda-EC algorithm on Yahoo dataset, where $k \leq 20$.

records $O(km)$, and (2) The higher variance of the map-side workload $O(2ck^2Var(x))$. As shown in Figure 8, the effect of k varies from one graph to another. In the case of the Yahoo dataset, the communication cost dominates the overall performance early, e.g., under $k = 20$. We attribute these different behaviors to the nature of the input datasets. Twitter—as one typical social network—has a lot of candidate triangles. In contrast, Yahoo is one typical hyperlink network with sparse connections and relatively fewer candidate triangles.

For execution efficiency, Bermuda-EC chooses to keep the relatively small core messages in the main memory, while allowing a sequential access to the relatively large pivot messages. For a given web-scale graph, a large setting of k is required to make the core messages fit into the available memory of an individual reducer. Unfortunately, the price is a bigger size of intermediate data $O(km)$, which leads to a serious network congestion, and even a job failure in some cases. In the case of the ClueWeb dataset, Bermuda-EC requires large number of reducers, e.g., in the order of 100s, which creates a prohibitively large size of intermediate data (in the order of 100TBs), which is not practical for most clusters.

Although the disk-based Bermuda-EC algorithm can work under smaller settings of k, its efficiency is limited because of a large amount of disk I/Os. Figure 9 presents the run time of the disk-based Bermuda-EC variation under different settings of k over the Yahoo dataset. When $k \geq 20$, the core messages can fit into memory, and its runtime is presented in Figure 8. When k equals 10, the disk-based Bermuda-EC algorithm takes less time in generating and shuffling the intermediate data, while more time is taken in the reduce phase. As expected, the runtime of the reduce step increases quickly and the benefits induced by the smaller settings of k disappear.

4.2.3 Message Sharing Management

In Figure 10, we present the empirical results of the different caching strategies on the Yahoo dataset where $k = 10$. As shown in Figure 10, the size of sharing messages grows rapidly, then overwhelms the size of the main memory available to a given reducer, which leads to a job failure. By tracking the re-use count, the *usage-based* tracking strategy is able to immediately *discard* useless pivot messages when their counter reaches zero. As a result the increase of the memory usage is slower. However, the retention of the pivot message having long re-use distance makes the discard strategy not very effective. By considering the access pattern of

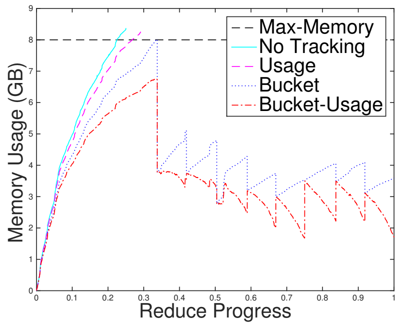


Figure 10: The cumulative size of sharing messages against progress for each reducer. All the statics are measured over the Yahoo dataset with the number of reducers $k = 10$.

	MR-Baseline	GP	Bermuda-EC	Bermuda-VC
Twitter	682	378	52	66
Yahoo	439	622	82	69
ClueWeb12	–	–	–	1528

Table 4: The run time (in minutes) of all algorithms on different real-world datasets. Bermuda algorithms outperform the MR-Baseline algorithm in all datasets and show more than 10x faster performance in twitter dataset.

the pivot messages at a smaller granularity, the *bucket-based* strategy avoids the retention of the pivot messages having too long re-use distance. As shown in Figure 10, the bucket-based strategy achieves better memory utilization. In the case of the Yahoo dataset, the size of sharing pivot message is practical for a commodity machine with the *bucket-based* strategy. The combination of the two strategies, i.e., the *bucket-usage* strategy can further reduce the size of sharing message by avoiding the memory storage of *idle* messages inside each bucket.

4.2.4 Execution Time Performance

Table 4 presents the runtime of all algorithms on the three datasets. For the Bermuda-EC algorithm, the number of reducers k is set to 40 and 20, for the Twitter and Yahoo datasets, respectively. For the Bermuda-VC algorithm, the number of reducers k is set to 40, 10, and 10, for the Twitter, Yahoo and ClueWeb datasets, respectively. The settings of reducer number k are determined by the cluster and the given datasets. Only Bermuda-VC manages to list all triangles in ClueWeb dataset, whereas MR-Baseline, GP and Bermuda-EC fail to finish due to the lack of disk space. As shown in Table 4, Bermuda methods outperform the other algorithms on the Twitter and Yahoo datasets. Bermuda -EC algorithm shows more than 5x faster performance on the Twitter dataset compared to the GP algorithm. Moreover, compared to Bermuda-EC, Bermuda-VC is able to get a better trade-off between the communication cost and the reduce-side computations. It shows a better performance over the Yahoo dataset under $k = 10$. Moreover, with a relatively small cluster, Bermuda-VC can scale up to larger datasets, e.g., ClueWeb graph dataset (688GB), while the other techniques fail to finish.

5. RELATED WORKS

Triangle listing is a basic operation of the graph analysis. Many research works have been conducted on this problem, which can be classified into three categories: in-memory algorithms, external-memory algorithms and distributed algorithms. Here, we briefly review these works.

In-Memory Algorithm. The majority of previously introduced triangle listing algorithms are the In-Memory processing approaches. Traditionally, they can be further classified as Node-Iterator [1, 4, 28] and Edge-Iterator ones [8, 15] with the respect to iterator-type. Authors [8, 15, 28]improved the performance of in-memory algorithms by adopting degree-based ordering. Matrix multiplication is used to count triangles [1]. However, all these algorithms are inapplicable to massive graphs which do not fit in memory.

External-Memory Algorithms. In order to handle the massive graph, several external-memory approaches were introduced [14, 18, 19]. Common idea of these methods is: (1) Partition the input graph to make each partition fit into main-memory, (2) Load each partition individually into main-memory and identify all its triangles, and then remove edges which participated in the identified triangle, and (3) After the whole graph is loaded into memory buffer once, the remaining edges are merged, then repeat former steps until no edges remain. These Algorithms require a lot of disk I/Os to perform the reading and writing of the edges. Authors [14, 18] improved the performance by reducing the amount of disk I/Os and exploiting multi-core parallelism. External-Memory Algorithms show great performance in time and space. However, the parallelization of external-memory algorithms is limited. External-memory approaches cannot easily scale up in terms of computing resources and parallelization degree.

Distributed Algorithms. Another promising approach to handle triangle listing on large-scale graphs is the distributed computing. Suri *et al.* [31] introduced two MapReduce adaptions of NodeIterator algorithm and the well-known Graph Partitioning (GP) algorithm to count triangles. The Graph Partitioning algorithm utilizes one universal hash partition function over nodes to distribute edges into overlapped graph partitions, then identifies triangles over all the partitions. Park *et al.* [24] further generalized Graph Partitioning algorithm into multiple rounds, significantly increasing the size of the graphs that can be handled on a given system. The authors compare their algorithm with GP algorithm [31] across various massive graphs then show that they get speedups ranging from 2 to 5. In this work, we show such large or even larger speedup (from 5 to 10) can also be obtained by reducing the size intermediate result directly via Bermuda methods. Teixeira *et al.* [32] presented Arabesque, one distributed data processing platform for implementing subgraph mining algorithms on the basis of MapReduce framework. Arabesque automates the process of exploring a very large number of subgraphs, including triangles. However, these MapReduce algorithms must generate a large amount of intermediate data that travel over the network during the shuffle operation, which degrade their performance. Arifuzzaman *et al.* [2] introduced an efficient MPI-based distributed memory parallel algorithm (Patric) on the basis of NodeIterator algorithm. The Patric algorithm introduced degree-based sorting preprocessing step for efficient set intersection operation to speed up execution. Furthermore, several distributed solutions designed for subgraph mining on large graph were also proposed [21,29]. Shao *et al.* introduced the PSgl framework to iteratively enumerate subgraph instance. Different from other parallel approaches, the PSgl framework completes relies on the graph traversal and avoids the explicit join operation. These distributed memory parallel algorithms achieve impressive performance over large-scale graph mining tasks. These methods distributed the data graph among the worker's memory, thus they are not suitable for processing large-scale graph with small clusters.

6. CONCLUSION

In this paper, we addressed the problem of listing triangles over massive graphs using the MapReduce infrastructure. We proposed the *Bermuda* technique that aims at reducing the size of the intermediate data by eliminating the communication redundancy and enabling data sharing across the graph nodes. We introduced several optimizations for better scalability and performance based on the processing order and the locality of shared intermediate records. An extensive experimental evaluation using real-world graph datasets confirms that Bermuda achieves significant savings in communication cost, execution time, and space requirements. The experimental evaluation also shows that Bermuda can bring the triangle listing over massive graphs to the realm of feasibility even under relatively small computing clusters.

As part of future work, we plan to generalize our idea beyond triangle listing to the listing of more complex subgraphs. We also plan to study additional optimizations that could possibly lead to further performance improvements such as balanced graph partitioning, and graph compression techniques.

7. REFERENCES

[1] N. Alon, R. Yuster, and U. Zwick. Finding and counting given length cycles. *Algorithmica*, 1997.

[2] S. Arifuzzaman, M. Khan, and M. Marathe. Patric: A parallel algorithm for counting triangles in massive networks. *CIKM*, 2013.

[3] N. Bao and T. Suzumura. Towards highly scalable pregel-based graph processing platform with x10. *WWW*, 2013.

[4] V. Batagelj and A. Mrvar. A subquadratic triad census algorithm for large sparse networks with small maximum degree. *Social networks*, 2001.

[5] L. Becchetti, P. Boldi, C. Castillo, and A. Gionis. Efficient semi-streaming algorithms for local triangle counting in massive graphs. *KDD*, 2008.

[6] J. Berry, B. Hendrickson, R. LaViolette, and C. Phillips. Tolerating the community detection resolution limit with edge weighting. *Physical Review E*, 2011.

[7] L. Buriol, G. Frahling, and S. Leonardi. Counting triangles in data streams. *VLDB*, 2006.

[8] N. Chiba and T. Nishizeki. Arboricity and subgraph listing algorithms. *SIAM Journal on Computing*, 1985.

[9] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 2008.

[10] J. Eckmann and E. Moses. Curvature of co-links uncovers hidden thematic layers in the world wide web. *Academy of Sciences*, 2002.

[11] J. Gonzalez, R. Xin, A. Dave, and D. Crankshaw. Graphx: Graph processing in a distributed dataflow framework. *GRADES,SIGMOD workshop*, 2014.

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

[13] W.-S. Han, S. Lee, K. Park, J.-H. Lee, M.-S. Kim, J. Kim, and H. Yu. Turbograph: a fast parallel graph engine handling billion-scale graphs in a single pc. *KDD*, 2013.

[14] X. Hu, Y. Tao, and C. Chung. I/O-Efficient algorithms on triangle listing and counting. *ACM Transactions on Database Systems*, 2014.

[15] A. Itai and M. Rodeh. Finding a minimum circuit in a graph. *SIAM*, 1978.

[16] G. Keramidas and P. Petoumenos. Cache replacement based on reuse-distance prediction. *ICCD*, 2007.

[17] S. Khuller and B. Saha. On finding dense subgraphs. *Automata*, 2009.

[18] J. Kim, W. Han, S. Lee, K. Park, and H. Yu. Opt: a new framework for overlapped and parallel triangulation in large-scale graphs. *SIGMOD*, 2014.

[19] A. Kyrola, G. Blelloch, and C. Guestrin. Graphchi: Large-scale graph computation on just a pc. *OSDI*, 2012.

[20] J. Lin. The curse of zipf and limits to parallelization: A look at the stragglers problem in mapreduce. *LSDR-IR workshop*, 2009.

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

[22] R. Milo, S. Shen-Orr, S. Itzkovitz, N. Kashtan, and D. Chklovskii. Network motifs: simple building blocks of complex networks. *Academy of Sciences*, 2002.

[23] G. W. Oehlert. A note on the delta method. *The American Statistician*, 1992.

[24] H. Park, F. Silvestri, U. Kang, and R. Pagh. MapReduce triangle enumeration with guarantees. *CIKM*, 2014.

[25] P. Petoumenos and G. Keramidas. Instruction-based reuse-distance prediction for effective cache management. 2009.

[26] S. Salihoglu and J. Widom. Gps: A graph processing system. *SSDBM*, 2013.

[27] A. D. Sarma, F. N. Afrati, S. Salihoglu, and J. D. Ullman. Upper and lower bounds on the cost of a map-reduce computation. *PVLDB*, 2013.

[28] T. Schank. Algorithmic aspects of triangle-based network analysis. *Phd in computer science*, 2007.

[29] Y. Shao, B. Cui, L. Chen, L. Ma, J. Yao, and N. Xu. Parallel subgraph listing in a large-scale graph. *SIGMOD*, 2014.

[30] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The hadoop distributed file system. *MSST*, 2010.

[31] S. Suri and S. Vassilvitskii. Counting triangles and the curse of the last reducer. *KDD*, 2011.

[32] C. H. C. Teixeira, A. J. Fonseca, M. Serafini, G. Siganos, M. J. Zaki, and A. Aboulnaga. Arabesque: a system for distributed graph mining. *SOSP*, 2015.

[33] T. White. Hadoop: The definitive guide. 2010.