Triangle Listing in Massive Networks

SHUMO CHU, Nanyang Technological University, Singapore
JAMES CHENG, The Chinese University of Hong Kong

Triangle listing is one of the fundamental algorithmic problems whose solution has numerous applications especially in the analysis of complex networks, such as the computation of clustering coefficients, transitivity, triangular connectivity, trusses, etc. Existing algorithms for triangle listing are mainly in-memory algorithms, whose performance cannot scale with the massive volume of today's fast growing networks. When the input graph cannot fit in main memory, triangle listing requires random disk accesses that can incur prohibitively huge I/O cost. Some streaming, semi-streaming, and sampling algorithms have been proposed but these are approximation algorithms. We propose an I/O-efficient algorithm for triangle listing. Our algorithm is exact and avoids random disk access. Our results show that our algorithm is scalable and outperforms the state-of-the-art in-memory and local triangle estimation algorithms.

Categories and Subject Descriptors: H.2.8 [**DATABASE MANAGEMENT**]: Database Applications—*Data mining*; G.2.2 [**DISCRETE MATHEMATICS**]: Graph Theory—*Graph algorithms*

General Terms: Algorithms, Experimentation, Performance

Additional Key Words and Phrases: Triangle Listing, Triangle Counting, Clustering Coefficients, Large Graphs, Massive Networks

ACM Reference Format:
Chu, S., and Cheng, J. 2011. Triangle Listing in Massive Networks. ACM Trans. Knowl. Discov. Data. V, N, Article A (January YYYY), 33 pages.
DOI = 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

1. INTRODUCTION

We study the problem of **triangle listing** in a simple undirected graph G, that is, *listing all triangles in* G, where a triangle is a complete subgraph of G that consists of three vertices. Our focus is to design an efficient algorithm for triangle listing when the input graph G is too large to fit in main memory and is disk resident.

Triangles are one of the fundamental types of small subgraphs most commonly used in the analysis of complex graphs/networks. In particular, a triangle is also the shortest non-trivial cycle (i.e., a cycle of length 3) and the smallest non-trivial clique (i.e., a clique of size 3). The concept of triangle is at the heart of the definitions of many important measures for network analysis, such as the clustering coefficients (of a single vertex and of the entire network) [Watts and Strogatz 1998], transitivity [Wasserman and Faust 1994; Newman et al. 2002], triangular connectivity [Batagelj and Zaveršnik

This research was done when the second author was at Nanyang Technological University, Singapore, and supported in part by the A*STAR Thematic Strategic Research Programme Grants (112 172 0013 and 102 158 0034).
Author's addresses: S. Chu, Division of Information Systems, School of Computer Engineering, Nanyang Technological University, Singapore; J. Cheng (corresponding author), Department of Computer Science and Engineering, the Chinese University of Hong Kong.

2007], etc. All these measures can be directly computed from the result of triangle listing.

The aforementioned triangle-centered measures have a large number of important applications. In addition, triangle listing also has a broad range of applications in other areas, such as the discovery of dense subgraphs [Wang et al. 2010], the computation of trusses (i.e., subgraphs of high connectivity) [Cohen 2009], spam detection [Becchetti et al. 2008; 2010], the study of motif occurrences [Milo et al. 2002], the uncovering of hidden thematic relationships in the Web [Eckmann and Moses 2002], etc. In all these applications, triangle listing plays a vital role in their computation.

Although many algorithms have been proposed for triangle listing, these existing algorithms [Itai and Rodeh 1977; 1978; Alon et al. 1997; Batagelj and Mrvar 2001; Schank and Wagner 2005; Schank 2007; Latapy 2008; Eppstein and Spiro 2009] all fall into the category of *in-memory algorithms*. The best existing in-memory algorithms require space that is asymptotically linear in the size of the input graph. However, many real-world networks have grown exceedingly large in recent years and are continuing to grow at a steady rate. For example, the Web graph has over 1 trillion webpages (by Google in 2008), most social networks (e.g., Facebook, MSN) have millions to billions of users, many citation networks (e.g., DBLP, Citeseer) have millions of publications, other networks such as phone-call networks, email networks, stock-market networks, etc., are also massively large and still growing fast.

For handling large graphs that cannot fit in main memory, a number of approximation algorithms have been proposed [Alon et al. 1999; Bar-Yossef et al. 2002; Coppersmith and Kumar 2004; Buriol et al. 2006; Becchetti et al. 2008; Tsourakakis et al. 2009; Becchetti et al. 2010]. However, all these algorithms are restricted to the approximation of *triangle counting*, i.e., estimating the number of triangles in a graph or that formed at each vertex. Algorithms for estimating the total number of triangles in a graph only are considerably accurate [Alon et al. 1999; Bar-Yossef et al. 2002; Coppersmith and Kumar 2004; Buriol et al. 2006; Tsourakakis et al. 2009], but the range of their applications is significantly more limited than that of triangle listing. Algorithms for estimating the number of triangles formed at each vertex in a graph, also called *local triangle counting*, have a wider range of applications but algorithms for local triangle counting [Becchetti et al. 2008; 2010] may not be accurate enough for certain applications. Moreover, the set of applications of triangle counting is only a small subset of that of triangle listing, as the result of triangle counting is directly obtainable from that of triangle listing.

We propose an I/O-efficient algorithm for exact triangle listing. Designing such an algorithm is difficult because triangle listing requires to access the neighbors of the neighbor of a vertex, which may appear arbitrarily in any position in the graph stored on disk. Thus, random access to the disk-resident graph is required, which incurs huge I/O cost.

Our algorithm iteratively partitions the input graph G into a set of subgraphs that can fit in main memory and processes triangle listing in each local subgraph in memory. To ensure the correctness and completeness of the final result computed iteratively from the local subgraphs, we categorize the triangles into three types. We devise an mechanism that lists all Type 1 and Type 2 triangles, and then converts the remaining Type 3 triangles into Type 1 and Type 2 by a new partition of a shrinking graph at the next iteration. To limit the total number of iterations, we show that we can remove all intra-partition edges at the end of each iteration, thus shrinking G until it becomes empty.

We propose three effective algorithms for graph partitioning for the task of triangle listing in our framework. The first algorithm sequentially scans the input graph only once to partition the graph, thus achieving high efficiency in practice. The sequential graph partitioning algorithm, however, does not have any theoretical guarantee on the number iterations required in the overall process of triangle listing. To this end, we propose another graph partitioning algorithm that requires two scans of the input graph and, by grouping neighboring vertices together based on the application of the dominating set, attains a theoretical upper bound on the total number of iterations needed. However, to apply the dominating set the algorithm requires $O(|V_G|)$ memory space, where $|V_G|$ is the number of vertices in the input graph G. To address this problem, we propose a randomized graph partitioning algorithm, which removes the memory space requirement, and with which we establish a bound on the total number of iterations with a high probability.

Many real-world networks undergo frequent updates. We propose a compact disk-based data structure for efficient update of the set of triangles when the input graph is updated. We discuss the operations for updating the data structure as well as for updating the set of triangles. The data structure is also useful in applications where the set of triangles needs to be materialized on disk.

We evaluated our algorithm on large real datasets with up to 106 million vertices and 1,877 million edges, by comparing with the state-of-the-art in-memory algorithm [Latapy 2008] and the approximation algorithm for local triangle counting [Becchetti et al. 2008]. Our algorithm achieves comparable performance with the in-memory algorithm when the graph can fit in main memory. For large graphs that cannot fit in main memory, our results show that our algorithm is superior to the approximation algorithm [Becchetti et al. 2008]: at comparable running time and memory usage, the approximation algorithm records a high error rate while ours returns the exact result. The results also show that our data structure supports efficient update of the set of triangles in a dynamic network.

Paper Organization. Section 2 gives the notations and problem definition. Section 3 describes an in-memory algorithm. Section 4 discusses the I/O-efficient algorithm for triangle listing and Section 5 proposes the three graph partitioning algorithms. Section 6 presents an I/O-efficient algorithm for triangle listing in graphs with extremely high degree vertices. Section 7 discusses update in dynamic networks. Section 8 presents two applications of our algorithm. Section 9 reports the experimental results. Section 10 gives the related work. Section 11 concludes the paper.

2. NOTATIONS AND PROBLEM DEFINITION

Let $G = (V_G, E_G)$ be a simple undirected graph, where V_G is the set of vertices and E_G is the set of edges. We define the set of *adjacent vertices* (or *neighbors*) of a vertex v in G as $adj_G(v) = \{u : (u, v) \in E_G\}$, and the *degree* of v in G as $deg_G(v) = |adj_G(v)|$.

We assume that the graph is stored in its adjacency list representation (whether in memory or on disk), which is a common data format used for graph storage. Each vertex in the graph is assigned a unique vertex ID. Given any two vertices u and v, we use $u < v$ or equivalently $v > u$ to denote that u is ordered before v according to the order of their vertex IDs. In the adjacency list representation, the vertices are ordered in the ascending order of their vertex IDs.

Given three distinct vertices, $u, v, w \in V_G$, we say that u, v and w form a *triangle* in G if $(u, v), (u, w), (v, w) \in E_G$. We use $\triangle_{uvw}$ to denote the triangle formed by the vertices u, v and w.

The set of triangles that consist of a vertex v, denoted by $\triangle(v)$, is defined as

$$\triangle(v) = \{\triangle_{uvw} : u, w \in adj_G(v), (u, w) \in E_G\}. \tag{1}$$

The *triangle number* of v, denoted by $N_\triangle(v)$, is defined as $N_\triangle(v) = |\triangle(v)|$.

Let $\triangle(G)$ be the set of all triangles in G. Then, $\triangle(G)$ is given by

$$\triangle(G) = \bigcup_{v \in V_G} \triangle(v). \tag{2}$$

The number of triangles in G, denoted by $N_\triangle(G)$, is defined as $N_\triangle(G) = |\triangle(G)|$, which is also given as follows

$$N_\triangle(G) = \frac{1}{3} \sum_{v \in V_G} N_\triangle(v). \tag{3}$$

Equation 3 holds because every triangle $\triangle_{uvw}$ is counted three times, once for each of the three vertices u, v and w.

Given a vertex $v \in V_G$, we say that u, v and w form an *open triangle* centered at v if $u, w \in adj_G(v)$. An open triangle is considered as a potential triangle. A triangle $\triangle_{uvw}$ may be regarded as a *closed triangle* and by definition, $\triangle_{uvw}$ contains three open triangles, centered at u, v, and w, respectively.

The number of open triangles centered at v, denoted by $N_\vee(v)$, is defined as

$$N_\vee(v) = \frac{1}{2} deg_G(v)(deg_G(v) - 1). \tag{4}$$

Intuitively, $N_\vee(v)$ defines the maximum number of triangles that can be potentially formed from v.

The following example illustrates the concepts.

Example 2.1. Let G be the graph given in Figure 1. Consider the vertices b and e, we have $\triangle(b) = \{\triangle_{abc}, \triangle_{bcg}, \triangle_{bgi}\}$ and $\triangle(e) = \{\triangle_{dej}, \triangle_{efh}\}$. Thus, $N_\triangle(b) = 3$ and $N_\triangle(e) = 2$. By Equation 4, $N_\vee(b) = 6$ and $N_\vee(e) = 6$ since $deg_G(b) = 4$ and $deg_G(e) = 4$. From Figure 1, we can also easily find that $\triangle(G) = \{\triangle_{abc}, \triangle_{bcg}, \triangle_{bgi}, \triangle_{dej}, \triangle_{efh}, \triangle_{jkl}\}$ and $N_\triangle(G) = |\triangle(G)| = 6$. $\square$

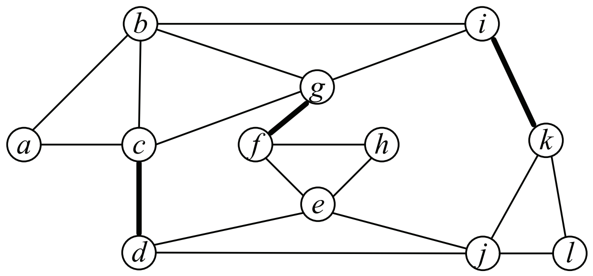

Fig. 1. A Graph G

Problem Definition. This paper studies the problem of **triangle listing** defined as follows. Given a graph $G = (V_G, E_G)$, output $\triangle(G)$. In particular, we design I/O-efficient algorithms when G cannot fit in main memory, i.e., $(|V_G| + |E_G|) > M$, where M is the size of the available main memory.

For the complexity analysis of I/O-efficient algorithms, we use the standard I/O model [Aggarwal and Vitter 1988] with the following parameters: M is the available main memory size and B is the disk block size, where $1 \ll B \le M/2$.

Table I. Frequently Used Notations

Notation	Description
$G = (V_G, E_G)$	A simple undirected graph
$adj_G(v)$	The set of adjacent vertices of v in G
$deg_G(v)$	The degree of v in G
$\triangle_{uvw}$	A triangle formed by u, v and w
$\triangle(v)$	The set of triangles that contains the vertex v (Eq. 1)
$N_\triangle(v)$	The triangle number of v, $N_\triangle(v) = \lvert\triangle(v)\rvert$
$\triangle(e)$	The set of triangles that contains the edge e (Section 7.2)
$\triangle(G)$	The set of all triangles in G (Eq. 2)
$N_\triangle(G)$	The number of triangles in G (Eq. 3)
$N_\vee(v)$	The number of open triangles centered at v (Eq. 4)
M	The available main memory size
B	The disk block size
$scan(N)$	$\Theta(N/B)$
$sort(N)$	$\Theta(\frac{N}{B}\log_{M/B}\frac{N}{B})$

We also use the following standard I/O complexity notations: $scan(N)$ I/Os $=\Theta(N/B)$ I/Os and $sort(N)$ I/Os $= \Theta(\frac{N}{B}\log_{M/B}\frac{N}{B})$ I/Os, where N is the amount of data being read or written from/to disk.

Table I gives the frequently-used notations in the paper.

3. IN-MEMORY TRIANGLE LISTING

In this section, we first present an in-memory algorithm for triangle listing and use it to explain the difficulties of triangle listing in the case when main memory is insufficient to hold the input graph.

We sketch the algorithm in Algorithm 1. The algorithm intersects the adjacency list of each vertex u with the adjacency list of each neighbor v of u. Clearly, each vertex w as the result of the intersection is a neighbor of both u and v, and as u and v are also neighbors, we obtain a triangle $\triangle_{uvw}$.

A naive algorithm for triangle listing processes every neighbor $v \in adj_G(u)$, and intersects the entire $adj_G(u)$ and $adj_G(v)$. This involves much redundant processing and also outputs duplicate triangles. In Algorithm 1, we only process a neighbor v that is ordered after u (Line 3), because if v is ordered before u, i.e., $v < u$, then v has been processed before u and hence the triangle $\triangle_{vuw}$ must have been already listed (note that $\triangle_{vuw} = \triangle_{uvw}$). For the intersection between $adj_G(u)$ and $adj_G(v)$ (Line 4), we also skip those vertices that are ordered before v (and hence also u) in $adj_G(u)$ and $adj_G(v)$. The above process is similar to the state-of-the-art in-memory algorithm for triangle listing [Latapy 2008], except that their work assumes that the adjacent list of each vertex is sorted in the non-increasing order of the vertex degree, which requires costly pre-processing for the sorting for a large graph. Note that it is not common to store a graph with adjacency lists sorted by the vertex degree, since update to such a storage data format is expensive.

Algorithm 1 is efficient when the input graph can fit in main memory. However, when the input graph G cannot fit in main memory, the algorithm requires huge I/O cost due to random disk access. Most existing in-memory algorithms [Itai and Rodeh 1977; 1978; Alon et al. 1997; Batagelj and Mrvar 2001; Schank 2007; Latapy 2008; Eppstein and Spiro 2009] require random access to each $adj_G(v)$ for each $v \in adj_G(u)$ (as in Line 4 of Algorithm 1 for the intersection). Note that each $adj_G(u)$ in Algorithm 1 is read sequentially as we read G, but $adj_G(v)$ can be in an arbitrary position on disk where G is stored. Other existing in-memory algorithms [Schank and Wagner 2005] use an additional array for each vertex in G and the total size of these arrays is in the

Algorithm 1 *In-Memory Triangle Listing*

Input: A graph $G = (V_G, E_G)$
Output: $\triangle(G)$

```
1.  $\triangle(G) \leftarrow \emptyset$;
2.  for each $u \in V_G$ do
3.      for each $v \in adj_G(u)$, where $v > u$, do
4.          for each $w \in (adj_G(u) \cap adj_G(v))$, where $w > v$, do
5.              $\triangle(G) \leftarrow (\triangle(G) \cup \{\triangle_{uvw}\})$;
            end
        end
    end
6.  return $\triangle(G)$;
```

order of the size of the input graph; thus these arrays need to be stored on disk and random access is again inevitable.

When G cannot fit in main memory, Algorithm 1 requires $O(|E_G| \cdot scan(d_{max}))$ I/Os in the worst case, where d_{max} is the maximum vertex degree in G, since we need to randomly access $adj_G(v)$ for each edge $(u, v) \in E_G$ and $deg_G(v) = O(d_{max})$. This I/O cost can be prohibitively large especially when G is large.

4. I/O-EFFICIENT TRIANGLE LISTING

In this section, we present an I/O-efficient algorithm for triangle listing in a large graph when main memory is not sufficient to hold the entire graph. We first sketch the framework of our algorithm and then present the details of the algorithm.

4.1. Algorithm Framework

When the input graph G cannot fit in main memory, we can only load a portion (i.e., a subgraph) of G that can fit in main memory each time. Thus, our algorithm iteratively performs triangle listing in a subgraph of G that fits in main memory. We outline the framework of our algorithm as follows.

- Each iteration:
  - Partition G into a set of subgraphs, $\mathcal{P} = \{G_1, \ldots, G_i, \ldots, G_p\}$, such that each G_i can fit in main memory;
  - Load each G_i in main memory and perform triangle listing in G_i;
  - Remove from G those edges of G_i that can no longer contribute to triangle listing.
- Repeat the above iteration until G becomes empty.

The main idea of our algorithm is to iteratively partition the graph and perform triangle listing in each local subgraph G_i separately, as to avoid random access to arbitrary vertices (and their adjacency lists) in the graph.

The concept is simple but there are a number of technical challenges: (1) ensuring the correctness and completeness of the final result obtained from the iterative local computations; (2) an effective and efficient partitioning algorithm for triangle listing; and (3) bounding the overall I/O complexity of the algorithm (i.e., the I/O complexity at each step and the number of iterations). We discuss the above three issues in each of the following subsections.

4.2. Correctness and Global-Completeness of Local Triangle Listing

We first propose an algorithm that ensures the correctness of triangle listing in each local subgraph of G as well as the completeness of the global result obtained from all local computations.

The design of our algorithm is based on the following Lemma.

LEMMA 4.1. *Let $\mathcal{P} = \{G_1, \ldots, G_i, \ldots, G_p\}$ be a partition of a graph $G = (V_G, E_G)$, where $\cup_{1 \le i \le p} V_{G_i} = V_G$ and $V_{G_i} \cap V_{G_j} = \emptyset$ for $1 \le i < j \le p$. Then, $\triangle(G) = \triangle 1 \cup \triangle 2 \cup \triangle 3$, where $\triangle 1$, $\triangle 2$, and $\triangle 3$ are disjoint sets defined as follows.*

- $\triangle 1 = \cup_{1 \le i \le p} \{\triangle_{uvw} : u, v, w \in V_{G_i}\}$.
- $\triangle 2 = \cup_{1 \le i, j \le p \, \wedge \, i \ne j} \{\triangle_{uvw} : u, v \in V_{G_i}, w \in V_{G_j}\}$.
- $\triangle 3 = \cup_{1 \le i < j < k \le p} \{\triangle_{uvw} : u \in V_{G_i}, v \in V_{G_j}, w \in V_{G_k}\}$.

PROOF. First, $(\triangle 1 \cup \triangle 2 \cup \triangle 3) \subseteq \triangle(G)$, since the elements in $\triangle 1$, $\triangle 2$, and $\triangle 3$ are triangles in G.

Next we show $\triangle(G) \subseteq (\triangle 1 \cup \triangle 2 \cup \triangle 3)$. For any triangle $\triangle_{uvw} \in \triangle(G)$, there are only three cases where u, v, and w can be located in the subgraphs in $\mathcal{P}$:

(1) u, v, and w are all in the same subgraph G_i.
(2) Two of them are in the same subgraph G_i while the other in another different subgraph G_j; that is, without the loss of generality, we may assume that $u, v \in V_{G_i}, w \in V_{G_j}, i \ne j$.
(3) u, v, and w are in three different subgraphs G_i, G_j, and G_k; that is, without the loss of generality, we may assume that $u \in V_{G_i}, v \in V_{G_j}, w \in V_{G_k}, i < j < k$.

The above three cases correspond to the three types $\triangle 1$, $\triangle 2$, and $\triangle 3$, and thus $\triangle(G) \subseteq (\triangle 1 \cup \triangle 2 \cup \triangle 3)$. □

The triangles in $\triangle 1$, $\triangle 2$, and $\triangle 3$ are also called *Type 1*, *Type 2*, and *Type 3* triangles, respectively. The following example illustrates the concept of the three types of triangles.

Example 4.2. Figure 2 shows a partition, $\mathcal{P} = \{G_1, G_2, G_3\}$, of the graph G shown in Figure 1. In the figure, $\triangle_{abc}$, $\triangle_{efh}$ and $\triangle_{jkl}$ are Type 1 triangles because all the three vertices in each of these three triangles are in the same subgraph. We only have one Type 2 triangle, $\triangle_{bcg}$, because its vertices are in two subgraphs, G_1 and G_2, in $\mathcal{P}$. We have two Type 3 triangles, $\triangle_{bgi}$ and $\triangle_{dej}$, because all the three vertices of each of the two triangles are in three different subgraphs in $\mathcal{P}$. □

According to Lemma 4.1, a triangle $\triangle_{uvw}$ can be listed by searching a subgraph G_i alone if and only if u, v and w are all in G_i (i.e., Type 1 triangles). However, the number of Type 1 triangles may be quite limited. More critically, we cannot remove any edge (and hence any vertex) of G_i from G even after we list all Type 1 triangles, because an edge (u, v) in $\triangle_{uvw}$ may form another triangle $\triangle_{uvx}$ with a vertex x in another subgraph G_j.

To enable the removal of edges after all triangles containing these edges are listed, and at same time to ensure the completeness of the global result, we introduce the notion of *extended subgraph* as follows.

Definition 4.3 (*Extended Subgraph*). Let $H = (V_H, E_H)$ be a subgraph of $G = (V_G, E_G)$. An *extended subgraph* of H in G, denoted by H^+, is a *directed* graph defined as $H^+ = (V_{H^+}, E_{H^+})$, where $V_{H^+} = V_H \cup \{v : u \in V_H, v \in V_G \backslash V_H, (u, v) \in E_G\}$ and $E_{H^+} = \{(u, v) : (u, v) \in E_G, u \in V_H\}$.

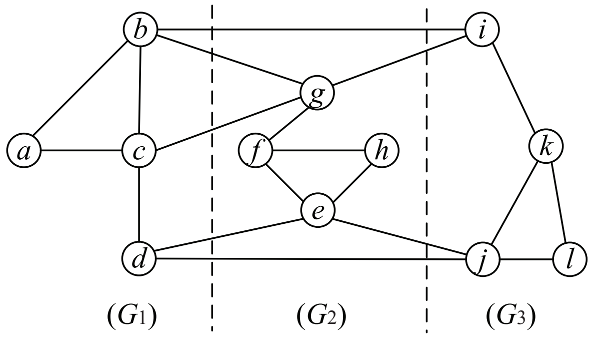

Fig. 2. A Partition of the Graph G in Figure 1: $\mathcal{P} = \{G_1, G_2, G_3\}$

Intuitively, an extended subgraph of H is a subgraph obtained by adding (to H) those directed edges from the vertices in H to those vertices not in H. Note that $\forall v \in V_{H^+} \backslash V_H, adj_{H^+}(v) = \emptyset$. For now, we assume that if $V_H = \{u, v\}$, for any $(u, v) \in E$, then the corresponding H^+ fits in main memory, which is a realistic assumption with the available memory size of a commodity PC today. We remove this assumption in Section 6.

We give an example of an extended subgraph as follows.

Example 4.4. Figure 3 depicts the extended subgraphs of G_1, G_2, and G_3 in Figure 2, i.e., G_1^+, G_2^+, and G_3^+. The shaded vertices are the vertices in each G_i, while the directed edges (i.e., those with an arrow) show the extension to vertices outside each G_i. All the other edges that are without an arrow are considered as the bi-directional edges in G_1^+, G_2^+, and G_3^+. □

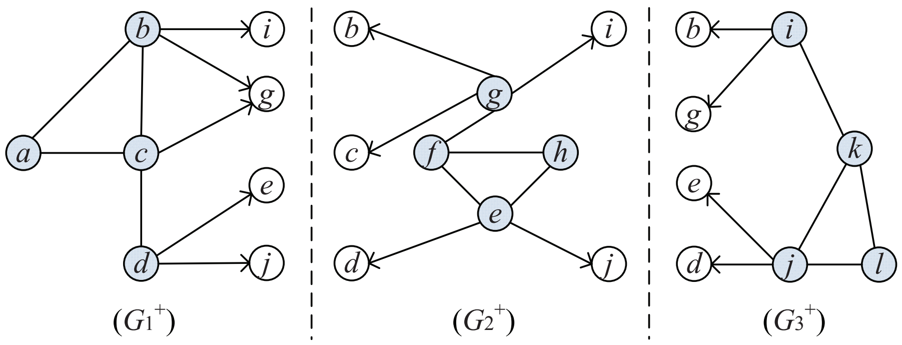

Fig. 3. The Extended Subgraphs of G_1, G_2, G_3 in Figure 2

Based on Definition 4.3, we have the following lemma for triangle listing in an extended subgraph.

LEMMA 4.5. *Let H^+ be an extended subgraph of a subgraph H of G. Then:*

- *Let $\triangle 1(H^+) = \{\triangle_{uvw} : \triangle_{uvw} \in \triangle 1, \quad u, v, w \in V_H\}$. Then, $\forall \triangle_{uvw} \in \triangle 1(H^+)$, $\triangle_{uvw}$ can be listed by searching H^+ alone.*

Algorithm 2 *I/O-Efficient Triangle Listing*

```
Input: A graph $G = (V_G, E_G)$
Output: $\triangle(G)$
1.  while($G$ is not empty)
2.      Partition $G$ into $\mathcal{P} = \{G_1, \ldots, G_i, \ldots, G_p\}$;
3.      for each extended subgraph $G_i^+$ of $G_i \in \mathcal{P}$ do
4.          List all triangles in $\triangle 1(G_i^+)$ and $\triangle 2(G_i^+)$ (by Algorithm 3);
5.          Remove all edges in $G_i$ from $G$;
        end
    end
```

- *Let* $\triangle 2(H^+) = \{\triangle_{uvw} : \triangle_{uvw} \in \triangle 2, \ \ u, v \in V_H\}$*. Then,* $\forall \triangle_{uvw} \in \triangle 2(H^+)$*,* $\triangle_{uvw}$ *can be listed by searching* H^+ *alone.*

In addition, for any edge $(u, v) \in E_H$*,* (u, v) *does not exist in any triangle in* $\triangle(G) \backslash (\triangle 1(H^+) \cup \triangle 2(H^+))$.

PROOF. First, $\forall \triangle_{uvw} \in \triangle 1(H^+)$, $\triangle_{uvw}$ can be listed by searching H^+ alone because all the three edges (u, v) (u, w), and (v, w) of $\triangle_{uvw}$ are in H and hence also in H^+. Second, $\forall \triangle_{uvw} \in \triangle 2(H^+)$, $\triangle_{uvw}$ can be listed by searching H^+ alone because $u, v \in V_H$, $w \in adj_{H^+}(u)$ and $w \in adj_{H^+}(v)$, which means that $\triangle_{uvw}$ can be found by intersecting $adj_{H^+}(u)$ and $adj_{H^+}(v)$.

Finally, for any edge $(u, v) \in E_H$, (u, v) does not exist in any triangle in $\triangle(G) \backslash (\triangle 1(H^+) \cup \triangle 2(H^+))$ because any triangle containing (u, v) must be in either $\triangle 1(H^+)$ or $\triangle 2(H^+)$. □

Lemma 4.5 implies that we can list all Type 1 and Type 2 triangles from the extended subgraph G_i^+ of each subgraph G_i in the partition $\mathcal{P}$ of G. More importantly, after listing the two types of triangles in each G_i^+, we can remove all edges in G_i (i.e., those bi-directed edges in G_i^+), since all triangles containing these edges have been already listed.

Listing all Type 1 and Type 2 triangles alone is not enough since we still miss all Type 3 triangles. We devise an efficient algorithm that iteratively converts Type 3 triangles into Type 1 and Type 2 triangles so that all triangles can be listed, while at the same time reducing the size of the graph to reduce the I/O cost as well as the search space for subsequent iterations of triangle listing. We outline our algorithm in Algorithm 2.

Algorithm 2 is essentially an iterative computation of $\triangle 1(G_i^+)$ and $\triangle 2(G_i^+)$ from the extended subgraph G_i^+ of each $G_i \in \mathcal{P}$, as defined in Lemma 4.5, where $\mathcal{P}$ is the new partition of the new graph G at each iteration (note that some edges of G are deleted at the end of each iteration, i.e., Step 5 of Algorithm 2). Note that Algorithm 2 deletes the original graph G, but we can first make a copy of G on disk and disk copy is relatively much cheaper compared with triangle listing. For the removal of edges in Step 5 of Algorithm 2, we simply remove all vertices in V_{G_i} from the adjacency list of each vertex in G_i and write the new adjacency lists back to disk.

At the end of each iteration, we remove all edges in each G_i in the current partition $\mathcal{P}$ and obtain a shrunk new graph. Then, at the beginning of the next iteration, we re-partition the new shrunk graph. The new partition $\mathcal{P}$ defines new sets of $\triangle 1(G_i^+)$ and $\triangle 2(G_i^+)$ for the extended subgraph G_i^+ of each $G_i \in \mathcal{P}$. Thus, Algorithm 2 iteratively converts the old set of Type 3 triangles at the previous iteration into Type 1 and Type

Algorithm 3 *Triangle Listing in Extended Subgraph*

Input: An extended subgraph $H^+ = (V_{H^+}, E_{H^+})$
Output: $\triangle 1(H^+)$ and $\triangle 2(H^+)$

```
1.  for each $u \in V_H$ do
2.      for each $v \in adj_H(u)$, where $v > u$, do
3.          for each $w \in (adj_{H^+}(u) \cap adj_{H^+}(v))$ do
4.              if($w > v$ or $w \notin V_H$)
5.                  List $\triangle_{uvw}$;
                end
            end
        end
    end
```

2 triangles with respect to the new partition at the current iteration. This process continues until all edges in G are removed.

Another purpose of graph partition is to make sure that each subgraph in $\mathcal{P}$ is small enough to fit in main memory, so as to avoid random disk access for triangle listing. Meanwhile, we also want to take full utilization of the available memory and hence each $G_i \in \mathcal{P}$ should be as big as possible under the condition that $(|V_{G_i}| + |E_{G_i}|) \leq M$. Thus, if the shrinking graph G becomes small enough to fit in main memory at any iteration, Step 2 of Algorithm 2 computes a partition consisting of only one subgraph, i.e., $\mathcal{P} = \{G\}$, after which the algorithm terminates since all edges in G will be removed at the end of the iteration.

Step 4 of Algorithm 2 invokes Algorithm 3 to compute $\triangle 1(G_i^+)$ and $\triangle 2(G_i^+)$ from G_i^+, i.e., H^+ in Algorithm 3. The extended subgraph G_i^+ can be easily obtained along with the computation of the partition $\mathcal{P}$, which we discuss in Section 5.

Algorithm 3 is an in-memory algorithm similar to Algorithm 1. The only difference is that the extended graph H^+ contains two sets of vertices, V_H and $V_{H^+} \backslash V_H$. Algorithm 3 only intersects the adjacency lists of those vertices in V_H. Let w be a vertex found in $(adj_{H^+}(u) \cap adj_{H^+}(v))$, where $u, v \in V_H$. If $w \in V_H$, we also require $w > v$ (and hence also $w > u$) in order to avoid duplicate listing of the triangle $\triangle_{uvw}$. If $w \notin V_H$, then we simply list $\triangle_{uvw}$ since $\triangle_{uvw}$ cannot be listed elsewhere.

Finally, although in many applications the output of our algorithm is pipelined as the input of another algorithm, there are also applications where the set of triangles needs to be materialized on disk. In the worst case, the size of the output for any general graph is $O(|V_G|^3)$, that is, in the case of a complete graph. More precisely, the worst case output size of a graph is given by $O(\frac{1}{3}\sum_{v \in V_G} N_{\vee}(v))$, which depends on the degree of a vertex (see Equation 4). Although the average case output size may be much smaller, it can still be considerably large and should be stored on disk. We present a compact data structure for storing the set of triangles in Section 7.1.

We now prove the correctness and completeness of Algorithm 2.

THEOREM 4.6. *Given a graph $G = (V_G, E_G)$, Algorithm 2 lists all triangles in G and no false or duplicate triangle is listed.*

PROOF. Lemma 4.5 ensures that (1) all triangles containing a removed edge are listed, (2) all edges of any triangle not yet listed are still in the current graph G, and (3) the already listed triangles will not be listed again at any future iterations because at least one of their edges has been removed from G. By (1) and (2), all triangles in G are listed because G becomes empty when Algorithm 2 terminates. Since Algorithm 3 does not list any duplicate triangle due to the enforced vertex ordering, by (3) Algorithm 2 does not list any duplicate triangle. Finally, since all triangles listed by Algorithm 3 are real triangles in G, Algorithm 2 does not list any false triangle. □

The overall complexity of triangle listing by Algorithm 2, however, also depends on the graph partitioning algorithm being used. Therefore, we give the overall complexity analysis of Algorithm 2 in Section 5.4 after the discussion of the graph partitioning algorithms.

5. GRAPH PARTITIONING ALGORITHMS FOR TRIANGLE LISTING

The objectives of graph partitioning for the task of triangle listing are: (1) each subgraph in the partition should fill the available memory as much as possible; and (2) each subgraph should contain as many intra-partition edges (i.e., edges within the same subgraph) as possible. The first objective is to fully utilize memory, while the second objective is to remove as many edges as possible at each iteration of Algorithm 2 in order to reduce the search space at each iteration and the number of total iterations.

Graph partitioning that fulfills the above two objectives, however, is known to be APX-hard [Andreev and Racke 2004] when the number of subgraphs in the partition is more than 2. There have been a number of approximation algorithms proposed [Kernigham and Lin 1970; Fiduccia and Mattheyses 1982; Karypis and Kumar 1999; Feige and Krauthgamer 2000; Feige et al. 2000; Abou-Rjeili and Karypis 2006], but they are in-memory algorithms that are not suitable for triangle listing in massive networks that cannot fit in memory. Other graph preprocessing techniques may be applied to improve the quality of graph partitioning, and hence the efficiency of triangle listing. For example, by discovering vertices that share many common neighbors through a frequent itemset mining process on the adjacency lists [Buehrer and Chellapilla 2008], and then grouping them into the same subgraphs in the partition. Though useful for many other graph computations, many of these algorithms are either in-memory algorithms or too costly as a preprocessing step, which can considerably increase the overall cost of triangle listing.

For the task of triangle listing in a large graph that cannot fit in memory, we need an efficient algorithm that partitions the graph with limited memory consumption. We propose three efficient graph partitioning algorithms, two of them are streaming algorithms that require only one scan of the input graph, while the other scans the graph only twice. All three algorithms have linear CPU time complexity. Moreover, all the three partitioning algorithms output the extended subgraphs required in the triangle listing algorithm.

5.1. Sequential Graph Partitioning

Sequential graph partitioning is a simple, efficient streaming algorithm, which works as follows: we sequentially read the input graph G from disk, whenever the available memory is filled up, the portion of G being read in memory forms a subgraph in the partition. Note that this subgraph is actually an extended subgraph, because each vertex v being read is associated with its adjacency list $adj_G(v)$.

After we scan G once, we obtain a partition with approximately $(|V_G| + |E_G|)/M$ subgraphs in it. Since the algorithm scans G only once, it requires only $O(scan(|V_G| + |E_G|))$ I/Os and $O(|V_G| + |E_G|)$ CPU time. Furthermore, since the subgraphs in the partition are obtained sequentially one after another as we read G, it allows pipelining such that we can process triangle listing in each subgraph as soon as it is obtained, rather than starting triangle listing until the entire partitioning process finishes.

Sequential graph partitioning is effective when the input graph exhibits high locality, i.e., vertices are naturally clustered according to the sequential order by which the graph is stored. For example, in a road network, proximate vertices are assigned

Algorithm 4 *DominatingSet*

Input: A graph $G = (V_G, E_G)$
Output: A dominating set, D, of G

```
1.  $D \leftarrow \emptyset$;
2.  Create a bit array $A$ of size $|V_G|$ and set each bit in $A$ to $0$;
3.  for each $v \in V_G$, where $A[v] = 0$, do
4.      $D \leftarrow D \cup \{v\}$;
5.      Set $A[v]$ and $A[u]$, for all $u \in adj_G(v)$, to $1$;
    end
6.  return $D$;
```

Algorithm 5 *Dominating-Set-based Graph Partitioning*

Input: A graph $G = (V_G, E_G)$
Output: A partition of G, $\mathcal{P} = \{G_1, \ldots, G_i, \ldots, G_p\}$

```
1.  Compute a dominating set $D$ of $G$ by Algorithm 4;
2.  Divide $D$ into $p$ disjoint subsets of roughly the same size;
3.  Create $p$ subgraphs for $\mathcal{P}$ out of the $p$ subsets of $D$;
4.  for each $v \in V_G$, where $v \notin D$, do
5.      Add $v$ and $adj_G(v)$ to the smallest subgraph in $\mathcal{P}$ that has
            at least $(deg_{\mathcal{P}}(v)/p)$ neighbors of $v$;
    end
6.  return $\mathcal{P}$;
```

consecutive vertex IDs and they are stored sequentially in nearby positions in the adjacency list graph representation. In social network graphs, local communities may also be stored together.

5.2. Dominating-Set-based Graph Partitioning

Sequential graph partitioning may be efficient in practice but it gives no theoretical guarantee on the number of iterations Algorithm 2 may take. To this end, we propose another graph partitioning algorithm based on the concept of *dominating set*. The partitioning algorithm takes two scans of the input graph, one for the computation of the dominating set and one for graph partitioning.

A dominating set of a graph G is a subset of vertices $D \subseteq V_G$ such that every vertex in G is either in D or a neighbor of some vertex in D. Computing the minimum dominating set is known to be NP-hard. However, for our purpose of graph partitioning, we do not require a minimum dominating set. Thus, we devise an efficient one-pass algorithm to compute a dominating set for G as shown in Algorithm 4.

In Algorithm 4, we first initialize a bit array A of size $|V_G|$ and set all bits in A to 0. Then, we read G from disk sequentially and for each vertex v (together with $adj_G(v)$) read, we add v to D only if $A[v] = 0$. If v is added to D, then we also set v and all v's neighbors to 1 in A. Thus, all vertices in $V_G \backslash D$ are neighbors of some vertex in D.

We then use D to compute a partition of G, as outlined in Algorithm 5. For triangle listing, we want the vertices in the same subgraph in the partition to be highly connected with each other. We divide D into $p = \frac{|E_G|}{M}$ subsets and create p initial subgraphs in $\mathcal{P}$ (if $|D| < p$, we can simply randomly select $(p - |D|)$ extra vertices from G and add them to D). Then, we use the dominating vertices in each subset as seeds to grow each of the p subgraphs by attracting their neighbors. Again, we read G sequentially from disk. For each vertex v (together with $adj_G(v)$) read, let $deg_{\mathcal{P}}(v)$ be the *current* total number of neighbors of v in all the subgraphs in the *current* partition $\mathcal{P}$, which can be easily obtained by scanning $adj_G(v)$ and checking which subgraph in $\mathcal{P}$ each neighbor of v belongs to. We choose the subgraph that has at least $(deg_{\mathcal{P}}(v)/p)$ neighbors of v currently, and add v to that subgraph. If there are more than one such subgraph, we add v to the subgraph with the smallest size so far. Upon adding v, we also add $adj_G(v)$ to the subgraph, so that the resultant subgraph is an extended subgraph ready for triangle listing in Algorithm 2.

Whenever the size of a subgraph becomes greater than B (i.e., the block size), we write a block of the subgraph to disk. Thus, we need extra I/Os to first write all subgraphs in $\mathcal{P}$ to disk and then read each subgraph in $\mathcal{P}$ into memory for triangle listing in Algorithm 2. However, the asymptotic I/O complexity of Algorithm 5 is still $O(scan(|V_G| + |E_G|))$. The CPU time complexity is $O(|V_G| + |E_G|)$ since we only need to scan $adj_G(v)$ for each v. But to compute $deg_{\mathcal{P}}(v)$ efficiently we need a look-up table to keep which subgraph in $\mathcal{P}$ a vertex in V_G belongs to. The look-up table requires $(|V_G|\log_2 p)$ bits, which is a problem if $(|V_G|\log_2 p) > M$.

Dominating-set-based graph partitioning has two advantages for the task of triangle listing. First, the method groups neighborhood vertices together, which leads to a larger number of Type 1 and Type 2 triangles to be listed in each local subgraph in the partition and hence also a larger number of edges to be deleted at the end of each iteration. Second, the method gives a guaranteed lower bound on the number of intra-partition edges, i.e., edges that can be removed at the end of each iteration of Algorithm 2, as we prove in the following lemma.

LEMMA 5.1. *Let $\mathcal{P} = \{G_1, \ldots, G_i, \ldots, G_p\}$ be a partition of G computed by Algorithm 5. Then, the number of intra-partition edges of $\mathcal{P}$ (i.e., edges that are incident on vertices within the same subgraph in $\mathcal{P}$) is at least $\frac{|E_G|}{p}$.*

PROOF. Let $\mathcal{P}_{t(v)}$ be the current partition $\mathcal{P}$ at the time when a vertex v is added to $\mathcal{P}$ at Step 5 of Algorithm 5. For each vertex v, v is added to the smallest subgraph in $\mathcal{P}_{t(v)}$ that has at least $((deg_{\mathcal{P}_{t(v)}}(v))/p)$ neighbors of v. First, there must exist such a subgraph in $\mathcal{P}_{t(v)}$ when v is being added, because v is the neighbor of at least one vertex in D. Thus, the total number of intra-partition edges is at least $\sum_{v \in V_G \setminus D}((deg_{\mathcal{P}_{t(v)}}(v))/p)$. We have $\sum_{v \in V_G \setminus D}(deg_{\mathcal{P}_{t(v)}}(v)) = |E_G|$ because each edge is counted once by one of its end vertices and no edge exists between any two vertices in D according to Algorithm 4. The result thus follows. □

The significance of Lemma 5.1 is further shown when we apply it to obtain an upper bound on the total number of iterations required in Algorithm 2 in Section 5.4.

5.3. Randomized Graph Partitioning

Dominating-set-based graph partitioning has a theoretical guarantee on the number of iterations performed in Algorithm 2, but it requires $O(|V_G|)$ memory space. For a very large graph, it is possible that $|V_G| > M$. To address this problem, we devise another graph partitioning algorithm that not only has a theoretical guarantee on the number of iterations required in Algorithm 2, but also does not have the memory space problem.

The algorithm is very simple and efficient. It is a streaming algorithm that scans the input graph only once, as shown in Algorithm 6. The algorithm uniformly at random maps each vertex $v \in V_G$ to one of the $p = \frac{|E_G|}{M}$ subgraphs in $\mathcal{P}$. We can create p buffers in memory, one for each subgraph in $\mathcal{P}$. When we map a vertex v to a subgraph $G_i \in \mathcal{P}$,

Algorithm 6 *Randomized Graph Partitioning*

```
Input: A graph $G = (V_G, E_G)$
Output: A partition of $G$, $\mathcal{P} = \{G_1, \ldots, G_i, \ldots, G_p\}$
1.  Create $p$ empty subgraphs for the initial partition $\mathcal{P}$;
2.  Let $h(v)$ be a function that maps a vertex $v \in V_G$ into $[1..p]$ uniformly at random;
3.  for each $v \in V_G$, where $v \notin D$, do
4.      Add $v$ and $adj_G(v)$ to $G_{h(v)} \in \mathcal{P}$;
    end
5.  return $\mathcal{P}$;
```

we add v and $adj_G(v)$ to G_i's buffer. Whenever a buffer is filled with a block (of size B) of data, we write the block of data to disk and clear it from the buffer.

The following lemma gives the expected number of intra-partition edges of a partition computed by the randomized graph partitioning algorithm.

LEMMA 5.2. *Let $\mathcal{P} = \{G_1, \ldots, G_i, \ldots, G_p\}$ be a partition of G computed by Algorithm 6. Let E_{intra} be the set of intra-partition edges of $\mathcal{P}$, i.e., edges that are incident on vertices within the same subgraph in $\mathcal{P}$. Then, the expectation of $|E_{intra}|$ is given by $\mathbf{E}[|E_{intra}|] = \frac{|E_G|}{p}$.*

PROOF. For each vertex $v \in V_G$, the probability that v is mapped to a subgraph $G_i \in \mathcal{P}$ is given by $\mathbf{Pr}[h(v) = i] = \frac{1}{p}$. Given two vertices, u and v, the event that u is mapped to a subgraph in $\mathcal{P}$ and the event that v is mapped to a subgraph in $\mathcal{P}$ are independent. Thus, given an edge (u, v), we have $\mathbf{Pr}[(u, v) \in E_{intra}] = p \times \frac{1}{p} \times \frac{1}{p} = \frac{1}{p}$. As a result, the expectation of $|E_{intra}|$ is given by $\mathbf{E}[|E_{intra}|] = \sum_{(u,v)\in E_G} \mathbf{Pr}[(u, v) \in E_{intra}] = \frac{|E_G|}{p}$. $\square$

Lemma 5.2 shows that the expected number of intra-partition edges obtained by the randomized graph partitioning is equal to the lower bound on the number of intra-partition edges obtained by the dominating-set-based graph partitioning, but without requiring $O(|V_G|)$ memory space as does the dominating-set-based graph partitioning algorithm. In Section 5.4, we also apply Lemma 5.2 to obtain a bound on the total number of iterations required in Algorithm 2 by applying Algorithm 6 with a high probability.

5.4. Bounding I/O Complexity by Graph Partitioning

We now analyze the overall complexity of our algorithm for triangle listing. For all the three graph partitioning algorithms, only $O(scan(|V_G| + |E_G|))$ I/Os are required. More precisely, only one scan of the input graph is required for the sequential and the randomized graph partitioning algorithms, while two scans are required for the dominating-set-based graph partitioning algorithm.

At each iteration of Algorithm 2, we read each extended subgraph in the partition into main memory only once. Thus, the overall I/O complexity for each iteration is $O(scan(|V_G| + |E_G|))$, but for a shrinking graph $G = (V_G, E_G)$.

From the above analysis, the overall I/O complexity of Algorithm 2 depends on the total number of iterations. Thus, we analyze the number of iterations required in Algorithm 2 when each of the three graph partitioning algorithms is applied.

5.4.1. ***Sequential Graph Partitioning.*** If we apply sequential graph partitioning in Algorithm 2, the number of iterations depends largely on the locality of the graph data, which varies for different datasets and is difficult to analyze. There is no graph model that characterizes this property for real-world graphs. Thus, we use synthetic graph dataset to investigate the behavior of the partitioning technique, as we show in our experiments.

5.4.2. ***Dominating-Set-based Graph Partitioning.*** If we apply dominating-set-based graph partitioning, then we can obtain an upper bound on the total number of iterations required in Algorithm 2, as shown in Lemma 5.3.

LEMMA 5.3. *If Algorithm 2 partitions G by Algorithm 5, then the total number of iterations required in Algorithm 2 is $O(\frac{|E_G|}{M})$.*

PROOF. According to Lemma 5.1, the number of intra-partition edges of $\mathcal{P}$ is at least $\frac{|E_G|}{p}$, where $p = \frac{|E_G|}{M}$. This means that $\frac{|E_G|}{p} = \frac{|E_G|}{|E_G|/M} = M$ edges are removed at the end of each iteration in Algorithm 2. Thus, the total number of iterations required in Algorithm 2 is $O(\frac{|E_G|}{M})$. □

With the result of Lemma 5.3, we give the overall I/O complexity of Algorithm 2, when dominating-set-based graph partitioning is applied, as follows.

THEOREM 5.4. *If Algorithm 2 partitions G by Algorithm 5, then Algorithm 2 requires $O(\frac{|E_G|}{M} scan(|V_G| + |E_G|) - scan(\frac{|E_G|^2}{M}))$ I/Os, where $(|V_G| + |E_G|)$ is the size of the original input graph G.*

PROOF. According to Lemma 5.3, at least M edges are removed from G at the end of each iteration of Algorithm 2. Thus, at the start of the t-th iteration, $(t-1)M$ edges have been removed from the original graph G. Summing up, the overall I/O complexity of Algorithm 2 is given by $O(\sum_{t=1}^{|E_G|/M}(scan(|V_G|+|E_G|-(t-1)M))) = O(\frac{|E_G|}{M} scan(|V_G| + |E_G|) - scan((\frac{|E_G|}{M})^2 M)) = O(\frac{|E_G|}{M} scan(|V_G| + |E_G|) - scan(\frac{|E_G|^2}{M}))$ I/Os. □

Although applying dominating-set-based graph partitioning in Algorithm 2 gives a bounded I/O complexity, the overall process requires at least $(|V_G| \log_2 p)$ bits of memory space. The randomized graph partitioning removes this requirement on memory.

5.4.3. ***Randomized Graph Partitioning.*** If randomized graph partitioning is applied, we can establish a bound on the total number of iterations required in Algorithm 2 with a high probability as follows.

LEMMA 5.5. *Let $X = |E_{intra}|$ be the number of intra-partition edges of a partition computed by Algorithm 6. Then, at each iteration of Algorithm 2, we have $Pr[X \geq (1-\epsilon)M] \geq 1 - \frac{1}{\epsilon^2 M}$, where $0 < \epsilon < 1$.*

PROOF. Let $|E_{G^t}|$ be the number of edges in the t-th iteration. Let X_k be the indicator random variable for each edge $e_k \in E_{G^t}$, where $X_k = 1$ if $e_k \in E_{intra}$, and $X_k = 0$ otherwise. Then, we have $X = \sum X_k$.

Recall in Algorithm 6, we uniformly at random pick each vertex to be added to one of the $(p_t = \frac{|E_{G^t}|}{M})$ subgraphs. Thus, we obtain the variance of X as follows:

$$\begin{aligned} Var[X] &= Var[\sum X_k] \\ &= E[(\sum X_k)^2] - (E[\sum X_k])^2 \end{aligned}$$

$$
\begin{aligned}
&= E[\sum X_k^2 + \sum_{j\neq k} X_j X_k] - (E[\sum X_k])^2 \\
&= |E_{G^t}|\frac{1}{p_t} + |E_{G^t}|(|E_{G^t}|-1)\frac{1}{p_t^2} - (\frac{|E_{G^t}|}{p_t})^2 \\
&= \frac{(p_t-1)|E_{G^t}|}{p_t^2} \ .
\end{aligned}
$$

Let $\mu = E[X] = M$, where $E[X] = M$ is obtained in Lemma 5.2. We first obtain $Pr[X < (1-\epsilon)M]$ as follows:

$$
\begin{aligned}
& Pr[X < (1-\epsilon)M] \\
=\ & Pr[\ \mu - X > \mu - (1-\epsilon)M\ ] \\
\leq\ & Pr[\ |\mu - X| > \epsilon M\ ] \\
=\ & Pr[\ |X-\mu| \geq \epsilon\frac{|E_{G^t}|}{p_t}\ ] \ . \qquad (5)
\end{aligned}
$$

By applying Chebyshev's inequality, we have:

$$
\begin{aligned}
Pr[X < (1-\epsilon)M] &\leq Pr[\ |X-\mu| \geq \epsilon\frac{|E_{G^t}|}{p_t}\ ] \\
&\leq \frac{Var[X]}{(\frac{\epsilon|E_{G^t}|}{p_t})^2} \\
&= \frac{p_t - 1}{\epsilon^2|E_{G^t}|} \ . \qquad (6)
\end{aligned}
$$

Thus, we obtain:

$$
\begin{aligned}
Pr[X \geq (1-\epsilon)M] &> 1 - \frac{p_t-1}{\epsilon^2|E_{G^t}|} \\
&> 1 - \frac{p_t}{\epsilon^2|E_{G^t}|} \ . \qquad (7)
\end{aligned}
$$

Since $p_t = \frac{|E_{G^t}|}{M}$, from Equation 7 we obtain the desired result:

$$
Pr[X \geq (1-\epsilon)M] > 1 - \frac{1}{\epsilon^2 M} \ .
$$

□

With the result of Lemma 5.5, the following theorem establishes a bound on the total number of iterations required in Algorithm 2 with a high probability.

THEOREM 5.6. *Let n be the total number of iterations required in Algorithm 2 by applying Algorithm 6. Then, $n \leq \frac{|E_G|}{(1-\epsilon)M}$ with a probability of at least $(1-\frac{1}{\epsilon^2 M})^{\frac{|E_G|}{(1-\epsilon)M}}$, where $0 < \epsilon < 1$.*

PROOF. Let A_t be the event that at the t-th iteration of Algorithm 2, $|E_{intra}| \geq (1-\epsilon)M$. Then, we have:

$$\begin{aligned} Pr[\cap_{t=1}^{k} A_t] &= Pr[A_1] \times Pr[A_2|A_1] \ldots Pr[A_k| \cap_{t=1}^{k-1} A_t] \\ &\geq (1 - \frac{1}{\epsilon^2 M})^k \ . \end{aligned} \tag{8}$$

Let $k = \frac{|E_G|}{(1-\epsilon)M}$, then $\cap_{t=1}^{k} A_t$ implies that Algorithm 2 terminates in at most k iterations, since at each iteration, the number of intra-partition edges is at least $(1-\epsilon)M$ according to Lemma 5.5. By substituting k with $\frac{|E_G|}{(1-\epsilon)M}$ in Equation 8, the proof follows. □

The following example gives an idea how tight the bound obtained in Theorem 5.6 may be.

Example 5.7. Let $\epsilon = 0.1$ and $M = 10^9$. If $|E_G| = 10^{10}$, then by Theorem 5.6, Algorithm 2 runs at most 11 iterations with a probability of greater than 0.99999. In fact, with this reasonable setting of $\epsilon = 0.1$ and $M = 10^9$, for any graph with $|E_G| \leq (9 \times 10^{13})$ edges, we obtain a probability greater than 0.99 that Algorithm 2 terminates within the bound specified in Theorem 5.6, i.e., $n \leq \frac{|E_G|}{(1-\epsilon)M}$. For a graph of a size with more than (9×10^{13}) edges, it is certainly reasonable to assume a larger available memory size M, with which we can establish the bound again with a high probability. □

With the result of Theorem 5.6, we obtain a bound on the overall I/O complexity of Algorithm 2 with a high probability as follows.

THEOREM 5.8. *If Algorithm 2 partitions G by Algorithm 6, then with a probability of at least $(1 - \frac{1}{\epsilon^2 M})^{\frac{|E_G|}{(1-\epsilon)M}}$, the I/O complexity of Algorithm 2 is $O(\frac{|E_G|}{(1-\epsilon)M} scan(|V_G| + |E_G|) - scan(\frac{|E_G|^2}{(1-\epsilon)M}))$ I/Os, where $0 < \epsilon < 1$ and $(|V_G| + |E_G|)$ is the size of the original input graph G.*

PROOF. The proof is similar to that of Theorem 5.4, by replacing M in the proof of Theorem 5.4 with $(1-\epsilon)M$. □

Finally, we remark that the CPU time complexity for triangle listing at each iteration of Algorithm 2 is similar that of the counter-part in-memory algorithm with the same input graph. We thus refer the readers to the related work [Latapy 2008] for details.

6. TRIANGLE LISTING FOR HIGH DEGREE VERTICES WITH LIMITED MEMORY

In the previous sections, we assume that if $V_H = \{u, v\}$, for any $(u, v) \in E$, then the corresponding extended subgraph H^+ fits in main memory (the assumption is given after Definition 4.3 in Section 4.2). In other words, we assume that $(deg_G(u) + deg_G(v)) < M$. This assumption is required because if for some $u, v \in V_G$, $adj_G(u)$ and $adj_G(v)$ cannot fit in memory but they appear in the same extended subgraph, then it incurs extra I/O cost when we apply the in-memory triangle listing algorithm in this extended subgraph. Note that to list any triangles in a local subgraph in a partition, V_H must consist of at least two vertices in the corresponding extended subgraph $H^+ = (V_{H^+}, E_{H^+})$; otherwise, either we have listed all triangles in the graph or we need to re-partition the graph.

Algorithm 7 *I/O-Efficient Triangle Listing for High Degree Vertices*

Input: A set of high degree vertices S, and $adj_G(v)$ for each $v \in S$
Output: The set of triangles with two end vertices in S, i.e., $\triangle(S) = \{\triangle_{uvw} : u, v \in S\}$

```
1.  for each $u \in S$ do
2.      for each block, $B_1$, of $adj_G(u)$ read from disk do
3.          for each $v$ in $B_1$, where $v \in S$ and $v > u$, do
4.              for each block, $B_2$, of $adj_G(v)$ read from disk do
5.                  for each $w \in (B_1 \cap B_2)$ do
6.                      if($w > v$ or $w \notin S$)
7.                          List $\triangle_{uvw}$;
                        end
                    end
                end
            end
        end
    end
```

The above assumption is realistic with the available memory size of a commodity PC today because for a machine with 4GB of memory, $adj_G(u)$ and $adj_G(v)$ should consist of 10^9 adjacent vertices in order to use up the memory. Even with very large power-law graphs [Faloutsos et al. 1999; Newman 2003] that can consist of few vertices with large degree, it is still extremely rare that we have such a large real graph with this extreme vertex degree. Note that we are not referring to the number of vertices in a graph but the degree of a vertex in a graph.

The above estimation shows that our I/O-efficient algorithm can handle most of the real-world large graphs. For those extremely large graphs for which the assumption is not valid, we propose an algorithm to handle this extreme case as follows.

Let S be the set of high degree vertices that violate the above-mentioned assumption. We first remove S, together with $adj_G(v)$ for each $v \in S$, from the input graph G, which gives a new graph G'. Then, we apply Algorithm 2 to list all triangles in G'. Finally, we list the rest of the triangles in G as shown in Algorithm 7.

Algorithm 7 is similar to Algorithm 3, except that now each $adj_G(u)$ and $adj_G(v)$ cannot fit in main memory. Therefore, to avoid random disk access, we use a *block nested-loop join* to allow sequential disk scans described as follows.

In the block nested-loop join, both the outer relation and the inner relation are the adjacency lists $adj_G(u)$ and $adj_G(v)$ for each pair of vertices $u, v \in S$, where $(u, v) \in E_G$ and $u < v$. The outer relation reads $adj_G(u)$ from disk, which is then joined with $adj_G(v)$ in the inner relation to find the common neighbor w of u and v, which form the triangle $\triangle_{uvw}$.

The I/O complexity analysis of Algorithm 7 is similar to that of the standard block nested-loop join. The number of disk blocks occupied by the outer relation is given by $\alpha = \sum_{u \in S} scan(adj_G(u))$, since the outer relation is scanned only once. For each $u \in S$ processed in Algorithm 7, the number of disk blocks occupied by the inner relation is given by $\beta(u) = \sum_{v \in adj_G(u) \wedge v \in S \wedge v > u} scan(adj_G(v))$. If $adj_G(u)$ can fit in memory, then the algorithm requires $O(\alpha + \sum_{u \in S} \beta(u))$ I/Os. Otherwise, the algorithm requires $O(\alpha + (\alpha/(M/B-2)) \cdot (\sum_{u \in S} \beta(u)))$ I/Os.

The I/O cost can be high if S is large, i.e., there are many extremely high degree vertices in the graph. However, according to [Faloutsos et al. 1999; Newman 2003], there are only a very small number of high degree vertices in a large real world graph, which can be estimated as follows.

According to [Faloutsos et al. 1999], large real-world graphs follow a power law degree distribution given by the following equation:

$$deg_G(v) = \frac{1}{|V_G|^r}(rank_G(v))^r, \tag{9}$$

where $rank_G(v)$ is the ***degree rank*** of a vertex v in G, i.e., v is the $(rank_G(v))$-th highest degree vertex in G, and r is the ***rank exponent***, where $r < 0$ is a ***constant*** with a typical value between -0.8 and -0.7 for most real-world networks [Faloutsos et al. 1999].

According to Equation 9, if $r = -0.7$ and $(deg_G(u) + deg_G(v)) \approx 10^9$, where u and v are the two highest degree vertices in G, then $|V_G| \approx (4 \times 10^{12})$. In other words, for a power-law graph G that has 4 trillion vertices, it only has two vertices u and v in G such that $adj_G(u)$ and $adj_G(v)$ would use up 4GB of memory, i.e., $|S| = 2$ in Algorithm 7.

7. UPDATE IN DYNAMIC NETWORKS

Many real-world networks undergo frequent updates. When the network graph is updated, the set of triangles of the graph also needs to be updated. The challenge, however, is that a single edge insertion or deletion can trigger a series of updates to the set of triangles. This can be a costly operation because the set of triangles may be large and stored on disk, and thus the updates require random disk accesses to read and write the triangles.

In this section, we present a compact data structure for efficient update of the set of triangles when the input graph undergoes frequent updates. We consider two types of updates: edge insertion and edge deletion. Vertex insertion/deletion can be considered as a sequence of edge insertions/deletions preceded/followed by the insertion/deletion of an isolated vertex. Note that it is trivial to handle the insertion/deletion of an isolated vertex, since it does not trigger an update to the set of triangles.

7.1. A Compact Data Structure for Storage of Triangles

We first present the data structure that supports efficient update of the set of triangles. Apart from supporting efficient update, the data structure is also useful in applications where the set of triangles needs to be materialized on disk.

The set of triangles can be naturally represented as a prefix tree structure. We define the data structure, called *triangle-tree*, as follows.

Definition 7.1 (*Triangle-Tree*). Let $\triangle(G)$ be the set of all triangles in a graph G. The *triangle-tree* of G is a prefix tree defined as follows.

— The root of the tree represents an empty set.
— Each $\triangle_{uvw} \in \triangle(G)$, where $u < v < w$, is represented by a path in the tree as follows: u is a child of the root, v is a child of u, and w is a child of v.
— The prefix tree order follows the order of the IDs of the vertices in the triangles.

Excluding the root, the triangle-tree has three levels, corresponding to the three vertices in a triangle. The compactness comes from the sharing of prefixing vertices among the triangles. Further compression is possible to reduce the space for storage purpose; however, our focus here is to support efficient update operations on disk.

To support the update of the set of triangles in a dynamic graph, we need to support efficient insertion and deletion of nodes in the triangle-tree. Both node insertion and deletion can be done in logarithmic time if the operations are performed in main memory. However, the triangle-tree data structure is stored on disk and the update operations need to be reflected on disk. If the tree is represented as pointers, then random disk accesses to the tree nodes are obviously too expensive. If the nodes in the triangle-tree is stored sequentially on disk, then a single node insertion may cause all the data following the node to be shifted backwards on disk in order to maintain the sequential order, which is prohibitive if the tree is large or the update is frequent.

We propose an effective disk storage scheme for the triangle-tree as follows. The storage scheme consists of three levels, corresponding to the three levels (excluding the root) in the triangle-tree. The nodes at each level of the triangle-tree are stored, according to their level-order traversal sequence, in a linked list of disk blocks (note that only the blocks are stored as a linked list, not the data items inside each block). To enable access to their children, each internal node (i.e., nodes at the first two levels) of the triangle-tree is also associated with a pointer to the block where its children are stored.

To avoid the overflow of a disk block due to node insertions or the under-utilization of disk space due to node deletions, we leave some free space in each of the disk blocks. Then, the following invariant is used to maintain the tree storage for efficient update: *at least $\frac{2}{3}$B of storage space is utilized for every pair of neighboring disk blocks*. Note that it does not mean that we only use $\frac{2}{3}$B out of every 2B of storage space, the $\frac{2}{3}$B is only an invariant to be maintained during the update process but we can use much more space than that.

With the above invariant, a node insertion requires a single I/O if the insertion does not cause an overflow. Let Y be the disk block where the node is to be inserted, and X and Z be the two disk blocks at either end of Y. Now suppose that we have an overflow, i.e., the block Y is full. Then, if either X or Z still has free space to insert the node, we simply shift the data within the two blocks Y and X (or Z), and then insert the node there. If both X and Z are full, then we split Y into two new blocks, Y_1 and Y_2, each taking approximately $\frac{1}{2}$B of the data from Y. Then, we insert the node into either Y_1 or Y_2 according to its order. It is easy to see that now the three pairs of neighboring blocks, i.e., (X, Y_1), (Y_1, Y_2), and (Y_2, Z), utilize $\frac{3}{2}$B, 1B, and $\frac{3}{2}$B of storage space, respectively. Therefore, the invariant is always maintained for an insertion operation, which takes $O(1)$ I/Os in the worst case.

Now we discuss a node deletion. Let Y be the disk block where the node is deleted, and X and Z be the two disk blocks at either end of Y. To maintain the invariant, we need to check if the pair of blocks (X, Y) or (Y, Z) now use less than $\frac{2}{3}$B of storage space. If both pairs still utilize at least $\frac{2}{3}$B of storage space, then the variant is still maintained. But assume that, without the loss of generality, X and Y now uses less than $\frac{2}{3}$B of storage space, then we simply merge X and Y into one single block. Clearly, the merging re-establishes the invariant. Thus, a node deletion takes $O(1)$ I/Os in the worst case.

There is another type of update that we also need to consider, that is, the update of the pointer of a node to its children's disk block. For an insertion/deletion of a triangle into/from the triangle-tree, we access the tree from parent to child by loading their respective disk blocks into memory; thus, the pointer information to the child's block can be easily updated in memory before it is written back to disk. However, in case of a disk block split/merge due to a node insertion/deletion at the child level, it may trigger an update of the pointer information of some nodes at the parent level. Let X be the block at the parent level currently loaded in memory. Now if the pointer information to be updated at the parent level is within the block X, it can be done directly in X in memory and then write X back to disk. However, if the parent block in which the pointer information needs to be updated is not the block X in memory, then it must be a neighboring block of X on disk, because the nodes in the triangle-tree are stored according to their level-order traversal sequence. Thus, we can read the corresponding neighboring block of X into memory, update the pointer information to the child blocks, and then write the block back to memory. In the worst case, such an update also takes $O(1)$ I/Os only.

7.2. Edge Insertion and Deletion

Having discussed the data structure that supports efficient update of triangles, we now discuss how the update to the set of triangles is performed when a new/old edge is inserted/deleted into/from the input graph.

7.2.1. ***Edge Insertion****.* We first consider edge insertion. The insertion of an edge, $e = (u, v)$, may create a number of new triangles, which are to be inserted into the triangle-tree. Let $\triangle(e)$ be the set of new triangles created with the insertion of e. Assume that, without the loss of generality, $u < v$. We can divide $\triangle(e)$ into the following three categories:

- $\triangle_1(e) = \{\triangle_{uvw} : \triangle_{uvw} \in \triangle(e), u < v < w\}$,
- $\triangle_2(e) = \{\triangle_{uvw} : \triangle_{uvw} \in \triangle(e), u < w < v\}$,
- $\triangle_3(e) = \{\triangle_{uvw} : \triangle_{uvw} \in \triangle(e), w < u < v\}$.

To insert the triangles in $\triangle_1(e) = \{\triangle_{uvw_1}, \cdots, \triangle_{uvw_k}\}$ into the triangle-tree, we process as follows. We first try to locate u (by binary search) among the children of the root. If u is not a child of the root, then we create a new node u to be inserted as a child of the root. Note that even if u is already a child of the root in the triangle-tree, v cannot be a child of u in the tree because the edge (u, v) is a new edge. Thus, in either case, we create a new node v to be inserted among the children of u or as the only child of u if u is newly created. Then, we create new nodes, w_1, $\cdots$, and w_k, to be inserted as the children of v in the triangle-tree.

The insertion of a triangle $\triangle_{uvw}$ in the second category, $\triangle_2(e)$, into the triangle-tree is processed as follows. We first try to locate u among the children of the root and then w among the children of u. If u and w are already there in the triangle-tree, then we create a new node v to be inserted among the children of w. If u and/or w are not in the triangle-tree, we create u and/or w to be inserted into the tree, followed by the creation of a new node v to be inserted as a child of w.

The insertion of a triangle $\triangle_{uvw}$ in the third category, $\triangle_3(e)$, into the triangle-tree is processed in a similar way as the insertion of a triangle in $\triangle_2(e)$, except that the orders of u and w are now reversed.

The insertion of a new node into the triangle-tree and its cost/complexity is discussed in Section 7.1. The extra cost required in the above process of inserting a triangle is the cost of locating a node among the children or determining the position where a node should be inserted among the children, both of which can be done by a binary search in memory.

7.2.2. ***Edge Deletion****.* Next, we consider edge deletion. The deletion of an edge, $e = (u, v)$, may invalidate a number of existing triangles, which thus need to be deleted from the triangle-tree. Let $\triangle(e)$ be the set of triangles that are originally in G but are no longer triangles after the removal of e from G. As in Section 7.2.1, we assume that $u < v$ and divide $\triangle(e)$ into three categories.

To delete the triangles in $\triangle_1(e) = \{\triangle_{uvw_1}, \cdots, \triangle_{uvw_k}\}$ from the triangle-tree, we process as follows. First, we locate u among the children of the root and then v among the children u. We delete all the children of v, which are exactly the set $\{w_1, \cdots, w_k\}$, and then also delete v from among the children of u. If after the deletion of v, u has no other child, then we also delete u from the triangle-tree.

The deletion of a triangle $\triangle_{uvw}$ in the second category, $\triangle_2(e)$, from the triangle-tree is processed as follows. First, we locate u among the children of the root and then w

among the children u. Then, we locate v among the children of w and delete v. We also delete w if after the deletion of v, w has no other child. Similarly, we delete u if w is the only child of u.

The deletion of a triangle $\triangle_{uvw}$ in the third category, $\triangle_3(e)$, from the triangle-tree is processed in a similar way as the deletion of a triangle in $\triangle_2(e)$, except that the orders of u and w are now reversed.

The deletion of a node from the triangle-tree and its cost/complexity is discussed in Section 7.1. The extra cost required in the above process of deleting a triangle is the cost of locating a node among the children, which can be done by a binary search in memory.

8. APPLICATIONS OF TRIANGLE LISTING

Triangle listing has many important applications. Here, we focus on a few popular ones and demonstrate how our algorithm can be applied to benefit these applications in massive networks that cannot fit in main memory.

8.1. Triangle Counting, Clustering Coefficients, and Transitivity

Our algorithm can be readily applied to compute triangle counting, clustering coefficients, and transitivity. For completeness, we also give the definitions of these concepts here.

We first define *clustering coefficient* [Watts and Strogatz 1998], which is a popular index for network analysis, as follows.

Definition 8.1 (*Clustering Coefficient*). The clustering coefficient of a vertex $v \in V_G$, where $deg_G(v) > 1$, denoted by $\mathcal{C}(v)$, is defined as

$$\mathcal{C}(v) = \frac{N_\triangle(v)}{N_\vee(v)}. \tag{10}$$

When $deg_G(v) \leq 1$, we define $\mathcal{C}(v) = 0$.

The clustering coefficient of G, denoted by $\mathcal{C}(G)$, is defined as

$$\mathcal{C}(G) = \frac{1}{|V_G|} \sum_{v \in V_G} \mathcal{C}(v). \tag{11}$$

Intuitively, the clustering coefficient of a vertex v, also called the *local clustering coefficient* or *neighborhood density* of v, defines the probability that two vertices are also neighbors of each other if they are both the neighbors of v. In a social network setting, local clustering coefficient implies how likely the two friends of a person are also themselves friends of each other.

The clustering coefficient of a network/graph G is the average of the clustering coefficient of all the vertices in G. G is a *small-world* network if $\mathcal{C}(G)$ is high (with respect to that of a random graph constructed on the same set of vertices) and G also has short average path length [Watts and Strogatz 1998].

Next we define *transitivity* [Wasserman and Faust 1994; Newman et al. 2002] of a network as follows.

Definition 8.2 (*Transitivity*). The transitivity of a graph G, denoted by $\mathcal{C}(G)$, is defined as

$$\mathcal{T}(G) = \frac{N_\triangle(G)}{\frac{1}{3}\sum_{v \in V_G} N_\vee(v)}. \tag{12}$$

Algorithm 8 *Triangle Counting, Clustering Coefficients, and Transitivity*

Input: A graph $G = (V_G, E_G)$
Output: $N_\triangle(v)$ and $\mathcal{C}(v)$ for each $v \in V_G$, $\mathcal{C}(G)$, and $\mathcal{T}(G)$

```
 1.  $\forall v \in V_G, N_\triangle(v) \leftarrow 0;$
 2.  $\mathcal{C}(G) \leftarrow 0;\ \ N_\triangle(G) \leftarrow 0;\ \ N_\vee(G) \leftarrow 0;$
 3.  for each triangle $\triangle_{uvw}$ listed by Algorithm 2 do
 4.      $N_\triangle(x) \leftarrow N_\triangle(x) + 1$, for all $x \in \{u, v, w\}$;
     end
 5.  for each $v \in V_G$ do
 6.      $\mathcal{C}(v) \leftarrow \frac{2N_\triangle(v)}{deg_G(v)(deg_G(v)-1)}$;
 7.      $\mathcal{C}(G) \leftarrow (\mathcal{C}(G) + \mathcal{C}(v))$;
 8.      $N_\triangle(G) \leftarrow (N_\triangle(G) + N_\triangle(v))$;
 9.      $N_\vee(G) \leftarrow (N_\vee(G) + \frac{1}{2} deg_G(v)(deg_G(v) - 1))$;
     end
10.  $\mathcal{C}(G) \leftarrow \frac{\mathcal{C}(G)}{|V_G|}$;
11.  $\mathcal{T}(G) \leftarrow \frac{3N_\triangle(G)}{N_\vee(G)}$;
```

The transitivity of a network/graph G is a measure of the degree to which the vertices in G tend to cluster together. It is often known as the *global clustering coefficient*, because it is an indication of clustering in the whole network, i.e., globally as contrast to the local clustering coefficient $\mathcal{C}(v)$. Note that $\mathcal{T}(G) \neq \mathcal{C}(G)$ in general.

When the input graph is too large to fit in main memory, our algorithm is far more efficient than the existing algorithms for computing these network measures. Algorithm 2 can be pipelined to compute all the above measures as shown in Algorithm 8, i.e., we do not need to first compute all triangles and perform a post-processing. The algorithm is self-explanatory.

Steps 5-9 of Algorithm 8 requires another scan of G, but the asymptotic I/O complexity of the algorithm is the same as Algorithm 2. Updating $N_\triangle(v)$ for all $v \in V_G$ may require $O(|V_G|)$ space, but this can be avoided by writing $N_\triangle(v)$ after processing each extended subgraph and then merging the results of all extended subgraphs. The asymptotic I/O complexity still remains the same since the size of $N_\triangle(v)$ for all v in an extended subgraph is bounded by the size of the subgraph.

8.2. Triangular Vertex Connectivity

Triangular vertex connectivity (also called *3-gonal connectivity*) defines stronger connectivity in a network than single link connectivity, since edges that are in short cyclic component (e.g., triangles) are considered as strong ties [Granovetter 1973; Batagelj and Zaveršnik 2007]. Triangular vertex connectivity is formally defined as follows [Schank 2007].

Definition 8.3 (*Triangular Vertex Connectivity*). Two vertices u and v are *triangularly vertex-connected* if there exists a sequence of triangles $\langle \triangle_1, \ldots, \triangle_n \rangle$ such that u is in $\triangle_1$, v is in $\triangle_n$, and either (1) $n = 1$, or (2) for $1 \leq i < n$, $\triangle_i$ and $\triangle_{i+1}$ share at least one common vertex.

Intuitively, if u and v are connected by a single path, then they become disconnected if any edge on the path is removed. On the contrary, if u and v are connected by a sequence of triangles, then removing any edge does not disconnect them.

Triangular vertex connectivity is important in many applications [Eckmann and Moses 2002; Fritzke 1993; Taubin and Rossignac 1998; Watts and Strogatz 1998]. It defines an equivalence relation $\mathcal{V}^{\triangle}$ on V_G. Two vertices u and v belong to the same equivalence class if they are triangularly vertex-connected. The following example further explains the concept.

Example 8.4. In the graph G given in Figure 1, there are two equivalence classes defined by triangular vertex connectivity, $C_1 = \{a, b, c, g, i\}$ and $C_2 = \{d, e, f, h, j, k, l\}$, which can be obtained by removing the three edges (in bold lines) that are not part of any triangle. Let $G[C_1]$ and $G[C_2]$ be induced subgraph of G by C_1 and C_2, respectively. Removing any edge from $G[C_1]$ or $G[C_2]$ does not disconnect the vertices in $G[C_1]$ or $G[C_2]$, which demonstrates that the connectivity between the vertices within the graphs are stronger. $\Box$

The existing algorithm for computing the equivalence classes of $\mathcal{V}^{\triangle}$ is based on triangle listing [Batagelj and Zaveršnik 2007]. However, the algorithm assumes that all computations, both equivalent class computation and triangle list, are processed in main memory, which is thus impractical when the input graph is too large and disk resident.

We show that the result of Algorithm 2 can be easily pipelined to compute the equivalence classes of $\mathcal{V}^{\triangle}$ as follows. Upon processing each extended subgraph, we first mark the directed edges (u, v), (u, w), (v, w), where $u < v < w$, for each triangle $\triangle_{uvw}$ listed. Then, we write these marked edges to disk. When Algorithm 2 terminates, we read the marked edges one by one to find which equivalent class each $v \in V_G$ belongs to. This can be done by using two lookup tables, C and A, where $C[j] = i$ indicates that i is the smallest class ID that Class j is connected to, and $A[v] = i$ indicates v belongs to Class i. Initially, $C[i] = i$ and $A[v] = \infty$. We keep a counter c which is initialized to 0. For each marked edge (u, v) read, we do the following: (1) if $A[u] = A[v] = \infty$, we set $A[u] = A[v] = c$ and increment c; (2) if $A[u] \neq A[v]$, without the loss of generality, assume that $A[u] < A[v]$, we set $C[A[v]] = min(C[A[v]], A[u])$ and $A[v] = A[u]$; (3) otherwise, do nothing. Finally, we scan C once to update each $C[i]$ to the smallest class ID that Class i is connected to, and update $A[v] = C[A[v]]$, which indicates the class v belongs to.

Since the number of marked edges written to disk for each extended subgraph cannot exceed the size of the subgraph, the asymptotic I/O complexity of computing the equivalence classes of $\mathcal{V}^{\triangle}$ by the above algorithm is the same as that of Algorithm 2.

9. EXPERIMENTAL RESULTS

We compare our algorithms with the state-of-the-art in-memory triangle listing algorithm (denoted by **In-Mem**) [Latapy 2008] and the semi-streaming local triangle estimation algorithm (denoted by **Semi-Stream**) [Becchetti et al. 2008]. We ran all experiments on a machine with an Intel Xeno 2.67GHz CPU and 4GB RAM, running CentOS 5.4.

Dataset. We use four real datasets: *LiveJournal* (**LJ**), *U.S. road network* (**USRD**), *World Wide Web of UK* (**WebUK**), and *Billion Triple Challenge* (**BTC**). LJ is a social network (http://www.live-journal.com, http://snap.stanford.edu), where vertices are members and edges represent friendship between members. USRD is the road network of United States, where vertices represent intersections and endpoints, and edges represent the roads connecting these intersections or road endpoints. WebUK is obtained from the YAHOO webspam dataset (http://barcelona.research.yahoo.net), where vertices are pages and edges are hyperlinks. BTC is a semantic graph converted from the Billion Triple Challenge 2009 RDF dataset (http://vmlion25.deri.ie), where each vertex represents an object such as a person, a document, and an event, and each edge represents the relationship between two vertices such as “has-author”, “links-to”, and “has-title”.

To examine the behavior of the graph partitioning algorithms, we also use a synthetic dataset, which is denoted as **Synthetic** in our discussion. We give the number of vertices and edges, the storage size on disk, and the size of the graph in memory (in binary format), of all the datasets in Table II. Note that the actual memory used for in-memory triangle listing is larger than the size of the graph in memory because the in-memory algorithm also uses extra data structures such as hashtable to speed up the process.

Table II. Datasets (M=10^6, G=10^9)

	LJ	**USRD**	**WebUK**	**BTC**	**Synthetic**
$\vert V_G\vert$	4.8M	24M	106M	165M	604M
$\vert E_G\vert$	69M	58M	1,877M	773M	1,208M
Disk size	809.1MB	969.6MB	20.3GB	10.0GB	21.4GB
Memory size	363.9MB	402.8MB	7.8GB	4.1GB	9.0GB

9.1. Effectiveness of Graph Partitioning Algorithms

We first show the effectiveness of the three graph partitioning algorithms, *sequential graph partitioning* (**Sequential**), *dominating-set-based graph partitioning* (**Dominating**), and *randomized graph partitioning* (**Randomized**). We set the available memory size M for partitioning to be $256MB$, $512MB$, $1GB$, $2GB$, and $4GB$, respectively.

Tables III and IV reports the total number of iterations Algorithm 2 takes by applying Sequential, Dominating, or Randomized, on the two large datasets, WebUK and BTC. The theoretical upper bound on the total number of iterations by applying Dominating and that by applying Randomized (with a probability of at least 0.99, where $\epsilon = 0.1$) are also given as reference.

Table III. The Number of Iterations of Algorithm 2 using Different Graph Partitioning Algorithms on the WebUK Dataset

	M=256MB	M=512MB	M=1GB	M=2GB	M=4GB
Sequential	3	3	2	2	1
Dominating	–	3	2	2	1
Dominating (Upper Bound)	–	16	8	4	2
Randomized	33	16	8	3	2
Randomized (Upper Bound)	35	18	9	5	3

Table IV. The Number of Iterations of Algorithm 2 using Different Graph Partitioning Algorithms on the BTC Dataset

	M=256MB	M=512MB	M=1GB	M=2GB	M=4GB
Sequential	$\rightarrow \infty$	$\rightarrow \infty$	7	2	1
Dominating	–	6	3	2	1
Dominating (Upper Bound)	–	9	5	3	2
Randomized	18	9	4	2	2
Randomized (Upper Bound)	19	10	5	3	2

The results show that for both WebUK and BTC, Dominating is most effective in all cases when the available memory is large enough, which is an expected result since dominating-vertex-based graph partitioning groups neighboring vertices together. These neighboring vertices tend to form Type 1 and Type 2 triangles, resulting in more intra-partition edges being deleted at the end of each iteration and hence less total number of iterations. However, Dominating becomes infeasible when the available memory is smaller than $O(|V_G|\log_2 p)$ bits, which happens to both WebUK and BTC when $M = 256$MB.

The results show that Sequential is effective in practice for both WebUK and BTC. In fact, the number of iterations of Algorithm 2 by applying Sequential is as small as that by applying Dominating in most cases. Thus, as shown in Table VI, applying Sequential in Algorithm 2 can be more efficient than applying Dominating since Dominating requires two more scans of the input graph at each iteration of Algorithm 2.

However, when the available memory becomes smaller, adopting Sequential shows no sign of termination since there is a point that all or almost all triangles are Type 3 triangles with respect to the partition. Thus, no or very few edges are removed at the end of an iteration. Such a situation happens to the BTC dataset when $M = 256$MB or $M = 512$MB, since BTC has a lower locality than the WebUK dataset.

In this case, Dominating shows its advantage as it gives a guaranteed upper bound on the number of iterations, while our result shows that it indeed has consistent performance. The result also shows that in all cases, the actual number of iterations needed by applying Dominating is always less than its theoretical upper bound.

The results also show that Randomized is not as effective as Sequential and Dominating in some cases, mainly because the randomized process destroys the locality that exists in the real datasets. However, when both Sequential and Dominating fail, the results show that Randomized is still effective, which demonstrates the advantage of applying Randomized in Algorithm 2. We also show that the total number of iterations of Algorithm 2 by applying Randomized almost matches its corresponding upper bound in all cases, for both WebUK and BTC.

We also remark that when $M = 4$GB, applying either Sequential or Dominating in Algorithm 2 requires only 1 iteration, for both WebUK and BTC. This result may seem to be be impossible since both datasets cannot fit in the 4GB memory and therefore one may expect that at least 2 iterations are needed. We examined the details and found that there are only two subgraphs in the partition, each of them (with the more compact binary format than the ascii format in disk storage) can fit in the 4GB memory (just fit for WebUK). In this case, there is no Type 3 triangles (since there are only two subgraphs in the partition) and therefore all triangles, either Type 1 or Type 2, are listed at the first iteration.

To further analyze the behaviors of the graph partitioning algorithms, we generate the Synthetic dataset as follows. Let G be the Synthetic graph. We divide G sequentially into three parts, G_1, G_2, and G_3. The graph G is generated in such a way that there is no triangle within each G_i ($i \in \{1, 2, 3\}$) and across any G_i and G_j ($i \neq j$); that is, there is no Type 1 and Type 2 triangles with respect to the partition $\{G_1, G_2, G_3\}$ of G. The size of each G_i, when loaded in memory, is larger than 2GB but smaller than 4GB.

Table V. The Number of Iterations of Algorithm 2 using Different Graph Partitioning Algorithms on the Synthetic Dataset

	M=256MB	M=512MB	M=1GB	M=2GB	M=4GB
Sequential	∞	∞	∞	∞	3
Dominating	–	–	–	2	2
Dominating (Upper Bound)	–	–	–	5	3
Randomized	45	23	10	5	2
Randomized (Upper Bound)	40	20	10	5	3

We show the results for the three graph partitioning algorithms in Table V. For Sequential, the algorithm obviously enters a dead loop when the available memory is less than 4GB, since there is no Type 1 and Type 2 triangles in these cases. For Dominating, the algorithm fails when the available memory is smaller than $O(|V_G| \log_2 p)$ bits, but is very effective when more memory ($M \geq$ 2GB) is available. For Randomized, it works in all settings of available memory and thus the result again demonstrates its advantage over Sequential and Dominating. However, for the cases when Dominating works, Randomized is not as effective as Dominating, since Dominating groups neighboring vertices together even though the neighboring vertices may be widely scattered over the Synthetic graph. In the case when Sequential also works ($M =$ 4GB), Randomized is more effective than Sequential, mainly because Sequential requires more on data locality which is lacking in the Synthetic dataset.

9.2. Performance Comparison with Existing Algorithms

We now report the performance of Algorithm 2 by applying Sequential (denoted by **TL-Sequential**), Dominating (denoted by **TL-Dominating**), and Randomized (denoted by **TL-Randomized**), compared with In-Mem [Latapy 2008] and Semi-Stream [Becchetti et al. 2008]. We set the available memory size for our I/O-efficient algorithms is to be 2GB.

Table VI reports the running time (wall-clock time in seconds) of all the algorithms. We do not report the memory consumption since the smaller datasets LJ and USRD can fit in memory and all algorithms use roughly the same memory (less than 1GB), while our algorithms and Semi-Stream use all available memory for the larger datasets BTC and WebUK. In-Mem is extremely slow on the larger datasets BTC and WebUK due to too many I/O swaps as a result of insufficient memory.

Table VI. Running Time (wall-clock time in seconds) of Our Algorithms Compared with the Existing Algorithms

	LJ	**USRD**	**BTC**	**WebUK**
TL-Sequential	29.63	6.24	350	2411
TL-Dominating	29.43	12.46	412	2503
TL-Randomized	20.60	29.03	465	2991
In-Mem	32.98	6.68	–	–
Semi-Stream ($\overline{\sigma}(N_\triangle(v)) \approx 0.8$)	306	321	3402	7032
Semi-Stream ($\overline{\sigma}(N_\triangle(v)) \approx 0.5$)	1275	1683	13711	34722

The result shows that the performance of our algorithm, whichever of the three graph partitioning algorithms is applied, is very competitive. When the graphs can fit in memory, our algorithm has comparable performance with In-Mem. When the graphs are too large to fit in memory, our algorithm demonstrates significant advantage over Semi-Stream, which makes multiple passes over the original input graph (in contrast to our algorithm that makes multiple iterations over a shrinking graph).

Let $\overline{\sigma}(N_\triangle(v))$ be the *average approximation error rate* for $N_\triangle(v)$, defined as $(|\text{approximate value} - \text{exact value}|/\text{exact value})$ averaged over all vertices. We find that Semi-Stream takes significantly longer time in order to obtain a low $\overline{\sigma}(N_\triangle(v))$. Table VI reports the running time for Semi-Stream by setting $\overline{\sigma}(N_\triangle(v)) \approx 0.8$ and $\overline{\sigma}(N_\triangle(v)) \approx 0.5$. The result shows that Semi-Stream is many times slower than our algorithm in the case of $\overline{\sigma}(N_\triangle(v)) \approx 0.8$ and up to orders of magnitude slower in the case of $\overline{\sigma}(N_\triangle(v)) \approx 0.5$. Note that our algorithm is an exact algorithm.

From another angle of comparison, we choose a setting for Semi-Stream that it takes slightly longer time than our algorithm TL-Sequential, and we report the error rate

Table VII. Error Rate of Semi-Stream at Comparable Running Time with TL-Sequential

LJ	**USRD**	**BTC**	**WebUK**
97.6%	133.6%	115.4%	95.0%

of Semi-Stream in Table VII. The result shows that at comparable running time with TL-Sequential, the error rate $\overline{\sigma}(N_{\triangle}(v))$ of Semi-Stream increases significantly.

9.3. Performance on Update in a Dynamic Network

In this subsection, we assess the performance of updating the set of triangles when the input graph is updated. We use a real dynamic network, the *Technorati blog* (**blog**) dataset, whose edges are associated with a time stamp when they were created. The Technorati blog network was collected from the top-15 popular queries published by Technorati (technorati.com) every three hours over a period of over a year. For each query, the top-50 results are retrieved. In the blog network, vertices are blogs and edges indicate that two blogs appear in the search result of the same query.

Update in the blog network is performed as follows. We first choose a starting point, that is the blog network obtained by the end of February 2007. Then, to test the update performance on edge insertion, we insert the edges that were created in the months of March and April 2007, in the order of their time stamp. To test the update performance on edge deletion, we delete the edges that were created in the months of February and January 2007, in the reverse order of their time stamp. For each edge insertion and deletion, we update the set of triangles accordingly.

Table VIII reports the size of the blog network at the starting point T, i.e., the end of February 2007, two months before T, denoted by T_-, and two months after T, denoted by T_+. We also report the number of triangles in the blog network at T_-, T, and T_+, respectively. The result shows that with the deletion of 1.63 million edges created in the period from T_- to T, we need to delete 5.41 million triangles. This implies that for each edge deletion, on average 3.32 triangles need to be deleted. On the other hand, the insertion of 1.91 million edges from T to T_+ triggers the insertion of 6.81 million triangles. On average, we have 3.57 triangles that need to be inserted for each edge insertion.

Table VIII. Dataset Size, Triangle Number, Storage Size of Triangles, Triangle-Tree Sizes for Storage and Update, and Disk Utilization (for Update Only)

	$\lvert V_G\rvert$	$\lvert E_G\rvert$	$N_{\triangle}(G)$	Storage Size of $\triangle(G)$	Triangle-Tree Size for Storage	Triangle-Tree Size for Update	Disk Utilization (Update Only)
T_-	0.09M	1.47M	5.06M	58MB	25MB	67MB	42.97%
T	0.17M	3.10M	10.47M	120MB	49MB	91MB	65.63%
T_+	0.27M	5.01M	17.28M	198MB	78MB	123MB	79.69%

Table VIII also reports the disk storage size of the set of triangles, the size of the triangle-tree (for storage purpose only), and that of the triangle-tree (for update purpose), at T_-, T, and T_+, respectively. The result shows that the triangle-tree saves the storage space considerably, especially when the tree is used for storage purpose only, i.e., there is not pointer information maintained for update purpose and every disk block is fully used. The disk utilization rate is considerably lower at T_- than at T_+. This is mainly because the update is performed as a series of continuous edge deletions or edge insertions, respectively, from the starting point T. As a result, the utilization rate at T_- is lower than at T while that at T_+ is higher than at T, which results in the greater gap between T_- and T_+. However, a continuous load of edge deletions and that of edge insertions stand for the two severest update conditions. And the result shows that in all cases, the invariant given in Section 7 is always maintained.

Table IX reports the update time and the number of I/Os used for the update, averaged over all edges or triangles. The result shows that inserting or deleting triangles is highly efficient, which is only about 0.01 milliseconds per triangle update. The number of I/Os required is only slightly more than 1 I/Os for each of the three vertices in a triangle. For edge insertion and edge deletion, they are more costly because an edge insertion/deletion may trigger the insertion/deletion of a number of triangles.

Table IX. Average Update Time (wall-clock time in milliseconds) and Average Number of I/Os for Update

	Insertion (Per Edge)	Insertion (Per △)	Deletion (Per Edge)	Deletion (Per △)
Avg. Update Time (msec)	0.11	0.015	0.07	0.011
Avg. Number of I/Os	33.43	4.69	26.45	4.00

In summary, the results show that the triangle-tree is efficient for storage as well as for frequent updates in dynamic networks.

9.4. Performance on Applications

The experimental results in Section 9.2 have demonstrated that our algorithm has significant advantages over the existing algorithms when the input graphs are too large to fit in main memory.

Table X reports the error rates of clustering coefficient and transitivity of a network approximated by Semi-Stream, denoted by $\sigma(\mathcal{C}(G))$ and $\sigma(\mathcal{T}(G))$, respectively. The result shows that, although the error rate of Semi-Stream is not too large for those global measures, our algorithm is able to obtain the exact results (the running time is almost the same as that shown in Table VI).

Table X. Error Rate of Semi-Stream for $\sigma(\mathcal{C}(G))$ and $\sigma(\mathcal{T}(G))$

BTC ($\sigma(\mathcal{C}(G))$)	**BTC** ($\sigma(\mathcal{T}(G))$)	**WebUK** ($\sigma(\mathcal{C}(G))$)	**WebUK** ($\sigma(\mathcal{T}(G))$)
26.5%	40.2%	12.7%	15.3%

We also assess the performance of using our algorithm for computing triangular-vertex-connectivity-based equivalence classes, compared with the state-of-the-art in-memory algorithm [Batagelj and Zaveršnik 2007] (also denoted by **In-Mem** here for simplicity).

Table XI. Triangular Vertex Connectivity

	LJ	**USRD**	**BTC**	**WebUK**
TL-Sequential	173.1	11.5	380.2	3670.3
In-Mem	138.1	9.0	N.A.	N.A.

Table XI shows that the running time of our algorithm is comparable with that of In-Mem on the smaller datasets LJ and USRD. But on the larger datasets, BTC and WebUK, In-Mem becomes infeasible due to insufficient memory while our algorithm still records high efficiency. The result again demonstrates that our algorithm is I/O-efficient for processing large graphs.

10. RELATED WORK

The algorithms for triangle listing or counting can be categorized into *exact* algorithms and *approximation* algorithms.

The first non-trivial exact algorithm was a spanning-tree-based algorithm [Itai and Rodeh 1977; 1978], which achieves a running time of $O(|E_G|^{1.5})$. This is the optimal worst-case time complexity for an in-memory algorithm for triangle listing because there are $O(|V_G|^3) = O(|E_G|^{1.5})$ triangles in a graph in the worst case. However, a number of practical fast algorithms have been proposed that use vertex ordering and efficient data structures such as lookup tables to facilitate the intersection of the adjacency lists of the neighboring vertices [Schank and Wagner 2005; Schank 2007; Latapy 2008]. For triangle counting, the number of triangles can be counted in $O(|E_G|^{1.41})$ time with a fast matrix multiplication algorithm [Alon et al. 1997]. The basic triangle listing algorithm was also extended to count triads (directed subgraph with three vertices) in directed graph [Batagelj and Mrvar 2001]. Maintaining the number of triangles in a dynamic graph was discussed in [Eppstein and Spiro 2009]. All the aforementioned algorithms are in-memory algorithm and require at least $O(|V_G|+|E_G|)$ memory space.

Approximation algorithms have been proposed for triangle counting in large graphs that cannot fit in main memory. Accurate streaming algorithms [Alon et al. 1999; Bar-Yossef et al. 2002; Coppersmith and Kumar 2004; Buriol et al. 2006] and sampling algorithm [Tsourakakis et al. 2009] have been proposed to estimate the total number of triangles in a graph. More closely related to our work is the semi-streaming algorithm that estimates the number of triangles formed locally at each vertex in a graph [Becchetti et al. 2008; 2010]. All these algorithms, however, cannot handle triangle listing, which has a broader range of applications.

For triangle counting in a large graph that cannot fit in main memory, parallel algorithms that apply the MapReduce framework were proposed recently [Suri and Vassilvitskii 2011]. Their algorithms are exact and do not require to keep the entire input graph in main memory at each individual machine. Our approach is orthogonal to their approach of parallelization for triangle counting. However, we note that the MapReduce framework may not be suitable for the task of triangle counting. As a cross reference, the experiments in [Suri and Vassilvitskii 2011] show that, for the same dataset LJ, the fastest of their parallel algorithms running on 1,636 machines takes 5.33 minutes, while our algorithms running on a single machine use less than 0.5 minute. Note that their algorithm is proven to be work efficient. The much longer running time may be due to the hidden cost needed in the shuffling phase between Mappers and Reducers, because to make triangle counting work in a MapReduce framework, the algorithm has to produce a huge amount of intermediate data. Moreover, many researchers or the average users may not have access to or may not be willing to pay for the computing resource of hundreds to thousands of machines. On the contrary, our algorithms are efficient and require only one ordinary machine.

Compared with the preliminary version of this paper [Chu and Cheng 2011], we proposed a new graph partitioning algorithm for the purpose of triangle listing; that is, the randomized graph partitioning. The randomized graph partitioning addresses the limitations of both the sequential and dominating-set-based graph partitioning algorithms. It gives a theoretical guarantee on the I/O complexity of triangle listing (unlike sequential partitioning), and at the same time has a bound on the memory usage (unlike dominating-set-based partitioning). This paper also removes an assumption made in the preliminary version [Chu and Cheng 2011] and proposes an I/O-efficient algorithm for triangle listing in graphs with extremely high degree vertices. These two new additions are of significant importance with respect to the design of I/O-efficient algorithms, making our work not only practically useful but also theoretically feasible for the problem of triangle listing in very large graphs that cannot fit in main memory. In addition, we also discussed the update of triangles in a dynamic network, and proposed a disk-based data structure for efficient update and storage of the set of triangles.

Apart from triangle listing, Algorithms for listing more complex substructures, such as maximal cliques, in massive networks were also studied [Cheng et al. 2010; 2011]. Their algorithms also partition the input graph and then perform local computation. However, for each subgraph in their partition, they need to scan the graph once to extract the subgraph, while we extract each subgraph in our partition as we sequentially scan the graph.

11. CONCLUSIONS

We presented an I/O-efficient algorithm for exact triangle listing. To avoid random disk access, our algorithm partitions the input graph and only process one subgraph in the partition each time. By carefully extracting the subgraphs, we proved that triangle listing in those local subgraphs gives globally correct and complete result. We devised three effective partitioning strategies, one achieving high efficiency in practice while the other two bounding the I/O complexity theoretically.

Our experimental results on large graphs with up to 106 million vertices and 1,877 million edges show that our algorithm is either significantly more efficient or more accurate than the state-of-the-art approximation algorithm for local triangle counting [Becchetti et al. 2008]. Our results on a dynamic network show that the triangle-tree is able to support frequent updates efficiently. The results also demonstrate the efficiency of applying our algorithm on computing various network measures such as clustering coefficients and transitivity, as well as equivalence classes of the graph based on triangular vertex connectivity. Thus, we believe that our work can benefit many other applications in processing large graphs.

Finally, the graphs we study in this paper are undirected graphs. However, our algorithms can be extended to list triangles in directed graphs as follows. When the input graph G is a directed graph, the triangles in G also contain directed edges. In general, there are 7 types of triangles in a directed graph by considering the combinations of all possible directions of the edges (note that an edge in a triangle can be either bi-directional or uni-directional). To list all such triangles, we can convert G into an undirected graph and then apply our algorithms to list the set of all undirected triangles. Note that all directed triangles are also present in this set. To obtain the directions of the edges in each triangle, a post-processing is needed by scanning the directed edges of G to assign the directions of the edges to the triangles. When G cannot fit in memory, we read $O(M)$ edges into memory each time, scan the set of triangles once to assign the directions of these edges, and then continue to read the next $O(M)$ edges until all edges of G are processed. The I/O complexity required for the post-processing is $O(\frac{|E_G|}{M} scan(|\triangle(G)|))$.

For future work, we plan to study the application of triangle listing for the computation of *k-trusses* [Cohen 2009]. A k-truss of a graph G is the largest subgraph of G in which every edge is contained in at least $(k-2)$ triangles within the subgraph. A k-truss can be considered as the “core” of a k-core [Cheng et al. 2011], and thus more useful than k-core in many applications such as visualization and fingerprinting of large-scale networks, interpretation of cooperative processes in complex networks, and analysis of network connectivity.

ACKNOWLEDGMENTS

The authors would like to thank the reviewers of this paper as well as of the preliminary KDD 2011 submission, for their constructive comments that have helped improve the quality of the paper significantly.

REFERENCES

ABOU-RJEILI, A. AND KARYPIS, G. 2006. Multilevel algorithms for partitioning power-law graphs. In *IPDPS*.

AGGARWAL, A. AND VITTER, JEFFREY, S. 1988. The input/output complexity of sorting and related problems. *Commun. ACM 31,* 9, 1116–1127.

ALON, N., MATIAS, Y., AND SZEGEDY, M. 1999. The space complexity of approximating the frequency moments. *J. Comput. Syst. Sci. 58,* 1, 137–147.

ALON, N., YUSTER, R., AND ZWICK, U. 1997. Finding and counting given length cycles. *Algorithmica 17,* 3, 354–364.

ANDREEV, K. AND RACKE, H. 2004. Balanced graph partitioning. In *SPAA*. 120–124.

BAR-YOSSEF, Z., KUMAR, R., AND SIVAKUMAR, D. 2002. Reductions in streaming algorithms, with an application to counting triangles in graphs. In *SODA*. 623–632.

BATAGELJ, V. AND MRVAR, A. 2001. A subquadratic triad census algorithm for large sparse networks with small maximum degree. *Social Networks 23,* 3, 237–243.

BATAGELJ, V. AND ZAVERŠNIK, M. 2007. Short cycle connectivity. *Discrete Mathematics 307,* 3-5, 310 – 318.

BECCHETTI, L., BOLDI, P., CASTILLO, C., AND GIONIS, A. 2008. Efficient semi-streaming algorithms for local triangle counting in massive graphs. In *KDD*. 16–24.

BECCHETTI, L., BOLDI, P., CASTILLO, C., AND GIONIS, A. 2010. Efficient algorithms for large-scale local triangle counting. *TKDD 4,* 3.

BUEHRER, G. AND CHELLAPILLA, K. 2008. A scalable pattern mining approach to web graph compression with communities. In *WSDM*. 95–106.

BURIOL, L. S., FRAHLING, G., LEONARDI, S., MARCHETTI-SPACCAMELA, A., AND SOHLER, C. 2006. Counting triangles in data streams. In *PODS*. 253–262.

CHENG, J., KE, Y., CHU, S., AND ÖZSU, M. T. 2011. Efficient core decomposition in massive networks. In *ICDE*. 51–62.

CHENG, J., KE, Y., FU, A. W.-C., YU, J. X., AND ZHU, L. 2010. Finding maximal cliques in massive networks by h*-graph. In *SIGMOD Conference*. 447–458.

CHENG, J., KE, Y., FU, A. W.-C., YU, J. X., AND ZHU, L. 2011. Finding maximal cliques in massive networks. *ACM Trans. Database Syst. 36,* 4, 21.

CHU, S. AND CHENG, J. 2011. Triangle listing in massive networks and its applications. In *KDD*. 672–680.

COHEN, J. 2009. Graph twiddling in a mapreduce world. *Computing in Science and Engineering 11*, 29–41.

COPPERSMITH, D. AND KUMAR, R. 2004. An improved data stream algorithm for frequency moments. In *SODA*. 151–156.

ECKMANN, J.-P. AND MOSES, E. 2002. Curvature of co-links uncovers hidden thematic layers in the world wide web. *PNAS 99*, 5825–5829.

EPPSTEIN, D. AND SPIRO, E. S. 2009. The h-index of a graph and its application to dynamic subgraph statistics. In *WADS*. 278–289.

FALOUTSOS, M., FALOUTSOS, P., AND FALOUTSOS, C. 1999. On power-law relationships of the internet topology. In *SIGCOMM*. 251–262.

FEIGE, U. AND KRAUTHGAMER, R. 2000. A polylogarithmic approximation of the minimum bisection. In *FOCS*. 105–115.

FEIGE, U., KRAUTHGAMER, R., AND NISSIM, K. 2000. Approximating the minimum bisection size (extended abstract). In *STOC*. 530–536.

FIDUCCIA, C. M. AND MATTHEYSES, R. M. 1982. A linear time heuristic for improving network partitions. In *IEEE Design Automation Conference*.

FRITZKE, B. 1993. A self-organizing network for unsupervised learning. *TR-03-026 42*.

GRANOVETTER, M. 1973. The strength of weak ties. *American Journal of Sociology 78,* 6, 1360–1380.

ITAI, A. AND RODEH, M. 1977. Finding a minimum circuit in a graph. In *STOC*. 1–10.

ITAI, A. AND RODEH, M. 1978. Finding a minimum circuit in a graph. *SIAM J. Comput. 7,* 4, 413–423.

KARYPIS, G. AND KUMAR, V. 1999. Parallel multilevel k-way partitioning for irregular graphs. *SIAM Review 41,* 2, 278–300.

KERNIGHAM, B. W. AND LIN, S. 1970. An efficient heuristic procedure for partitioning graphs. *Bell System Technical Journal*, 291–308.

LATAPY, M. 2008. Main-memory triangle computations for very large (sparse (power-law)) graphs. *Theor. Comput. Sci. 407,* 1-3, 458–473.

MILO, R., SHEN-ORR, S., ITZKOVITZ, S., KASHTAN, N., CHKLOVSKII, D., AND ALON, U. 2002. Network Motifs: Simple Building Blocks of Complex Networks. *Science 298,* 5594, 824–827.

NEWMAN, M. E. J. 2003. The structure and function of complex networks. *SIAM Review Vol. 45, No. 2*, 167–256.

NEWMAN, M. E. J., WATTS, D. J., AND STROGATZ, S. H. 2002. Random graph models of social networks. *PNAS 99*, 2566–2572.

SCHANK, T. 2007. Algorithmic aspects of triangle-based network analysis. *Ph.D. Dissertation, Universität Karlsruhe, Fakultät für Informatik*.

SCHANK, T. AND WAGNER, D. 2005. Finding, counting and listing all triangles in large graphs, an experimental study. In *WEA*. 606–609.

SURI, S. AND VASSILVITSKII, S. 2011. Counting triangles and the curse of the last reducer. In *WWW*. 607–614.

TAUBIN, G. AND ROSSIGNAC, J. 1998. Geometric compression through topological surgery. *ACM Trans. Graph. 17,* 2, 84–115.

TSOURAKAKIS, C. E., KANG, U., MILLER, G. L., AND FALOUTSOS, C. 2009. Doulion: counting triangles in massive graphs with a coin. In *KDD*. 837–846.

WANG, N., ZHANG, J., TAN, K.-L., AND TUNG, A. K. H. 2010. On triangulation-based dense neighborhood graphs discovery. *PVLDB 4,* 2, 58–68.

WASSERMAN, S. AND FAUST, K. 1994. Social network analysis: Methods and applications. *Cambridge University Press*.

WATTS, D. J. AND STROGATZ, S. H. 1998. Collective dynamics of ‘small-world’ networks. *Nature 393,* 6684, 440–442.