Triangle Listing in Massive Networks and Its Applications

Shumo Chu
Nanyang Technological University, Singapore
shumo.chu@acm.org

James Cheng
Nanyang Technological University, Singapore
j.cheng@acm.org

ABSTRACT

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 coefficient, transitivity, triangular connectivity, 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 into main memory, triangle listing requires random disk accesses that can incur prohibitively large I/O cost. Some 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 local triangle estimation algorithm.

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

Keywords

Triangle Listing, Triangle Counting, Clustering Coefficient, Large Graphs, Massive Networks

1. INTRODUCTION

We study the problem of **triangle listing** in a simple undirected graph G, that is, *listing all triangles in* G. Our focus is to design efficient algorithm for triangle listing when G is too large to fit into 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 definition of many important measures for network analysis, such as the clustering coefficient (of a single vertex and of the entire network) [36], transitivity [35, 28], triangular connectivity [8], 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 in the discovery of dense subgraphs [34], in the detection of spamming activities [9], in the study of motif occurrences [27], in the uncovering of hidden thematic relationships in the Web [15], 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 [22, 23, 4, 7, 30, 29, 26, 16] 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. Unfortunately, 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.

For handling large graphs that cannot fit into main memory, a number of approximation algorithms have been proposed [3, 6, 14, 10, 9, 33]. However, all these algorithms are restricted to 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 more accurate [3, 6, 14, 10, 33], but their applications are also much more limited than those 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 the state-of-the-art algorithm [9] is still not accurate enough. 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. Thus, random access to the graph stored on disk is required, which incurs huge I/O cost.

Our algorithm iteratively partitions the input graph G into a set of subgraphs that fit into memory and processes triangle listing in

KDD'11, August 21–24, 2011, San Diego, California, USA.

each local subgraph. 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 Types 1 and 2 by a new partition at the next iteration. To limit the total number of iterations, we show that we can remove all edges in each subgraph at the end of each iteration, thus shrinking G until it becomes empty. We propose two effective algorithms for graph partitioning for the task of triangle listing in our framework, one achieving high efficiency in practice while the other giving theoretical bound on the total number of iterations (also efficient in practice).

We evaluate 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 [26] and the most related approximation algorithm [9]. Our algorithm achieves comparable performance with the in-memory algorithm when the graph can fit into memory. For large graphs that cannot fit into memory, the error rate of the approximation algorithm [9] can be as large as 95% to 133% while ours returns exact result, with comparable running time and memory. When we attempt to attain a lower error rate, e.g., at 50%, the approximation algorithm is already orders of magnitude slower than our exact algorithm.

Paper Organization. Section 2 gives the notations and problem definition. Section 3 describes an in-memory algorithm. Section 4 discusses our main algorithm. Section 5 presents two applications of our algorithm. Section 6 reports the experimental results. Section 7 gives the related work. Section 8 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 the adjacency list representation, with vertices ordered according to their ID.

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

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

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

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

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

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

Equation 3 holds because every triangle $\triangle_{uvw}$ is counted three times for 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.

Table 1: Notations

Symbol	Description
$G = (V_G, E_G)$	A simple undirected graph
$adj_G(v)$	The set of adjacent vertices of v in G
$deg_G(v)$	Degree of v in G
$\triangle_{uvw}$	A triangle formed by u, v and w
$\triangle(v)$	The set of triangles that contains v (Eq. 1)
$N_\triangle(v)$	The triangle number of v, $N_\triangle(v) = \|\triangle(v)\|$
$\triangle(G)$	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	Available main memory size
B	Disk block size
$scan(N)$	$\Theta(N/B)$ I/Os
$sort(N)$	$\Theta(\frac{N}{B} \log_{M/B} \frac{N}{B})$ I/Os

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

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

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

The following example illustrates the concepts.

Example 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 $deg_G(e) = 4$. We can also easy find $\triangle(G) = \{\triangle_{abc}, \triangle_{bcg}, \triangle_{bgi}, \triangle_{dej}, \triangle_{efh}, \triangle_{jkl}\}$ and $N_\triangle(G) = |\triangle(G)| = 6$. □

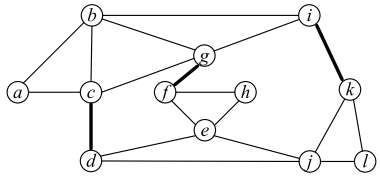

Figure 1: A Graph G

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 into main memory, i.e., $(|V_G| + |E_G|) > M$, where M is the size of available main memory.

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

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

Table 1 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 highlight the difficulty for triangle listing when main memory is insufficient.

We sketch the algorithm in Algorithm 1. Assume that the input graph G is in its adjacency-list representation and the vertices are

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. $\triangle(G) \leftarrow (\triangle(G) \cup \{\triangle_{uvw}\})$;
6. **return** $\triangle(G)$;

ordered in ascending order of their vertex ID, which is the most common format used for graph storage. The algorithm intersects the adjacency-list of each vertex u with the adjacency-list of u's neighbor v. 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. 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 (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 [26], except that their work orders the vertices in non-increasing order of their degree, which requires pre-processing.

When G cannot fit into main memory, however, the I/O cost becomes a bottleneck. Most existing in-memory algorithms [22, 23, 4, 7, 29, 26, 16] require random access to each $adj_G(v)$ for each $v \in adj_G(u)$ (as in Line 4 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. Others [30] use an additional array for each vertex in G and the total size of these arrays is in the 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 into 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 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 into memory, we could only load a portion (i.e., a subgraph) of G that can fit into memory each time. Thus, our algorithm iteratively performs triangle listing in a subgraph of G that fits in 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 into memory;
  - Load each G_i into 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-list) in the graph.

The concept is simple but there are a number of challenging technical issues: (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 1. *Let $\mathcal{P} = \{G_1, \ldots, G_i, \ldots, G_p\}$ be a partition of 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: (1) they 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 (WLOG, we may assume that $u, v \in V_{G_i}, w \in V_{G_j}, i \ne j$); (3) they are in three different subgraphs G_i, G_j, and G_k (WLOG, we may assume that $u \in V_{G_i}, v \in V_{G_j}, w \in V_{G_k}, i < j < k$). The three cases correspond to the three types $\triangle 1$, $\triangle 2$, and $\triangle 3$, and thus $\triangle(G) \subseteq (\triangle 1 \cup \triangle 2 \cup \triangle 3)$. □

We call triangles in $\triangle 1$, $\triangle 2$, and $\triangle 3$, *Type 1*, *Type 2*, and *Type 3* triangles. The following example illustrates the concept of the three types of triangles.

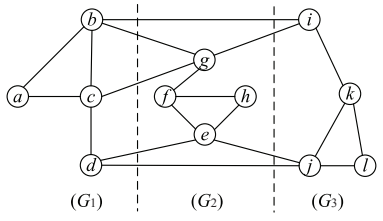

Figure 2: A partition of G in Figure 1: $\mathcal{P} = \{G_1, G_2, G_3\}$

Example 2. Figure 2 shows a partition, $\mathcal{P} = \{G_1, G_2, G_3\}$, of the graph shown in Figure 1. In the figure, $\triangle_{abc}$, $\triangle_{efh}$ and $\triangle_{jkl}$

are Type 1 triangles because all the three vertices of each triangle are in the same subgraph. We only have one Type 2 triangle, $\triangle_{bcg}$, because its vertices are in two subgraphs in $\mathcal{P}$. We have two Type 3 triangles, $\triangle_{bgi}$ and $\triangle_{dej}$, because all the three vertices of each triangle are in three different subgraphs in $\mathcal{P}$. □

According to Lemma 1, a triangle $\triangle_{uvw}$ can be listed by searching any subgraph G_i alone 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 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*.

Definition 1 (EXTENDED SUBGRAPH). *Let $H = (V_H, E_H)$ be a subgraph of G. An* extended subgraph *of H in G, denoted by H^+, is a* directed *subgraph 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 $E_{H^+} = \{(u, v) : (u, v) \in E_G, u \in V_H\}$.*

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. In this paper, we assume that if $V_H = \{v\}$, for any $v \in V$, then the corresponding H^+ fits into memory. We give an example of extended subgraph as follows.

Example 3. Figure 3 depicts the extended subgraphs of G_1, G_2, and G_3 in Figure 2. The shaded vertices are the vertices in each G_i, while the directed edges show the extension to vertices outside each G_i. □

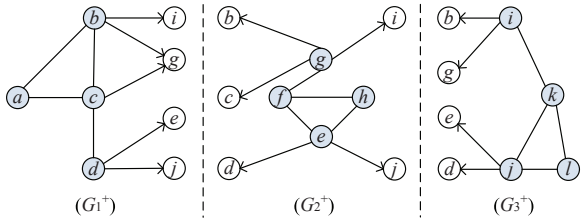

Figure 3: The extended subgraphs of G_1, G_2, G_3 in Figure 2

Based on Definition 1, we have the following lemma for triangle listing in an extended subgraph.

LEMMA 2. *Let H^+ be an extended subgraph of a subgraph H of G. Then:*

- *Let $\triangle 1(H^+) = \{\triangle_{uvw} : \triangle_{uvw} \in \triangle 1, \ u, v, w \in V_H\}$. Then, $\forall \triangle_{uvw} \in \triangle 1(H^+)$, $\triangle_{uvw}$ can be listed by searching H^+ alone.*
- *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 $u, v \in V_H$, $w \in adj_{H^+}(u)$, $w \in adj_{H^+}(v)$, and $\triangle_{uvw}$ 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 $\triangle 2(H^+)$. □

Algorithm 2 *I/O-Efficient Triangle Listing*

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

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;

Algorithm 3 *Triangle Listing in Extended Subgraph*

Input: An extended subgraph $H^+ = (V_{H^+}, E_{H^+})$
Output: A listing of $\triangle 1(H^+)$ 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 $\triangle_{uvw}$;

Lemma 2 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.

The above scheme lists all Type 1 and Type 2 triangles but still misses 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. 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 2, where $\mathcal{P}$ is the new partition of G at each iteration. (G_i^+ can be easily obtained along with the computation of $\mathcal{P}$, which we discuss in Section 4.3.)

At the end of each iteration, we remove all edges in each G_i in the current $\mathcal{P}$ 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 G_i^+ of each $G_i \in \mathcal{P}$. Thus, Algorithm 2 iteratively converts the old set of Type 3 triangles in the previous iteration into Type 1 and Type 2 triangles with respect to the new partition at the current iteration. This process continues until all edges in G are removed.

Note that another purpose of graph partition is to make sure that each subgraph in $\mathcal{P}$ is small enough to fit into 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 G at any iteration is small enough to fit into memory, Step 2 of Algorithm 2 computes a partition consisting of only one element, 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. 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 it cannot be listed elsewhere.

We now prove the correctness and completeness of Algorithm 2.

THEOREM 1. *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 2 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 G, and (3) the already listed triangles will not be listed again in 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. □

4.3 Graph Partitioning 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 edge as possible at each iteration of Algorithm 2 in order to reduce the number of total iterations.

Graph partitioning that fulfills the above two objectives, however, is known to be APX-hard [5] when the number of subgraphs in the partition is more than 2. There have been a number of approximation algorithms proposed [25, 19, 24, 17, 18, 1], but they are in-memory algorithms that are not suitable for triangle listing in massive networks that cannot fit into memory. Instead, we need an efficient algorithm that partitions a large graph with limited memory consumption. We devise two efficient graph partitioning algorithms that require only one scan or two scans of the input graph and has linear CPU time complexity.

4.3.1 Sequential Graph Partitioning

Sequential graph partitioning is a simple but very efficient 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 is obtained sequentially one after another as we read G, it allows pipelining such that we can process triangle listing for each subgraph as soon as it is produced, rather than starting triangle listing until the entire partitioning process finishes.

Sequential graph partitioning is effective when graphs exhibit 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 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.

Procedure 4 *DominatingSet(G)*

1. Create a bit array A of size $|V_G|$ and set each bit to 0;
2. **for each** v in G, where $A[v] = 0$, **do**
3. Add v to D and mark v and all v's neighbors as 1 in A;
4. **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 invoking *DominatingSet*(G);
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;
6. **return** $\mathcal{P}$;

4.3.2 Dominating-Set-based Graph Partitioning

Sequential graph partitioning may be efficient in practice but it gives no 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*.

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. More specifically, we devise a one-pass algorithm to compute a dominating set for G as shown in Procedure 4.

In Procedure 4, we first initialize a bit array A of size $|V_G|$ and set each bit 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 $p = O((|V_G| + |E_G|)/M)$ subsets and create p initial subgraphs in $\mathcal{P}$. 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* $\mathcal{P}$, which can be easily obtained by scanning $adj_G(v)$. 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 Algo-

rithm 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 of size $(|V_G| \log_2 p)$ bits. However, in this case, the algorithm is considered as a semi-external-memory algorithm.

Dominating-set-based graph partitioning not only groups neighborhood vertices together, but more importantly it gives a 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.

LEMMA 3. *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 $|E_G|/p$.*

PROOF. Let $P_{t(v)}$ be the current partition P at the time when a vertex v is added to 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 $(deg_{\mathcal{P}(v)_{t(v)}}/p)$ 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 Procedure 4. The result thus follows. □

We further show that Lemma 3 implies an upper bound on the total number of iterations required for Algorithm 2 in Lemma 4 in the following subsection.

4.4 Bounding I/O Complexity

We now analyze the overall complexity of our algorithm. As discussed in Section 4.3, graph partitioning at each iteration of Algorithm 2 requires $O(scan(|V_G|+|E_G|))$ I/Os. For triangle listing at each iteration, we only read once the extended subgraph of each subgraph in the partition into memory. Thus, the overall I/O complexity for each iteration is $O(scan(|V_G|+|E_G|))$ (for a shrinking G). Note that both partitioning algorithms in Section 4.3 actually output the extended subgraphs.

From the above analysis, the overall I/O complexity of Algorithm 2 depends on the total number of iterations needed. If we use sequential graph partitioning, the number of iterations depends largely on the locality of the graph data, which varies for different datasets. If we use dominating-set-based graph partitioning, then we can obtain an upper bound on the number of iterations (but require $|V_G| \log_2 p$ bits of memory) for Algorithm 2 as follows.

LEMMA 4. *If Algorithm 2 partitions G by Algorithm 5, then the number of iterations in Algorithm 2 is $O(|E_G|/M)$.*

PROOF. According to Lemma 3, the number of intra-partition edges of $\mathcal{P}$ is at least $|E_G|/p$, which means that at least $|E_G|/p \approx |E_G|/(|E_G|/M) = M$ edges can be removed at the end of each iteration since $p = O((|V_G|+|E_G|)/M) \approx |E_G|/M$. Thus, the total number of iterations in Algorithm 2 is approximately $|E_G|/M$. □

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

PROOF. According to Lemma 4, at least M edges are removed from G at the end of each iteration of Algorithm 2. Thus, at the start of the i-th iteration, $(i-1)M$ edges have been removed from the original G. Summing up, the overall complexity of Algorithm 2 is $O(\sum_{i=1}^{|E_G|/M}(scan(|V_G| + |E_G|) - (i-1)M)) = O(\frac{|E_G|}{M} scan(|V_G| + |E_G|) - (\frac{|E_G|}{M})^2 M)$. □

Algorithm 6 *Triangle Counting, Clustering Coeff., 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. $\quad N_\triangle(x) \leftarrow N_\triangle(x) + 1$, for $x \in \{u, v, w\}$;
5. **for each** $v \in V_G$ **do**
6. $\quad \mathcal{C}(v) \leftarrow 2N_\triangle(v)/deg_G(v)(deg_G(v) - 1)$;
7. $\quad \mathcal{C}(G) \leftarrow \mathcal{C}(G) + \mathcal{C}(v)$;
8. $\quad N_\triangle(G) \leftarrow N_\triangle(G) + N_\triangle(v)$;
9. $\quad N_\vee(G) \leftarrow N_\vee(G) + deg_G(v)(deg_G(v) - 1)/2$;
10. $\mathcal{C}(G) \leftarrow \mathcal{C}(G)/|V_G|$;
11. $\mathcal{T}(G) \leftarrow 3N_\triangle(G)/N_\vee(G)$;

Finally, we remark that the CPU time complexity for triangle listing at each iteration of Algorithm 2 is the same as that of the counter-part in-memory algorithm with the same input graph. We thus refer the readers to the related work [26] for details.

5. 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 into main memory.

5.1 Triangle Counting, Clustering Coefficients, and Transitivity

Our algorithm can be readily applied to compute triangle counting, clustering coefficients, and transitivity. For completeness, we first give the definition of the concepts.

Clustering coefficient [36], which is a popular index for network analysis, is defined as follows.

Definition 2 (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*

$$\mathcal{C}(v) = \frac{N_\triangle(v)}{N_\vee(v)}. \tag{5}$$

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

$$\mathcal{C}(G) = \frac{1}{|V_G|} \sum_{v \in V_G} \mathcal{C}(v). \tag{6}$$

Transitivity [35, 28] is defined as follows.

Definition 3 (TRANSITIVITY). *The transitivity of a graph G, denoted by $\mathcal{C}(G)$, is defined as*

$$\mathcal{T}(G) = \frac{N_\triangle(G)}{\frac{1}{3}\sum_{v \in V_G} N_\vee(v)}. \tag{7}$$

Instead of first computing all triangles and performing a post-processing, our algorithm can be pipelined to compute all the above measures as shown in Algorithm 6. The algorithm is self-explanatory.

Lines 5-9 of Algorithm 6 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.

5.2 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 [21, 8]. Triangular vertex connectivity is formally defined as follows [29].

Definition 4 (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 [15, 20, 32, 36]. 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 4. In the graph in Figure 1, there are two equivalence classes defined by triangular vertex connectivity, $\{a, b, c, g, i\}$ and $\{d, e, f, h, j, k, l\}$, which can be obtained by removing the three edges (in bold lines) that are not part of any triangle. Removing any edge from the induced subgraph by any class does not disconnect the vertices in the class, which shows a stronger connectivity. □

The existing algorithm for computing the equivalence classes of $\mathcal{V}^{\triangle}$ is based on triangle listing [8]. However, the algorithm requires random disk access when the graph is stored in disk.

The result of Algorithm 2 can be easily pipelined to compute the equivalence classes of $\mathcal{V}^{\triangle}$. 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 $A[u] \neq A[v]$, WLOG assuming $A[u] < A[v]$, 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 small 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 Algorithm 2.

6. EXPERIMENTAL RESULTS

We compare our algorithms with the semi-streaming local triangle estimating algorithm (denoted by **Semi-stream**) [9] and the state-of-the-art in-memory triangle listing algorithm (denoted by **In-mem**) [26]. We run the experiments on a machine with an Intel Xeno 2.67GHz CPU and 4GB RAM, and we use g++ 4.4.5 compiler on 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.livejournal.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". We give the number of vertices and edges, and the storage size on disk, of the datasets in Table 2.

Table 2: Datasets

	LJ	USRD	WebUK	BTC
$\|V_G\|$	4.8M	24M	106M	165M
$\|E_G\|$	69M	58M	1,877M	773M
Disk size	809.1MB	969.6MB	20.3GB	10.0GB

6.1 Effectiveness of Partitioning Algorithms

We first show the effectiveness of two graph partitioning algorithms, *sequential graph partitioning* (**Seq**) and *dominating-set-based graph partitioning* (**DS**). We set the available memory size M for partitioning to be $512MB$, $1GB$, $2GB$, and $4GB$, respectively. Table 3 reports the number of iterations Algorithm 2 takes by adopting Seq or DS, on the two large datasets BTC and WebUK. The theoretical upper bound on the number of iterations by adopting DS is also shown.

Table 3: Number of Iterations of Algorithm 2

	M=512MB	M=1GB	M=2GB	M=4GB
BTC (Seq)	$\rightarrow \infty$	7	2	1
BTC (DS)	6	3	2	1
BTC (Theoretical)	19	10	5	3
WebUK (Seq)	3	2	2	1
WebUK (DS)	3	2	2	1
WebUK (Theoretical)	29	15	7	4

The result shows that Seq is effective in practice. The number of iterations of Algorithm 2 by adopting Seq is as small as that by adopting DS in most cases, but as shown in Table 4 in Section 6.2 adopting Seq is more efficient than adopting DS since DS requires two more scans of G at each iteration of Algorithm 2.

However, when the available memory becomes smaller (M = 512MB), adopting Seq does not terminate since there is a point that all triangles are Type 3 triangles w.r.t. the partition; thus no more edges can be removed. Such a situation may also happen to other datasets that exhibit low locality. In this case, DS 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 adopting DS is always less than its theoretical upper bound.

6.2 Triangle Listing

We now report the performance of Algorithm 2 by adopting Seq (denoted by **TL-Seq**) and by adopting DS (denoted by **TL-DS**), compared with In-mem and Semi-stream. We set the available memory size to be 2GB.

Table 4 reports the running time of the different 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 BTC and WebUK. In-mem runs out of memory for BTC and WebUK (BTC and WebUK requires about 10GB and 20GB of memory).

Table 4: Running time (wall-clock time in seconds)

	LJ	USRD	BTC	WebUK
TL-Seq	29.63	6.24	350	2411
TL-DS	29.43	12.46	412	2503
In-mem	32.98	6.68	N.A.	N.A.
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 algorithms is very competitive. When the graphs can fit into memory, our algorithms have comparable performance compared with In-mem. When the graphs are too large to fit into memory, our algorithms demonstrate great advantage over Semi-stream, which makes multiple passes over the data as do our algorithms.

Let $\overline{\sigma}(N_\triangle(v))$ be the *average approximation error rate* for $N_\triangle(v)$, defined as ($|$approximate value $-$ exact value$|$/exact value) averaged over all vertices. We find that Semi-stream takes prohibitively long time to obtain a low $\overline{\sigma}(N_\triangle(v))$. Table 4 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 both our algorithms for $\overline{\sigma}(N_\triangle(v)) \approx 0.8$ and up to orders of magnitude slower for $\overline{\sigma}(N_\triangle(v)) \approx 0.5$.

From another angle of comparison, if we choose a setting for Semi-stream that it takes slightly longer time than our algorithm TL-Seq, we also report its error rate in Table 5. The result shows that at comparable time, the error rate $\overline{\sigma}(N_\triangle(v))$ of Semi-stream increases significantly.

Table 5: Error rate of Semi-stream at comparable time as TL-Seq

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

6.3 Performance on Applications

When the graphs are too large to fit into memory, our algorithm shows significant advantages over approximation algorithms. Table 6 reports the error rate of clustering coefficient of a network and of transitivity 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, we obtain exact results at comparable time.

Table 6: 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 [8] (also denoted by **In-mem** here for simplicity).

Table 7 shows that the running time of our algorithm is comparable with In-mem for LJ and USRD. But for the larger datasets, BTC and WebUK, In-mem runs out of memory while our algorithm still records high efficiency. The result again demonstrates our algorithms are I/O-efficient for processing large graphs.

Table 7: Triangular Vertex Connectivity

	LJ	USRD	BTC	WebUK
TL-Seq	173.1	11.5	380.2	3670.3
In-mem	138.1	9.0	N.A.	N.A.

7. 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 [22, 23], which achieves a running time of $O(|E_G|^{1.5})$. The asymptotic complexity of in-memory exact triangle listing has not been improved since then. 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 [30, 29, 26]. For triangle counting, the number of triangles can be counted in $O(|E_G|^{1.41})$ time with a fast matrix multiplication algorithm [4]. The basic triangle listing algorithm was also extended to count triads (directed subgraph with three vertices) in directed graph [7]. Maintaining the number of triangles in a dynamic graph was discussed in [16]. All the aforementioned algorithms are in-memory algorithm and require at least $O(|V_G| + |E_G|)$ space.

Approximation algorithms have been proposed for triangle counting in large graphs that cannot fit in memory. Accurate streaming algorithms [3, 6, 14, 10] and sampling algorithm [33] have been proposed to estimate the total number of triangles in a graph. More closely related to estimate the number of triangles formed locally at each vertex in a graph. All these algorithms, however, cannot handle triangle listing, which has a broader range of applications.

Very recently, a parallel algorithm for triangle counting using MapReduce framework [31] has been proposed. Their algorithm is exact and do not require to keep the entire input graph in memory at each individual machine. Algorithms for listing more complex subgraphs are also studied in [13, 12].

8. CONCLUSIONS

We propose an I/O-efficient algorithm for exact triangle listing. To avoid random disk access, we partition the input graph and only process one subgraph in the partition each time. By carefully extracting the subgraphs, we prove that triangle listing in those local subgraphs gives globally correct and complete result. We devise two effective partitioning strategies, one achieving high efficiency in practice while the other 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 significantly more efficient than the state-of-the-art approximation algorithm for local triangle counting [9]: at comparable running time and memory, their algorithm records a very high error rate while ours returns the exact result. We 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.

For future work, we plan to study the relationship between the triangles and the k-cores [11] for analyzing massive networks.

Acknowledgment

The authors would like to thank the reviewers for their constructive comments. This research is supported in part by the AcRF Tier-1 Grant (M52020092) from Ministry of Education of Singapore.

9. REFERENCES

[1] A. Abou-Rjeili and G. Karypis. Multilevel algorithms for partitioning power-law graphs. In *IPDPS*, 2006.

[2] A. Aggarwal and S. Vitter, Jeffrey. The input/output complexity of sorting and related problems. *Commun. ACM*, 31(9):1116–1127, 1988.

[3] N. Alon, Y. Matias, and M. Szegedy. The space complexity of approximating the frequency moments. *J. Comput. Syst. Sci.*, 58(1):137–147, 1999.

[4] N. Alon, R. Yuster, and U. Zwick. Finding and counting given length cycles. *Algorithmica*, 17(3):354–364, 1997.

[5] K. Andreev and H. Racke. Balanced graph partitioning. In *SPAA*, pages 120–124, 2004.

[6] Z. Bar-Yossef, K. Hildrum, and F. Wu. Incentive-compatible online auctions for digital goods. In *SODA*, pages 964–970, 2002.

[7] V. Batagelj and A. Mrvar. A subquadratic triad census algorithm for large sparse networks with small maximum degree. *Social Networks*, 23(3):237–243, 2001.

[8] V. Batagelj and M. Zaveršnik. Short cycle connectivity. *Discrete Mathematics*, 307(3-5):310 – 318, 2007.

[9] L. Becchetti, P. Boldi, C. Castillo, and A. Gionis. Efficient semi-streaming algorithms for local triangle counting in massive graphs. In *KDD*, pages 16–24, 2008.

[10] L. S. Buriol, G. Frahling, S. Leonardi, A. Marchetti-Spaccamela, and C. Sohler. Counting triangles in data streams. In *PODS*, pages 253–262, 2006.

[11] J. Cheng, Y. Ke, S. Chu, and M. T. Özsu. Efficient core decomposition in massive networks. In *ICDE*, pages 51–62, 2011.

[12] J. Cheng, Y. Ke, A. W.-C. Fu, J. X. Yu, and L. Zhu. Finding maximal cliques in massive networks. *ACM Transactions on Database Systems*.

[13] J. Cheng, Y. Ke, A. W.-C. Fu, J. X. Yu, and L. Zhu. Finding maximal cliques in massive networks by h*-graph. In *SIGMOD Conference*, pages 447–458, 2010.

[14] D. Coppersmith and R. Kumar. An improved data stream algorithm for frequency moments. In *SODA*, pages 151–156, 2004.

[15] J.-P. Eckmann and E. Moses. Curvature of co-links uncovers hidden thematic layers in the world wide web. *PNAS*, 99:5825–5829, 2002.

[16] D. Eppstein and E. S. Spiro. The *h*-index of a graph and its application to dynamic subgraph statistics. In *WADS*, pages 278–289, 2009.

[17] U. Feige and R. Krauthgamer. A polylogarithmic approximation of the minimum bisection. In *FOCS*, pages 105–115, 2000.

[18] U. Feige, R. Krauthgamer, and K. Nissim. Approximating the minimum bisection size (extended abstract). In *STOC*, pages 530–536, 2000.

[19] C. M. Fiduccia and R. M. Mattheyses. A linear time heuristic for improving network partitions. In *IEEE Design Automation Conference*, 1982.

[20] B. Fritzke. A self-organizing network for unsupervised learning. *TR-03-026*, 42, 1993.

[21] M. Granovetter. The strength of weak ties. *American Journal of Sociology*, 78(6):1360–1380, 1973.

[22] A. Itai and M. Rodeh. Finding a minimum circuit in a graph. In *STOC*, pages 1–10, 1977.

[23] A. Itai and M. Rodeh. Finding a minimum circuit in a graph. *SIAM J. Comput.*, 7(4):413–423, 1978.

[24] G. Karypis and V. Kumar. Parallel multilevel k-way partitioning for irregular graphs. *SIAM Review*, 41(2):278–300, 1999.

[25] B. W. Kernigham and S. Lin. An efficient heuristic procedure for partitioning graphs. *Bell System Technical Journal*, pages 291–308, 1970.

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

[27] R. Milo, S. Shen-Orr, S. Itzkovitz, N. Kashtan, D. Chklovskii, and U. Alon. Network Motifs: Simple Building Blocks of Complex Networks. *Science*, 298(5594):824–827, 2002.

[28] M. E. J. Newman, D. J. Watts, and S. H. Strogatz. Random graph models of social networks. *PNAS*, 99:2566–2572, 2002.

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

[30] T. Schank and D. Wagner. Finding, counting and listing all triangles in large graphs, an experimental study. In *WEA*, pages 606–609, 2005.

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

[32] G. Taubin and J. Rossignac. Geometric compression through topological surgery. *ACM Trans. Graph.*, 17(2):84–115, 1998.

[33] C. E. Tsourakakis, U. Kang, G. L. Miller, and C. Faloutsos. Doulion: counting triangles in massive graphs with a coin. In *KDD*, pages 837–846, 2009.

[34] N. Wang, J. Zhang, K.-L. Tan, and A. K. H. Tung. On triangulation-based dense neighborhood graphs discovery. *PVLDB*, 4(2):58–68, 2010.

[35] S. Wasserman and K. Faust. Social network analysis: Methods and applications. *Cambridge University Press*, 1994.

[36] D. J. Watts and S. H. Strogatz. Collective dynamics of 'small-world' networks. *Nature*, 393(6684):440–442, 1998.