OPT: A New Framework for Overlapped and Parallel Triangulation in Large-scale Graphs

Jinha Kim, Wook-Shin Han[*], Sangyeon Lee, Kyungyeol Park, Hwanjo Yu
Pohang University of Science and Technology (POSTECH)
Pohang, South Korea
goldbar@postech.ac.kr, {wshan.postech,syleeDB,realpky85}@gmail.com,
hwanjoyu@postech.ac.kr

ABSTRACT

Graph triangulation, which finds all triangles in a graph, has been actively studied due to its wide range of applications in the network analysis and data mining. With the rapid growth of graph data size, disk-based triangulation methods are in demand but little researched. To handle a large-scale graph which does not fit in memory, we must iteratively load small parts of the graph. In the existing literature, achieving the ideal cost has been considered to be impossible for billion-scale graphs due to the memory size constraint. In this paper, we propose an overlapped and parallel disk-based triangulation framework for billion-scale graphs, OPT, which achieves the ideal cost by (1) full overlap of the CPU and I/O operations and (2) full parallelism of multi-core CPU and FlashSSD I/O. In OPT, triangles in memory are called the *internal triangles* while triangles constituting vertices in memory and vertices in external memory are called the *external triangles*. At the macro level, OPT overlaps the internal triangulation and the external triangulation, while it overlaps the CPU and I/O operations at the micro level. Thereby, the cost of OPT is close to the ideal cost. Moreover, OPT instantiates both vertex-iterator and edge-iterator models and benefits from multi-thread parallelism on both types of triangulation. Extensive experiments conducted on large-scale datasets showed that (1) OPT achieved the elapsed time close to that of the ideal method with less than 7% of overhead under the limited memory budget, (2) OPT achieved linear speed-up with an increasing number of CPU cores, (3) OPT outperforms the state-of-the-art parallel method by up to an order of magnitude with 6 CPU cores, and (4) for the first time in the literature, the triangulation results are reported for a billion-vertex scale real-world graph.

Categories and Subject Descriptors

H.3.3 [**Information search and retrieval**]: Search process

Keywords

Triangulation; Big data; Parallel processing

1. INTRODUCTION

Graph triangulation enumerates triangles in a graph, and its significance is well identified in the network analysis and data mining area. Various network analysis metrics can be obtained directly from graph triangulation. Clustering coefficients [19], transitivity [18], and trigonal connectivity [6] are representative. The clustering coefficient and transitivity are two important metrics which quantify density. Trigonal connectivity measures the tightness of a connection between a pair of vertices. Graph triangulation also provides insight into data mining applications. Becchetti et al. [7] exploit the number of triangles used in detecting spam pages in web graphs and in measuring content quality in social networks. Prat-Pérez et al. [26] propose a community detection method based on the observation that a good community has many triangles. Eckmann and Moses [14] study the hidden thematic relationship in web graphs using the transitivity metric.

Subsequently, graph triangulation methods have been actively studied. In early studies, most of the proposed methods assume that graphs fit in main memory [2, 5, 21, 24, 27, 28]. In-memory triangulation methods are classified into edge-iterator and vertex-iterator methods. However, emerging graphs of interest, such as social networks and web graphs, do not fit in main memory. To alleviate the memory size restriction, approximation methods were proposed [1, 7, 9, 13, 31]. However, such methods cannot support general graph triangulation but *approximate* triangle counting only. Thus, their applications are significantly limited [12].

To support triangulation in a cost-efficient way, the exact, disk-based triangulation methods are in great demand due to the emergence of large-scale graphs. Nowadays, online social networks such as Facebook reach a billion vertices [17]. The Yahoo web graph, which is publicly available, consists of over 1.4 billion vertices. Moreover, billion-scale web graphs can easily be obtained by ordinary users using open source crawlers such as Apache Nutch. In order to support efficient triangulation for billion-scale graphs, one may use the existing approaches which use either large-scale clusters or expensive high-performance servers. However, buying and maintaining such an expensive hardware environment is very hard for ordinary users or for small research groups. Accordingly, devising an efficient, *disk-based parallel* triangulation method in a single PC capable of handling billion-scale graphs is important and in great demand for the energy and economic benefits.

Figure 1 illustrates a motivating example that shows an example graph G and two types of triangles. Let us denote a triangle which is composed of three distinct vertices u, v and w as $\triangle_{uvw}$. When the memory buffer holds edges to which a, b, c, and d belong, among five triangles in G, $\triangle_{abc}$ and $\triangle_{cdf}$ are identified using edges in the memory buffer. However, $\triangle_{def}$, $\triangle_{cfg}$, and $\triangle_{cgh}$ can be identified only when edges in the external memory – (e, f), (f, g), and (g, h) – are loaded in the main memory. We call the first type of triangles the *internal triangles* and the second type the *external triangles*. Any disk-based triangulation method must identify both types of triangles efficiently.

[*]Corresponding author

SIGMOD'14, June 22–27, 2014, Snowbird, UT, USA.
Copyright 2014 ACM 978-1-4503-2376-5/14/06 ...$15.00.
http://dx.doi.org/10.1145/2588555.2588563.

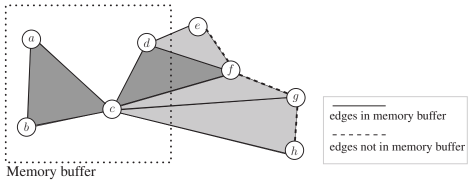


Figure 1: An example graph G and two types of triangles (dark: internal triangle, light: external triangle)

In the existing literature, disk-based triangulation methods have been little researched. To our knowledge, Chu and Cheng [12], Kyrola et al. [23], and Hu et al. [20] proposed the state-of-the-art methods. The method of Chu and Cheng [12] first partitions a graph. Then, for each partition, it identifies triangles, removes edges which participate the identified triangles, and saves the remaining edges. The above process is repeated until no edges remain. GraphChi [23] is the state-of-the-art *parallel* disk-based graph processing system and provides a triangulation method as one of its applications. The overall procedure of the application is similar to [12]. However, both methods have a severe drawback in that they require a significant amount of I/Os of reading and writing the remaining edges to disk at each iteration. Most recently, Hu et al. [20] proposed a disk-based variant of the vertex-iterator triangulation method. The method only reads the input graph as many times as the number of iterations. Although this *serial* method outperforms the method of [12], it is an instance of our triangulation framework and has a heavier cost than ours.

In this paper, we propose a highly-scalable, *overlapped* and *parallel*, disk-based triangulation framework, OPT, in a single PC of the multi-core CPU and the FlashSSD. OPT exploits (1) *full parallelism* of multi-core CPU and FlashSSD I/O and (2) *full overlap* of the CPU and I/O operations using asynchronous I/Os. The cost of the ideal triangulation method is the sum of the I/O cost of reading a graph once and the CPU cost of identifying triangles, provided that the buffer size is sufficient to make the graph resident in main memory. In the existing literature, such cost has been considered to be impossible due to the memory size constraint. Remarkably, OPT achieves the cost close to the ideal by using *a two-level overlapping strategy* (macro level and micro level).

At the macro level, OPT overlaps the internal triangulation and the external triangulation. OPT organizes its memory buffer into the *internal area* and the *external area*. In addition, it exploits two types of threads, the *main thread* and the *callback thread* which are dedicated to the internal and external triangulation. First, the main thread loads adjacency lists until they fully fill the internal area. While loading the data, the callback thread identifies the external candidate vertices whose adjacency lists can constitute the external triangles. After that, the two types of triangulation are overlapped by the asynchronous read request to the FlashSSD. The main thread (1) sends asynchronous read requests to the FlashSSD for the adjacency lists of the external candidate vertices, and (2) continues to find the internal triangles. While the internal triangles are found, as soon as the page requested by an asynchronous read is loaded on the external area, the callback thread simultaneously (1) finds the external triangles using both areas of the memory buffer and (2) issues remaining asynchronous I/O requests. This procedure is repeated until each adjacency list is loaded in the internal area once. The total elapsed time is the sum of the elapsed time of each iteration which is the sum of the I/O cost of loading the internal area and the maximum between (1) the CPU cost of the internal triangulation and (2) the I/O and CPU costs of the external triangulation.

At the micro level, OPT overlaps I/O and CPU operations of the external triangulation using asynchronous I/O. After asynchronous read requests are issued by the main thread, while the callback thread identifies the external triangles related to the data loaded by the asynchronous read, remaining asynchronous read requests are handled by the FlashSSD simultaneously. Consequently, OPT fully overlaps the I/O and CPU cost in the external triangulation. The elapsed time of the external triangulation only takes the CPU time, and its I/O time can be hidden, since triangulation is a CPU bound problem. Such I/O cost hiding incurs that the I/O cost of OPT becomes reading the input graph only once.

Note that OPT is *generic* in that any in-memory triangulation method is pluggable to OPT. By plugging appropriate operations for identifying internal triangles, external candidate vertices, and external triangles, OPT supports both vertex-iterator and edge-iterator methods in a disk-based manner.

To fully utilize CPU resource, OPT exploits *thread morphing* in which the type of one thread is morphed into the other type when one thread terminates earlier than the other. The full CPU utilization incurred by thread morphing maximizes the parallelization effect.

Moreover, OPT fully parallelizes the CPU operations using the multi-core parallelism (e.g. OpenMP). In OPT, several lines of OpenMP meta-language expressions enable parallel execution. When multiple CPU cores are used, OPT achieves the linear speed-up with an increasing number of CPU cores.

Our contributions are summarized as follows.

- We propose the first framework for overlapping I/O and CPU operations in parallel triangulation (Section 3). Specifically, we propose a two-level overlapping strategy. At the macro level, the internal triangulation and the external triangulation are overlapped. At the micro level, the I/O and CPU operations of the external triangulation are overlapped.
- Our triangulation framework is *generic* in that any vertex-iterator (Algorithms 11, 12, and 13) and edge-iterator (Algorithms 6, 8, and 10) triangulation models are pluggable.
- Through theoretical analysis, we show that, when the micro level overlapping is only applied, the cost of OPT is close to the ideal cost (Section 3.3).
- Experimental results showed that (1) OPT reached the ideal cost with less than 7% overhead (Section 5.3), (2) OPT achieved the linear speed-up with an increasing number of CPU cores (Section 5.6), and (3) OPT was an order of magnitude faster than the state-of-the-art parallel triangulation method when 6 CPU cores are used (Section 5.6).
- We report the triangulation results on a billion-vertex scale real-world graph, which is believed to be the first time in the literature (Section 5.7).

The rest of this paper is organized as follows. In Section 2, the problem definition is stated, and existing in-memory triangulation solutions are introduced. In Section 3, our triangulation framework OPT is introduced and how OPT overlaps and parallelizes triangulation is described in detail. In Section 4, the existing solutions and related works are reviewed. In Section 5, our experimental result is reported. Finally, in Section 6, our conclusion is presented.

2. PRELIMINARY

2.1 Problem Definition and Notation

First, we state the triangulation problem as follows.

Definition 1. [The exact triangulation problem]
When a simple undirected graph $G(V, E)$ is given, the triangulation problem identifies all triangles existing in G.

Before describing the triangulation methods, let us define the notation related to the problem and the method description. $G(V, E)$ is a simple undirected graph where V is a set of vertices and, E is a set of edges. $id(v) : V \to \mathbb{N}$ is a one-to-one mapping function from a vertex v to its id. $n(v)$ is the adjacency list of v. $n_\succ(v)$ is a sub-list of $n(v)$ which is defined as follows

$$n_\succ(v) = [u|u \in n(v) \text{ and } id(u) \succ id(v)]$$
$$\text{where } \succ \text{ represents a partial order.}$$

$n_\prec(v)$ is a sub-list of $n(v)$ which is defined as follows

$$n_\prec(v) = [u|u \in n(v) \text{ and } id(u) \prec id(v)].$$

A binary operation $\cap$ of two ordered lists returns their intersection. $\cup$ of two ordered lists returns their union.

Table 1 shows the notation frequently used in the paper.

Table 1: Summary of notation

Symbol	Description
$id(v)$ / $n(v)$	the vertex id/ the adjacency list of v
$n_\succ(v)$ / $n_\prec(v)$	the sub-list of $n(v)$ whose elements have higher/lower id than v
$\triangle_{uvw}$	the triangle which consists of u, v, and $w(id(u) \prec id(v) \prec id(w))$
$P(G)$	the number of pages of the graph G
m	the number of pages of the memory buffer
m_{in}	the number of pages of the internal area
m_{ex}	the number of pages of the external area

2.2 Iterator Models for In-memory Triangulation

The state-of-the-art in-memory triangulation methods follow the iterator model which iterates over vertices or edges [27]. The vertex-iterator finds a triangle $\triangle_{uvw}$ when, for each vertex u, a combination $(v, w) \in n(u) \times n(u)$ is included in E. The edge-iterator finds a triangle $\triangle_{uvw}$ when, for an edge $(u, v) \in E$, there exists a common neighbor w between u and v. In addition, in order to identify each triangle only once, the ordering constraint embedded in $n_\succ(v)$ enforces each triangle to be identified only once [27]. Algorithms 1 and 2 outline the in-memory vertex-iterator and edge-iterator methods which identify each triangle $\triangle_{uvw}$ only once which satisfies $id(u) \prec id(v) \prec id(w)$. In addition to helping unique triangle identification, the vertex mapping, $id(v)$, influences the efficiency of the in-memory triangulation method. Schank and Wagner [28] show that the degree-based heuristic, where $id(u) \prec id(v)$ if $degree(u) < degree(v)$, boosts the elapsed time over orders of magnitude in power-law graphs. The intuition behind the heuristic is that assigning a high id to a high-degree vertex v makes $|n_\succ(v)|$ small, and eventually reduces the intersection cost in VertexIterator$_\succ$ and EdgeIterator$_\succ$ (See Eq. 3).

Algorithm 1 VertexIterator$_\succ(G)$

```
1: for each $u \in V$ do
2:     for each $v \in n_\succ(u)$ do
3:         for each $w \in n_\succ(u)$ do
4:             if $(v, w) \in E, id(w) \succ id(v)$ then
5:                 output $\triangle_{uvw}$
```

Algorithm 2 EdgeIterator$_\succ(G)$

```
1: for each $(u, v) \in E$ do
2:     $W_{uv} \leftarrow n_\succ(u) \cap n_\succ(v)$
3:     for each $w \in W_{uv}$ do
4:         output $\triangle_{uvw}$
```

Both VertexIterator$_\succ$ and EdgeIterator$_\succ$ have $O(\alpha|E|)$ time complexity where α is the arboricity of a graph [11]. Arboricity has the following property which is used to bound the time complexity of triangulation methods [11].

$$\sum_{(u,v)\in E} \min(|n(u)|, |n(v)|) = O(\alpha|E|) \tag{1}$$

According to [20], when an $O(1)$ time hash for checking $(u, v) \in E$ exists, the time complexity of the VertexIterator$_\succ$ is $O(\alpha|E|)$. Similarly, when an $O(1)$ time hash for checking $u \in n_\succ(v)$ exists, the time complexity of the EdgeIterator$_\succ$ is derived as follows:

$$\sum_{(u,v)\in E} cost(n_\succ(u) \cap n_\succ(v)) \tag{2}$$
$$= \sum_{(u,v)\in E} \min(|n_\succ(u)|, |n_\succ(v)|) \quad \text{(by using hash)} \tag{3}$$
$$\le \sum_{(u,v)\in E} \min(|n(u)|, |n(v)|) \tag{4}$$
$$= O(\alpha|E|) \quad \text{(by Eq.1).} \tag{5}$$

3. OPT: OVERLAPPED AND PARALLEL TRIANGULATION

In this section, we describe our overlapped and parallel triangulation framework, OPT, which exploits the advanced overlapping and parallelism features of the multi-core CPU and the FlashSSD. When a graph cannot fit in memory, the *internal triangles* and the *external triangles* are classified, and the challenge of disk-based triangulation methods is identified (Section 3.1). To resolve the challenge, OPT exploits a two-level overlapping strategy – (1) overlapping two types of triangulation and (2) overlapping the I/O and CPU operations in the external triangulation (Section 3.2). Then, we formally analyze the cost of OPT and show that it is close to the ideal cost (Section 3.3). To achieve the full CPU utilization and linear speed-up, thread morphing and multi-core parallelism are applied to OPT (Section 3.4).

Although OPT provides a general framework for triangulation, for ease of understanding, in Sections 3.1~3.3, we will describe how OPT works by using a specific instance of OPT for EdgeIterator$_\succ$ (Algorithm 2) with $id(v)$ which follows alphabetical order. To show the generalization power of OPT, Section 3.5 describes how OPT instantiates the VertexIterator$_\succ$ and the method of [20].

3.1 Two Types of Triangles

When a graph is too large to be loaded into main memory, triangles are classified into two types – the *internal triangles* and the *external triangles*. We denote a triangle $\triangle_{uvw}$ as an internal triangle only if both $n(u)$ and $n(v)$ are loaded in main memory. When $n(u)$ and $n(v)$ reside in main memory, $\triangle_{uvw}$ is found as an internal triangle if $w \in n_\succ(u) \cap n_\succ(v)$ (Line 4 of Algorithm 2). If $n(u)$ is in main memory and $n(v)$ is not, $\triangle_{uvw}$ is an external triangle.

For example, let us recall the graph G in Figure 1. Suppose that only $n(a)$, $n(b)$, $n(c)$, and $n(d)$ are resident in main memory. Then, $\triangle_{abc}$ and $\triangle_{cdf}$ are the internal triangles identified by $c \in n_\succ(a) \cap n_\succ(b)$ and $f \in n_\succ(c) \cap n_\succ(d)$. However, $\triangle_{cfg}$, $\triangle_{cgh}$, and $\triangle_{def}$ are the external triangles because $n(f)$, $n(g)$, and $n(e)$ are not available in main memory. Thus, to find all triangles, the adjacency lists (e.g. $n(f)$), which are required to find the external triangles (e.g. $\triangle_{cfg}$), should be loaded in main memory.

When the graph does not fit in main memory, the in-memory triangulation method incurs severe performance degradation due to the excessive random I/O. Suppose that to get $n_\succ(v)$ Algorithm 2 loads $n(v)$ from disk whenever it is not resident in main memory. When we retrieve neighbors of a second vertex of (u, v) (i.e., when reading $n_\succ(v)$) (Line 2 of Algorithm 2), random I/O is inevitable since each $n(v)$ for $v \in n_\succ(u)$ is scattered across the disk. Consequently, the excessive random read requests on small data fragments from disk are required to fetch $n(v)$ which is not loaded in main memory. The same phenomenon occurs in Algorithm 1.

3.2 Overlapped Processing of OPT

In this subsection, we describe how OPT identifies triangles efficiently by using a two-level overlapping strategy.

Graph Representation in Disk. When storing $n(v)$ for $v \in V$ in the disk, OPT uses the slotted page structure which is widely used in database systems. Globally, each $(v, n(v))$ for $v \in V$ are stored throughout the slotted pages, but the storage ordering of $(v, v(n))$ does not need to match the original id of v. When the size of an adjacency list is larger than the size of a slotted page, a list of slotted pages is used to store it.

Memory Buffer Organization. OPT splits the memory buffer into the *internal area* and the *external area*. The internal area holds the adjacency lists with which the internal triangles are recognized. As in [12, 23], the internal area size must be large enough to load at least one adjacency list, which is reasonable. The external area is a temporary area to identify the external triangles under the adjacency lists currently loaded in the internal area. For later use, let us denote the number of pages of the memory buffer m, that of the internal area as m_{in}, and that of the external area as m_{ex}.

Figure 2 shows an example of how OPT exploits the memory buffer where two pages are assigned to the internal area and one page is to the external area. When p_1 and p_2 are loaded into the internal area, $\triangle_{abc}$ and $\triangle_{cdf}$ are identified as the internal triangles. To identify the external triangles, p_3 and p_4 should be loaded into the external area. When p_3 is loaded into the external area temporarily, $\triangle_{def}$ and $\triangle_{cfg}$ are recognized as the external triangles. When p_4 is loaded into the external area, $\triangle_{cgh}$ is recognized.

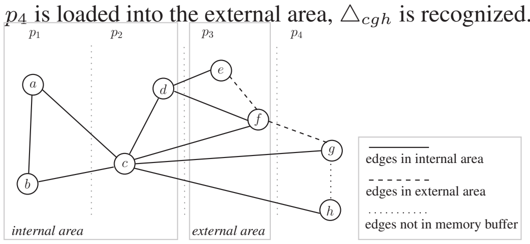


Figure 2: How OPT utilizes memory buffer

Here, we introduce a concept of *the external candidate vertex* whose adjacency list should be loaded into the external area. In EdgeIterator$_{\succ}$, for a triangle $\triangle_{uvw}$ where $id(u) \prec id(v) \prec id(w)$, $n(u)$ and $n(v)$ are required to identify it. When only $n(u)$ is in the internal area, v is called the external candidate vertex. For example, suppose that the adjacency lists of a, b, c, and d are loaded in the internal area (Figure 2). e and f are the external candidate vertices for $n(d)$ which are not in the internal area and thus $n(e)$ and $n(f)$ should be loaded in the external area to identify the external triangles.

Asynchronous Read Function. We provide a core function AsyncRead(pid, $Callback$, $Args$), in order to allow asynchronous reads to the FlashSSD. Here, pid is the page id to be loaded, $Callback$ is a callback function, and $Args$ is the list of arguments of $Callback$. AsyncRead(pid, $Callback$, $Args$) issues an asynchronous read for the page pid to the FlashSSD and registers $Callback$ with $Args$ to the operating system. On completion of reading, $Callback$ with $Args$ is called. For example, AsyncRead(1, ExampleCallback, $\{a, b\}$) issues a read request for page 1, and on completion of the asynchronous read, ExampleCallback(a, b) for the page 1 is called.

A Two-Level Overlapping Strategy. The asynchronous I/Os to the FlashSSD and the callback functions play a crucial role on a two-level overlapping strategy of OPT which enables two levels of simultaneous executions.

At the macro level, OPT overlaps the internal and external triangulation by using two types of threads. The *main thread* and the *callback thread* are dedicated to identifying the internal triangles and the external triangles, respectively. When the main thread requests an asynchronous read to the FlashSSD, it feeds the data and the corresponding task to the callback thread. Because the asynchronous I/Os do not wait for completion of the I/O requests, the two types of threads can be executed simultaneously.

At the micro level, OPT overlaps the CPU and I/O operations in the external triangulation. The asynchronous I/O and the callback functions enable the independent execution of the I/O and the CPU operations. Consequently, while the callback thread finds the external triangles by calling the callback function, the FlashSSD processes the asynchronous read request simultaneously.

Main Thread. The main thread controls the overall procedure of OPT. First, it allocates the internal area and the external area of the memory buffer. It fills the internal area with a part of the graph and identifies the external candidate vertices. It issues asynchronous read requests to FlashSSD for the adjacency lists of the external candidate vertices and delegates the external triangulation to the callback thread. It finds the internal triangles using the adjacency lists in the internal area. Until all adjacency lists are loaded in the internal area once, the above procedure is repeated.

Algorithm 3 describes the detailed procedure of OPT. When the main thread starts, it first initializes the external candidate vertex set, V_{ex}, as an empty set (Line 2). It allocates m_{in} pages of the internal area and m_{ex} pages of the external area (Line 3). After the memory allocation, the main thread repeats $\lceil P(G)/m_{in} \rceil$ iterations (Lines 4-13). For each iteration, it first loads m_{in} from disk by calling AsyncRead(j, IdentifyExternalCandidateVertex, $\{j\}$) m_{in} times (Lines 6-7). On completion of each read request, the callback function, IdentifyExternalCandidateVertex (Algorithm 7), collects the external candidate vertex set, V_{ex}. The main thread waits until all asynchronous read requests for the internal area are finished. (Line 8). After filling the internal area, the main thread delegates the external triangulation to the callback thread by calling DelegateExternalTriangle (Line 9). Note that DelegateExternalTriangle only issues the *asynchronous* read requests to the Flash-SSD, and the actual triangulation is conducted in the callback thread. Then, the internal triangles under the current internal area are identified by calling InternalTriangle (Line 10). After all the internal triangles are identified, the main thread waits until the external triangulation is finished (Line 11). Finally, all pages in the internal area are unpinned to allow the replacement policy to freely evict those pages (Lines 12-13).

Algorithm 3 OPT(m_{in}, m_{ex}, d)

```
1:  Require: $m_{in}$: # of pages, $m_{ex}$: # of pages, $P(G)$: # of pages
2:  $V_{ex} \leftarrow \phi$
3:  allocate the memory buffer of $m_{in}$ for the internal area and $m_{ex}$ pages for the
    external area
4:  for $i \leftarrow 1$ to $\lceil P(G)/m_{in} \rceil$ do
5:      $pid_s \leftarrow (i-1) \times m_{in} + 1, pid_e \leftarrow i \times m_{in}$
6:      for $j \leftarrow pid_s$ to $pid_e$ do
7:          AsyncRead($j$, IdentifyExternalCandidateVertex, $\{j\}$)
8:      wait until IdentifyExternalCandidateVertex executions are finished
9:      DelegateExternalTriangle($V_{ex}$, $P(G)$, $pid_e$, $m_{in}$, $m_{ex}$)
10:     InternalTriangle($pid_s$, $pid_e$)
11:     wait until DelegateExternalTriangle executions are finished
12:     for $j \leftarrow pid_s$ to $pid_e$ do
13:         unpin a page of id $j$
```

For example, consider the graph G in Figure 2. At the first iteration, the main thread executes the following step. First, three pages are allocated in the memory buffer, two pages are assigned to the internal area, and one page is assigned to the external area. Next, p_1 and p_2 are loaded into the internal area by calling AsyncRead(j, IdentifyExternalCandidateVertex, $\{j\}$) $(j = 1, 2)$. When p_1 is loaded, $n(a)$ and $n(b)$ become available in memory. When p_2 is loaded, $n(c)$ and $n(d)$ become available, and $\{e, f, g, h\}$ is identified as V_{ex}. By calling DelegateExternalTriangle ($\{e, f, g, h\}$, 4, 2, 2, 1), the main thread delegates the external triangulation to the callback thread, and $\triangle_{cfg}$, $\triangle_{cgh}$, and $\triangle_{def}$ are found as the external triangles in the callback thread. The internal triangles are identified by calling InternalTriangle(1,2), and $\triangle_{abc}$ and $\triangle_{cdf}$ are identified. At the second iteration, the same procedure is conducted, but no triangles are identified.

Algorithm 4 constructs asynchronous I/O requests by grouping the candidate vertices by their page IDs and issues those requests to the FlashSSD. Specifically, it first groups the external candidate vertices by their corresponding page IDs and makes a request list L whose element is a pair of page id, i, and the external candidate vertex set, V_{ex}^{i} (Lines 3-6). When the number of the requested pages, $|L|$, is larger than the number of pages for the external area, m_{ex}, the request list is split into two lists – L_{now} and L_{later} such that $|L_{now}| = \min(m_{ex}, |L|)$ (Line 7). Then, $|L_{now}|$ times asynchronous reads are requested to the FlashSSD, and the external triangulation is delegated to the callback thread which executes the callback function ExternalTriangle (Algorithm 9) (Lines 8-9). The remaining $|L_{later}|$ requests are issued in ExternalTriangle.

Algorithm 4 DelegateExternalTriangle($V_{ex}, P(G), id_e, m_{in}, m_{ex}$)

```
1: Require: $V_{ex}$: vertex set, $P(G)$: # of pages, $id_s$ : page id, $id_e$ : page id,
   $m_{ex}$: # of pages
2: $L \leftarrow \phi$
3: for $i \leftarrow (\cdots, id_e + m_{in}, \cdots, id_e + 1)$ do
4:     $V_{ex}^{i} \leftarrow \{v|n(v)$ in the page of id $i$ and $v \in V_{ex}\}$
5:     if $V_{ex}^{i} \neq \phi$ then
6:         append $\{(i, V_{ex}^{i})\}$ to $L$
7: split $L$ into $L_{now}$ and $L_{later}$ s.t $|L_{now}| = min(m_{ex}, |L|)$
8: for each $(j, V_{ex}^{j}) \in L_{now}$ do
9:     AsyncRead($j$, ExternalTriangle, $\{j, V_{ex}^{j}, L_{later}\}$)
```

For example, DelegateExternalTriangle($\{e, f, g, h\}$, 4, 2, 2, 1) from the first iteration executes the following. First, the requested adjacency lists, $n(e)$, $n(f)$, $n(g)$, and $n(h)$, are translated into a request list of $L = [(4, \{g, h\}); (3, \{e, f\})]$. As the page size of the external area is 1, L is split into $L_{now} = [(4, \{g, h\})]$ and $L_{later} = [(3, \{e, f\})]$. Finally, an asynchronous I/O request for p_4 is submitted, and the request for p_3 will be submitted in ExternalTriangle after the external triangles related to p_4 are found.

Note that the pages in the internal area for the next iteration should be the last pages loaded in the external area for the current iteration. The above condition on the page loading order avoids repetitive loading of those pages. The page loading order of the internal area (Line 6 of Algorithm 3) and that of the external area (Line 3 of Algorithm 4) satisfies this condition. When the pages of id from id_s to id_e are loaded in the internal area, the last m_{in} pages loaded in the external area are the pages of id ($id_e+m_{in}, \cdots, id_e+1$), and thus, those pages in the external area in the current iteration can be used for the internal area in the next iteration. Thus, OPT can even outperform the ideal method as we will see in Section 5.3.

The asynchronous read request in Line 9 of Algorithm 4 is the critical point where the two-level overlapping strategy of OPT is implemented. Here, both types of triangulation can be executed simultaneously (the macro level overlapping). Also, while the FlashSSD processes an asynchronous read request on the page i, the callback thread identifies the external triangles related to the distinct page $j (i \neq j)$ simultaneously (the micro level overlapping).

Algorithm 5 identifies the internal triangles in parallel using multiple CPU cores. By plugging a specific triangulation method to InternalTriangleImpl, OPT can instantiate various triangulation methods. For example, to instantiate EdgeIterator$_{\succ}$, InternalTriangleEdgeIterator$_{\succ}$ (Algorithm 6) is plugged to InternalTriangleImpl. Moreover, when multiple CPU cores are available for the internal triangulation, the parallelization can be applied on the basis of pages (Lines 2-4).

When generating results, we use a nested representation to avoid generating repetition of the triangle prefixes. Specifically, for those

Algorithm 5 InternalTriangle(pid_s,pid_e)

```
1: Require: $pid_s$: page id, $pid_e$: page id
2: for $j \leftarrow pid_s$ to $pid_e$ parallel do
3:     for each $(u, n(u))$ in the page of id $j$ do
4:         InternalTriangleImpl(u);
```

triangles having the same u and v, $\triangle_{uvw_1} \sim \triangle_{uvw_k}$, to be generated in Line 5 of Algorithm 6, we output the results in the form of $< u, v, \{w_1, \cdots, w_k\} >$. In order to increase the performance, each thread accumulates results into a memory buffer and flushes the buffer to the FlashSSD using asynchronous write requests.

Algorithm 6 InternalTriangleEdgeIterator$_{\succ}(u)$

```
1: for each $(u, v)$ where $v \in n_{\succ}(u)$ do
2:     if $n(v)$ is in internal area then
3:         $W_{uv} \leftarrow n_{\succ}(u) \cap n_{\succ}(v)$
4:         for each $v \in W_{uv}$ do
5:             output $\triangle_{uvw}$
```

Callback Thread. On completion of an asynchronous I/O request, the callback thread catches the I/O completion signal from the FlashSSD via the operating system and executes the callback function which conducts the CPU operations related to the loaded data. We use two callback functions in OPT for (1) identifying the external candidate vertices and (2) finding the external triangles.

On completion of loading the page pid in the internal area, Algorithm 7 collects the external candidate vertices as a response to the asynchronous read request from AsyncRead (pid, IdentifyExternalCandidateVertex, $\{pid\}$) (Line 7 of Algorithm 3). It first pins the loaded page in the internal area to prevent the page from being evicted (Line 3). Then, it determines the external candidate vertices whose adjacency lists must be loaded in the external area (Line 5). The condition of identifying external candidate vertices depends on the specific instance of OPT and such condition should be plugged in ExternalCandidateVertexImpl.

To instantiate EdgeIterator$_{\succ}$, ExternalCandidateVertexEdgeIterator$_{\succ}$ (Algorithm 8) is used. In EdgeIterator$_{\succ}$, when an adjacency list $n(u)$ is loaded in the internal area, $n(v)$ for $v \in n_{\succ}(u)$ should be resident in the memory buffer to find all triangles in which u and v participate. If such $n(v)$s are not in the internal area, ExternalCandidateVertexEdgeIterator$_{\succ}$ identifies v as the external candidate vertex (Lines 2-4).

Algorithm 7 IdentifyExternalCandidateVertex(pid)

```
1: Require: $pid$: page id
2: Ensure: $V_{ex}$: external candidate vertex set is updated
3: pin a page $p$ of $pid$
4: for each $(u, n(u)) \in p$ do
5:     $V_{ex} \leftarrow V_{ex} \cup$ ExternalCandidateVertexImpl($u$)
```

Algorithm 8 ExternalCandidateVertexEdgeIterator$_{\succ}(u)$

```
1: $ret \leftarrow \phi$
2: for each $v \in n_{\succ}(u)$ do
3:     if $n(v)$ is not in the internal area then
4:         $ret \leftarrow ret \cup \{v\}$
5: return $ret$
```

From the recurring example of G, when p_2 is loaded in the internal area, the external candidate vertex set V_{ex}, is identified as $\{e, f, g, h\}$. From $n_{\succ}(c)$, f, g, and h become the external candidate vertices, and from $n_{\succ}(d)$, e and f do.

On completion of reading the page pid in the external area, Algorithm 9 identifies the external triangles as a response to the asynchronous read request from AsyncRead (pid, ExternalTriangle, $\{pid$, $V_{ex}^{pid}, L_{later}\}$) (Line 9 of Algorithm 4). Like IdentifyExternalCandidateVertex, it first pins the loaded page (Line 3). V_{ex}^{pid} contains the external candidate vertices whose adjacency lists are located in page pid. For each $v \in V_{ex}^{pid}$, the vertex set V_{req}^{v}, whose element requests v as the external candidate vertex, is identified (Line 5).

Then, from a combination of $(u \in V^v_{req}, v \in V^{pid}_{ex})$, external triangles are identified using ExternalTriangleImpl((u, v)) (Line 7). After all the external triangles in the loaded page are found, OPT unpins the page to yield the space that the loaded page occupies (Line 8). Finally, If L_{later} is not empty, the next page to be loaded in the external area is popped from L_{later}, and another asynchronous I/O request is issued (Lines 9-13). When multiple CPU cores are available for the callback thread, the asynchronous I/O request (Lines 9-13) should be atomic, since L_{later} is a shared variable.

Algorithm 10 (ExternalTriangleEdgeIterator$_\succ$) is the EdgeIterator$_\succ$ implementation of identifying external triangles. It identifies external triangles by intersecting $n_\succ(u)$ in the internal area and $n_\succ(v)$ in the external area.

Algorithm 9 ExternalTriangle($pid, V^{pid}_{ex}, L_{later}$)

```
1: Require: $pid$: page id, $V^{pid}_{ex}$: vertex set, $L_{later}$: list of pairs of page id and vertex set
2: Ensure: the external triangles related to the page $pid$ are counted.
3: pin a page $p$ of $pid$
4: for each $v \in V^{pid}_{ex}$ do
5:    $V^v_{req} \leftarrow \{u | u \in n_\prec(v)$ and $n(u)$ is in the internal area$\}$
6:    for each $u \in V^v_{req}$ do
7:       ExternalTriangleImpl($u, v$)
8: unpin $p$
9: atomic {
10:    if $L_{later}$ is not empty then
11:       $(pid', V'_{ex}) \leftarrow$ pop the first element of $L_{later}$
12:       AsyncRead($pid'$, ExternalTriangle, $\{pid', V'_{ex}, L_{later}\}$)
13: }
```

Algorithm 10 ExternalTriangleEdgeIterator$_\succ$(u, v)

```
1: $W_{uv} \leftarrow n_\succ(u) \cap n_\succ(v)$
2: for $w \in W_{uv}$ do
3:    output $\triangle_{uvw}$
```

For example, when p_4 is loaded in the external area, ExternalTriangle (4, $\{g, h\}$, $[(3, \{e, f\})]$) is executed as follows. From $V^4_{ex} = \{g, h\}$, $V^g_{req} = \{c\}$ and $V^h_{req} = \{c\}$ are extracted. Then, $\triangle_{cgh}$ is identified from $n_\succ(c) \cap n_\succ(g)$. After the external triangles related to p_4 are processed, the next page p_3 is requested by an asynchronous I/O.

Correctness. To prove the correctness, we first prove that the EdgeIterator$_\succ$ instance of OPT executes the same set of adjacency list intersections to EdgeIterator$_\succ$(Algorithm 2).

THEOREM 1. *The adjacency list intersections executed in the EdgeIterator$_\succ$ instance of* OPT *are same to those executed in EdgeIterator$_\succ$ (Algorithm 2).*

PROOF. For each vertex $u \in V$, $n_\succ(u)$ must be intersected with $n_\succ(v)$ for $v \in n_\succ(u)$ (Line 2 of Algorithm 2). In EdgeIterator$_\succ$ instance of OPT, only part of $n_\succ(v)$s are located in the internal area. Let us denote $n^{internal}_\succ(u) = \{v | v \in n_\succ(u)$ and $n(v) \in$ the internal area$\}$ and $n^{external}_\succ(u) = \{v | v \in n_\succ(u)$ and $n(v) \notin$ the internal area$\}$.

Case $\mathbf{v \in n^{internal}_\succ(u)}$: Because $n(v)$ is loaded into the internal area, $n_\succ(u) \cap n_\succ(v)$ is executed when finding the internal triangles (Line 10 of Algorithm 3).

Case $\mathbf{v \in n^{external}_\succ(u)}$: Each v, whose adjacency list, $n(v)$, is not in the internal area, is identified in IdentifyExternalCandidateVertex (Lines 4-5 of Algorithm 7). The page p to which $n(v)$ belongs is identified in DelegateExternalTriangle (Lines 3-6 of Algorithm 4). When the page p is loaded into the external area, ExternalTriangle finds $u \in V^p_{ex}$ and executes $n_\succ(u) \cap n_\succ(v)$ to find the external triangles (Line 7 of Algorithm 9).

From both cases, for all $v \in n_\succ(u)$, $n_\succ(u) \cap n_\succ(v)$ is executed in OPT. Since this analysis is applied to all vertices, OPT executes the same intersections as EdgeIterator$_\succ$. □

Theorem 1 naturally induces the correctness of the EdgeIterator$_\succ$ instance of OPT (Lemma 1).

LEMMA 1. OPT *correctly identifies triangles.*

PROOF. By Theorem 1, all set intersections in both methods are the same. Triangles of a graph is directly obtained from the set intersection result. Therefore, as long as EdgeIterator$_\succ$ identifies triangles correctly, so does the EdgeIterator$_\succ$ instance of OPT. □

3.3 Cost Analysis

By the cost analysis of OPT, we want to show that (1) when a single CPU core is available, the cost OPT is close to the cost of the ideal method with a small overhead and (2) when multiple CPU cores are available, an additional treatment is required to fully utilize the CPU resource. To unify the I/O and CPU cost in terms of time complexity, let us denote the ratio of the I/O cost reading a page to the CPU cost executing an operation as a constant c. Because OPT is a generic triangulation framework, we analyze the I/O and CPU cost of the EdgeIterator$_\succ$ instance of OPT.

The ideal cost of EdgeIterator$_\succ$ is the sum of the I/O cost of reading a graph once ($cP(G)$) and the CPU cost of identifying triangles ($Cost_{CPU}$). Note that, according to Eq.5, $Cost_{CPU}$ has the same asymptotic time complexity $O(\alpha|E|)$ of the method in [20]. Such cost is only achievable only when the in-memory triangle method is executed under the infinite-sized main memory. Let us denote such method as ideal. Formally, the cost of ideal is expressed as follows.

$$Cost_{\mathsf{ideal}} = \quad cP(G) + Cost_{CPU} \tag{6}$$

Note that CPU cost in this analysis follows EdgeIterator$_\succ$ (Algorithm 2). Let us denote a serial version of OPT as $\mathsf{OPT_{serial}}$ when a single CPU core is available. To make $\mathsf{OPT_{serial}}$ use only one CPU core, OPT is modified to disable the macro level overlapping – at each iteration, the external triangles are identified after the internal triangulation is completed.

When a single core is available, the cost of $\mathsf{OPT_{serial}}$, $Cost_{\mathsf{OPT_{serial}}}$, consists of the cost of ideal and the overhead induced by the incomplete I/O and CPU overlap in the external triangulation. The cost of $\mathsf{OPT_{serial}}$ is the sum of the internal triangulation cost and the external triangulation cost. As $k = \lceil P(G)/m_{in} \rceil$ iterations are executed in the outer loop of $\mathsf{OPT_{serial}}$, the overall cost is the summation of the cost of each iteration. At the ith iteration, let us denote the edges participating in the internal/external/total triangulation as E_{in^i}/E_{ex^i}/E_i. We also denote the request list generated in Algorithm 4 as L_i. E_{in^i}, $E_{ex^i i}$, and E_i are formally defined as follows.

$$E_{in^i} = \{(u, v) | n(u) \text{ is in internal area}, v \in n^{internal}_\succ(u)\}$$
$$E_{ex^i} = \{(u, v) | n(u) \text{ is in internal area}, v \in n^{external}_\succ(u)\}$$
$$E_i = E_{in^i} \cup E_{ex^i}$$

The cost at each iteration is the sum of the internal triangulation cost and the external triangulation cost. The cost of the internal triangulation at the ith iteration is the I/O cost of loading data into the internal area, $c \times m_{in}$, minus the saved I/O cost by the pages buffered at the previous iteration (Algorithm 4), $c \times \Delta^{in_i}_{I/O}$, and plus the CPU cost of finding the internal triangles, $\sum_{(u,v) \in E_{in^i}}$ $\min(|n_\succ(u)|, |n_\succ(v)|)$ by Eq.5. The cost of external triangulation of ith iteration is the maximum of the I/O cost of loading data into the external area, $c|L^i|$, and the CPU cost of finding the external triangles, $\sum_{(u,v) \in E_{ex^i}} \min(|n_\succ(u)|, |n_\succ(v)|)$, due to the micro level overlapping. The maximum can be interpreted as the sum of CPU cost of finding external triangles and the non-overlapped I/O cost, $c \times \Delta^{ex_i}_{I/O} (\geq 0)$. After re-organizing the cost into the I/O cost and the CPU cost, the CPU cost becomes that of the in-memory triangulation method by Theorem 1. The I/O cost becomes the cost of reading the input graph once, $cP(G)$, minus the saved I/O cost by the buffered pages, $c \times \Delta^{in}_{I/O} (= \sum_{i=1}^{k} c \times \Delta^{in_i}_{I/O})$, and plus the non-overlapped I/O cost in the external triangulation, $c \times \Delta^{ex}_{I/O} (= \sum_{i=1}^{k} c \times \Delta^{ex_i}_{I/O})$. The final cost of OPT$_{\text{serial}}$ becomes as follows.

$$\begin{aligned}
& Cost_{\text{OPT}_{\text{serial}}} \\
= \; & \textstyle\sum_{i=1}^{k} \{c(m_{in} - \Delta^{in_i}_{I/O}) + \sum_{(u,v)\in E_{in^i}} \min(|n_{\succ}(u)|, |n_{\succ}(v)|) \\
& + \max(c|L_i|, \textstyle\sum_{(u,v)\in E_{ex^i}} \min(|n_{\succ}(u)|, |n_{\succ}(v)|))\} \\
= \; & \textstyle\sum_{i=1}^{k} \{c(m_{in} - \Delta^{in_i}_{I/O}) + \sum_{(u,v)\in E_{in^i}} \min(|n_{\succ}(u)|, |n_{\succ}(v)|) \\
& + \textstyle\sum_{(u,v)\in E_{ex^i}} \min(|n_{\succ}(u)|, |n_{\succ}(v)|) + c \times \Delta^{ex_i}_{I/O}\} \\
= \; & \textstyle\sum_{i=1}^{k} \{c(m_{in} - \Delta^{in_i}_{I/O} + \Delta^{ex_i}_{I/O}) \\
& + \textstyle\sum_{(u,v)\in E^i} \min(|n_{\succ}(u)|, |n_{\succ}(v)|)\} \\
= \; & cP(G) + \textstyle\sum_{(u,v)\in E} \min(|n_{\succ}(u)|, |n_{\succ}(v)|) + c(\Delta^{ex}_{I/O} - \Delta^{in}_{I/O}) \\
= \; & cP(G) + Cost_{CPU} + c(\Delta^{ex}_{I/O} - \Delta^{in}_{I/O}) \\
= \; & Cost_{\text{ideal}} + c(\Delta^{ex}_{I/O} - \Delta^{in}_{I/O})
\end{aligned}$$

However, the cost of OPT$_{\text{serial}}$ is still close to or even less than that of ideal. Because the triangulation problem is a CPU bound task, most of I/O cost in the external triangulation becomes hidden behind the CPU operations by the asynchronous I/O. In addition, the saved I/O cost by the buffered pages, $c \times \Delta^{in}_{I/O}$, reduces the total I/O cost. Consequently, $c(\Delta^{ex}_{I/O} - \Delta^{in}_{I/O})$, which is the cost gap between OPT$_{\text{serial}}$ and ideal, is small in the triangulation. In Section 5.3, the empirical evaluation shows that the I/O cost overhead, $c(\Delta^{ex}_{I/O} - \Delta^{in}_{I/O})$, becomes negative or does not exceed 7% of the cost of ideal in term of the elapsed time.

When two CPU cores are available, OPT reduces its cost, $Cost_{\text{opt}}$, by applying both levels of overlapping, but does not fully utilize the CPU resource. By applying the macro level overlapping additionally, OPT can overlap the internal triangulation and the external triangulation. The lower bound of I/O cost of OPT is the cost of reading the graph once, minus the saved cost by buffered pages. The CPU cost of OPT becomes the summation of the maximum of the CPU cost in the internal triangulation and that in the external triangulation at each iteration as follows.

$$\begin{aligned}
& Cost_{\text{OPT}} = \\
& c(P(G) - \Delta^{in}_{I/O}) + \textstyle\sum_{i=0}^{k} \{\max(\sum_{(u,v)\in E_{in^i}} \min(|n_{\succ}(u)|, |n_{\succ}(v)|), \\
& \textstyle\sum_{(u,v)\in E_{ex^i}} \min(|n_{\succ}(u)|, |n_{\succ}(v)|) + c \times \Delta^{ex_i}_{I/O})\}
\end{aligned}$$

The overhead induced by non-overlapped I/O cost, $c \times \Delta^{ex_i}_{I/O}$ has an additional chance to be hidden behind the CPU cost of internal triangulation.Because, in most cases, the CPU costs of the two types of the triangulation are not the same, the CPU cost of OPT is larger than half of the CPU cost of OPT$_{\text{serial}}$ although two CPU cores are used. Consequently, OPT does not fully utilize the CPU resource.

3.4 Thread Morphing and Parallel Processing

By adapting thread morphing to the macro level overlapping, OPT accomplishes full CPU utilization. At each iteration of OPT, if the callback thread terminates earlier than the main thread, the callback thread is morphed into the main thread and continues identifying the internal triangles. If the main thread terminates earlier, the opposite happens. We call such thread type changes as thread morphing. From the cost analysis of OPT in Section 3.3 OPT does not fully exploit the CPU resource if the two types of triangulation at each iteration do not terminate at the same time. However, using thread morphing, OPT always utilizes the CPU resources and achieves full speed-up when at least two CPU cores are available.

When more than two CPU cores are available, OPT further improves the elapsed time using parallelism such as OpenMP. Basically, OPT applies the parallelization on the internal triangulation, the for loop of iterating slotted pages (Lines 2-5 of Algorithm 5). The parallelization of the external triangulation is enhanced by thread morphing. If the internal triangulation is terminated earlier than the external triangulation, the main thread identifies the external triangles. With the full parallelization on both types of triangulation, OPT achieves linear speed-up with an increasing number of CPU cores (Section 5.6).

3.5 Instantiation of OPT for Vertex Iterator

In this section, to show generalization power of OPT, we present two OPT instances of VertexIterator$_{\succ}$ (Algorithm 1) and MGT [20]. First, the internal triangles and external triangles should be identified in the vertex-iterator perspective. In VertexIterator$_{\succ}$, the key task is checking $(v, w) \in E$ (Line 4 of Algorithm 1). Because holding all (v, w)s is impossible with the limited memory budget, only part of edges, E_{in}, are loaded in the internal area. Then, for each $u \in V$, all candidate combinations of (v', w'), where (1) $v', w' \in n_{\succ}(u)$, (2) $n(v') \in$ the internal area, and (3) $id(v') \prec id(w')$, are checked if they are included in the internal area. Thus, if $n(u)$ is loaded in the internal area, $\triangle_{uvw}$ is identified as the internal triangle, otherwise, it is identified as the external triangle. The above procedure continues until all edges are loaded in the internal area. When a single CPU is used, the CPU cost of the VertexIterator$_{\succ}$ instance of OPT follows that of VertexIterator$_{\succ}$ (Algorithm 1) and the I/O cost is $c(P(G) + \Delta^{ex}_{I/O} - \Delta^{in}_{I/O})$ which is same to that of the EdgeIterator$_{\succ}$ instance of OPT. The proof is omitted due to the space limit, but it follows the same steps mentioned in Section 3.2.

Algorithm 11 identifies the internal triangles using VertexIterator$_{\succ}$ when $n(u)$ is loaded in the internal area. Algorithm 12 adds $u \in n_{\prec}(v)$ as an external candidate vertex if $n(u)$ is not loaded in the internal area. Algorithm 13 identifies the external triangles using VertexIterator$_{\succ}$ when $n(u)$ is loaded in the external area.

Algorithm 11 InternalTriangleVertexIterator$_{\succ}(u)$

```
1: for each $v \in \{v | v \in n_{\succ}(u), n(v)$ is in internal area$\}$ do
2:     for each $w \in \{w | w \in n_{\succ}(u), id(w) \succ id(v)\}$ do
3:         if $(v, w) \in E_{in}$ then
4:             output $\triangle_{uvw}$
```

Algorithm 12 ExternalCandidateVertexVertexIterator$_{\succ}(v)$

```
1: $ret \leftarrow \phi$
2: for each $u \in n_{\prec}(v)$ do
3:     $ret \leftarrow u$
4: return $ret$
```

Algorithm 13 ExternalTriangleVertexIerator$_{\succ}(v, u)$

```
1: for each $w \in \{w | w \in n_{\succ}(u), id(w) \succ id(v)\}$ do
2:     if $(v, w)$ is loaded in external area then
3:         output $\triangle_{uvw}$
```

MGT [20] is also an instance of OPT. To instantiate it, (1) no task is conducted in internal triangulation, (2) all vertices become the external candidate vertices, (3) ExternalTriangleVertexIterator is used for external triangulation, and (4) synchronous I/O is used instead of asynchronous I/O. In summary, MGT [20] is a *serial*, disk-based, *vertex iterator* method which exploits only *synchronous* I/Os. Thus, although the CPU cost of MGT is the same as the VertexIterator$_{\succ}$ instance of OPT$_{\text{serial}}$, the I/O cost of MGT is worse than OPT$_{\text{serial}}$ as follows.

$$\begin{aligned}
& Cost^{I/O}_{\text{OPT}_{\text{serial}}} = \\
& c(P(G) - \Delta^{in}_{I/O} + \Delta^{ex}_{I/O}) < cP(G) + \textstyle\sum_{i=1}^{\lceil P(G)/m_{in} \rceil} c|L_i| \\
& < (1 + \lceil P(G)/m_{in} \rceil) cP(G) = Cost^{I/O}_{\text{MGT}}
\end{aligned} \tag{7}$$

4. RELATED WORK

In-memory Methods. The early stage of triangulation methods assumed that the input graph would fit in memory. Traditionally, triangulation methods are classified into two categories depending on the iterator type. The vertex-iterator finds a triangle $\triangle_{uvw}$ when, for each vertex u, a combination $(v, w) \in n(u) \times n(u)$ is included in E. The edge-iterator finds a triangle $\triangle_{uvw}$ when, for an edge $(u, v) \in E$, there exists a common neighbor w between u and v. [2] theoretically improved the worst case complexity of the vertex-iterator method. Specifically, it first divides vertices into a high-degree vertex set V_{high} and a low-degree vertex set V_{low}. Matrix multiplication is used to count triangles in the induced subgraph of V_{high} (step 1), and the vertex-iterator without the ordering constraint is used to count triangles in which at least one vertex in V_{low} is included (step 2). The time complexity of step 1 ($O(|E|^{2\omega/\omega+1}$) dominates that of step 2 ($O(|E|^{2(\omega-1)/\omega+1})$) and becomes the time complexity of [2], where ω is the matrix multiplication exponent (e.g. 2.804 in the Strassen's algorithm). However, as we will see in Section 5.3, the method of [2] shows longer elapsed time than VertexIterator$_{\succ}$ and EdgeIterator$_{\succ}$, because the step 1 took less than 1% of the elapsed time, and the step 2 showed longer elapsed time than VertexIterator$_{\succ}$ and EdgeIterator$_{\succ}$. [28] improved the edge-iterator method using the vertex ordering based on degree. All these methods are inapplicable to large-scale graphs which do not fit in memory.

Approximation Methods. To detour the memory constraint, approximation methods were proposed. Streaming algorithms [1, 9, 13] scan the whole graph several times and estimate the triangle count. [31] samples the input graph and approximate the triangle counting using Map-Reduce. However, such methods support approximate triangle counting only, and thus their applications are significantly limited [12].

Exact Disk-based Methods. Recently, the *serial*, exact, disk-based triangulation methods were proposed [12, 20, 23]. The methods of [12] first partition the input graph to make each partition fit into the memory buffer. For each partition, it loads the partition, identifies all triangles which exist in the memory buffer, and removes edges which participate in the recognized triangles. After the whole graph is loaded into the memory buffer once, the remaining edges are merged. The partition-identifying-merging sequence is repeated until no edges remain. The method requires a significant amount of disk I/Os to conduct a sequence of reading and writing remaining edges. Such I/O overhead degrades the efficiency of those methods. Most recently, [20] proposed a disk-based method that performs read I/O only. After re-ordering vertices based on the degree, the method of [20] is a disk-based variant of the vertex-iterator triangulation method. As mentioned in Section 3.5, its I/O cost is reading the input graph as many times as the number of iterations ($\lceil P(G)/m \rceil$). Consequently, it improves the efficiency by reducing the I/O cost, but is still far from the ideal cost.

GraphChi [23] is a *parallel* disk-based graph processing system. It follows the vertex-centric programming model which processes graph operations by updating vertex values and passing messages between vertices via edge values. To support the vertex-centric programming model in a disk-based manner, GraphChi divides vertices into P execution intervals and each execution interval has a shard file which contains all edges whose target vertices are included in the execution interval. Then, it conducts a load-update-store sequence of the sub-graph for each execution interval.

For efficient graph processing, GraphChi exploits asynchronous I/O and multi-core parallelism, but the underlying mechanism is completely different from OPT. Asynchronous I/Os are conducted on only loading and storing outgoing edges of a vertex in the execution interval. For incoming edges, synchronous I/Os are used, which hinders the overlapping of CPU and I/O operations. Moreover, when both vertices of an edge are included in the same execution interval, GraphChi enforces the sequential-order processing to prevent data hazard. The enforced sequential-order processing has a negative impact on the multi-core parallelism. Accordingly, the triangle counting application of GraphChi shows much worse parallelization performance than OPT, which will be detailed in Section 5.6.

The triangle counting application of GraphChi allocates the additional memory buffer for pivoting a part of graph. At every odd iteration, it loads a part of graph into the additional memory buffer and removes edges that participate triangles identified at the previous iteration. At every even iteration, it identifies triangles by intersecting the adjacency lists in the additional memory buffer and all adjacency lists. The iteration continues until no edges remain. Like [12], the application suffers from a sequence of reading and writing remaining edges.

Distributed Method. The distributed triangle counting methods [16, 30] and triangulation method [3] which exploit Hadoop or MPI are also proposed in parallel with the disk-based triangulation method. [30] proposed a MapReduce-based triangle counting method. In the map phase, the input graph is partitioned by sending edges to reducers using a universal hash over vertices. In the reduce phase, in each partition, triangles are counted using the obtained edges to that partition. To handle triangles which are counted in multiple partitions, such triangles are accumulated to the triangle count by 1 over the number of occurrences across partitions. [3] proposed an MPI-based vertex-iterator triangulation method. The method distributes a partition of the input graph to cluster nodes, identifies triangles in each cluster node, and merges all identified triangles. [16] proposed a distributed graph processing engine named PowerGraph, and PowerGraph has a triangle counting method as one of its applications. Like GraphChi, PowerGraph proposes the Gather-Apply-Scatter (GAS) model which follows the vertex-centric programming model. To support it in a distributed manner, PowerGraph partitions the input graph using a balanced p-way vertex-cut. Sticking to the GAS model, a triangle counting method can be implemented in PowerGraph.

5. EXPERIMENT RESULT

The goals of the experiment are as follows:

- We validate the cost analysis of OPT$_{\text{serial}}$ which claims that $Cost_{\text{OPT}_{\text{serial}}}$ is close to $Cost_{\text{ideal}}$ with small overhead $c(\Delta^{ex}_{I/O} - \Delta^{in}_{I/O})$ (Section 5.3).
- We show that OPT along with thread morphing achieves ideal speed-up (Section 5.4).
- We show the insensitiveness of the elapsed time of OPT for varying the memory buffer size (Section 5.5).
- We show the linear speed-up of OPT with an increasing number of CPU cores (Section 5.6).
- We report the triangulation results on a billion-vertex scale real-world graph. To our knowledge, it is the first time such results have been reported in the literature (Section 5.7).
- We perform sensitivity analysis by varying several parameter values using a synthetic graph generator (Section 5.8).
- We show significantly better performance of OPT compared to the distributed triangulation methods (Section 5.9).

5.1 Experiment Setup

Datasets. Five real-world graph datasets were used in the experiments. LJ [4] is a sample of the LiveJournal blogger network in which bloggers are vertices, and the friend relationships between them are edges. ORKUT [25] is a sample of the orkut network which is an online social network operated by Google. TWITTER [22] is a sample of the Twitter network, which is one of the largest online social networks. UK [8] is a web graph where web pages are vertices, and hyperlinks are edges. YAHOO[1] is one of the largest real-world graphs which have over *one-billion* vertices. Table 2 shows basic statistics for the five datasets. Note that all the datasets are downloaded from the original websites.

Table 2: Basic statistics on the datasets

	LJ	ORKUT	TWITTER	UK	YAHOO
$\|V\|$	$4,847,571$	$3,072,627$	$41,652,230$	$105,896,555$	$1,413,511,394$
$\|E\|$	$68,993,773$	$223,534,301$	$1,468,365,182$	$3,738,733,648$	$6,636,600,779$
# of $\triangle$	$285,730,264$	$627,584,181$	$34,824,916,864$	$286,701,284,103$	$85,782,928,684$

Methods. OPT was compared with four state-of-the-art methods, GraphChi-Tri [23], CC-Seq [12], CC-DS [12], and MGT [20]. We implemented MGT using our OPT framework as stated in Section 3.5. OPT used EdgeIterator$_{\succ}$, which shows shorter elapsed time than VertexIterator$_{\succ}$ [28]. The memory buffer is evenly divided into the internal area and the external area to maximize the buffering effect of Line 3 of Algorithm 4. Specifically, when m pages of the memory buffer are available, $m_{in} = m_{ex} = m/2$.

To exclude the OS file system cache effect, we made OPT, MGT, CC-Seq, and CC-DS use direct I/O and made GraphChi-Tri clear the OS file system cache at each iteration. Note that when mapping vertices to ids, we used the degree-based heuristic [28] mentioned in Section 2.2, since all five methods above are based on VertexIterator$_{\succ}$ or EdgeIterator$_{\succ}$ which benefit from the degree-based heuristic.

Measure. To measure the cost, the elapsed time is used. To measure the parallelization effect, the speed-up is used, which is the elapsed time of the single thread execution over that of the multiple thread execution.

Running Environment. We conducted the experiments on two machines having the same hardware – an Intel Core i7-3930K CPU (a total of 6 CPU cores), $16GB$ RAM, and a $512GB$ FlashSSD (Samsung 830). OPT, MGT, CC-Seq, and CC-DS were executed on Windows 7, while GraphChi-Tri was executed on Linux, since OPT framework currently supports Windows platform only, while the GraphChi-Tri currently supports Linux platform only. Although GraphChi-Tri does not officially support Window 7, we ported GraphChi to Windows 7. However, due to a faster file system support in Linux[2], the ported GraphChi showed over 20% longer elapsed time in all experiments. Thus, we report the result of GraphChi-Tri on Linux.

5.2 Output Writing Cost

We performed experiments that measure the output writing times on LJ, ORKUT, TWITTER, and UK. Since the original binary of CC-Seq and CC-DS do not support output generation, in our implementation of those algorithms, we applied the bulk write method of the original implementation of MGT. Since the output writing times of CC-Seq and CC-DS were almost the same, we report that of CC-Seq. GraphChi-Tri was excluded because it is a triangle counting method which only focuses on the number of triangles. All methods used the same nested representation described in Section 3.2, and the output was written to another FlashSSD ($2TB$ RevuAhn RT8500). The memory buffer size was set to 15% of the graph size.

Table 3 shows the output writing times of OPT$_{\text{serial}}$, MGT, and CC-Seq. In all experiments, OPT$_{\text{serial}}$ shows the least output writing time, since it fully overlaps write I/O processing and CPU processing. Among the methods that do not support such overlapping,

Table 3: Output writing times of triangulation methods (sec)

	LJ	ORKUT	TWITTER	UK
OPT$_{\text{serial}}$	3.74	7.38	379.10	2858.24
MGT	6.65	12.81	555.04	3328.80
CC-Seq	16.56	41.53	1976.00	17146.80

MGT shows the least output writing time, since triangles identified by MGT have more common triangle prefixes than CC-Seq and CC-DS. Note that OPT$_{\text{serial}}$ and MGT have almost the same output sizes. Note also that OPT with the output writing step shows slightly better speedup than OPT without it. These experiments confirm that OPT is a true, parallel disk-based triangulation method regardless of output generation. In the following sections, we report the elapsed time excluding the output writing time, since our focus is to efficiently identify triangles.

5.3 Validation of Cost Analysis on OPT$_{\text{serial}}$ and Comparison to In-memory methods

To validate the analysis of OPT$_{\text{serial}}$ and to compare OPT$_{\text{serial}}$ to the state-of-the-art in-memory methods, we measured the relative elapsed time of OPT and those of the in-memory methods including VertexIterator$_{\succ}$, EdgeIterator$_{\succ}$, and [2]. The relative elapsed time is the ratio of the elapsed time of a method to that of ideal. Note that, ideal is equivalent to EdgeIterator$_{\succ}$ because OPT used EdgeIterator$_{\succ}$. When measuring the elapsed time of ideal and the in-memory methods, we temporarily used more RAM to make main memory hold the whole graph. Note that [2] is not a triangle listing method, but a triangle counting method. When implementing [2], in the matrix multiplication step, we used several state-of-the-art matrix-matrix multiplication libraries including Eigen and Intel's implementation of Strassen's algorithm, and, we reported the best elapsed time. In addition, in the vertex-iterator step, we further improved it by applying the ordering constraint when we count the triangles consisting of low-degree vertices only.

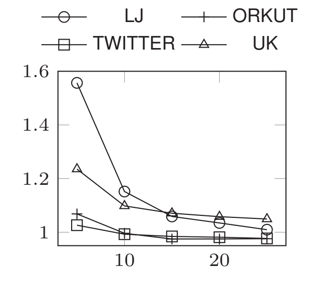
Figure 3a shows the trend of relative elapsed time of OPT$_{\text{serial}}$ with the change of the memory buffer size in LJ, ORKUT, TWITTER, and UK. The memory buffer size was varied from 5% of the graph size to 25% with 5% increments. In all datasets, the relative elapsed time decreased until 15% of the graph size was used as the memory buffer, and after that elbow point, the relative elapsed time became stabilized. At the elbow point, the relative overhead of OPT$_{\text{serial}}$ was 5.8%, -2.5%, -1.5%, and 7% in LJ, ORKUT, TWITTER, and UK, respectively. With the moderate memory buffer size (15%), OPT$_{\text{serial}}$ showed less than 7% of relative overhead and even negative overhead in the ORKUT and TWITTER datasets.

The relative overhead of OPT$_{\text{serial}}$ comes from $c(\Delta^{ex}_{I/O}$ - $\Delta^{in}_{I/O})$. As stated in Section 3.2, the page loading order of OPT can lead to the good buffering effect of the page loaded in the internal area at each next iteration. Thus, when the saved I/O cost ($c\Delta^{in}_{I/O}$) exceeds the non-overlapped I/O cost in the external triangulation ($c\Delta^{ex}_{I/O}$), we can have such negative overhead.

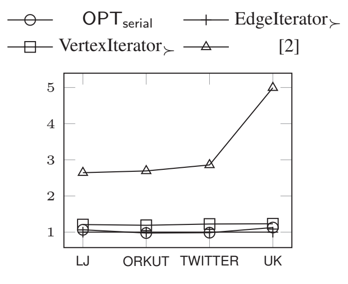
Figure 3b shows the relative elapsed time of the state-of-the-art in-memory methods compared to OPT$_{\text{serial}}$. The in-memory methods include graph loading times for fair comparison. For OPT$_{\text{serial}}$, the memory buffer size was set to 15% of the graph size. Among the in-memory methods, EdgeIterator$_{\succ}$ consistently showed the least elapsed time. Even though VertexIterator$_{\succ}$ has the same time complexity of $O(\alpha|E|)$, it was about 20% slower than EdgeIterator$_{\succ}$ in all cases, which is consistent with the results in [28]. Even though [2] has theoretically lower time complexity than EdgeIterator$_{\succ}$ and VertexIterator$_{\succ}$, it showed the longest elapsed time. This is because that (1) although counting triangles which have only high-degree vertices dominates the time complexity in theory, it took less than 1% of the elapsed time in practice, and (2) although the vertex-

[1]`http://webscope.sandbox.yahoo.com/`
[2]`http://www.phoronix.com/scan.php?page=article&item=ubuntu_win7_ws&num=4`

iterator, which is used to count the remaining triangles, is improved by applying the ordering constraint, it still shows longer elapsed time than VertexIterator$_{\succ}$ and EdgeIterator$_{\succ}$ counting each triangle only once. OPT$_{\text{serial}}$ was close to EdgeIterator$_{\succ}$ and showed better performance than VertexIterator$_{\succ}$ or [2].



(a) OPT$_{\text{serial}}$ by varying buffer size



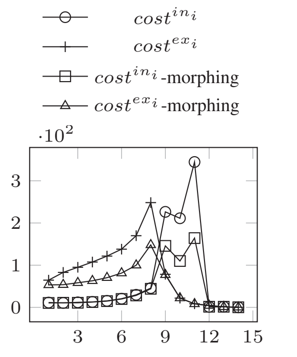
(b) OPT$_{\text{serial}}$ and in-memory methods

Figure 3: Relative elapsed time of OPT$_{\text{serial}}$ and in-memory methods

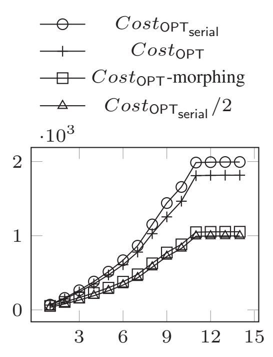
5.4 Validation of Cost Analysis on OPT and Thread Morphing

To validate the analysis of OPT and thread morphing, we compared the elapsed time of two types of threads of OPT at each iteration. In the experiment, the memory buffer size was set to 15% of the graph size, and OPT used two CPU cores and assigned one CPU core to each type of thread. The iteration count is $14(= \lceil 100/7.5 \rceil)$ as half of the memory buffer is used for the internal area.

Figure 4a shows the elapsed time trends of the main thread (the internal triangulation) and the callback thread (the external triangulation) at each iteration in UK with and without applying thread morphing. Without thread morphing, the main thread became idle until the eighth iteration, and after that, the callback thread became idle. With thread morphing, any idle thread continues to process either external or internal triangulation. Thus, the main thread was morphed to identify external triangles until the eighth iteration, while the callback thread was morphed to identify internal triangles after the eighth iteration.



(a) elapsed time at each iteration



(b) cumulative elapsed time

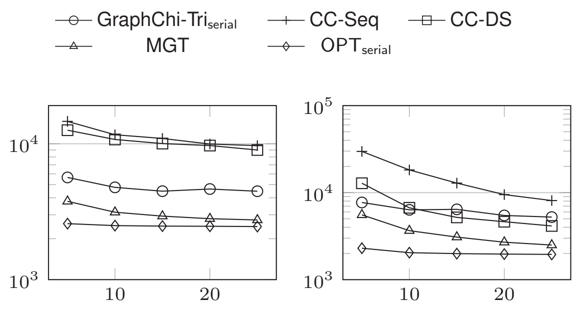
Figure 4: Thread morphing effect in UK dataset (X axis : iteration number, Y axis : time (sec)

Figure 4b shows the cumulative elapsed time trend of OPT. With thread morphing, OPT showed almost two times shorter cumulative elapsed time than that of OPT$_{\text{serial}}$. Without thread morphing, however, the cumulative elapsed time was only 1.1 ~1.3 times shorter.

5.5 Effect of Memory Buffer Size

To see the effect of the memory buffer size on the elapsed time, we varied the memory buffer size from 5% of the graph size to 25% with 5% increments. For parallel methods (OPT and GraphChi-Tri), we configured them to use a single thread. That is, OPT$_{\text{serial}}$ was used for OPT, and the configuration variable `execthread` was set to 1 for GraphChi-Tri.

Figure 5 shows the elapsed time trends of the five methods in the TWITTER and UK datasets. Due to the space limit, we omit the experiment result of the LJ and ORKUT datasets since the performance trends are similar to those of the TWITTER and UK datasets. Regardless of datasets and memory buffer size, OPT$_{\text{serial}}$ always outperformed the other four triangulation methods. GraphChi-Tri, CC-Seq, and CC-DS were 2 to 10 times slower than OPT$_{\text{serial}}$. In particular, when the memory buffer size was small, those three methods suffered from performance degradation. MGT was the closest method to OPT$_{\text{serial}}$, but as the input graph size increased, the elapsed time gap between two methods also increased – 1.11 times slower in TWITTER, and 1.25 times slower in UK.



(a) TWITTER (b) UK

Figure 5: Effect of Memory Buffer size (X axis: ratio of memory buffer to database size (%), Y axis : elapsed time (sec))

The triangulation methods are classified into two groups based on the elapsed time result – the slow group (GraphChi-Tri, CC-Seq, and CC-DS) and the fast group (MGT and OPT$_{\text{serial}}$). The main difference between the two groups is that the methods in the slow group write the remaining edges at each iteration, while those in the fast group always exploit the original input graph. Therefore, the methods in the slow group are inherently slower than those in the fast group. This analysis clearly explains the different elapsed time trends between the two groups (Figure 5). Because the methods in the fast group execute read operations only, they are relatively insensitive to the memory buffer size. However, the methods in the slow group read the whole graph and write the remaining edges at each iteration. Consequently, they are very sensitive to the memory buffer size and suffer from performance degradation in the case of small memory buffer due to excessive I/O operations.

Even though both MGT and OPT$_{\text{serial}}$ conduct read I/O operations only, OPT$_{\text{serial}}$ is always faster than MGT because it overlaps the CPU and I/O operations in the external triangulation. By overlapping the CPU and I/O operations in the external triangulation, OPT$_{\text{serial}}$ achieves less I/O cost than MGT (see Eq.7).

The performance results also show that OPT is more efficient than the others when the buffer size is small. This feature is especially important when we handle very large graphs with a limited-size buffer in a single PC.

5.6 Effect of Number of CPU Cores

To assess the parallelization effect, we compared the elapsed times and the speed-ups of the parallel triangulation methods by varying the number of CPU cores. In this experiment, the memory buffer size was fixed as 15% of the graph size. The number of CPU cores was varied from 1 to 6.

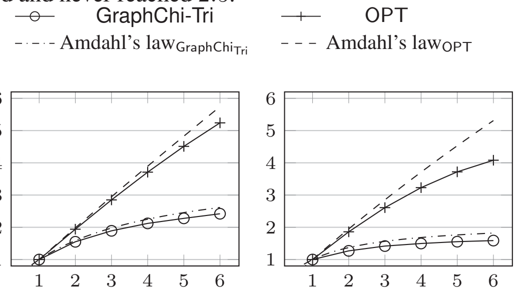
Table 4 shows the elapsed times of OPT and GraphChi-Tri in the LJ, ORKUT, TWITTER, and UK datasets using 1 and 6 CPU cores. In every combination of datasets and the number of CPU

Table 4: Elapsed time comparison of OPT and GraphChi-Tri using 1 and 6 CPU cores

	LJ	ORKUT	TWITTER	UK
$\text{OPT}_{\text{serial}}$	17.05	83.32	2477.55	1966.93
$\text{GraphChi-Tri}_{\text{serial}}$	105.427	304.361	4477.29	6424.59
OPT	6.39	18.51	469.40	480.918
GraphChi-Tri	85.87	196.95	1850.26	4046.77
GraphChi-Tri/OPT	13.44	10.64	3.94	8.41

cores, OPT always showed shorter elapsed time than GraphChi-Tri. OPT outperformed GraphChi-Tri by up to 13.44 times.

Figures 6a and 6b show the trends of the relative speed-up of OPT and GraphChi-Tri in the TWITTER and UK datasets as the number of CPU cores increases. As the number of cores increased, the speed-up of OPT increased linearly. In all datasets, OPT always showed much higher speed-up than GraphChi-Tri regardless of the number of cores. In contrast, the speed-up of GraphChi-Tri saturated and never reached 2.5.



(a) TWITTER (b) UK

Figure 6: Effect of CPU cores (X axis: # of CPU cores, Y axis: speed-up)

The main reason for the different speed-up of the two parallel methods is that OPT has a greater parallelizable portion in its operations than GraphChi-Tri. According to Amdahl's law, theoretically, when c cores are available, and $p \in (0, 1]$ is the parallel fraction of a parallel method, the upper bound of speed-up, ub^c, is $\frac{1}{(1-p)+\frac{p}{c}}$. The dashed lines are the upper bound of both methods inferred by Amdahl's law. In OPT and GraphChi-Tri, the CPU operations that intersect two adjacency lists are parallelizable, and the rest of the CPU and I/O operations are not. Tables 5 shows (1) the parallel fraction of OPT and GraphChi-Tri, (2) the upper bound of the speed-up, and (3) the empirical speed-up when 6 cores were used. From the table, in all datasets, OPT always has higher parallel fraction ($> 95\%$) than GraphChi-Tri ($< 75\%$). That leads to the higher upper bound speed-up and empirical speed-up of OPT than those of GraphChi-Tri.

Table 5: Speed-up of OPT and GraphChi-Tri using 6 cores

method	measure	LJ	ORKUT	TWITTER	UK
OPT	p	0.961	0.980	0.989	0.975
	ub^6	5.03	5.45	5.70	5.34
	$speedup^6$	2.62	4.45	5.24	4.08
GraphChi-Tri	p	0.271	0.490	0.747	0.544
	ub^6	1.30	1.69	2.68	1.83
	$speedup^6$	1.23	1.54	2.42	1.59

5.7 Comparison on 1-billion Vertex Graphs

We performed the experiments on the larger graph which has over 1-billion vertices. For this purpose, we obtained the YAHOO dataset which is considered to be as the largest real-world graph data publicly available and has 1.4 billion vertices. In these experiments, OPT, MGT, and GraphChi-Tri were used, and CC-Seq and CC-DS were excluded, since they are clearly inferior to the other methods. The memory buffer size was set to $10GB$.

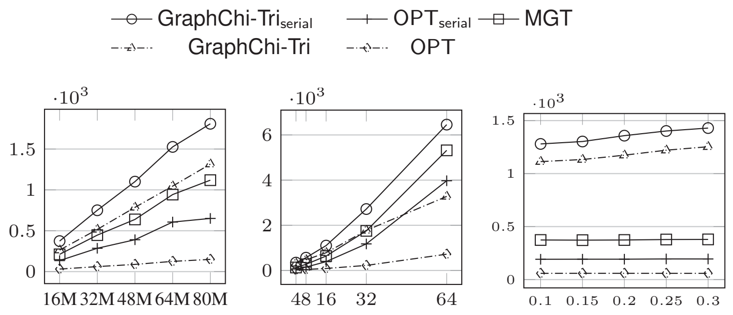
Table 6 shows the elapsed times of the triangulation methods on the YAHOO dataset. To the best of our knowledge, this is the first time that triangulation result on a billion-vertex scale real-world graph is reported. $\text{OPT}_{\text{serial}}$ showed 2.04 and 5.25 times shorter elapsed time than MGT and $\text{GraphChi-Tri}_{\text{serial}}$. When 6 cores are used, OPT showed 31.36 times shorter elapsed time than GraphChi-Tri. Although the number of triangles in this dataset is relatively small compared with the other real datasets, the speed-up of OPT reached 3.25, while that of GraphChi-Tri was only 1.11. In summary, OPT shows reasonable performance even for billion-scale graphs, consistently achieving the shortest elapsed time compared with its competitors.

Table 6: Elapsed time on YAHOO (sec)

$\text{OPT}_{\text{serial}}$	MGT	$\text{GraphChi-Tri}_{\text{serial}}$	OPT	GraphChi-Tri
2665	5445	28568	819	25686

5.8 Comparison on Synthetic Datasets

We compare performance of triangulation methods in synthetic datasets. We generated synthetic datasets that using the R-MAT model [10], which is well known for its simplicity and expressive power that subsumes Erdos-Renyi model [15] and power-law distribution. We used the publicly available implementation of R-MAT[3] with the default parameter used in [10]. We varied the number of vertices, $|V|$, and the density of graph, $|E|/|V|$ – when varying $|V| = 16M, 32M, 48M, 64M$, and $80M$, we fixed $|E|/|V| = 16$, and when varying $|E|/|V| = 4, 8, 16, 32$, and 64, we fixed $|V| = 48M$. The memory buffer size was set to 15% of the graph size. The same set of methods in Section 5.7 was used.



(a) varying number of vertices (b) varying density (c) varying clustering coefficient

Figure 7: Comparison on synthetic datasets (X axis: # of vertices, density, clustering coefficient, Y axis: elapsed time)

Figure 7a shows the trend of the elapsed time as the number of vertices increases. For the serial methods which are plotted as straight lines, $\text{OPT}_{\text{serial}}$ showed shorter elapsed time than MGT, and the processing time gap between them increased with the increase of $|V|$. When $|V| = 16M$, $\text{OPT}_{\text{serial}}$ took 1.57 times shorter time than MGT, and when $|V| = 80M$, $\text{OPT}_{\text{serial}}$ took 1.72 times shorter time. For the parallel methods which are plotted as dashed lines, the speed-up of both methods did not change much – the speed-up of OPT was around 4.5, and that of GraphChi-Tri was around 1.4. Again, OPT showed shorter elapsed time and higher speed-up than GraphChi-Tri in all cases. When $|V| = 80M$, OPT showed 12.13 shorter elapsed time than GraphChi-Tri, and the speed-up of OPT was 4.35 but that of GraphChi-Tri was 1.37.

Figure 7b shows the trend of the elapsed time as the density of graph increases. For the serial methods, $\text{OPT}_{\text{serial}}$ showed 1.33~2.01 times shorter elapsed time than MGT. For the parallel methods, OPT and GraphChi-Tri showed higher speed-up as the density increases – OPT increased from 2.67 to 5.51, and GraphChi-Tri increased from 1.12 to 1.97. Again, OPT achieved shorter elapsed time and higher speed-up than GraphChi-Tri.

[3]http://www.cse.psu.edu/madduri/software/GTgraph

Along with varying the density, we conducted the experiment by varying the clustering coefficient. Because the R-MAT model cannot control the clustering coefficient, we extensively surveyed the literature and found the model of [19] that follows the power-law degree distribution and *controls* the clustering coefficient. Following the clustering coefficient range of the real-world graphs (LJ: 0.28, ORKUT: 0.17), we varied the clustering coefficient from 0.1 to 0.3 with a 0.05 interval, while fixing $|V| = 48M$. To get the indented range of the clustering coefficient, the average degree, $|E|/|V|$, was set to 10.

Figure 7c shows the trend of the elapsed time as the clustering coefficient increases. The elapsed times of OPT, $\text{OPT}_{\text{serial}}$, and MGT remained constant regardless of the clustering coefficient since the time complexity of intersection of two adjacency lists depends on the average degree, not on the clustering coefficient. OPT showed about two times shorter elapsed time than MGT. The speed-up of OPT reached 3.25, while that of GraphChi-Tri was only 1.16. When 6 CPU cores were used, OPT showed 21.60 times shorter elapsed time than GraphChi-Tri.

Additionally, we conducted experiments using another synthetic graph generator [29] which can control the clustering coefficient. In this experiment, we varied the clustering coefficient from 0.1 to 0.5 with a 0.1 interval. We observed that the elapsed time of OPT also remained constant.

5.9 Comparison to Distributed System Implementation

We compared the relative performance of OPT to the state-of-the-art distributed triangle counting methods (SV [30] and PowerGraph [16]) and triangulation method (AKM [3]). We implemented SV in Hadoop and AKM in C++-MPI, and used the publicly available C++ source code of PowerGraph. For fair comparison, we used a 32-node cluster system. The distributed triangulation methods used 31 nodes, while OPT used one node. Each node is equipped with two Intel Xeon X5650 CPUs (a total of 12 CPU cores) and $24GB$ RAM. The number of threads was set to the number of the available CPU cores. Specifically, OPT used 12 threads, and SV, AKM, and PowerGraph used $372 (= 12 \times 31)$ threads.

Table 7 shows the distributed triangulation methods, their hardware settings, and their elapsed times for the TWITTER dataset. OPT took 7.03 minutes using the memory buffer of 15% of the graph size. SV [30] took 452.2 minutes and showed 64.32 times longer elapsed time than OPT. AKM [3] took 10.14 minutes and showed 1.44 times longer elapsed time. PowerGraph [16] took 5.38 minutes and showed 1.31 times shorter elapsed time. Considering that the distributed methods use 31 nodes, OPT shows 1994.05, 44.71, and 23.72 times better relative performance than SV, AKM, and PowerGraph, respectively.

Table 7: Comparison with distributed methods in TWITTER

Method	Framework	Hardware setting	# of machines	Elapsed time
OPT		2 CPUs, 12 cores, $24GB$ RAM	1	7.03min
SV	Hadoop			452.2min
AKM	MPI	2 CPUs, 12 cores, $24GB$ RAM	31	10.14min
PowerGraph	MPI			5.38min

6. CONCLUSION

In this paper, we proposed an overlapped and parallel disk-based triangulation framework, OPT, in a single PC of the multi-core CPU and the FlashSSD. When a graph does not fit in main memory, we first identify two types of triangles – the internal triangles and the external triangles. The overlap of the I/O and CPU processing and the multi-core parallelism make OPT exploit a two-level overlapping strategy. At the macro level, OPT overlaps the two types of graph triangulation using the multi-core parallelism and FlashSSD parallelism. The macro level overlapping and thread morphing make OPT achieve the linear speed-up with an increasing number of CPU cores. At the micro level, OPT overlaps the I/O and CPU processing using the I/O and CPU processing overlapping using the asynchronous I/Os of the FlashSSD. The micro level overlapping makes OPT have the cost close to that of the ideal triangulation method. In addition, OPT is generic in that OPT can instantiate both vertex-iterator and edge-iterator triangulation models. Extensive experiments conducted on large-scale datasets showed that OPT achieved the ideal cost with less than 7% overhead even under the limited memory budget and achieved the linear speed-up and more than an order of magnitude shorter elapsed time than the state-of-the-art parallel triangulation method, when 6 CPU cores were used. Overall, we believe our overlapped and parallel triangulation method provides comprehensive insight and a substantial framework for future research such as the subgraph listing problem.

Acknowledgement

This research was supported by Next-Generation Information Computing Development Program through the National Research Foundation of Korea(NRF) funded by the Ministry of Education, Science and Technology (No. 2012M3C4A7033342, No. 2012M3C4-A7033344).

References

[1] N. Alon et al. The space complexity of approximating the frequency moments. STOC '96.
[2] N. Alon et al. Finding and counting given length cycles. *Algorithmica*, 1997.
[3] S. Arifuzzaman et al. Patric: A parallel algorithm for counting triangles in massive networks. CIKM '13.
[4] L. Backstrom et al. Group formation in large social networks: membership, growth, and evolution. KDD '06.
[5] V. Batagelj and A. Mrvar. A subquadratic triad census algorithm for large sparse networks with small maximum degree. *Social Networks*, 2001.
[6] V. Batagelj and M. Zaveršnik. Short cycle connectivity. *Discrete Mathematics*, 2007.
[7] L. Becchetti et al. Efficient semi-streaming algorithms for local triangle counting in massive graphs. KDD '08.
[8] P. Boldi et al. Layered label propagation: a multiresolution coordinate-free ordering for compressing social networks. WWW '11.
[9] L. S. Buriol et al. Counting triangles in data streams. PODS '06.
[10] D. Chakrabarti et al. R-mat: A recursive model for graph mining. SDM'04.
[11] N. Chiba and T. Nishizeki. Arboricity and subgraph listing algorithms. *SIAM J. Comput.*, Feb. 1985.
[12] S. Chu and J. Cheng. Triangle listing in massive networks and its applications. KDD '11.
[13] D. Coppersmith and R. Kumar. An improved data stream algorithm for frequency moments. SODA '04.
[14] J.-P. Eckmann and E. Moses. Curvature of co-links uncovers hidden thematic layers in the World Wide Web. PNAS'02.
[15] P. Erdos and A. Renyi. On random graphs i. *Publ. Math. Debrecen*, 1959.
[16] J. E. Gonzalez et al. Powergraph: distributed graph-parallel computation on natural graphs. OSDI'12.
[17] W.-S. Han et al. Turbograph: a fast parallel graph engine handling billion-scale graphs in a single pc. KDD '13.
[18] F. Harary and H. J. Kommel. Matrix measures for transitivity and balance. *Journal of Mathematical Sociology*, 1979.
[19] P. Holme and B. J. Kim. Growing scale-free networks with tunable clustering. *Phys. Rev. E*, 2002.
[20] X. Hu et al. Massive graph triangulation. SIGMOD '13.
[21] A. Itai and M. Rodeh. Finding a minimum circuit in a graph. STOC '77.
[22] H. Kwak et al. What is twitter, a social network or a news media? WWW '10.
[23] A. Kyrola et al. Graphchi: large-scale graph computation on just a pc. OSDI'12.
[24] M. Latapy. Main-memory triangle computations for very large (sparse (power-law)) graphs. *Theor. Comput. Sci.*, 2008.
[25] A. Mislove et al. Measurement and analysis of online social networks. IMC '07.
[26] A. Prat-Pérez et al. Shaping communities out of triangles. CIKM '12.
[27] T. Schank. *Algorithmic Aspects of Triangle-Based Network Analysis*. PhD thesis, Universität Karlsruhe, 2007.
[28] T. Schank and D. Wagner. Finding, counting and listing all triangles in large graphs, an experimental study. WEA'05.
[29] C. Seshadhri et al. Community structure and scale-free collections of erdős-rényi graphs. *Physical Review E*, 85(5):056109, 2012.
[30] S. Suri and S. Vassilvitskii. Counting triangles and the curse of the last reducer. WWW '11.
[31] C. E. Tsourakakis et al. Doulion: counting triangles in massive graphs with a coin. KDD '09.