Distributed Massive Graph Triangulation

Ilias Giechaskiel
Magdalene College

UNIVERSITY OF CAMBRIDGE

A dissertation submitted to the University of Cambridge in partial fulfillment of the requirements for the degree of Master of Philosophy in Advanced Computer Science (M.Phil Project Option)

University of Cambridge
Computer Laboratory
William Gates Building
15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom

Email: ig305@cam.ac.uk

June 11, 2014

Declaration

I Ilias Giechaskiel of Magdalene College, being a candidate for the M.Phil in Advanced Computer Science, hereby declare that this report and the work described in it are my own work, unaided except as may be specified below, and that the report does not contain material that has already been used to any substantial extent for a comparable purpose.

Total word count: 14,315 (excluding Appendices A and B)

Signed:

Date: June 11, 2014

Acknowledgments

I would like to thank Dr. Eiko Yoneki for being my adviser, and Dr. Weixiong Rao for his helpful bootstrapping remarks. I would additionally like to thank Barouch Giechaskiel and Bryan Richter for their comments on preliminary versions of this document and Emilia Koufa for her overall support and encouragement. Finally, I am grateful to Magdalene College and the John L. Goulandris scholarship, without which I would have been unable to attend the University of Cambridge and the Computer Laboratory.

Abstract

Triangle *listing* — finding and storing all the 3-cliques (or "triangles") of a graph — and the related problem of triangle *counting* — finding the total number of triangles in a graph — have recently attracted attention in the algorithmic community. Applications of triangle listing include dense neighborhood discovery and triangular connectivity, as well as important density metrics such as the clustering coefficient and the transitivity ratio, for which triangle listing can be used as a blackbox. Although triangles can be easily counted when graphs fit in memory, social and other online networks often exceed tens of gigabytes in size, causing in-memory algorithms to incur frequent trashing of memory due to the need for non-sequential access to the neighbors of a vertex. As a result, the triangle counting literature has been focusing on external-memory algorithms that minimize random I/Os. These external-memory algorithms are sequential, so other researchers have invented new ways to parallelize triangle counting across machines, but at the expense of random disk accesses.

This project set out to combine these two approaches, which are inherently in conflict, and produced the first framework that provides guarantees for CPU, I/O, Network, and Memory efficiency. We based our algorithm on Hu et al.'s state-of-the-art external-memory algorithm Massive Graph Triangulation (MGT) [HTC13] as a starting point, to create our Distributed Massive Graph Triangulation (DMGT) framework. We also provided theoretical guarantees and proofs for our algorithm's performance, and uncovered flaws, missing assumptions, and implementation errors in the MGT binary provided by its authors, which we fixed in our system.[1] We also showed that our algorithm performs well both in a single-core, low-memory environment, as well as in a multi-machine, multi-core distributed cluster — scaling well, and outperforming the state-of-the-art distributed triangle counting system PATRIC [AKM13] by 3×, using 36% fewer resources.

Our contributions, however, are not triangle-listing specific: our approach shows that by bringing external storage back into the distributed environment, we can build scalable algorithms, without sacrificing performance.

[1] No source code was provided by the original authors, so our implementation was completely independent.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Graphs have become important abstractions to model real-world situations, ranging from social relationships to communication, web, and road networks. Various local and global properties pertaining to the connectivity and density of graphs can be measured, and one such metric of interest is the *number of triangles*. As we discuss in more details in Chapter 2, triangle counting is often used as a blackbox in further algorithms pertaining to the density and clustering coefficient of subgraphs, but is also an important metric on its own, since triangles are the smallest non-trivial cliques and cycles.

When the graph is small enough to fit in memory, triangle listing and enumeration is straightforward (Section 3.1), but, as graphs grow in size beyond millions of vertices and billions of edges, frequent trashing of memory to the disk occurs, due to the inherent need for non-sequential access. Publicly available graphs of social networks and other online communities range in the tens of millions of vertices and over a billion edges, occupying several gigabytes of space [KLPM10, YL12]. For this reason, external-memory, I/O-efficient algorithms have been dominating the triangle-listing literature [Dem06, Men10, CC11, CC12, HTC13] (Section 3.2), but their sequential approach makes them time-consuming for larger graphs: we show in Section 5.4 that it takes 45 minutes to count the number of triangles on the Twitter graph [KLPM10], even when allocating tens of GBs of memory to the

state-of-the-art algorithm.

On the other hand, triangle-counting distributed systems are few and far between (Section 3.3), and either use frameworks like MapReduce, which are inherently unsuited for such task and generate lots of intermediate networking data [Coh09, SV11, PC13], or ignore the time taken for loading and distributing the graph across machines [GLG+12, Low13] (Section 3.5). Alternatives include using streaming online-algorithms and approximations to overcome the inherent difficulty of this problem, but such algorithms often do not provide the necessary accuracy (Section 3.4).

As far as we know, only one distributed framework specific to triangle counting, PATRIC [AKM13], provides overall efficiency, but at the cost of higher memory requirements, and lack of I/O-efficiency, as it relies on random disk accesses. Thus, this project set out to combine the conflicting requirements of I/O, CPU, Memory, and Network efficiency, and we believe it has successfully done so by introducing external-memory accesses to the distributed environment. Specifically, we altered the state-of-the-art external-memory triangle listing algorithm, Massive Graph Triangulation (MGT) [HTC13], and implemented it in the distributed environment, creating our Distributed Massive Graph Triangulation (DMGT) framework.

In summary, our contributions are as follows:

- We explain why no other external-memory algorithm would be suitable for such a distributed environment (Section 4.1).
- We uncover hidden assumptions in the proofs and implementation of the closed-source MGT algorithm (Section 4.2).
- We modify the theoretical MGT algorithm to correspond to its implementation (Section 4.2.3), and prove that our modifications do not alter the theoretical efficiency of the algorithm (Section 4.2.4).
- We create a general, extensible, and cross-platform framework for triangle listing and counting that implements both distributed and single-machine versions of our algorithm (Section 4.3.1).

- We prove the theoretical efficiency of our algorithm (Section 4.3.3), making it the first triangle listing algorithm that provides efficient and well-understood bounds on CPU, I/O, Memory, and Network utilization, across multiple environments. Specifically, we prove Theorem 4.3.1, stating that for triangle listing, and using R machines with P processors and M memory per processor each, our DMGT algorithm incurs a total of:
  - $\Theta\left(RP + R|E| + T\right)$ Network traffic
  - $\mathcal{O}\left(RP|E| + \frac{|E|^2}{M} + \alpha|E|\right)$ CPU computations
  - $\mathcal{O}\left(RP\frac{|E|}{B} + \frac{|E|^2}{MB} + \frac{T}{B}\right)$ I/Os

  where E is the edge set of the graph, B the disk block size, T the number of triangles, and α the arboricity of the graph (Definition 4.2.1).

- We test our algorithm with graphs up to 66 million vertices and 1.8 billion edges, occupying over 14GB of space in binary format. Our algorithm works on a variety of environments, from one single-core machine to multiple multi-core machines, with typically less than 1GB of memory per core. We experimentally show that our algorithm is highly scalable across multiple cores and multiple machines, with lower memory requirements compared to other approaches (Chapter 5).

- We compare our algorithm with the state-of-the-art distributed system PATRIC [AKM13], which uses 200 processors and 4GB of memory per processor. On the largest common dataset, the Twitter graph [KLPM10] with 42 million vertices and 1.2 billion edges, our algorithm is 3 times faster than PATRIC when using 128 processors and 1GB of memory per processor, and is still faster than PATRIC, even when using only 16 cores and 1GB of memory per core (Section 5.6).

- Finally, we also discover that the closed-source implementation of MGT by its authors contains an implementation flaw that causes the program to miscount triangles in larger graphs (Section 5.7).

Though the exact algorithmic details of our DMGT framework might not be transferable to other problems, the idea of duplicating the graph across machines can prove to be beneficial in the parallelization of other external-memory algorithms which require super-linear disk accesses. Overall, we believe that our novel approach of focusing on disk-based access on a distributed system is a more scalable alternative than requiring large amounts of memory on each of the machines — especially for larger graph processing — and we hope that our work will inspire future research in this area, even outside of the triangle-listing community.

Chapter 2

Background

In this chapter, we remind the reader of some pertinent definitions of graph concepts (Section 2.1) and discuss applications of triangle listing (Section 2.2), so that the need for efficient triangle listing can be appreciated more fully. We also present an I/O model which will help us compute the asymptotic behavior of our disk accesses (Section 2.3). Notation introduced here and in subsequent chapters is summarized in Appendix A.

2.1 Definitions

In this section we introduce some useful preliminary notions. Unless otherwise noted, we are concerned only with undirected (bi-directional) graphs $G = (V, E)$ on $n = |V|$ vertices and $m = |E|$ edges, where edge $(u, v) \in E$ if and only if $(v, u) \in E$. We denote by $N_G(u) = \{v : (u, v) \in E\}$ the set of *neighbors* (or *adjacency list*) of u, and $d_G(v) = |N_G(u)|$ its degree. We may omit the qualifier G when doing so is clear and cannot cause confusion. Additionally, we let $d_{max}(G) = \max_{v \in V} d_G(v)$ denote the maximum degree in G. Finally, we assume that G is simple — i.e. does not contain parallel edges or loops — and contains no isolated vertices — i.e. vertices v of degree $d(v) = 0$. These assumptions are not necessary, but simplify the discussion,

by allowing more concise descriptions.

Definition 2.1.1 (Triangle)**.** *Given an undirected graph $G = (V, E)$, a triangle is a set of three (distinct) vertices $\{u, v, w\} \subseteq V$, such that all of (u, v), (v, w) and (w, u) are edges in E.*

Triangles are interesting structures within a graph, because they are also the smallest non-trivial examples of cliques and cycles. Finding the set K of all such triangles is called triangle *listing*, and merely reporting on their number $T = |K|$ is called triangle *counting*. It is useful when reporting triangles to define an ordering on the vertices such that triangles are uniquely identified by a single permutation out of the six possible ones for the set $\{u, v, w\}$.

Definition 2.1.2 (Strict Total Order)**.** *Given a set S, the binary relation $\prec$ is a strict total order on S, if both the following hold for all $a, b, c \in S$:*

- *Exactly one of $a \prec b$, $a = b$, and $b \prec a$ holds (trichotomy)*
- *If $a \prec b$ and $b \prec c$, then $a \prec c$ holds (transitivity)*

Definition 2.1.3 (Orientation)**.** *Given a graph $G = (V, E)$, and a strict total order $\prec$ on V, define the directed graph $G^* = (V, E^*)$ by $(u, v) \in E^*$ if and only if $(u, v) \in E$ and $u \prec v$. We call G^* the orientation of G under $\prec$.*

It is clear by the definition of the strict total order that not both (u, v) and (v, u) can be edges in G^*, so a strict total order uniquely associates the triangle $\{u, v, w\}$ where $u \prec v \prec w$ with the tuple (u, v, w). To unambiguously define this triangle, the following notation has been introduced [HTC13]:

Definition 2.1.4 (Cone Vertex, Pivot Edge)**.** *Given a triangle (u, v, w) with $u \prec v \prec w$ in G^*, we call u its cone vertex, and (v, w) its pivot edge.*

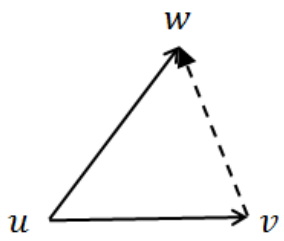


Figure 2.1: A triangle with cone vertex u and pivot edge (v, w)

Definition 2.1.4 is illustrated in Figure 2.1.

2.1.1 Graph Representation

Because operations on integers are fast, we identify the vertex set V with the set of the first n non-negative integers $[n] = \{0, \ldots, n-1\}$.[1] Additionally, even though the natural relation $<$ on the the set of natural numbers already defines a strict total order on V, it is beneficial to define a new strict total order for graphs, which leads to a better asymptotic runtime for our algorithm (Theorem 4.2.3).

Definition 2.1.5 (Degree-Based Order)**.** *Given an undirected graph $G = (V, E)$, the degree-based order $\prec$ on V is defined as follows: $u \prec v$ if and only if $d(u) < d(v)$ or $d(u) = d(v)$ and $u < v$.*

2.2 Applications

The need for efficient triangle listing and counting becomes apparent, once one realizes the magnitudes of publicly available datasets. LiveJournal, an online, and relatively small, blogging community, boasts over 4.8 million members, and 69.0 million friendships [LLDM08]. Data from Twitter released by researchers show over 41.7 million profiles, with 1.5 billion relations among them [KLPM10]. Finally, the Friendster gaming network contains 65.6 milion nodes, with over 1.8 billion edges among them [YL12], while Facebook, a larger social network, had, in 2011, more than 721 million registered users forming 68.7 billion friendships [UKBM11]. These quantities will only increase in years to come. At the same time, the versatility of the graph abstraction does not restrict us to social graphs. Instead, it also allows analysis of even road, citation, and other large networks of similarly large size [CC11, CC12, HTC13, SW05].

[1] This assumption is by no means restrictive, as we can use a mapping between any set of unique identifiers and $[n]$, but the simplicity and efficiency in graph representation and processing poses a considerable advantage, as we will see in Chapter 4.

The applications of triangle listing and counting also have a wide range. On the more theoretical end, triangle counting is a special case of counting cycles of given length [AYZ97], counting complete subgraphs [KKM95], and finding small subgraphs, called "motifs" [MSOI+02]. However, triangles are also useful in network analysis, and one metric of interest involving the number of triangles is the clustering coefficient [WS98]:

Definition 2.2.1 (Clustering Coefficient)**.** *Given a graph $G = (V, E)$, the quantity $C(v) = \frac{2T(v)}{d(v)(d(v)-1)}$ is the (local) clustering coefficient (or neighborhood density [CC12]) for vertex $v \in V$, where $T(v)$ denotes the number of triangles in G containing v. The (global) clustering coefficient for the graph G is $C(G) = \frac{1}{|V|} \sum_{v \in V} C(v)$.*

Essentially, the clustering coefficient represents the proportion between the actual number of triangles $T(v)$ around v, and the maximum possible such number $\binom{d(v)}{2}$. Perhaps surprisingly, the most I/O-efficient algorithm for this measure of density resorts to triangle listing as a first step [CC12].

A related metric is the transitivity ratio [OP09]:

Definition 2.2.2 (Transitivity Ratio)**.** *Given $G = (V, E)$, let $D = \frac{1}{3} \sum_{v \in V} \binom{d(v)}{2}$ denote the maximum number of triangles, and $T = \frac{1}{3} \sum_{v \in V} T(v)$ the actual number of triangles. The transitivity ratio (or simply transitivity [CC12]) for the graph is defined by the ratio $\frac{T}{D}$.*

Though the clustering coefficient and transitivity ratio for a given graph are often distinct, the latter is often also referred to as the global clustering coefficient, causing some notational confusion. Even so, such metrics can be used to detect fake accounts in social networks [YWW+14], as well as web spam and content quality [BBCG08].

Dense neighborhood discovery [WZTT10] is a further application of triangle listing: finding connected subgraphs where all connected pairs of vertices are part of "many" triangles within that graph. The proposed algorithm utilizes triangle listing as a blackbox, and can be used as yet another density indicator. Moreover, *triangular clustering* (or *triangular connectivity*) [BZ07,

Sch07] partitions the graph into equivalence classes where $u \sim v$ if and only if there is a sequence of consecutive triangles with at least one vertex in common leading from one vertex to the other. Additionally, the *k-truss* of a graph G is its maximum subgraph such that every edge appears in at least $k-2$ triangles, and can be used to identify "important" sections of a graph [Coh09, WC12]. Unsurprisingly, both of these algorithms require a fast triangle listing subroutine.

These applications show that triangle listing and counting are important procedures for graph analysis, and thus require CPU-efficient algorithms. At the same time, the sizes of the graphs in question better motivate our goal of also focusing on I/O efficiency, since in the absence of enough memory, frequent disk accesses are necessary.

2.3 I/O Complexity

Like traditional CPU complexity analysis, I/O complexity analysis attempts to describe the asymptotic behavior of a program. Unlike CPU analysis, however, I/O analysis solely focuses on the number of reads and writes, and was first introduced by Aggarwal and Vitter in 1988 [AV88]. Due to the way rotational hard disk drives (HDDs) operate, I/O analysis is performed in terms of not individual elements accessed, but entire blocks of data. Although individual disks have different characteristics, the filesystem presents a uniform view to applications, through units called *blocks*.[2] Although there are multiple possible settings for the block *size* B, which depend on the operating system, filesystem, and user, B is a power of 2, and typically ranges from 512 bytes to 128 kilobytes (KB), with 4KB being most common on Windows [Sup]. Consequently, in accessing N elements in order, the disk performs $scan(N) = \Theta(N/B)$ I/Os, whereas random access can require $\Omega(N)$ I/Os in the worst case. This, of course, assumes that the memory size

[2]The term block in the literature is often used inconsistently, and typically blocks equal multiple physical sectors of the disk. We prefer to use the term to refer to filesystem blocks, as applications typically do not directly interact with the hardware.

M is sufficiently big to store B elements in memory, but such assumption is clearly realistic for the aforementioned block sizes. Sorting similarly takes $sort(N) = \Theta\left(\frac{N}{B} \log_{M/B} \frac{N}{B}\right)$ I/Os by external mergesort [AV88]. Even though the original as well as newer proposed models [VS94] accounted for the case of $P > 1$ "parallel block transfers", these have not, in practice, been adopted.

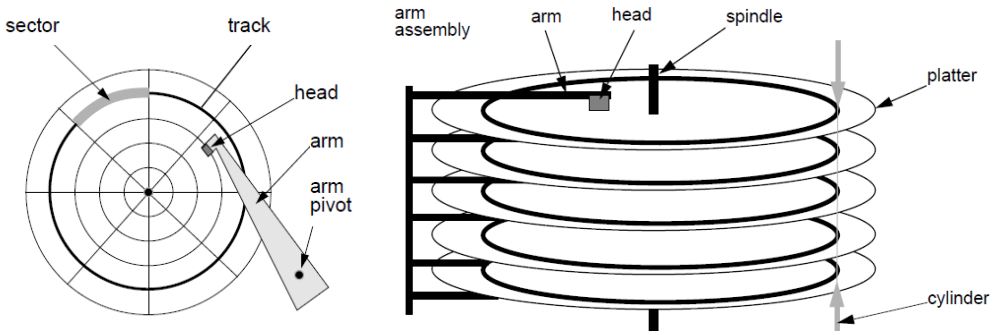
At the same time, however, there has been a trend of attempting to mitigate the I/O bottleneck present in breadth-first search (BFS) of large graphs, either by scalable, parallel BFS versions for specific platforms [SVP08, YCH+05], or by introducing special in-memory data structures and clever pre-fetching to hide the complexity from the programmer [YNDR14]. Nonetheless, both of these approaches are impractical for the triangle listing problem, since triangle counting does not require a full BFS.[3] Consequently, we have to resort to traditional disk accesses, but since older rotational HDDs, and newer solid state drives (SSDs) behave differently due to their intrinsic hardware dissimilarities, it is worth discussing them separately in Section 2.3.1.

2.3.1 Disk Characteristics

Older HDDs are primarily composed of mechanical parts, as shown in Figure 2.2. Each HDD contains a stack of rotating platters, with a dedicated head per platter that completes read and write operations, even though the overall disk has a unique read-write channel that switches between the heads [RW94]. The physical movement of the arm to place the head on the correct cylinder/track and perform the access operation dominates I/O time when accesses are not sequential, especially because it invalidates read-ahead caching of results [RW94]. The latter, however, is the responsibility of the micro-controller, which is in turn also determined by the disk's design and interface [ADR03].

SSDs, on the other hand, typically rely on NAND-flash packages, which are combined and interleaved in arrays enabling higher bandwidth, due to the

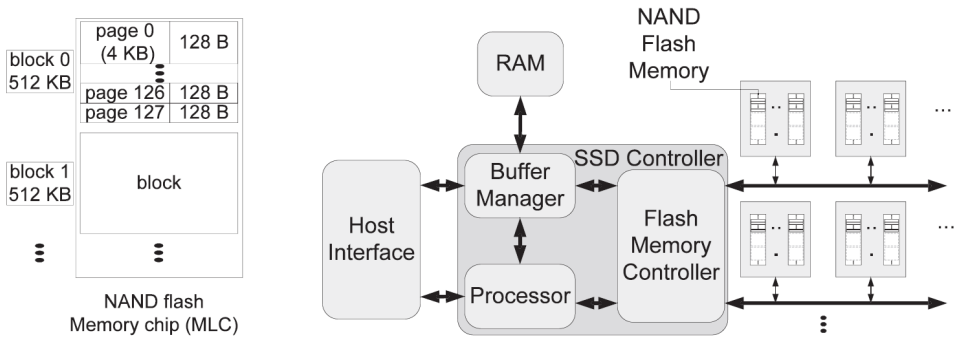
[3] Additionally, the former approach is overly system-specific, and the latter requires higher memory utilization.



(a) Platter (b) Mechanical parts

Figure 2.2: HDD components [RW94]

multiple levels of parallelism [APW+08] (Figure 2.3). The SSD controller is much more complicated than its HDD counterpart. The physical data layout must be conducive to internal parallelism, indicating that large writes are beneficial for subsequent large reads [CLZ11]. Thus, even though parallelizing random I/O requests certainly achieves better throughput than consecutive random I/Os, a single large read is preferable, due to the buffering effects of read-ahead [CLZ11].



(a) NAND-flash (b) SSD components

Figure 2.3: SSD internals [KSJ+12]

Consequently, despite the fact that HDDs and SSDs have orders of magnitude differences in performance in terms of latency and bandwidth, large sequential reads are preferable for both of them. Moreover, because operat-

ing systems often take advantage of pre-fetching [CFKL95], and because we do not want our algorithm and analysis to be dependent on the underlying hardware,[4] this simple principle of issuing large sequential reads will guide our algorithmic design in Chapter 4.

[4] Although most present-day computers make use of SSDs, even as just an intermediate caching layer, just 5 years ago, migrating servers to them was deemed too costly [NTD+09].

Chapter 3

Related Work

The problems of triangle listing and counting have been extensively studied since the late 1990s [AYZ97], with the related problem of determining whether any triangle exists in a graph first solved in 1977 [IR77]. In the following years, in-memory algorithms became focal (Section 3.1), but, recently, the size of the graphs in question have dictated alternatives which attempt to make the problem more tractable, from external memory algorithms (Section 3.2) and general-purpose systems which hide I/O considerations (Section 3.5), to distributed systems (Section 3.3) and approximation algorithms (Section 3.4).

3.1 In-Memory Algorithms

One way of representing a graph G of $n = |V(G)|$ nodes is through its $n \times n$ *adjacency matrix* $A(G)$, where entry (i, j) is equal to the number of edges from i to j.[1] It is easy to prove that entry (i, j) for the matrix $A^k(G)$ where $k \in \mathbb{N}$ represents the number of paths of length k starting at i and finishing at j. As a result, the (i, i) entry in $A^3(G)$ represents twice the number of

[1] Clearly, for undirected graphs this matrix is symmetric, and for simple graphs the lack of parallel edges forces the matrix to be binary, and the lack of self-loops means that the diagonal consists of 0s only.

triangles containing i as a node, hence the sum of the diagonal entries — the *trace* $tr\left(A^3\left(G\right)\right)$ — is equal to six times the number of triangles in the graph, as each triangle has been counted twice (due to the two directions) for each of its three nodes.

Consequently, triangle counting, when there is sufficient space to represent the adjacency matrix, comes down to matrix multiplication, which takes time $\Theta\left(n^{\omega}\right)$, where ω is the *matrix multiplication exponent*. A naive version, thus has $\omega = 3$, whereas the best known value for ω is currently < 2.373 [Wil11]. Early work by Alon et al. [AYZ97], as well as recent work by Björklund et al. [BPVWZ14], utilizes matrix multiplication as a subroutine, and distinguishes between dense and sparse graphs to improve performance, by considering runtime in terms of the number of edges m instead of the number of vertices n, when doing so is beneficial. Note that the 1977 work by Itai and Rodeh [IR77], as well as the current state of the art work by Björklund et al. [BPVWZ14] are output sensitive and make no additional assumptions for the graph, meaning that they are strictly better than other approaches when there are few triangles to be reported.

Algorithm 1 Node-Iterator

```
Input: G = (V, E)
Output: All triangles in G
for u ∈ V do
    for v ∈ N(u) do
        for w ∈ N(u) do
            if (v, w) ∈ E and u < v < w then
                Output (u, v, w)
```

Thomas Schank (with Dorothea Wagner) conducted an extensive experimental study in 2005 on triangle listing algorithms [SW05] and extended his work for his PhD thesis [Sch07]. He identified two classes of algorithms, those based on the "*node-iterator*" algorithm, which takes $\sum_{v \in V} \binom{d(v)}{2}$ time (Algorithm 1), and those based on the "*edge-iterator*" algorithm (Algorithm 2), of

cost $\sum_{v \in V} d(v)^2$. Thus, the two algorithms have the same asymptotic complexity, even though *edge-iterator* is faster in practice [SW05].

Algorithm 2 Edge-Iterator

```
Input: G = (V, E), where N(v) is sorted for all v ∈ V
Output: All triangles in G
for u ∈ V do
    for v ∈ N(u) do
        for w ∈ N(u) ∩ N(v) do
            if u < v < w then
                Output (u, v, w)
```

The reason why *edge-iterator* requires sorted adjacency lists is so that the set intersection can be performed quickly, by iterating through both at the same time, in $d(u) + d(v)$ steps. An alternative is by using hashing, though this was shown to be slower in practice [SW05]. Even though Schank lists some improvements to both *node-iterator* and *edge-iterator*, they are only faster when the graphs have *skewed degree distributions*, meaning that d_{max} has a high deviation away from the average degree. In this case, Latapy's [Lat08] improvements on Schank's [SW05, Sch07] *forward* algorithm runs in $\Theta\left(m^{\frac{3}{2}}\right)$ time and $\Theta(m)$ space in the worst case, and has improved performance for sparse, *power-law graphs*, where the proportion p_k of nodes of degree k behaves like $p_k \sim k^{-\alpha}$, where α is a constant, typically between 2 and 3. More recently, Ortmann and Brandes have revisited in-memory triangle listing algorithms and proved that the running time for nearly all of them is $\mathcal{O}(\alpha|E|)$ where α is the arboricity of the graph (Definition 4.2.1) [OB14].

3.2 I/O-Efficient Algorithms

I/O-efficient (or "external-memory") algorithms are not new in the graph community (e.g. [CGG+95, Zeh02, KM03]), but the specific problem of triangle listing was first looked at by Roman Dementiev in 2006 [Dem06].

Dementiev used a pipelined implementation of the *node-iterator* algorithm, called the *External Memory Node Iterator* (or EM-NI) to achieve an I/O complexity of $\mathcal{O}\left(|E|^{\frac{1}{2}} \cdot sort\left(|E|\right)\right) = \Theta\left(\frac{|E|^{1.5}}{B} \log_{M/B} \frac{|E|}{B}\right)$, which has a high dominant term of $\frac{|E|^{1.5}}{B}$ insensitive to the memory size M.[2] Four years later, Bruno Menegola [Men10] used Latapy's compact forward algorithm [Lat08] to create the *External Memory Compact-Forward* algorithm (or EM-CF), which has an I/O complexity of $\mathcal{O}\left(|E| + \frac{|E|^{1.5}}{B}\right)$. His algorithm is thus completely insensitive to M, and furthermore requires at least $|E|$ I/Os, which is prohibitively expensive.

Neither of the above two algorithms are sensitive to the total number of triangles T in the graph, but Chu and Cheng [CC11, CC12], using the idea of *graph partitioning* achieved an I/O complexity of $\mathcal{O}\left(\frac{|E|^2}{MB} + \frac{T}{B}\right)$. This is a strictly better asymptotic time, because $\frac{|E|}{M} < \sqrt{|E|}$ for reasonable values of M.[3] The idea is for a given graph $G = (V, E)$ to partition the vertex set V into $p = |E|/M$ disjoint sets $V_1, \ldots, V_p$. One can then distinguish three types of triangles based on the number of partition sets in which their endpoints belong. More concretely, a triangle is of *Type* i ($1 \leq i \leq 3$) if its endpoints are in exactly i of the V_j sets. The set V_i *induces* a subgraph $H_i = (V_i, E_i)$ where $E_i = \{(u, v) \in E : u, v \in V_i\}$, but graph partitioning relies on *extended subgraphs* (Definition 3.2.1):

Definition 3.2.1 (Extended Subgraph)**.** *Given a graph $G = (V, E)$ and $V_H \subseteq V$, we define the extended subgraph to be the directed $H^+ = (V_{H^+}, E_{H^+})$, where $V_{H^+} = V_H \cup \{v : u \in V_H, v \in V, (u, v) \in E\}$, and $E_{H^+} = \{(u, v) : (u, v) \in E, u \in V_H\}$.*

An extended subgraph H_i^+ builds upon the induced subgraph H_i, but also adds all the neighbors of vertices in V_i that are not already contained within V_i. As a result, by listing triangles within the extended subgraphs (Algorithm 3), one can find all Type 1 and Type 2 triangles in G, and, furthermore,

[2] Hu et al. later proved that the I/O complexity is actually $\mathcal{O}\left(\alpha sort(|E|)\right)$, where α is the arboricity of the graph (Definition 4.2.1) [HTC13].

[3] Hu et al. note that even if M can hold 1% of the edges, the inequality still holds when $|E| > 10,000$ [HTC13].

ensure that no edge fully within H_i is present in a Type 3 triangle. One can then remove these edges, and repeat the procedure, until the graph becomes empty. Though the partitioning can be sequential, using a partitioning based on a dominating (vertex) set of G (DGP) guarantees that the number of intra-partition edges that are removed at each stage is at least $|E|/p$ [CC11]. The authors have also proposed a randomized partitioning algorithm (RGP) with the same bound in expectation with high probability [CC12], making the I/O complexity for both approaches $\mathcal{O}\left(\frac{|E|^2}{MB} + \frac{T}{B}\right)$. Nonetheless, the graph partitioning algorithm assumes that each extended subgraph can fit in memory, and for the DGP implementation specifically, it is assumed that the vertex set can fit in memory. It was proven, however, that provided that $M \geq 24 d_{max} \ln |E|$, each of the extended partitions of RGP fits in memory with probability at least $1 - 1/|E|$ [HTC13], giving RGP better theoretical guarantees than DGP for smaller memory sizes.

Algorithm 3 Extended Subgraph Triangle Listing

Input: An extended subgraph H^+
Output: All Type 1 and 2 triangles in H^+
for $u \in V_H$ **do**
  for $v \in N_H(u)$ with $u < v$ **do**
    for $w \in N_{H^+}(u) \cap N_{H^+}(v)$ **do**
      if $v < w$ or $w \notin V_H$ **then**
        Output (u, v, w)

Even so, these assumptions might not hold, and in practice, the Massive Graph Triangulation (MGT) algorithm by Hu et al. [HTC13] has proven to be superior. MGT has the same I/O complexity as RGP and DGP, but does not make any additional assumptions on the graph. Furthermore, it has a CPU complexity of $\mathcal{O}\left(|E| \log |E| + |E|^2/M + \alpha|E|\right)$ where α is the arboricity of the graph (Definition 4.2.1), and both complexities are proven to be optimal up to constant factors in the worst case.[4] The key idea is to orient the graph based on the degree ordering of Definition 2.1.5, load the

[4]See the analysis in Section 4.2.4 for a discussion on this optimality.

next set of edges in memory, and report all triangles which have pivot edges (Definition 2.1.4) within the loaded set by scanning the entire graph. We will revisit this state-of-the-art external memory algorithm in Chapter 4, as it forms the building block for our Distributed Massive Graph Triangulation (DMGT) framework, which successfully addresses this sequential bottleneck present in MGT.

Finally, it is worth noting that Pagh and Silvestri recently proposed a new algorithm for triangle counting (but not listing) which has an I/O complexity of $\mathcal{O}\left(\frac{E^{1.5}}{\sqrt{M}B}\right)$, which improves the given bounds by a factor of $\min\left(\sqrt{E/M}, \sqrt{M}\right)$. Their algorithm was proven to be optimal in the sense that any algorithm enumerating a graph with T triangles must use time $\Omega\left(\frac{T}{\sqrt{M}B}\right)$, and there are graphs with $T = \Omega\left(E^{1.5}\right)$. Their randomized algorithm is cache-oblivious, and uses a coloring technique in combination with MGT, though the results are only theoretical, and do not have an experimental implementation. The algorithm can be made deterministic (though cache aware), under some additional assumptions on the memory size M.

3.3 Parallel Algorithms

With the introduction of MapReduce by Google in 2004 [DG04], and its open-source implementation Hadoop,[5] the environment for cloud computing changed radically. Developers need only specify two functions, `Map`, which produces intermediate $(key, value)$ pairs and `Reduce`, which combines these intermediate values based on their keys. Distribution of the input is transparent to the user, who only needs to supply the partitioning function for the `Reduce` instances if needed. Using MapReduce for graph algorithms, however, is not straightforward, as many metrics involve local properties of the graph, which are hard to evaluate in a distributed fashion. However, Cohen first proposed a MapReduce implementation for triangle counting in 2009 [Coh09]. His algorithm is based on the *node-iterator* algorithm, uses

[5] `https://hadoop.apache.org/`

a two-pass approach, and assumes that each pair of edges is accompanied by the degrees of its two endpoints. The first pass outputs all *open triangles* (u, v, w) such that (u, v) and (v, w) are edges in E,[6] and the second pass combines the open triangles with the existing edges to find all complete triangles. The algorithm has a CPU runtime complexity of $\mathcal{O}(|E|^{1.5})$, and it assumes that the `Reduce` instances have sufficiently large memory to accommodate high-degree nodes.

A couple of years later, Suri and Vassilvitskii [SV11] introduced the idea of partitioning the graph into ρ distinct vertex sets V_i, defining $V_{ijk} = V_i \cup V_j \cup V_k$ for pairwise distinct i, j, k, and looking at the induced subgraph on V_{ijk}. The `Map` phase associates an edge with all the possible induced subgraphs to which it belongs, while `Reduce` counts the triangles in memory, scaling the output to avoid double counting. By choosing the parameter ρ appropriately, the subgraphs can fit in memory, avoiding the high-degree node problem.

Nonetheless, this approach also generates too much intermediate (duplicated) network traffic, which Park and Chung addressed in a recent paper [PC13]. Their *Triangle Type Partition* (TTP) algorithm — the current state-of-the-art MapReduce algorithm for triangle listing — corrected for this problem by accounting for the type of triangle before emitting it in the `Map` function, thus reducing the duplicate data and the need for scaling during the `Reduce` computation. TTP also has a $\mathcal{O}(|E|^{1.5})$ runtime complexity, and is still too slow to be practical, as we also examine in Section 5.6.

PATRIC, created by Arifuzzaman et al., however, completed ignored the MapReduce paradigm, and used an MPI-based (Message Passing Interface) approach instead, making it 40 times faster than TTP [AKM13]. PATRIC also relies on graph partitioning, but instead of only listing Type 1 and 2 triangles in the extended subgraph, it also lists Type 3 triangles, by including the neighbors' neighbors' in the partition. As such, the algorithm is not I/O-efficient when constructing the partitions,[7] which it also assumes can completely fit in the memory of each machine. Despites these drawbacks,

[6] I.e. all paths of length 2.

[7] But neither are the MapReduce variants discussed above.

the key contributions of PATRIC lie in the load balancing mechanisms it proposes, which are calculated in parallel and do not pose a bottleneck. Even so, PATRIC appears to be targeting high-end data centers, which have hundreds of processors and multiple GBs of dedicated RAM per processor, as opposed to also utilizing idle computers with lower specifications. These issues are avoided in our DMGT implementation, which, importantly, maintains I/O-efficiency, while exhibiting similar (and superior) behavior to PATRIC (Chapter 5).

3.4 Approximation Algorithms

Although our exposition has so far focused on exact triangle listing, there exists a long history on *approximate* triangle counting, particularly in the streaming and semi-streaming context, starting with Bar-Yossef et al.'s seminal paper [BYKS02]. In the streaming model, one attempts to use sublinear space (in $n = |V|$), and a constant number of passes over the input [HRR99], whereas in the semi-streaming model, one can use $\mathcal{O}(n \cdot \operatorname{polylog} n)$ space, since graph problems often become intractable otherwise [FKM+05].[8] Unfortunately, Braverman et al. recently showed that for one-pass approximations, $\Omega(m)$ memory is needed (where $m = |E|$), and when more passes are allowed, $\Omega(m/T)$ memory is needed instead, where T is the number of triangles in the graph [BOV13].

Early work focused on *edge sampling*, where an edge and a node are randomly sampled, and then it is checked whether they form a triangle [BFL+06]. More recently, *wedge sampling* has been proposed, where open triangles of the form (u, v, w) are chosen with (u, v) and (v, w) in E, and it is checked whether (u, w) is also part of the edge set [SPK13]. This method of sampling has been applied both in the streaming environment by using the birthday paradox [JSP12], and in the parallel computing environment by using the MapReduce framework [KPP+13]. Other work has instead focused more on providing a

[8] Semi-streaming algorithms often use $\mathcal{O}(\log n)$ pases instead of a constant number of them.

parallel framework to deal with the size of the graphs in question [TKMF09, TPT13, AKM13], and algorithms often utilize *sparsification* to account for the quadratic space needed for dense graphs as identified above [TKMF09, AKM13]. Other methods include finding eigenvalues [Tso08], estimating intersection of sets [BBCG08], vertex partitioning [KMPT10], node coloring [PT12], and finding small vertex covers [GSK14].[9] Finally, it is worth noting that, recently, Kutzkov and Pagh proposed the first method for triangle counting in dynamic graph streams which has proven theoretical guarantees — even in the case of edge deletion — although their algorithm has not been implemented and verified experimentally [KP14].

Overall, however, the recent theorem by Braverman et al. [BOV13] indicates that approximate counting is not necessarily much more efficient than exact listing,[10] and, as a matter of fact, the state-of-the-art algorithms mentioned above often have errors of up to 4% on larger graphs (e.g. [TPT13, JSP12]), which can be impractical for some applications.[11] Consequently, although approximate algorithms may be accurate or fast enough for some purposes, we believe that better exact counting and listing algorithms are needed, further motivating the creation of our DMGT framework.

3.5 General-Purpose Systems

This chapter would be incomplete without mentioning general-purpose systems where the above algorithms can be implemented without having to make extensive I/O considerations.

[9] The interested reader may wish to consult the survey by McGregor [McG13] for more details on streaming algorithms for the triangle counting and other related problems.

[10] As can be verified by looking at their runtimes. However, Berry et al. recently proved that under some assumptions, approximate triangle counting can have expected linear-time performance for a certain class of graphs [BFN+14].

[11] Note that even a 1% error on the Twitter graph [KLPM10] results in tens of millions of mis-reported triangles.

3.5.1 Local Frameworks

Kyrola et al.'s GraphChi [KBG12] was arguably the first general-purpose framework to support computation on massive graphs on a commodity PC. The goal of the project was to show that its performance is *reasonable* for many applications, and the key idea was to split a large graph into "shards" which can fit in memory, and which contain all in-edges associated with an interval, sorted by source. As such, GraphChi has a heavy pre-processing step, but each iteration required at most $\Theta(P^2)$ non-sequential disk accesses, where $P < 1,000$ is the number of shards. However, its vertex-centric model, where vertex values are only propagated through adjacent edges, meant that the triangle counting algorithm "is quite complicated and requires 'trickery' to work", including deletion of edges.[12] Even so, the algorithm runs twice as fast as the state-of-the-art MapReduce TTP algorithm [PC13] (including the pre-processing step), indicating that performance is indeed reasonable (though by no means as fast as our algorithm, even in the single-core setting), and resorting to distributed systems is not always necessary [KBG12].

TurboGraph [HLP+13] is another single-PC system which claims to exploit multi-core and FlashSSD I/O parallelism, and to reduce bottlenecks by viewing graph algorithms as matrix multiplication problems. TurboGraph could thus be used for an implementation of the various matrix-based triangle counting algorithms, but its implementation is not available, so it cannot be explored further. X-Stream [RMZ13], on the other hand, uses an *edge-centric* model on unordered edges, and as a result does not have a pre-processing step bottleneck. X-Stream has good theoretical I/O bounds, and outperforms GraphChi by up to 6×, even when excluding GraphChi's pre-processing time. However, X-Stream's streaming, unordered model makes it unsuited for implementing an exact triangle counting algorithm, hence no comparison with GraphChi for this problem is available. That said, the framework does contain an implementation of Becchetti et al.'s semi-streaming algo-

[12] Source code comment at `https://github.com/GraphChi/graphchi-cpp/blob/master/example_apps/trianglecounting.cpp`

rithm [BBCG08],[13] but as discussed in Section 3.4, non-exact algorithms are not sufficient for our purposes.

3.5.2 Distributed Frameworks

PowerGraph [GLG+12] was introduced as a distributed abstraction to "exploit the structure of vertex-problems" in order to provide high parallelism, and reduced network communication and storage costs. PowerGraph programs implement three (mostly stateless) functions: `gather` (which combines incoming information), `scatter` (which transmits outgoing information), and `apply` (which computes and updates the values). PowerGraph can either be used synchronously or asynchronously, and the distribution of the graph happens transparently to the user. Of course, PowerGraph does not provide theoretical I/O guarantees, and does not work well in low memory scenarios, targeting the data-centric environment. Despite these limitations, PowerGraph's *edge-iterator* implementation (Algorithm 2) on over a thousand cores is currently the fastest known triangle counting implementation, especially after additional optimizations [Low13], beating even PATRIC [AKM13] by 37×, but using 5× more computational power. However, the PowerGraph measurements are not entirely transparent, as they do not include its set-up and distribution time, making it slower than was actually reported. Finally, it is worth mentioning that GraphX, built on top of Apache Spark, [XGFS13] uses a more database-oriented approach with clever data-structures to reduce communication between processing nodes. However, GraphX is up to 7× slower than PowerGraph, and aims to tackle productivity gains rather than absolute performance. Additionally, its triangle counting algorithm requires partitioning, and is not very reliable on larger datasets or more than one machines according to early reports on its mailing lists.[14]

[13] X-Stream's code is available at `http://labos.epfl.ch/x-stream`

[14] Details can be found at `http://apache-spark-user-list.1001560.n3.nabble.com/RDD-is-Lost-in-GraphX-When-doing-Triangle-Counting-td6006.html` and `http://apache-spark-user-list.1001560.n3.nabble.com/TriangleCount-amp-Shortest-Path-under-Spark-td2439.html`

3.5.3 Databases

It is worth briefly discussing databases, which can also be used for triangle counting. On the one end, graph databases [RWE13] have started to emerge as a way to persistently store graphs and perform computations on them efficiently. Graph databases are much more efficient in finding friends-of-friends at longer depths than traditional relational databases [RWE13], but triangle counting only requires a depth of 1. With that said, experiments have shown that smarter `joins` on databases actually outperform MapReduce implementations [Wal, Suv]. However, such a comparison is not entirely meaningful, given that MapReduce is ill-suited for this particular problem, as discussed in Section 3.3. Additionally, databases have a much higher set-up cost, have no theoretical I/O guarantees, and can use extensive memory, but if the graph data is to be re-used, for instance through dynamic modifications, distributed databases can be a worthy alternative to dedicated triangle counting algorithms.

Chapter 4

Design and Implementation

The goal for this project was to create a triangle listing algorithm which is I/O-, CPU-, Network-, and Memory-efficient. Our distributed algorithm should be able to run efficiently in a high-end data center environment, or be able to make use of low-end commodity machines during their idle time. To accommodate for both of these scenarios, the framework is assumed to contain R machines, each of which has P processors, with M bytes of memory for each of the processors.[1] By choosing these parameters appropriately, we can model a high-end data center, with multiple processors per machine, or even just a single computer with low available memory.

After first discussing in Section 4.1 why other communication techniques and external-memory algorithms were not chosen for our framework, we carefully discuss and analyze in Section 4.2 why modifying the MGT algorithm to use ordered array intersection instead of hash structures is beneficial. Finally, in Section 4.3 we explain in detail how our Distributed MGT (DMGT) algorithm works by duplicating the graph across the R machines, and assigning a unique, contiguous subset $S \subseteq E$ of the edges to each processor, which is then responsible for finding all triangles with pivot edges in S.

[1] In this dissertation, the terms "machines" and "nodes" are used interchangeably, as are the terms "processors" and "cores".

4.1 Non-Choices

In order to better motivate some of our decisions, it is worth briefly discussing some of the alternatives that were rejected. A matrix-based algorithm was quickly dismissed, because, first, distributed matrix operation are notoriously hard to get right (e.g. [CDPW92, QCK+12]), and, second, because matrix-based algorithms do not offer the same theoretical guarantees when the graph is too big to fit in memory. Additionally, existing parallel algorithms using MapReduce [Coh09, SV11, PC13] have shown MapReduce to be the inappropriate framework to use, due to the $\mathcal{O}(|E|^2)$ intermediate traffic, which appears to be necessary as neighbors are often sent to different machines. The most successful parallel algorithms [AKM13, Low13] both employ graph partitioning, and rely on basic in-memory algorithms (Section 3.1) to count triangles within a partition. Nonetheless, in the case of PowerGraph [GLG+12, Low13] the focus was to build a general purpose system, while PATRIC's [AKM13] contribution lay in its load-balancing techniques. Consequently, neither of the two systems provided theoretical analysis and efficiency guarantees, and both were ill-suited for lower-end environments, where available memory was limited.

Overall, we preferred an approach of minimal network communication: no data is exchanged beyond the the initial graph transfer and the final triangle list/count by the machines. The idea was that our communication model was simple enough to not warrant use of a Message Passing Interface (MPI) which adds complications in implementation, and additional latency, especially in non-dedicated clusters.[2] For similar reasons, we chose to avoid Remote Direct Memory Access (RDMA) [TMS11] and Distributed Shared Memory (DSM) [NL91] solutions, since not only would it be harder to make concrete performance guarantees, but such approaches would also not perform well in the wide variety of computational environments we target.

[2]Lockwood has made an excellent post on the intricacies of getting high-performance MPI on Amazon's EC2 servers at [Loc].

4.1.1 Graph Partitioning

In terms external-memory algorithms, Chu and Cheng's DGP [CC11] was easily rejected due to the linear (in $n = |V|$) memory requirement. The case against RGP [CC12] was less straightforward: we first prove Lemma 4.1.1, to understand the effect of extended subgraphs (Definition 3.2.1):

Lemma 4.1.1. *Given a graph $G = (V, E)$, a partition of V into I sets $V_1, \ldots, V_I$, and the corresponding extended subgraphs $H_1^+, \ldots, H_I^+$, the total number of edges $\sum_{i=1}^{I} \left| E_{H_i^+} \right|$ is equal to $2|E_G|$.*

Proof. The number of edges in each extended subgraph $H^+ = (V_{H^+}, H_{H^+})$ is equal to $|E_{H^+}| = \sum_{v \in V_H} d_G(v)$, as the graph is directed, so no edges are double counted. Additionally, because each vertex is in a unique set of the partition,

$$\sum_{i=1}^{I} \left| E_{H_i^+} \right| = \sum_{i=1}^{I} \sum_{v \in V_{H_i}} d_G(v) = \sum_{v \in V} d_G(v) = 2|E_G|$$

by the handshaking lemma. □

Because we assume that there are no isolated vertices, the total traffic that needs to be sent across at each iteration of the RGP algorithm is $\Theta(|E|)$. Because partitions must fit in memory (in a single machine), and by using Lemma 5.2 in [CC12], we know that the number of edges at each full iteration decreases by M in expectation. The total network traffic is thus

$$\Theta(|E| + (|E| - M) + (|E| - 2M) + \ldots + M) = \Theta\left(\frac{|E|^2}{M}\right)$$

Essentially, the network becomes yet another layer of (slower) I/O, and there is an added bottleneck of randomizing the partitions at the master. By thinking of each machine as a "supernode" of memory $P \cdot M$, which further delegates the partitioning to each of the individual processors, we conceptually see that R machines of P processors are the same as $R \cdot P$ machines of 1 processors in terms of network traffic. Hence, because MGT is faster in

practice with fewer assumptions, and because RGP is iterative, we chose to forgo the graph partitioning route.

4.2 Massive Graph Triangulation

In this section we present the MGT algorithm in detail, discussing its assumptions and performance.

4.2.1 Input Format

Given a graph $G = (V, E)$, where $V = [n]$, we wish for the edges to be sorted by their (left) endpoints from 0 to $n - 1$, tie-breaking on their other (right) endpoints. For instance, the edges of the graph in Figure 4.1 should be listed as $(0, 2)$, $(0, 3)$, $(1, 2)$, $(2, 0)$, $(2, 1)$, $(2, 3)$, $(3, 0)$, and $(3, 2)$. Though many graphs already come in this format, by a simple external sort, this format can be achieved in $\mathcal{O}(sort(|E|))$ I/Os and $\mathcal{O}(|E| \ln |E|)$ CPU time [AV88].

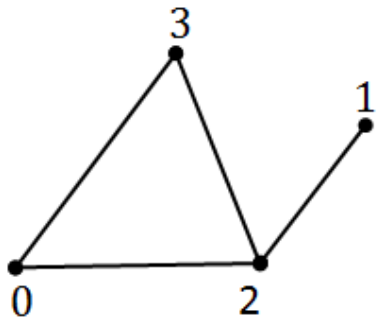


Figure 4.1: A graph with 4 vertices and 4 edges

In addition, because typically $|V| \ll |E|$, we also prefer to store the graph as two separate files, the adjacency list file (with extension `.adj`), and the degree file (with extension `.deg`). Specifically, we adopt the convention used by the implementation of MGT:[3] the files are stored in binary (non-textual)

[3]The binary (without source code) can be found at [HTCb] and its manual at [HTCa].

little-endian format to make processing faster, and each node is represented by a 32-bit integer. The degree file stores the degrees of the vertices as pairs $(i, d_G(i))$ and occupies $2n$ integers, i.e. $8n$ bytes. The adjacency list contains the neighbors of the vertices in order, and occupies $2m$ integers (as the graph is bi-directional), i.e. $8m$ bytes. For instance, the degree file for the graph of Figure 4.1 would contain (in binary format and without spaces, commas, or parentheses) the values $(0, 2), (1, 1), (2, 3), (3, 2)$, while its adjacency file would contain $(2, 3), (2), (0, 1, 3), (0, 2)$. It is clear that creation of these files only takes $\mathcal{O}(scan(|E|))$ I/Os and a linear CPU processing time.

Crucially (Theorem 4.2.3), for efficiency purposes, the algorithm orients the graph $G = (V, E)$ to obtain G^* (Definition 2.1.3) under the degree-based ordering (Definition 2.1.5). The analysis for the orientation presented in [HTC13], however, is flawed, as the authors do not account for the possibility that the degree array does not fit in memory. Specifically, given a graph in this two-file format, if the degree array fits in memory, it can be simply read completely in memory in $\mathcal{O}(scan(|V|))$ I/Os, and scan and output the oriented files in $\mathcal{O}(scan(|E|))$ I/Os and $\mathcal{O}(|E|)$ CPU time.

However, if the degree array does not entirely fit in memory, in the worst case (e.g. for the complete graph K_n), a vertex has a neighbor in every block. As a result, for each node, there must be $\mathcal{O}(|V|/B)$ I/Os, for a total of $\mathcal{O}(|V|^2/B)$ I/Os, just for the degree file. Because $|E| = \mathcal{O}(|V|^2)$, the total complexity is $\mathcal{O}(scan(|V|^2))$ I/Os and $\mathcal{O}(|E|)$ CPU time. This does not make a difference in dense graphs (except for the asymptotic constant), but it is still a point of omission for the analysis presented in [HTC13].

4.2.2 The Algorithm

In this section we summarize the CPU-efficient version of the baseline MGT algorithm [HTC13] (Algorithm 4). The algorithm keeps reading cM edges into memory, denoted by E_{mem}, where $c < 1$ is an implementation-specific constant. The algorithm then builds hash structures on V_{mem} (the endpoints of the edges in E_{mem}) and V^+_{mem} (those $v \in V_{mem}$ that have outgoing edges

in E_{mem}). For each vertex $u \in V$, the algorithm reads $N(u)$, which in this context denotes the out-neighbors of u which are also in V_{mem} and constructs $N^+(u) = N(u) \cap V^+_{mem}$, so that the algorithm can report all triangles (u, v, w) that have cone vertex u (for all $u \in V$) and a pivot edge (v, w) loaded in memory (Definition 2.1.4). It is clear that each triangle is reported exactly once this way, and that there are $\Theta(|E|/M)$ iterations. Moreover, the entire graph is read once per iteration,[4] so the total amount of I/Os is $\mathcal{O}\left(\frac{|E|^2}{MB}\right)$ plus the I/Os for listing the T triangles, for a total of $\mathcal{O}\left(\frac{|E|^2}{MB} + \frac{T}{B}\right)$. The proof for the CPU efficiency requires the *small-degree assumption*: that every vertex $v \in V$ has $d_{G^*}(v) \leq cM/2$.[5] The algorithm also poses an "all-or-nothing" requirement, where all of the edges incident to a vertex must be present, but we defer analysis to Section 4.2.4, where we show that this is not necessary.

Algorithm 4 MGT

```
Input: The oriented G* = (V*, E*)
Output: All triangles in G
while there are edges in E* to be read do
    Read the next cM edges into memory and call them E_mem
    Obtain and build hash structures on V_mem and V+_mem
    for u ∈ V do
        Read the neighbors of u from disk to obtain N(u)
        Construct N+(u) from N(u) and V+_mem
        for v ∈ N+(u) do
            for w ∈ N(u) do
                if (v, w) ∈ E_mem then
                    Output (u, v, w)
        Release N(u) and N+(u) from memory
```

The authors also discuss how to remove the small-degree assumption, by reporting all triangles involving high-degree vertices, and then removing these

[4] With an additional cM edges per iteration which can be ignored asymptotically.

[5] Recall that $d_{G^*}(v)$ is the (out-)degree of v in the oriented graph G^* under the degree-based ordering $\prec$.

edges from the graph (Algorithm 5). The idea is to find a high-degree vertex u, and load a subset S of its edges that can fit in memory. Let $T = \{v : (u, v) \in S\}$, and $T(v) = N(v) \cap T$. There are two types of triangles to be reported involving S: those that contain two edges in S, (u, v) and (u, w), and those involving only one, (u, w). By iterating over $v \in V$, with $v \neq u$ we can report both types of triangles efficiently. Specifically, if $v \in T$ (i.e. $(u, v) \in S$), then any $w \in T(v)$ completes a triangle of the first type, as $w \in T$, so $(u, w) \in S$, and $w \in N(v)$, so $(v, w) \in E$. If not, and if $u \in N(v)$, then any $w \in T(v)$ completes a triangle of the second type, because $(v, u) \in E$ and $(v, w) \in E$ by definition, and $(u, w) \in S$, since $w \in T$. It is clear to see that we don't miss or double count triangles, and that there are at most $\mathcal{O}(|E|/M)$ iterations, since $\Theta(M)$ edges are removed at each iteration. Additionally, the algorithm performs $\mathcal{O}(scan|E|)$ I/Os per iteration and uses $\mathcal{O}(|E|)$ time, which makes the pre-processing step consistent with the overall runtime, with a total of $\mathcal{O}\left(\frac{|E|^2}{MB}\right)$ I/Os and $\mathcal{O}\left(\frac{|E|^2}{M}\right)$ CPU computations.

Algorithm 5 Removing High-Degree Vertices

Input: The undirected graph $G = (V, E)$
Output: G' without high-degree vertices, with relevant triangles reported
while there is a high-degree vertex **do**
    Find a high-degree vertex u with $d(u) > cM/2$
    Load $cM/2$ edges of u in memory, and create a hash structure T
    for $v \in V \setminus \{u\}$ **do**
        Let $T(v) = N(v) \cap T$
        if $v \in T$ **then** ▷ Report triangles with two edges in memory
            for $w \in T(v)$ **do**
                if $v < w$ **then** ▷ Only report one of the two directions
                    Report (u, v, w)
        else if $u \in N(v)$ **then** ▷ Report triangles with one edge in memory
            for $w \in T(v)$ **do**
                Report (u, v, w)
    Remove the loaded edges from E
return $G' = G$

4.2.3 Implementation

Although the source code for MGT is not available,[6] during our experimentation we hypothesized that the implementation of MGT does not use either hash- or tree-sets, but arrays. Though we did not reverse engineer the binary to verify our claims, we believe them to be true, because if the adjacency list for any given vertex is not sorted, the given implementation misses triangles, though the manual [HTCa] does not make mention of such requirements. Clearly, if any types of sets were constructed, this need for a sorted adjacency list would not be present.

This belief was further verified by our own implementation, where using sets and maps of any kind, from C++'s `std::unordered_set` to Google's `google::dense_hash_set`,[7] made our implementation more than 10× slower. Consequently, our implementation deviates from the proposed high-level Algorithm 4 in the following ways:

- We do not make the "all-or-nothing" demand
- Since the maximum oriented degree d_{max} is known, we create 2 static arrays to hold $N(u)$ and $N^+(u)$ for any u, without the need to release, resize, or clear them
- We do not calculate or store V_{mem}. As a result, $N(u)$ denotes all the out-neighbors of u as opposed to those which are also present in V_{mem}
- We store the out-neighbors of E_{mem} in `edges`: a large consecutive array
- Because V^+_{mem} consists of consecutive vertices, we allocate `indices`: a large two-dimensional array, storing vertices between v_{low} and v_{high}, where $v_{high} - v_{low} \in \Theta(M)$. The array makes the following guarantees:
  - If $v < v_{low}$ or $v > v_{high}$, then $v \notin V^+_{mem}$
  - Otherwise, `indices[`$v - v_{low}$`]` stores the number of edges read for

[6] Only an executable is present at [HTCb].

[7] Described at `http://en.cppreference.com/w/cpp/container/unordered_set` and `https://code.google.com/p/sparsehash/` respectively.

v, and the index into `edges` for these edges[8]

– At the beginning of each iteration, it is cleared using `std::fill`

- The triangles around $v \in N^+(u)$ are found using set intersection between $N(u)$ and `edges[indices[`$v - v_{low}$`]]`
- To calculate the size of `edges` and `indices`, we remove the buffer sizes from the available memory, and allocate 2/3 of the remaining memory to `edges` and 1/3 to `indices`

4.2.4 Analysis

Because of the changes in implementation (Section 4.2.3), the analysis of the algorithm is also different, but is based on the analysis already given in [HTC13]. First of all, note that because of the requirement that there are no isolated vertices, the I/O complexity remains the same at $\mathcal{O}\left(\frac{|E|^2}{MB} + \frac{T}{B}\right)$, as `edges` still holds $\Theta(M)$ edges and `indices` $\Theta(M)$ (consecutive) vertices.[9]

The *arboricity* of a graph (Definition 4.2.1) is a key concept for the analysis of the runtime:

Definition 4.2.1 (Arboricity, Density [CN85])**.** *The arboricity $\alpha(G)$ of a graph G is the minimum number of edge-disjoint forests needed to cover the edges of G. It satisfies*

$$\alpha(G) = \max_{G' \subseteq G} density(G')$$

where G' ranges over all subgraphs of G with ≥ 2 vertices, and

$$density(G') = \left\lceil \frac{|E_{G'}|}{|V_{G'}| - 1} \right\rceil$$

[8] The array is two dimensional instead of two separate arrays to increase caching effects, which in practice sped up the code by 10%. The caching effects were even more pronounced when we tried parallelizing within a single iteration for fixed $v \in V$ using `OpenMP`: instead of achieving a speed-up, the code became 2.5× slower!

[9] Note that after the orientation, some vertices might only have in-edges, but it is easy to see that they don't change this bound.

Theorem 4.2.2 (Arboricity bounds [CN85])**.** *The arboricity of a graph* $G = (V, E)$ *satisfies:*

1. $\alpha \leq \left\lceil \sqrt{|E|} \right\rceil$
2. $\alpha = \mathcal{O}(1)$ *if* G *is planar*
3. $\sum_{(u,v)\in E} \min\{d(u), d(v)\} \leq \mathcal{O}(\alpha|E|)$

Note that the number of triangles T satisfies $T \leq \frac{1}{3} \sum_{(u,v)\in E} \min\{d(u), d(v)\}$, as any edge can appear in at most $\min\{d(u), d(v)\}$ triangles, so $T = \mathcal{O}(\alpha|E|)$.

Note that because $\Theta(M)$ edges are loaded at each step, there are $h = \Theta(|E|/M)$ iterations. Checking whether $v \in V^+_{mem}$ amounts to checking whether `indices`$[v - v_{low}]$ has a positive degree, which is an $\mathcal{O}(1)$ operation, so construction of E_{mem}, and V^+_{mem} (together with clearing it) takes $\Theta(|E_{mem}|) = \Theta(M)$ time. Construction of $N(u)$ and $N^+(u)$ thus also takes $\Theta(|N(u)|) = \Theta(d_{G^*}(u))$ time. Note that since each edge is examined once in a single iteration, each iteration incurs time $\mathcal{O}(|E|)$ for construction of these structures, for a total of $\Theta(|E|^2/M)$ time.

Additionally, set intersection of two ordered sets of size m, n takes time $\mathcal{O}(m + n)$ using a naive set intersection,[10] thus the total complexity for the triangle operations is

$$\sum_{i=1}^{h} \sum_{u\in V} \sum_{v\in N_i^+(u)} d_{G^*}(u) + d_{G^*}(v)$$

where $N_i^+(u)$ denotes $N^+(u)$ in the i-th iteration. First, note that any v is in at most 2 (consecutive) $N_i^+(u)$ for any given u. This is due to the small degree assumption, because if the adjacency is split the first time, the second time it will entirely fit in memory. Thus, we can change the order of summation as follows:

[10] We have also implemented a faster $\mathcal{O}(m \ln n)$ set intersection based on a paper by Baeza-Yates [BY04], but this algorithm is only used when m is small.

$$\begin{aligned} & \sum_{i=1}^{h} \sum_{u \in V} \sum_{v \in N_i^+(u)} \\ = & \sum_{u \in V} \sum_{i=1}^{h} \sum_{v \in N_i^+(u)} \\ \leq & \; 2 \sum_{u \in V} \sum_{v \in N^+(u)} \end{aligned}$$

We look at each term separately (and ignore the factor of 2 for clarity):

$$\sum_{u \in V} \sum_{v \in N^+(u)} d_{G^*}(u) = \sum_{u \in V} d_{G^*}^2(u)$$

Additionally,

$$\begin{aligned} & \sum_{u \in V} \sum_{v \in N^+(u)} d_{G^*}(v) \\ = & \sum_{v \in V} d_{G^*}(v) \left(d_G(v) - d_{G^*}(v) \right) \\ = & \sum_{v \in V} d_G(v) \cdot d_{G^*}(v) - \sum_{v \in V} d_{G^*}^2(v) \end{aligned}$$

because $d_G(v) - d_{G^*}(v)$ represents the number of incoming vertices to v. The sums of $d_{G^*}^2(v)$ cancel out, so we need to calculate $\sum_{v \in V} d_G(v) \cdot d_{G^*}(v)$. This is where the arboricity becomes useful (Theorem 4.2.3 is adapted from — but is not identical to — the one given in [HTC13]):

Theorem 4.2.3 (Ordering)**.** $\sum_{v \in V} d_G(v) \cdot d_{G^*}(v) = \mathcal{O}\left(\alpha |E|\right)$

Proof.

$$\begin{aligned} \sum_{v \in V} d_G(v) \cdot d_{G^*}(v) & = \sum_{v \in V} \sum_{u \in N^+(v)} d_G(v) \\ & = \sum_{(v,u) \in E^*} d_G(v) \\ \text{(by orientation)} & \leq \sum_{(v,u) \in E} \min\{d(v), d(u)\} \\ \text{(by Theorem 4.2.2)} & = \mathcal{O}\left(\alpha |E|\right) \end{aligned}$$

□

Theorem 4.2.4 (MGT Complexity)**.** *In summary, our implementation of MGT has an I/O complexity of*

$$\mathcal{O}\left(\frac{|E|^2}{MB} + \frac{T}{B}\right)$$

and CPU complexity of

$$\mathcal{O}\left(\frac{|E|^2}{M} + \alpha|E|\right)$$

If the graph is not already sorted, an additional $\mathcal{O}(sort(|E|))$ *I/Os and* $\mathcal{O}(|E| \log |E|)$ *computations are needed, and if* $|V| < M$*,* $\mathcal{O}(scan(|V|^2))$ *I/Os are necessary to orient the file.*

This complexity is identical to that of MGT, and the additional I/Os are also needed for the baseline implementation of MGT as well, but they are conveniently ignored in [HTC13].

Worst-Case Bounds

The authors of [HTC13] also claim that their algorithm is optimal (up to constants) in the worst case. Though this claim is correct within the narrow context of worst-case external-memory triangle listing, it is perhaps somewhat misleading. Specifically, it is clear that in order to actually list T triangles, T/B I/Os are necessary. Additionally, when memory is $\Omega(|E|)$, a complexity of $\Omega(|E|^2/(MB)) = \Omega(|E|/B)$ is necessary, just to read the files. An interesting question arises, however, when $M = \varo(|E|)$.

In this case, the authors suggest considering the complete graph on $|V|$ vertices, with $M \geq |V| = \Theta(|E|)$. Specifically, there are $\Omega(|V|^3)$ triangles, so they claim that the algorithm must incur a complexity of $\Omega(|V|^3/B) = \Omega(|E|^2/(|V|B)) = \Omega(|E|^2/(MB))$. Though, indeed, this complexity is necessary for *listing* the triangles, which is included in the $\mathcal{O}(T/B)$ bound, such complexity is *not* in fact necessary for merely counting them, as a recent theorem by Pagh and Silvestri indicates [PS13]. Specifically, Pagh and Silvestri instead show that for triangle counting, an I/O complexity of

$\Omega\left(T/(\sqrt{M}B)\right)$ is necessary for reporting triangles, and an algorithm with $\mathcal{O}\left(|E|^{3/2}/(\sqrt{M}B)\right)$ I/Os exists.

The argument for CPU efficiency is similar: The $\alpha|E|$ is indeed necessary to find all triangles as explained in Theorem 4.2.2, and in the worst case trumps the remaining terms. However, in-memory matrix algorithms suggest lower bounds, even for triangle listing (e.g. [BPVWZ14]).

Consequently, though no faster (in terms of both runtime and theoretical complexity) external-memory algorithm for triangle listing exists, there are faster algorithms for triangle counting (even in external memory), and the algorithm is thus optimal only in the worst case, where listing the triangles themselves is most expensive.

4.3 Distributed Massive Graph Triangulation

Our distributed protocol is a natural extension of the MGT algorithm: every available processor is allocated a (contiguous) set of edges S, and is responsible for finding all triangles in the graph which contain pivot edges in S. This is significantly different from the existing parallel literature, where different machines are responsible for different subsets of the vertices. Such approaches require careful load-balancing and have already been studied, for instance in PATRIC [AKM13]. Besides the novelty of using an edge-centric approach, there is also a significant algorithmic advantage: when the memory M is sufficient to hold the allocated set of edges, the processor need only perform a single iteration — because of the structures on E_{mem} and V^+_{mem} — whereas a vertex-oriented MGT algorithm would still have to perform $\mathcal{O}(|E|/M)$ iterations, re-creating the structures and eliminating any caching effects. Of course, this approach requires sending the entire graph to each machine, but we discuss in Section 4.3.3 why doing so is a reasonable choice.

Note that, in principle, only the orientation requires having a separate degree file. After that, we could have used alternative techniques of storing the

adjacency lists, such as Compressed Sparse Row, which are often used for efficient traversal [YNDR14, KBG12, PGA10]. However, such methods have worse I/O guarantees, and we additionally chose to maintain the two-file format for compatibility with the existing MGT implementation. Additionally, even though we could compress our files before sending them over the network, compression is too slow for our purposes, it changes the I/O analysis, and, overall, cannot produce a good compression rate for our data.[11]

4.3.1 The Framework

Though it would be impossible to discuss the over 2000 lines of cross-platform C++ code written from scratch for this project, it is worth discussing the high-level architecture of the system, before we delve into the network protocol and analysis.[12] The code can be compiled for both 32- and 64-bit platforms, and across operating-systems (tested on Windows and Linux), though the 64-bit binaries are necessary when the available memory per machine exceeds 2GB. Moreover, the platform-specific code is isolated to the two utility files `util.h/cpp` and `networkutil.h/cpp`.

As mentioned in Section 4.2.1, a vertex, represented by the type `vx`, is a 32-bit integer, but compiling with the option `BIT64` enables use of 64-bit integers, for the case where a graph has more than 4 billion vertices. Because I/O is expensive, we use default buffer sizes of 4MB, equal to multiple blocks, as guided by Section 2.3.1. For writing to the disk, a class called `FileBuffer` is used, whereas `ParserUtil` receives ordered edges (u, v) as input, and outputs the degree and adjacency list files.

For processing the two files, a `DegreeHandler` buffers (some of) the degrees, and automatically updates the buffer when the degree of a vertex out of bounds is requested. These requests are typically sequential, with

[11] As an example, compressing the Twitter adjacency list to the `.zip`, `.gz`, or `.7z` format took over an hour, for a compression rate of only 75%.

[12] The full code can be downloaded from `https://www.dropbox.com/s/ddwwo6bfgnhbbti/DMGT.zip`.

the exception of the graph orientation, as discussed in Section 4.2.1. Handling the adjacency list is trickier, and the class `AdjacencyHandler` is a virtual class with access to a `DegreeHandler`, which implements the method `processAdjacency(unsigned long long low, unsigned long long high)` for processing the adjacency list between offsets `low` (inclusive) and `high` (exclusive). Subclasses need to implement methods for set-up and tear-down, as well as the crucial `handleEdge(vx from, vx to, vx fromDegree)`. This way, subclasses are oblivious to how data is read from disk, and the framework is very extensible: `InMemAdjacencyHandler` implements the in-memory algorithm, `HighDegreeHandler` removes vertices of high degree, and, of course, `MGTAdjacencyHandler` implements MGT.

Though most of the framework works with the adjacency and degree files detailed above, `FileParser` provides utilities for converting a (sorted) textual file to the binary representation. Though for large files sorting, and converting the graph to the undirected format using external utilities is necessary,[13] the framework also provides utilities for doing so in-memory after the graph has been parser to the binary format. Finally, the `orient` method applies the degree-based order of Definition 2.1.5.

4.3.2 The Protocol

In our distributed scenario, there is a single machine, the *master*, dedicated to delegating responsibility to the R *client* machines, and combining the triangle listing results. Because MGT operates on oriented graphs, and because this orientation need only occur once, it is the responsibility of the master to apply the degree-based order (if not already applied) to the graph in question, before sending the graph over the network. Though for our analysis we assume that each client has P processors, with M memory per processor, the protocol is flexible enough, so that each processor may have a different amount of available memory, and the number of processors may vary per

[13] `grep -v "^#" graph.txt | perl -p -e "s/^(\S+)\s+(\S+)/\1\t\2\n\2\t\1/" | sort -u -nk 1,1 -nk 2,2` can be used to convert to a sorted, undirected graph.

machine. In this case, each machine is allocated a number of edges proportional to its share of the total memory. The master creates a thread for each client, and sends the graph, together with the indices that each processor on the machine is responsible for, and the memory allocated for that processor. Each client, on receipt of the graph, creates a thread for each of the processors, where `MGTAdjacencyHandler` is used to process the adjacency list between the specified indices. If triangle listing is requested, the client combines all the reported triangle files from each of the threads, and sends the file back to the master, together with the triangle count for the machine. The master atomically adds the total number of triangles, and also combines the received triangle files for more convenient access.

Our protocol is illustrated in Figure 4.2, where, for clarity, the master process is duplicated, and is shown to be run on a separate machine from the clients. In our illustration, boxes represent different processes and clients, and lines between boxes represent network traffic. Solid lines represent requests, and dotted lines calculated answers. Ovals within boxes represent threads, and summation happens atomically. Finally, $C_{i,j}$ represents the "configuration" for processor j on machine i: the memory allocated for that thread, together with the low and high indices within the oriented edge set for which the processor is responsible.

4.3.3 Analysis

The crucial mistake that existing parallel triangle counting systems like PATRIC [AKM13] and PowerGraph [GLG+12, Low13] make is that they work using graph partitioning. As we already explained in Section 4.1, constructing these partitions is I/O-inefficient. Additionally, the algorithms utilizing graph partitioning assume that the partitions can fit in memory. Though for smaller graphs this may be true, in graphs with large cliques, this is no longer the case. Take, for instance, the extreme case of the complete graph on n vertices, K_n. Any partition of this graph requires loading the entire graph in memory, which requires memory proportional to n^2 on

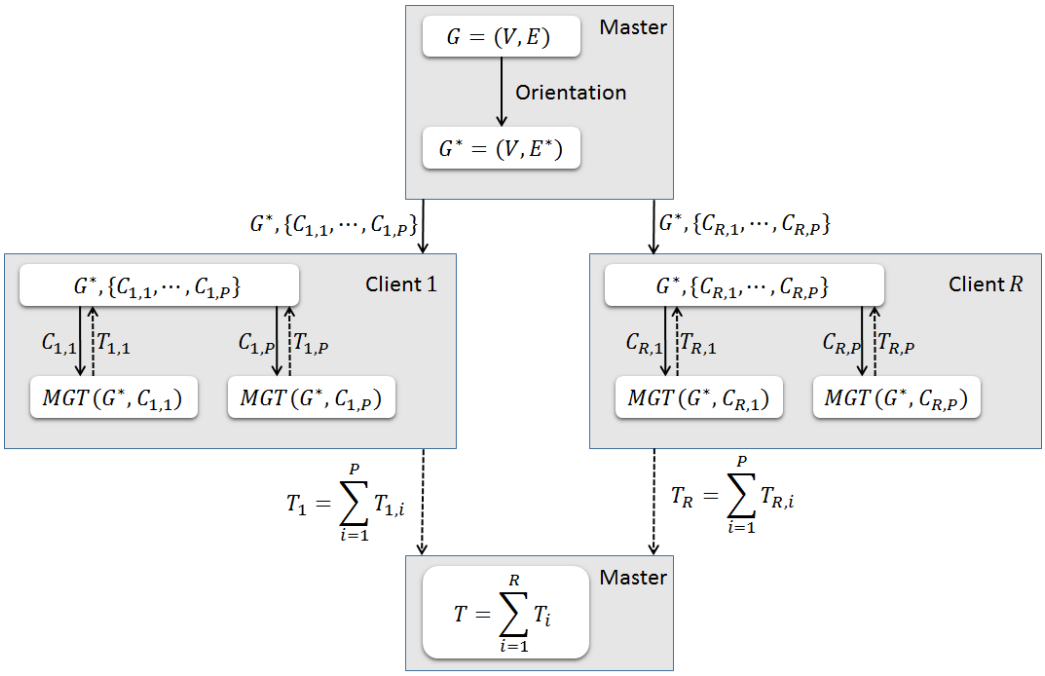


Figure 4.2: DMGT protocol overview

each of the processors. As a result, each of the RP processors in such a graph-partitioning system must receive the entire graph. Recall, however, that our algorithm only requires memory proportional to the maximum degree, which in this case would be n, and the graph is only duplicated once per each of the R machines, making our approach preferable in terms on network traffic in dense graphs. However, graph-partitioning would be preferable in terms of network traffic in highly disconnected networks, such as a graph consisting of RP copies of K_n, but even then, partitioning is either a master-bottleneck, or it requires extensive coordination between processors, and is not guaranteed to produce an optimal distribution.[14] Consequently, DMGT can accommodate a wider range of scenarios, including the high-end data centers towards which PATRIC and PowerGraph are targeted, but also smaller-sized ones with limited memory, and even single PCs. We now discuss Network, CPU, and I/O efficiency separately.

[14] [AKM13] discusses how the processors must ensure that "chunks" are not split, and how better load-balancing techniques require more levels of communication.

Network

Because the master sends the oriented graph, the graph size is equal to $4|E| + 8|V|$ bytes, as each vertex is represented by a single 4-byte integer.[15] Consequently, since it is sent to R machines, the total traffic for transmitting the graph is $R \cdot (4|E| + 8|V|)$ bytes. Though this may appear excessive at first sight, the Twitter graph [KLPM10], which is traditionally used as the biggest graph input for triangle counting algorithms, and has over 1.2 billion edges, has a size of less than 5GB. On a 10 Gigabit network, which is typical for data centers, this only takes 4 seconds per machine, which is a small proportion of the computation time (Chapter 5). Crucially, the graph is only sent once per machine, so our algorithm favors data centers with low R but high P.

Accounting for the communication cost, since $|V| \leq |E|$, the network traffic for each of the R machines is $\Theta\left(P + |E| + t_i\right)$, where t_i is the total number of triangles found in the i-th machine if the algorithm requires triangle listing, and 0 otherwise. The overall network traffic is thus $\Theta\left(R \cdot (P + |E|) + T\right)$, where T is the total number of triangles in the graph (or 0 if only triangle counting was requested).

The Master

The master is responsible for orienting the graph according to the degree-based order. Though a more complicated protocol — where different portions of the graph are oriented by different machines — could be devised, it was deemed as unnecessary since it would require extensive coordination, and a memory size proportional to V in all of the machines. Instead, and as discussed in Section 4.2.1, the proposed methodology takes $\mathcal{O}\left(scan(|E|)\right)$ I/Os and $\mathcal{O}\left(|E|\right)$ CPU time, assuming there is enough memory to hold V, which is typically a reasonable assumption. If needed, the master can remove high-degree vertices using Algorithm 5 for a CPU complexity of $\mathcal{O}\left(|E|^2/M\right)$ and

[15] When the programs have been compiled with the `BIT64` option for graphs with more than 4 billion vertices, the number of bytes is doubled.

an I/O complexity of $\mathcal{O}(|E|^2/MB + t/B)$, where t represents the number of triangles found. The master is also responsible for adding the triangle counts received (in parallel) and also concatenating the triangle listing (sequentially), for a CPU complexity of $\mathcal{O}(R+T)$ and an I/O complexity of $\mathcal{O}((T+R)/B)$, as there might be an extra additional block for each of the R machines.

CPU

Note that each processor is responsible for a unique (contiguous) section of the graph, meaning that there are no repeated computations. The chunk that each processor is responsible for has size $S = \frac{|E|}{RP}$, and each processor must make $N = \lceil \frac{S}{M} \rceil$ iterations over the graph. During these iterations, the graph is read once for creation of the vertex structures, and contributes $\mathcal{O}(|E|)$ processing time. Though it would be impossible to calculate exactly the amount of computations performed in each iteration for counting triangles as it depends closely on the graph structure, we know by the proof of Theorem 4.2.4 that over all processors, these computations sum to $\mathcal{O}(\alpha \cdot |E|)$.[16] Consequently, the total number of computations across all processors is

$$\mathcal{O}\left(RP \cdot \left\lceil \frac{|E|}{RPM} \right\rceil \cdot |E| + \alpha \cdot |E|\right) = \mathcal{O}\left(RP \cdot |E| + \frac{|E|^2}{M} + \alpha \cdot |E|\right)$$

because $\left\lceil \frac{|E|}{RPM} \right\rceil \leq \frac{|E|}{RPM} + 1$.

Consequently, one of the important distinctions between DMGT and frameworks like PATRIC and PowerGraph which load entire subgraphs in memory is that in DMGT, $|E|$ can still be larger than the total amount of available memory RPM, whereas existing distributed frameworks would fail in this case. Moreover, we see that when $RPM > |E|$, we can reduce M to $\frac{|E|}{RP}$ without affecting any individual processor, whereas the total amount of memory

[16] Technically, if S is smaller than the maximum degree, the complexity changes to $\mathcal{O}(RP \cdot \alpha \cdot |E|)$ as a single vertex can be split across RP machines, but this would be atypical for the quantities discussed here.

needed in PATRIC and PowerGraph can exceed $|E|$, due to overlapping subgraphs. Finally, it is important to note that the limiting factor after the graphs have been sent to all machines is the processor responsible for the highest number of triangles, so increasing the total number of processors is usually preferable, even with the same amount of total memory, as we also identify in Chapter 5.

I/O

As above, each processor makes $N = \left\lceil \frac{|E|}{RPM} \right\rceil$ iterations over the graph, and outputs a variable number of triangles t, making its I/O complexity equal to $\mathcal{O}\left(N \cdot scan(|E|) + scan(t)\right)$. Consequently, the total I/O over all processors is

$$\mathcal{O}\left(RP\frac{|E|}{B} + \frac{|E|^2}{MB} + \frac{T}{B}\right)$$

Overall Complexity

Our findings are summarized in Theorem 4.3.1:

Theorem 4.3.1 (DMGT Complexity)**.** *Using the convention that T represents the number of triangles in the case of triangle listing and 0 in the case of triangle counting, as well as denoting by S the number 1 if $\frac{|E|}{RP} > d_{max}(G^*)$ and RP otherwise, DMGT incurs a total of:*

- $\Theta\left(RP + R|E| + T\right)$ *Network traffic*
- $\mathcal{O}\left(RP|E| + \frac{|E|^2}{M} + S\alpha|E|\right)$ *CPU computations*
- $\mathcal{O}\left(RP\frac{|E|}{B} + \frac{|E|^2}{MB} + \frac{T}{B}\right)$ *I/Os*

The total amount of time taken from start to finish is equal to the cost of orientation, plus the cost of sending the graph across the network, plus the cost of the bottleneck processor responsible for pivot edges origination from the vertex of maximum degree.

Chapter 5

Evaluation

As explained in Chapter 3, there are few frameworks that can be used to evaluate a distributed algorithm for triangle listing and counting. As a result, we primarily focus on showing scalability across processors and machines, but also on empirically showing that our algorithm exhibits high performance across a wide range of scenarios, before comparing against other approaches in Section 5.6. The full raw data collected can be found in Appendix B.

5.1 Setup and Methodology

Though preliminary experiments were conducted on local Windows and Linux machines using both HDDs and SSDs, we rented 4 Amazon EC2 `c3.8xlarge` instances in the US Oregon region, each of which contained 32 vCPU units, 60GiB of memory, and 2×320GB of SSD storage. The instances were connected using a 10 Gigabit Ethernet network. To fully exploit the underlying "High Frequency Intel Xeon E5-2680 v2 (Ivy Bridge)" processors, we used Amazon's 64-bit Linux image (version 2014.03.1), which supported hardware virtualization. In order to account for random variation due to the underlying server load, we repeated each of our experiments three times, and

present the averages here.[1] Though our code fully supports triangle listing, our experiments only measure counting time, since the overhead would be the same for any algorithm used, and the reported timings of related literature are only for triangle counting. The testing scripts and detailed logs can be found at `https://www.dropbox.com/s/ddwwo6bfgnhbbti/DMGT.zip`, together with the rest of the source code, which was compiled with `G++`, using the `-O3` optimization option.

5.2 Datasets

For consistency and easier evaluation, we used the same datasets as those found in other works in distributed triangle counting [AKM13, SV11, PC13, GLG$^+$12, Low13], which are described in Table 5.1, and are sorted by size. The number of edges reported corresponds to the number of *undirected* edges, and the size column represents the total size of the adjacency and degree files, in *binary* format.[2] Most of our datasets come from Stanford's Large Network Dataset Collection (SNAP) [Les], except for the Twitter dataset which comes from the website of the authors of [KLPM10].[3]

It is important to stress here that the datasets used and evaluation methodology employed is consistent with the literature on distributed triangle counting, which focuses on the Twitter dataset [AKM13, GLG$^+$12, Low13, PC13, SV11], and which provides a good anchor point for comparing implementations, especially compared to privately-generated synthetic datasets. Finally, it is worth mentioning that our triangle counts are **correct**, as all but the Twitter datasets have reported triangle counts on the SNAP repository [Les]. As for the Twitter dataset, our value is consistent with the 34.8B triangles reported by PATRIC [AKM13], showing that the TTP MapReduce algorithm [PC13] mis-reports the count as 1.5B, possibly due to an integer overflow.

[1] Relative standard deviations were also calculated, and for the most part are below 1-2%, except for smaller graphs under high memory and core usage. These deviations do not alter our discussion, so we refer the reader to Appendix B for more details.

[2] The files in textual format are about 4.6× larger.

[3] Found at `http://an.kaist.ac.kr/traces/WWW2010.html`

Graph	Nodes	Edges	Triangles	Size	Source
web-BerkStan	685K	6.6M	64,690,980	56MB	[LLDM08]
as-Skitter	1.7M	11.1M	28,769,868	98MB	[LKF05]
soc-LiveJournal1	4.8M	43.1M	285,730,264	365MB	[LLDM08]
com-Orkut	3.1M	117.2M	627,584,181	917MB	[YL12]
Twitter	41.7M	1.2B	34,824,916,864	9.4GB	[KLPM10]
com-Friendster	65.6M	1.8B	4,173,724,142	14.3GB	[YL12]

Table 5.1: Graphs used for our experiments

5.2.1 Parsing

It is important to note that since the graphs found above were only available in textual format, we pre-processed them to convert them into bi-directional, binary graphs, in accordance with Sections 4.2.1 and 4.3.1. Though parsing text is time-consuming, we do not include this pre-processing time in our discussion, because virtually all algorithms require neighbors of a vertex to be grouped together and graphs to be bi-directional for exact counting. Moreover, all efficient storage techniques use binary data (e.g. [YNDR14, KBG12, RMZ13, PGA10]), and conversion between such formats and our format would just require a linear pass over their file. Consequently, we believe that the starting point for our measurements should be with the adjacency list in binary format and already sorted by source and destination, as described in more detail in Section 4.2.1. However, because degree-based ordering is non-standard, we consider the orientation cost separately in Section 5.3, and include it in our overall time measurements in Section 5.6.

5.3 Orientation

Since orientation is sequential and independent of the rest of our calculations, we time it separately, and present it in Table 5.2, together with the maximum

oriented degree $d^*_{max} = d_{max}(G^*)$.

Graph	d^*_{max}	Time (s)
web-BerkStan	201	0.20
as-Skitter	231	0.45
soc-LiveJournal1	687	2.08
com-Orkut	535	5.40
Twitter	4,102	87.59
com-Friendster	868	196.05

Table 5.2: Orientation time for our datasets

As we see in Section 5.4, these times are insignificant compared to sequential triangle counting, but in Section 5.5 we see that they represent 50% of the total computation time when distributing across multiple machines and processors. However, orientation is crucial, as it provides better theoretical guarantees and a $2-3\times$ speed-up on larger graphs, as we informally verified. It is also worth mentioning that the orientation times for Friendster had a higher deviation, and ranged from 181.68 to 221.32 seconds.

5.4 MGT

We initially tested our algorithm on a single machine, using various combinations of cores and memory per core. Because the results are copious, we only summarize our observations and remind the reader that the full data is available in Appendix B. We choose to discuss small and large graphs separately, and exclude orientation cost, which would always be a constant offset for each datapoint.

5.4.1 Small Graphs

Figure 5.1 illustrates the performance of our algorithm on a variable amount of cores, with a fixed amount of memory M equal to 400MB per core. As

can be seen, for the two smallest datasets, Skitter and BerkStan, increasing the number of cores beyond a certain amount (8 for BerkStan, 16 for Skitter) actually slows down our algorithm due to the initialization overhead.

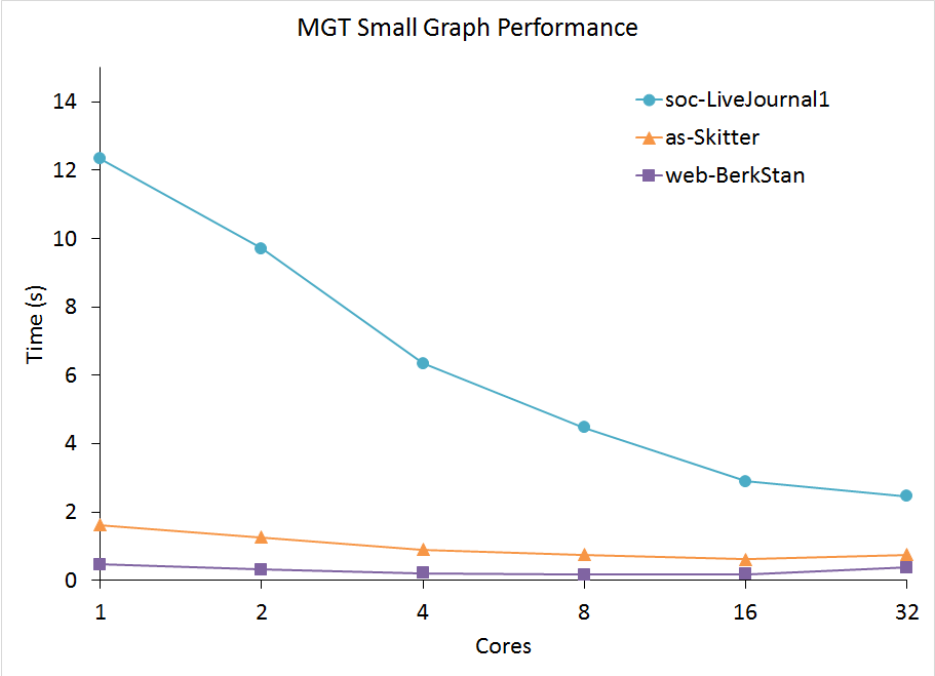


Figure 5.1: MGT on small graphs with 400MB of memory per core

A similar situation occurs with increasing memory beyond the size where edges can entirely fit in memory. Specifically, as can be seen in Figure 5.2 for the LiveJournal dataset, increasing memory beyond a certain point actually impedes our algorithm, as the time taken to clear the arrays dominates the computation time of processing the few edges that were assigned to each core. More concretely, increasing memory when few cores are available is beneficial, as the algorithm requires more than one iterations of the graph, but as memory and cores increase, so does the total amount of computation.[4]

However, as the graphs increase in size, increasing the number of cores is always beneficial, as can also be seen for the Orkut dataset (Figure 5.3), which was allocated a fixed memory of 500MB per core.

[4]PATRIC [AKM13] experiences a similar phenomenon for smaller graphs, due to the increased MPI communication cost between processors.

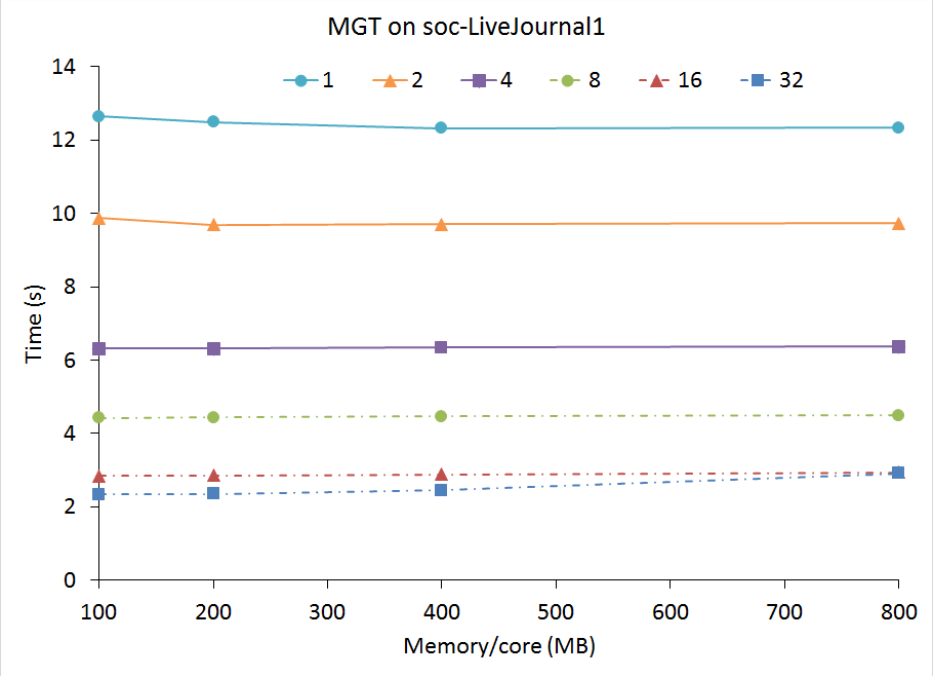


Figure 5.2: MGT on soc-LiveJournal1 with different cores and memory

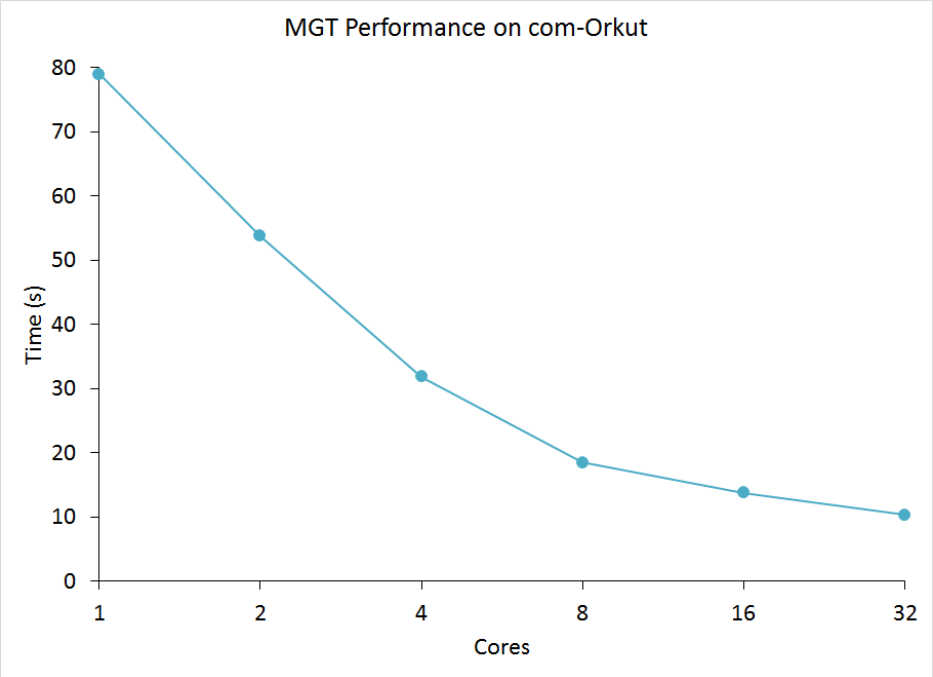


Figure 5.3: MGT on com-Orkut with 500MB of memory per core

5.4.2 Large Graphs

The situation for larger graphs is similar, but due to their size, the effects are more pronounced. For the two larger graphs, however, we chose not to use a fixed amount of memory per core, but instead allocate a total of 32,000MB of RAM and split it equally among the cores. For instance, in the single-core solution, our algorithm is allocated 32,000MB, but in the 32-core solution or algorithm uses 1,000MB per core. Figure 5.4 contains our findings, showing that for both the Twitter and the Friendster graphs, distributing the load across 32 cores brings in an 11-fold decrease in time over the baseline 1-core solution. Although not shown here, the effect of increasing the memory size is similar to the one discussed previously: for fewer cores, increasing memory size can be beneficial, but as the number of cores increases, a larger memory size can slow the algorithm down, but by an insignificant amount.

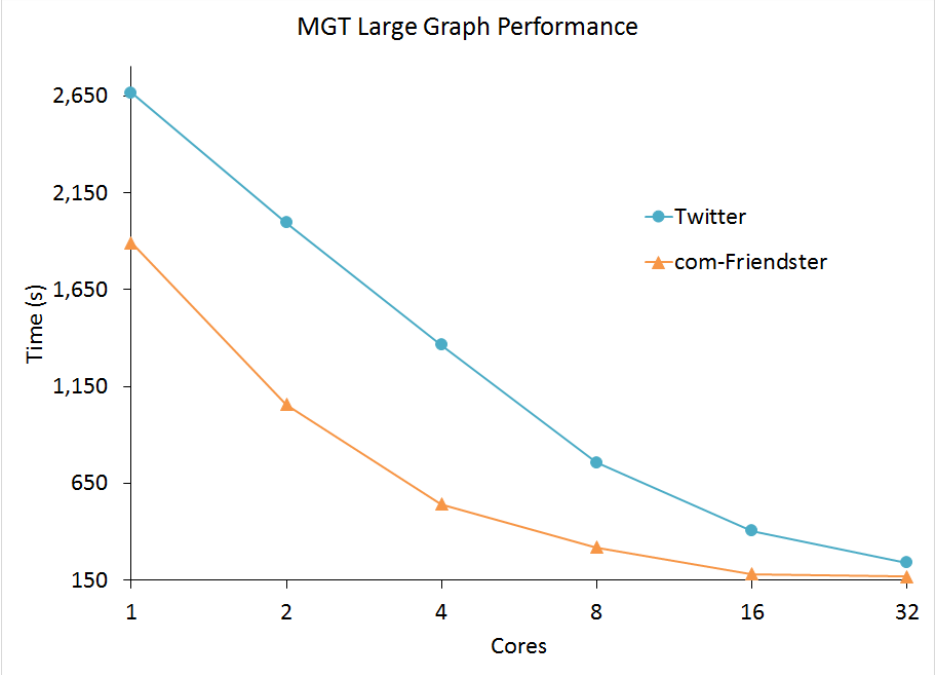


Figure 5.4: MGT on large graphs with a variable number of cores

5.5 DMGT

Despite the fact that, as we saw in Section 5.4, increasing the memory allocated per thread beyond the necessary minimum can increase runtime by a small amount, we ran the distributed version of our algorithm using 2, 3, and 4 servers,[5] and allocated 32 threads and 1000MB/thread on all of our datasets. Because the algorithm exhibits different performance characteristics for different graph sizes, we separate our discussion into small graphs (Figure 5.5) and large ones (Figure 5.6), again excluding orientation cost.

For the smaller graphs BerkStan, Skitter, and LiveJournal, we notice that performance is overall constant (with minor variations and higher relative standard deviations), because the added network and threading overhead neutralizes the effect of adding extra processing nodes. However, for the slightly larger Orkut graph, additional computational power is, in fact, beneficial initially, but makes little difference between 3 and 4 nodes.

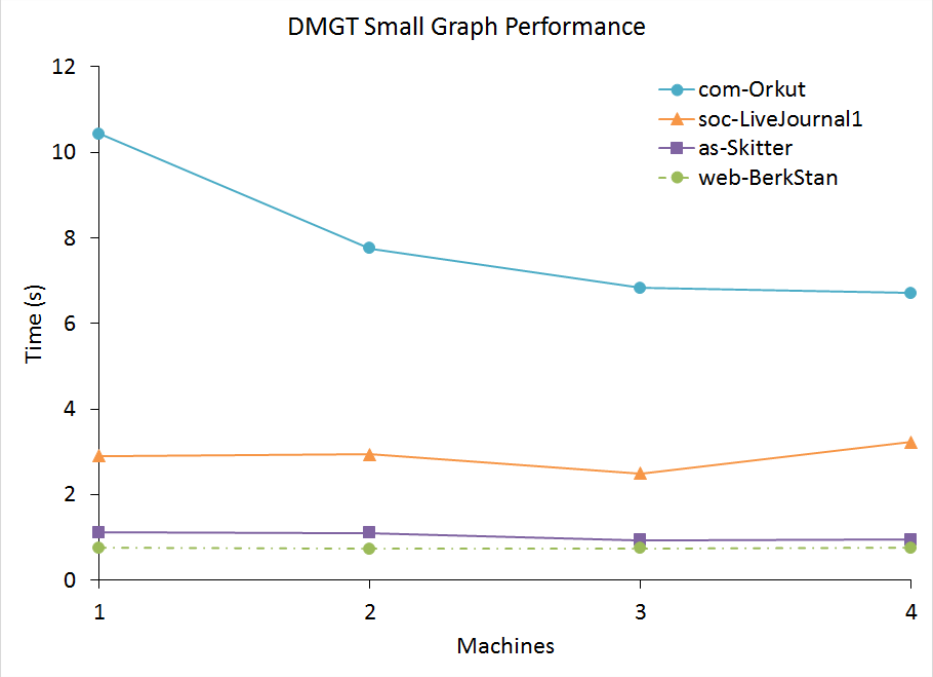


Figure 5.5: DMGT for small graphs

[5]Using just 1 server is equivalent to MGT, so it was not tested separately.

For the two larger graphs, the behavior appears to be more dependent on the concrete structure of the graph. Specifically, we see that adding more nodes always decreases computation time in the case of the Twitter graph, but the time plateaus between 3 and 4 nodes for the Friendster graph.

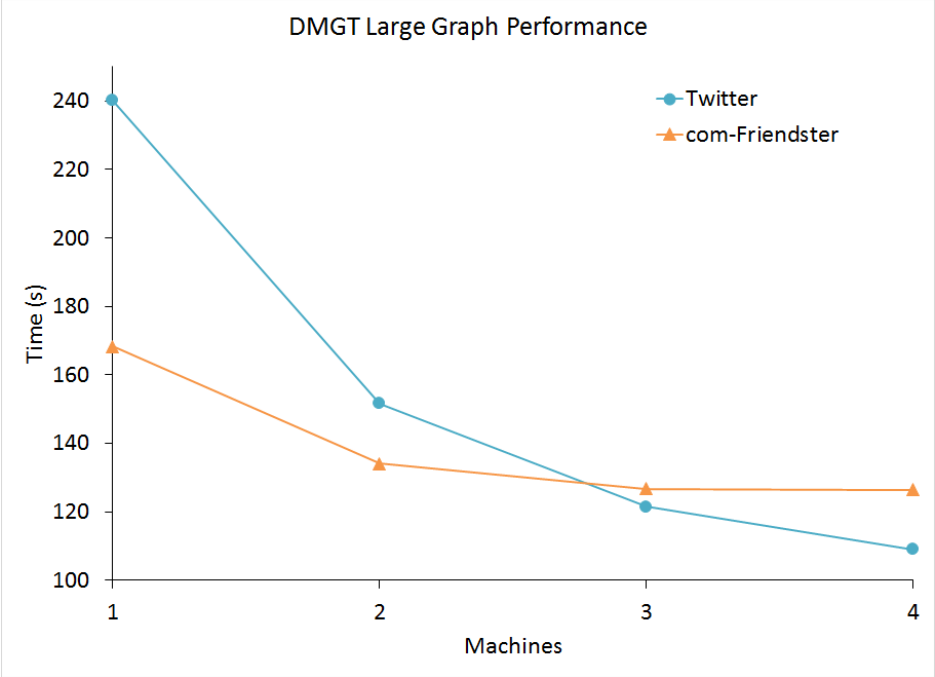


Figure 5.6: DMGT for large graphs

The reason for the behavior exhibited by the graphs here is related to the phenomenon we described in Section 4.3.3 and witnessed in Section 5.4: when the number of processors increases beyond a certain point, the amount of memory allocated per processor should be decreased, because initializing the memory becomes more costly than processing the few edges allocated to that processor. For the Twitter graph specifically, due to its large $d^*_{max} = 4,102$ and higher number of triangles, this behavior occurs at a later point, compared to the Friendster dataset, which has $d^*_{max} = 868$ (Table 5.2). These findings can be summarized in Figure 5.7, which illustrates how when the number of cores increases (with constant memory per core), large graphs gain substantial speedups, whereas performance degrades for small graphs due to

the communication and initialization costs.[6]

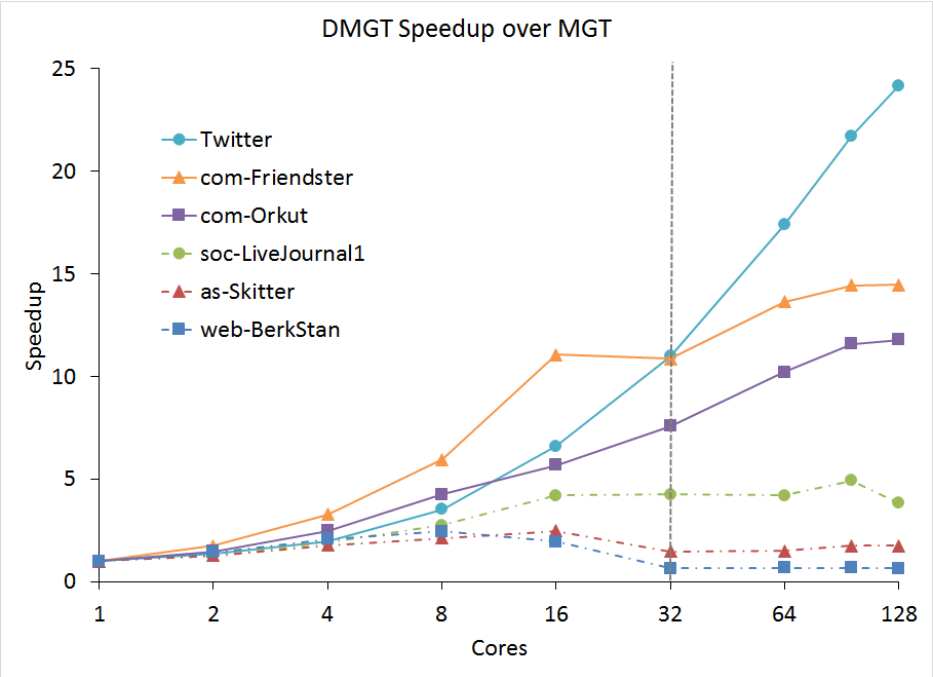


Figure 5.7: DMGT speedup over single-core MGT. The vertical line represents the transition from a single machine to multiple ones

5.6 Comparisons

Table 5.3 compares our DMGT algorithm using 128 processors, and 1000MB of memory/processor against PATRIC [AKM13], using 200 processors and 4GB memory/processor, and TTP [PC13], the MapReduce algorithm using 47 nodes with 4GB of memory/node.[7] Note that although TTP uses fewer processors, the comparison presented here is not unfair: even using only 32 processors, our algorithm takes less than 5.5 minutes on the Twitter dataset. Additionally, TTP would not benefit substantially from more processors, as

[6]This degradation typically occurs when there are fewer than $500,000$ edges per core.

[7]We note that although our algorithm always correctly calculates the exact number of triangles for each dataset (Table 5.1), we only list the magnitude of these quantities here for convenience.

it is network-bound: the other MapReduce algorithm GP took 514 minutes using 47 nodes [PC13], and 423 minutes using 1636 nodes [SV11]! Additionally, though it is unclear whether PATRIC includes orientation in their time measurements, we include it in ours for completeness. Finally, although PowerGraph claims to be able to count triangles on the Twitter dataset in 1.5 minutes [GLG+12] or even 15 seconds [Low13] using 1024 processors, we do not include these results here, as they completely ignore graph-loading and splitting time, as well as MPI set-up time, as can also be seen on their open-source implementation page.[8]

Graph	**Triangles**	**DMGT**	[AKM13]	[PC13]
web-BerkStan	64.7M	0.96s	0.10s	1.31m
as-Skitter	28.8M	1.40s	-	1.62m
soc-LiveJournal1	285.7M	5.31s	0.8s	3.63m
com-Orkut	627.6M	12.11s	-	-
Twitter	34.8B	3.28m	9.4m	213m
com-Friendster	4.2B	5.37m	-	-

Table 5.3: Comparison between distributed triangle counting algorithms

Consequently, we see that although for smaller graphs, PATRIC is faster by a few seconds, due to the overheads of traversing the entire graph on each processor, as graphs get larger, DMGT scales better, and is faster, even with fewer resources. As a matter of fact, Figure 5.8 illustrates more clearly that even using only 16 cores and 1GB of memory per core, our algorithm is faster than PATRIC's 200 processors and 4GB of memory per processor. Note that much like our algorithm on a graph of this size, PATRIC's performance on the Twitter graph increases as the number of processors increase, so our algorithm exhibits better performance than PATRIC, even when using 12.5× fewer processors and 50× less memory!

[8] `https://github.com/graphlab-code/graphlab/blob/master/toolkits/graph_analytics/undirected_triangle_count.cpp`

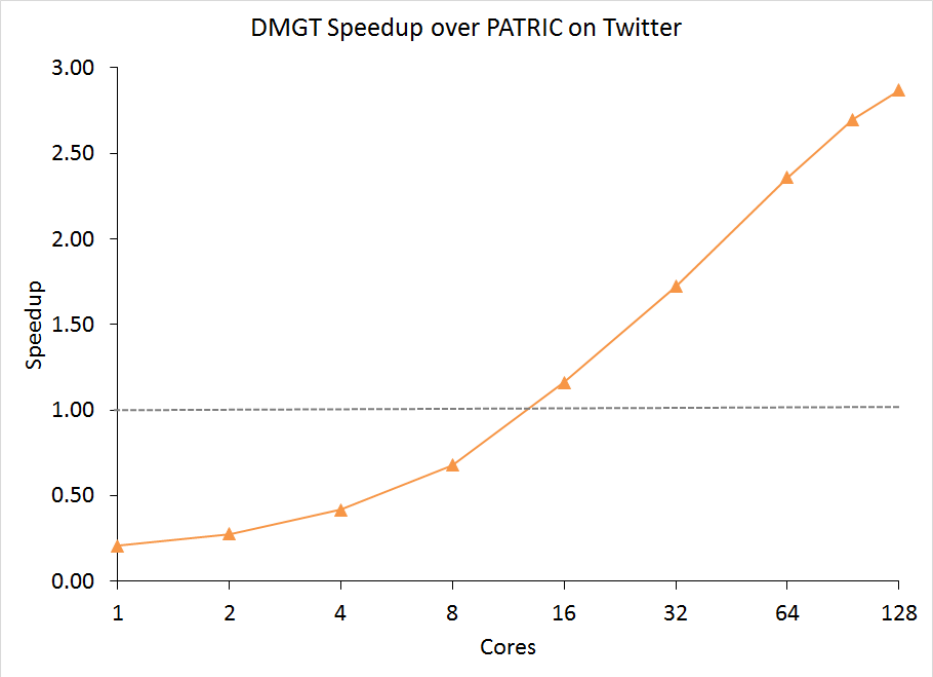


Figure 5.8: DMGT performance against PATRIC on the Twitter graph

5.7 A Note on External Memory

As was already established by the MGT authors [HTC13], MGT is over 3 times faster than all other external memory algorithms, a fact which we informally verified with the provided MGT binary, which also implemented all other external-memory triangle listing algorithms. However, the program sometimes crashed when running them, and since we could not find the source code or binaries from the authors of other external-memory algorithms, we could not run an independent, formal experiment focusing on external memory only, which was, after all, not the focus of DMGT.

In our investigations, we also discovered that the given MGT implementation had a coding error for larger graphs. For instance, MGT reported $627,506,739$ triangles for Orkut (compared to $627,584,181$) and $559,420,538$ triangles for Twitter (compared to 34.8 billion)! It is important to note that these miscounts cannot be attributed to an integer overflow, unlike the TTP [PC13] problem mentioned in Section 5.1.

Even so, for the sake of completeness, we note that for the graphs for which MGT reported the correct number of triangles, our implementation was 10% slower, perhaps due to lack of an additional optimization or due to the extensive modularity of our code.

5.8 Conclusions

The key take-away from this section is that our DMGT algorithm is well-suited for a variety of environments, from one single-core machine with limited memory, to multiple multi-core machines with extensive available memory. In all of these scenarios, our algorithm out-performs the state-of-the-art algorithms for larger graphs (even with 12.5× fewer resources), though it can be a few seconds slower for smaller ones, since allocating too few edges to each processor can slow the algorithm down. In line with our analysis (Section 4.3.3), our experiments show that transmitting the entire graph to all nodes is not a bottleneck in clusters, and thus our novel approach of focusing on SSDs in the distributed environment comes with impressive performance. We conclude that more virtual CPUs are preferable to more memory, and our algorithm is highly scalable and parallelizable. However, it does incur an orientation bottleneck, which can account for up to 50% of the total computation time when using 128 processors. Future work should thus focus on orienting the graph in a distributed fashion. Such an attempt could benefit from PATRIC's [AKM13] load-balancing methodology, but to a large extent, the orientation problem is more of a systems-design question of efficient message-passing, rather than an algorithmic design question, such as the triangle listing one we have answered with DMGT.

Chapter 6

Conclusions and Future Work

In this project we created a distributed triangle listing and counting algorithm that focuses on external-memory I/O efficiency, but also provides theoretical CPU and Network guarantees. Our extensible, flexible framework works well in a variety of computational environments, and is based on the recent MGT algorithm [HTC13]. In creating our framework, we reimplemented MGT due to its closed-source nature, and uncovered errors in the provided MGT binary [HTCb] as well as hidden assumptions and implementation details that were omitted from the theoretical proofs given by its authors. As a result, we modified the algorithm's original analysis to prove that the overall complexity remains the same. We proved that under reasonable assumptions, using R machines with P processors and M memory per processor, for triangle listing, our DMGT incurs a total of:

- $\Theta\left(RP + R|E| + T\right)$ Network traffic
- $\mathcal{O}\left(RP|E| + \frac{|E|^2}{M} + \alpha|E|\right)$ CPU computations
- $\mathcal{O}\left(RP\frac{|E|}{B} + \frac{|E|^2}{MB} + \frac{T}{B}\right)$ I/Os

where E is the edge set of the graph, B the disk block size, T the total number of triangles, and α the arboricity of the graph. DMGT is thus the first distributed triangle listing algorithm which provides theoretical guarantees

for efficiency for I/O, CPU, Memory, and Network traffic.

For large graphs, DMGT outperforms the state-of-the-art distributed system PATRIC [AKM13] by almost 3× when using 36% fewer resources, while for smaller graphs, PATRIC is only faster by a few seconds. Our approach of focusing on external memory in a distributed system is thus not only novel, but also achieves better performance and theoretical guarantees than other algorithms which load entire sub-graphs in memory. This makes our approach more scalable, since disk capacity is often cheaper than main memory, and our algorithm performs especially well in low-memory scenarios. In essence, we believe that as graphs become larger, the requirement of fitting the entire graph in the memory of the available machines will no longer be viable, increasing the need for our external memory algorithm.

Though our experiments were performed on Amazon's EC2 servers, informal experiments on local machines with SSDs and HDDs showed that our results are also reproducible under different disk configurations. Future work, however, could focus on more formally investigating performance under different disk arrangements. Moreover, we identified that our algorithm suffers from a sequential orientation bottleneck that can account up to 50% of the time spent for triangle counting. This bottleneck, however, is *non-essential*: our algorithm does not require sequential orientation, so an improvement in the orientation step will immediately improve our overall algorithm. Future work could thus focus on parallelizing the orientation step, and PATRIC presents load-balancing ideas which can be readily applied in this context. Moreover, future work could focus on making our algorithm dynamic: each processor would still be responsible for a subset of the edges, and it would count newly formed triangles when new edges arrive. A similar idea could be used to create an approximation algorithm, for instance through PATRIC's sparsification approach, but we should note that approximation algorithms traditionally tend to be streaming algorithms that avoid external memory.

Overall, we firmly believe in our disk-based approach in distributed systems, and hope that our framework and results can inspire researchers to focus on utilizing SSDs — rather than RAM — more efficiently and effectively.

Appendix A

Notation

Table A.1 summarizes our notation. Its first part describes properties of a graph $G = (V, E)$, without subscripts that may be used when G is ambiguous. Its second part describes the symbols used for our theoretical analysis.

Symbol	**Explanation**
G	A simple, undirected graph
V	G's vertex set
E	G's edge set
n	The size $\|V\|$ of the vertex set
m	The size $\|E\|$ of the edge set
α	The arboricity of G
T	The total number of triangles in G
A	The adjacency matrix of G
G^*	The degree-based orientation of G
H^+	The extended subgraph for $V_H \subseteq V$
$N(u)$	The set of neighbors of $u \in V$
$d(u)$	The degree $\|N(u)\|$ of $u \in V$
d_{max}	The maximum degree in G
$[n]$	The set $\{0, \ldots, n-1\}$
R	The number of machines
P	The number of processors per machine
M	The memory size per processor
B	The disk block size
$scan(N)$	$\Theta\left(N/B\right)$ I/Os
$sort(N)$	$\Theta\left(N/B \log_{M/B} N/B\right)$ I/Os

Table A.1: Notation

Appendix B

Raw Data

We include the raw data we collected, for independent verification. For details on the data collection procedure, please see Chapter 5. Unless otherwise noted, memory is in MB, and all measurements are in seconds, and do not include orientation cost. Relative standard deviation is also included and expressed as a percentage.

B.1 Orientation

This section contains timing measurements for the orientation of all datasets.

Graph	Run 1	Run 2	Run 3	Mean	%RSD
web-BerkStan	0.20	0.20	0.20	0.20	1.18
as-Skitter	0.44	0.45	0.47	0.45	3.87
soc-LiveJournal1	2.07	2.08	2.10	2.08	0.74
com-Orkut	5.49	5.40	5.30	5.40	1.76
Twitter	88.34	87.15	87.30	87.59	0.74
com-Friendster	221.32	185.15	181.68	196.05	11.20

Table B.1: Orientation times

B.2 MGT

This section contains the single-machine MGT measurements with various core and memory combinations for all graphs separately.

B.2.1 web-BerkStan

Memory	Run 1	Run 2	Run 3	Mean	%RSD
100	0.43	0.43	0.43	0.43	0.31
200	0.44	0.44	0.44	0.44	0.28
400	0.46	0.46	0.46	0.46	0.11
800	0.49	0.49	0.49	0.49	0.10

Table B.2: MGT for web-BerkStan with 1 core

Memory	Run 1	Run 2	Run 3	Mean	%RSD
100	0.29	0.29	0.29	0.29	0.12
200	0.30	0.30	0.30	0.30	0.10
400	0.31	0.31	0.31	0.31	0.23
800	0.35	0.35	0.35	0.35	0.13

Table B.3: MGT for web-BerkStan with 2 cores

Memory	Run 1	Run 2	Run 3	Mean	%RSD
100	0.17	0.18	0.18	0.18	0.44
200	0.18	0.18	0.18	0.18	0.24
400	0.20	0.20	0.20	0.20	0.27
800	0.24	0.24	0.24	0.24	0.20

Table B.4: MGT for web-BerkStan with 4 cores

Memory	Run 1	Run 2	Run 3	Mean	%RSD
100	0.13	0.13	0.14	0.13	0.90
200	0.14	0.14	0.14	0.14	1.25
400	0.16	0.16	0.16	0.16	0.95
800	0.20	0.20	0.20	0.20	0.39

Table B.5: MGT for web-BerkStan with 8 cores

Memory	Run 1	Run 2	Run 3	Mean	%RSD
100	0.13	0.14	0.14	0.14	4.30
200	0.14	0.15	0.15	0.15	3.14
400	0.17	0.18	0.17	0.17	1.49
800	0.30	0.23	0.23	0.25	15.58

Table B.6: MGT for web-BerkStan with 16 cores

Memory	Run 1	Run 2	Run 3	Mean	%RSD
100	0.19	0.22	0.20	0.20	7.46
200	0.22	0.23	0.23	0.23	3.98
400	0.47	0.32	0.34	0.38	20.36
800	0.90	0.69	0.67	0.75	16.51

Table B.7: MGT for web-BerkStan with 32 cores

B.2.2 as-Skitter

Memory	Run 1	Run 2	Run 3	Mean	%RSD
100	1.60	1.60	1.60	1.60	0.11
200	1.61	1.61	1.61	1.61	0.07
400	1.63	1.63	1.63	1.63	0.04
800	1.66	1.66	1.66	1.66	0.06

Table B.8: MGT for as-Skitter with 1 core

Memory	Run 1	Run 2	Run 3	Mean	%RSD
100	1.23	1.23	1.23	1.23	0.04
200	1.24	1.23	1.24	1.24	0.12
400	1.26	1.25	1.25	1.25	0.16
800	1.28	1.29	1.28	1.29	0.21

Table B.9: MGT for as-Skitter with 2 cores

Memory	Run 1	Run 2	Run 3	Mean	%RSD
100	0.87	0.87	0.87	0.87	0.16
200	0.88	0.88	0.88	0.88	0.03
400	0.89	0.89	0.89	0.89	0.08
800	0.93	0.93	0.93	0.93	0.21

Table B.10: MGT for as-Skitter with 4 cores

Memory	Run 1	Run 2	Run 3	Mean	%RSD
100	0.71	0.71	0.72	0.71	0.14
200	0.72	0.72	0.72	0.72	0.07
400	0.74	0.74	0.74	0.74	0.21
800	0.78	0.78	0.78	0.78	0.06

Table B.11: MGT for as-Skitter with 8 cores

Memory	Run 1	Run 2	Run 3	Mean	%RSD
100	0.58	0.58	0.58	0.58	0.34
200	0.59	0.59	0.59	0.59	0.29
400	0.61	0.62	0.62	0.62	0.67
800	0.66	0.67	0.68	0.67	0.88

Table B.12: MGT for as-Skitter with 16 cores

Memory	Run 1	Run 2	Run 3	Mean	%RSD
100	0.63	0.62	0.63	0.63	0.67
200	0.65	0.67	0.66	0.66	1.22
400	0.73	0.75	0.77	0.75	2.81
800	1.16	1.07	1.12	1.12	3.69

Table B.13: MGT for as-Skitter with 32 cores

B.2.3 soc-LiveJournal1

Memory	Run 1	Run 2	Run 3	Mean	%RSD
100	12.64	12.65	12.63	12.64	0.09
200	12.50	12.50	12.49	12.50	0.05
400	12.34	12.32	12.32	12.33	0.08
800	12.34	12.34	12.34	12.34	0.02

Table B.14: MGT for soc-LiveJournal1 with 1 core

Memory	Run 1	Run 2	Run 3	Mean	%RSD
100	9.88	9.87	9.88	9.88	0.07
200	9.69	9.69	9.69	9.69	0.03
400	9.72	9.71	9.71	9.71	0.07
800	9.74	9.73	9.74	9.74	0.08

Table B.15: MGT for soc-LiveJournal1 with 2 cores

Memory	Run 1	Run 2	Run 3	Mean	%RSD
100	6.32	6.31	6.31	6.31	0.13
200	6.34	6.29	6.32	6.31	0.38
400	6.34	6.35	6.33	6.34	0.23
800	6.37	6.38	6.36	6.37	0.15

Table B.16: MGT for soc-LiveJournal1 with 4 cores

Memory	Run 1	Run 2	Run 3	Mean	%RSD
100	4.42	4.43	4.42	4.42	0.10
200	4.43	4.43	4.44	4.43	0.06
400	4.46	4.47	4.47	4.46	0.11
800	4.49	4.49	4.49	4.49	0.07

Table B.17: MGT for soc-LiveJournal1 with 8 cores

Memory	Run 1	Run 2	Run 3	Mean	%RSD
100	2.85	2.84	2.86	2.85	0.29
200	2.86	2.87	2.86	2.86	0.29
400	2.89	2.89	2.89	2.89	0.08
800	2.93	2.93	2.96	2.94	0.64

Table B.18: MGT for soc-LiveJournal1 with 16 cores

Memory	Run 1	Run 2	Run 3	Mean	%RSD
100	2.40	2.28	2.34	2.34	2.57
200	2.33	2.34	2.40	2.36	1.65
400	2.41	2.40	2.55	2.45	3.43
800	2.89	2.78	3.05	2.91	4.75

Table B.19: MGT for soc-LiveJournal1 with 32 cores

B.2.4 com-Orkut

Memory	Run 1	Run 2	Run 3	Mean	%RSD
500	78.98	78.97	78.96	78.97	0.01
1,000	79.10	79.12	79.08	79.10	0.02
1,500	79.15	79.14	79.14	79.14	0.01
2,000	79.16	79.16	79.19	79.17	0.02

Table B.20: MGT for com-Orkut with 1 core

Memory	Run 1	Run 2	Run 3	Mean	%RSD
500	53.72	53.73	53.74	53.73	0.01
1,000	53.75	53.74	53.81	53.77	0.07
1,500	53.79	53.80	53.79	53.80	0.01
2,000	53.85	53.84	53.87	53.85	0.03

Table B.21: MGT for com-Orkut with 2 cores

Memory	Run 1	Run 2	Run 3	Mean	%RSD
500	31.87	31.87	31.88	31.87	0.01
1,000	32.00	31.93	31.93	31.95	0.14
1,500	31.95	31.96	31.96	31.96	0.02
2,000	32.02	32.01	32.00	32.01	0.03

Table B.22: MGT for com-Orkut with 4 cores

Memory	Run 1	Run 2	Run 3	Mean	%RSD
500	18.50	18.49	18.51	18.50	0.04
1,000	18.53	18.53	18.53	18.53	0.02
1,500	18.58	18.60	18.58	18.59	0.06
2,000	19.19	18.62	18.64	18.82	1.73

Table B.23: MGT for com-Orkut with 8 cores

Memory	Run 1	Run 2	Run 3	Mean	%RSD
500	13.78	13.80	13.80	13.79	0.10
1,000	14.16	13.83	13.84	13.94	1.37
1,500	14.33	14.41	14.32	14.35	0.32
2,000	14.54	14.52	14.51	14.52	0.07

Table B.24: MGT for com-Orkut with 16 cores

Memory	Run 1	Run 2	Run 3	Mean	%RSD
500	10.55	10.49	9.82	10.29	3.96
1,000	10.47	10.44	10.38	10.43	0.40
1,500	11.05	10.78	11.31	11.04	2.40

Table B.25: MGT for com-Orkut with 32 cores

B.2.5 Twitter

Memory	Run 1	Run 2	Run 3	Mean	%RSD
2,000	2,636.68	2,636.96	2,637.17	2,636.94	0.01
4,000	2,625.22	2,625.28	2,625.73	2,625.41	0.01
8,000	2,623.23	2,624.30	2,624.28	2,623.94	0.02
16,000	2,624.35	2,623.97	2,624.54	2,624.29	0.01
32,000	2,649.83	2,646.57	2,699.40	2,665.27	1.11

Table B.26: MGT for Twitter with 1 core

Memory	Run 1	Run 2	Run 3	Mean	%RSD
2,000	1,960.29	1,960.12	1,959.97	1,960.13	0.01
4,000	1,964.08	1,958.47	1,952.45	1,958.33	0.30
8,000	2,004.91	1,992.13	1,952.63	1,983.22	1.37
16,000	2,016.54	2,016.84	1,954.21	1,995.86	1.81

Table B.27: MGT for Twitter with 2 cores

Memory	Run 1	Run 2	Run 3	Mean	%RSD
2,000	1,358.05	1,348.74	1,348.13	1,351.64	0.41
4,000	1,384.01	1,349.03	1,348.32	1,360.45	1.50
8,000	1,390.40	1,349.80	1,350.25	1,363.48	1.71

Table B.28: MGT for Twitter with 4 cores

Memory	Run 1	Run 2	Run 3	Mean	%RSD
1,000	745.55	744.31	744.24	744.70	0.10
2,000	765.38	748.53	747.70	753.87	1.32
4,000	770.59	749.13	749.60	756.44	1.62

Table B.29: MGT for Twitter with 8 cores

Memory	Run 1	Run 2	Run 3	Mean	%RSD
1,000	402.11	396.55	396.38	398.35	0.82
2,000	408.92	400.18	399.92	403.00	1.27

Table B.30: MGT for Twitter with 16 cores

Memory	Run 1	Run 2	Run 3	Mean	%RSD
1,000	244.83	237.98	237.42	240.08	1.72
1,500	251.70	243.50	242.23	245.81	2.09

Table B.31: MGT for Twitter with 32 cores

B.2.6 com-Friendster

Memory	Run 1	Run 2	Run 3	Mean	%RSD
2,000	1,828.70	1,827.74	1,828.84	1,828.43	0.03
4,000	1,866.44	1,824.75	1,855.85	1,849.01	1.17
8,000	1,911.40	1,896.02	1,866.46	1,891.29	1.21
16,000	1,907.55	1,907.33	1,838.72	1,884.53	2.11
32,000	1,917.13	1,915.87	1,838.24	1,890.41	2.39

Table B.32: MGT for com-Friendster with 1 core

Memory	Run 1	Run 2	Run 3	Mean	%RSD
2,000	1,037.35	1,038.47	1,036.94	1,037.59	0.08
4,000	1,058.59	1,017.45	1,016.42	1,030.82	2.33
8,000	1,081.82	1,002.74	999.52	1,028.03	4.53
16,000	1,085.37	1,002.35	1,082.04	1,056.59	4.45

Table B.33: MGT for com-Friendster with 2 cores

Memory	Run 1	Run 2	Run 3	Mean	%RSD
2,000	558.45	556.89	555.90	557.08	0.23
4,000	542.73	542.28	542.53	542.51	0.04
8,000	542.69	542.30	542.39	542.46	0.04

Table B.34: MGT for com-Friendster with 4 cores

Memory	Run 1	Run 2	Run 3	Mean	%RSD
1,000	310.75	305.76	305.17	307.23	1.00
2,000	316.40	313.82	306.48	312.23	1.65
4,000	321.15	319.98	320.58	320.57	0.18

Table B.35: MGT for com-Friendster with 8 cores

Memory	Run 1	Run 2	Run 3	Mean	%RSD
1,000	166.35	165.05	164.45	165.28	0.59
2,000	190.32	177.52	176.13	181.33	4.31

Table B.36: MGT for com-Friendster with 16 cores

Memory	Run 1	Run 2	Run 3	Mean	%RSD
1,000	175.04	169.49	160.32	168.29	4.42
1,500	184.37	181.46	174.19	180.01	2.91

Table B.37: MGT for com-Friendster with 32 cores

B.3 DMGT

This section details our DMGT measurements for a different number of machines, with 32 cores/machines, and 1000MB of memory/core.

B.3.1 2 Machines

Graph	Run 1	Run 2	Run 3	Mean	%RSD
web-BerkStan	0.69	0.75	0.76	0.73	4.72
as-Skitter	1.14	1.17	1.00	1.10	8.43
soc-LiveJournal1	2.63	2.67	3.50	2.93	16.69
com-Orkut	7.55	8.05	7.66	7.75	3.36
Twitter	151.17	157.84	146.03	151.68	3.90
com-Friendster	134.30	131.27	136.75	134.11	2.05

Table B.38: DMGT with 2 machines

B.3.2 3 Machines

Graph	Run 1	Run 2	Run 3	Mean	%RSD
web-BerkStan	0.71	0.76	0.75	0.74	3.93
as-Skitter	0.93	0.94	0.95	0.94	0.94
soc-LiveJournal1	2.47	2.49	2.55	2.50	1.75
com-Orkut	6.81	6.98	6.68	6.82	2.21
Twitter	126.38	117.25	120.53	121.39	3.81
com-Friendster	127.59	126.91	125.56	126.69	0.82

Table B.39: DMGT with 3 machines

B.3.3 4 Machines

Graph	Run 1	Run 2	Run 3	Mean	%RSD
web-BerkStan	0.75	0.77	0.74	0.76	2.30
as-Skitter	0.93	0.94	0.96	0.94	1.73
soc-LiveJournal1	3.81	2.88	2.99	3.23	15.75
com-Orkut	6.62	6.85	6.66	6.71	1.81
Twitter	105.38	112.14	109.49	109.00	3.13
com-Friendster	124.45	129.81	124.82	126.36	2.37

Table B.40: DMGT with 4 machines

Bibliography

[ADR03] Dave Anderson, Jim Dykes, and Erik Riedel. More than an interface—SCSI vs. ATA. In *Proceedings of the 2Nd USENIX Conference on File and Storage Technologies*, FAST '03, pages 245–257, Berkeley, CA, USA, 2003. USENIX Association.

[AKM13] Shaikh Arifuzzaman, Maleq Khan, and Madhav Marathe. PATRIC: A parallel algorithm for counting triangles in massive networks. In *Proceedings of the 22Nd ACM International Conference on Conference on Information & Knowledge Management*, CIKM '13, pages 529–538, New York, NY, USA, 2013. ACM.

[APW+08] Nitin Agrawal, Vijayan Prabhakaran, Ted Wobber, John D. Davis, Mark Manasse, and Rina Panigrahy. Design tradeoffs for SSD performance. In *USENIX 2008 Annual Technical Conference on Annual Technical Conference*, ATC'08, pages 57–70, Berkeley, CA, USA, 2008. USENIX Association.

[AV88] Alok Aggarwal and S. Vitter, Jeffrey. The input/output complexity of sorting and related problems. *Commun. ACM*, 31(9):1116–1127, September 1988.

[AYZ97] N. Alon, R. Yuster, and U. Zwick. Finding and counting given length cycles. *Algorithmica*, 17(3):209–223, 1997.

[BBCG08] Luca Becchetti, Paolo Boldi, Carlos Castillo, and Aristides Gionis. Efficient semi-streaming algorithms for local triangle counting in massive graphs. In *Proceedings of the 14th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '08, pages 16–24, New York, NY, USA, 2008. ACM.

[BFL+06] Luciana S. Buriol, Gereon Frahling, Stefano Leonardi, Alberto Marchetti-Spaccamela, and Christian Sohler. Counting triangles in data streams. In *Proceedings of the Twenty-fifth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '06, pages 253–262, New York, NY, USA, 2006. ACM.

[BFN+14] Jonathan W. Berry, Luke K. Fostvedt, Daniel J. Nordman, Cynthia A. Phillips, C. Seshadhri, and Alyson G. Wilson. Why do simple algorithms for triangle enumeration work in the real world? In *Proceedings of the 5th Conference on Innovations in Theoretical Computer Science*, ITCS '14, pages 225–234, New York, NY, USA, 2014. ACM.

[BOV13] Vladimir Braverman, Rafail Ostrovsky, and Dan Vilenchik. How hard is counting triangles in the streaming model? In FedorV. Fomin, Rsi Freivalds, Marta Kwiatkowska, and David Peleg, editors, *Automata, Languages, and Programming*, volume 7965 of *Lecture Notes in Computer Science*, pages 244–254. Springer Berlin Heidelberg, 2013.

[BPVWZ14] Andreas Björklund, Rasmus Pagh, Virginia Vassilevska-Williams, and Uri Zwick. Listing triangles. In *Automata, Languages, and Programming - 41st International Colloquium, ICALP 2014, Copenhagen, Denmark, July 8-11, 2014, Proceedings, Part I*, 2014.

[BY04] Ricardo Baeza-Yates. A fast set intersection algorithm for sorted sequences. In SuleymanCenk Sahinalp, S. Muthukrishnan, and Ugur Dogrusoz, editors, *Combinatorial Pattern Matching*, volume 3109 of *Lecture Notes in Computer Science*, pages 400–408. Springer Berlin Heidelberg, 2004.

[BYKS02] Ziv Bar-Yossef, Ravi Kumar, and D. Sivakumar. Reductions in streaming algorithms, with an application to counting triangles in graphs. In *Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '02, pages 623–632, Philadelphia, PA, USA, 2002. Society for Industrial and Applied Mathematics.

[BZ07] V. Batagelj and M. Zavernik. Short cycle connectivity. *Discrete Mathematics*, 307(35):310 – 318, 2007. Algebraic and Topological Methods in Graph Theory.

[CC11] Shumo Chu and James Cheng. Triangle listing in massive networks and its applications. In *Proceedings of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '11, pages 672–680, New York, NY, USA, 2011. ACM.

[CC12] Shumo Chu and James Cheng. Triangle listing in massive networks. *ACM Trans. Knowl. Discov. Data*, 6(4):17:1–17:32, December 2012.

[CDPW92] Jaeyoung Choi, J.J. Dongarra, R. Pozo, and D.W. Walker. ScaLAPACK: a scalable linear algebra library for distributed memory concurrent computers. In *Frontiers of Massively Parallel Computation, 1992., Fourth Symposium on the*, pages 120–127, Oct 1992.

[CFKL95] Pei Cao, Edward W. Felten, Anna R. Karlin, and Kai Li. A study of integrated prefetching and caching strategies. *SIGMETRICS Perform. Eval. Rev.*, 23(1):188–197, May 1995.

[CGG+95] Yi-Jen Chiang, Michael T. Goodrich, Edward F. Grove, Roberto Tamassia, Darren Erik Vengroff, and Jeffrey Scott Vitter. External-memory graph algorithms. In *Proceedings of the Sixth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '95, pages 139–149, Philadelphia, PA, USA, 1995. Society for Industrial and Applied Mathematics.

[CLZ11] Feng Chen, Rubao Lee, and Xiaodong Zhang. Essential roles of exploiting internal parallelism of flash memory based solid state drives in high-speed data processing. In *High Performance Computer Architecture (HPCA), 2011 IEEE 17th International Symposium on*, pages 266–277, Feb 2011.

[CN85] Norishige Chiba and Takao Nishizeki. Arboricity and subgraph listing algorithms. *SIAM J. Comput.*, 14(1):210–223, February 1985.

[Coh09] Jonathan Cohen. Graph twiddling in a MapReduce world. *Computing in Science and Engg.*, 11(4):29–41, July 2009.

[Dem06] Roman Dementiev. *Algorithm engineering for large data sets*. PhD thesis, Saarland University, 2006.

[DG04] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. In *Proceedings of the 6th*

Conference on Symposium on Opearting Systems Design & Implementation - Volume 6, OSDI'04, pages 10–10, Berkeley, CA, USA, 2004. USENIX Association.

[FKM+05] Joan Feigenbaum, Sampath Kannan, Andrew McGregor, Siddharth Suri, and Jian Zhang. On graph problems in a semi-streaming model. *Theor. Comput. Sci.*, 348(2):207–216, December 2005.

[GLG+12] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. PowerGraph: Distributed graph-parallel computation on natural graphs. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 17–30, Berkeley, CA, USA, 2012. USENIX Association.

[GSK14] D. Garca-Soriano and K. Kutzkov. Triangle counting in streamed graphs via small vertex covers. In *Proceedings of the 2014 SIAM International Conference on Data Mining*, pages 352–360, 2014.

[HLP+13] Wook-Shin Han, Sangyeon Lee, Kyungyeol Park, Jeong-Hoon Lee, Min-Soo Kim, Jinha Kim, and Hwanjo Yu. TurboGraph: A fast parallel graph engine handling billion-scale graphs in a single PC. In *Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '13, pages 77–85, New York, NY, USA, 2013. ACM.

[HRR99] Monika R. Henzinger, Prabhakar Raghavan, and Sridhar Rajagopalan. Computing on data streams. *Dimacs Series In Discrete Mathematics And Theoretical Computer Science*, pages 107–118, 1999.

[HTCa] Xiaocheng Hu, Yufei Tao, and Chin-Wan Chung. MGT manual. `http://appsrv.cse.cuhk.edu.hk/~taoyf/paper/codes/trilist/manual`. Published: 2013. Accessed: 2014-06-03.

[HTCb] Xiaocheng Hu, Yufei Tao, and Chin-Wan Chung. MGT program. `http://appsrv.cse.cuhk.edu.hk/~taoyf/paper/codes/trilist/trilist.zip`. Published: 2013. Accessed: 2014-06-03.

[HTC13] Xiaocheng Hu, Yufei Tao, and Chin-Wan Chung. Massive graph triangulation. In *Proceedings of the 2013 ACM SIGMOD In-*

ternational Conference on Management of Data, SIGMOD '13, pages 325–336, New York, NY, USA, 2013. ACM.

[IR77] Alon Itai and Michael Rodeh. Finding a minimum circuit in a graph. In *Proceedings of the Ninth Annual ACM Symposium on Theory of Computing*, STOC '77, pages 1–10, New York, NY, USA, 1977. ACM.

[JSP12] Madhav Jha, C. Seshadhri, and Ali Pinar. From the birthday paradox to a practical sublinear space streaming algorithm for triangle counting. *CoRR*, abs/1212.2264, 2012.

[KBG12] Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. GraphChi: Large-scale graph computation on just a PC. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 31–46, Berkeley, CA, USA, 2012. USENIX Association.

[KKM95] T. Kloks, D. Kratsch, and H. Mller. Finding and counting small induced subgraphs efficiently. In Manfred Nagl, editor, *Graph-Theoretic Concepts in Computer Science*, volume 1017 of *Lecture Notes in Computer Science*, pages 14–23. Springer Berlin Heidelberg, 1995.

[KLPM10] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. What is Twitter, a social network or a news media? In *WWW '10: Proceedings of the 19th international conference on World wide web*, pages 591–600, New York, NY, USA, 2010. ACM.

[KM03] Irit Katriel and Ulrich Meyer. Elementary graph algorithms in external memory. In Ulrich Meyer, Peter Sanders, and Jop Sibeyn, editors, *Algorithms for Memory Hierarchies*, volume 2625 of *Lecture Notes in Computer Science*, pages 62–84. Springer Berlin Heidelberg, 2003.

[KMPT10] MihailN. Kolountzakis, GaryL. Miller, Richard Peng, and CharalamposE. Tsourakakis. Efficient triangle counting in large graphs via degree-based vertex partitioning. In Ravi Kumar and Dandapani Sivakumar, editors, *Algorithms and Models for the Web-Graph*, volume 6516 of *Lecture Notes in Computer Science*, pages 15–24. Springer Berlin Heidelberg, 2010.

[KP14] Konstantin Kutzkov and Rasmus Pagh. Triangle counting in dynamic graph streams. *CoRR*, abs/1404.4696, 2014.

[KPP+13] Tamara G. Kolda, Ali Pinar, Todd Plantenga, C. Seshadhri, and Christine Task. Counting triangles in massive graphs with MapReduce. *CoRR*, abs/1301.5887, 2013.

[KSJ+12] Jaehong Kim, Sangwon Seo, Dawoon Jung, Jin-Soo Kim, and Jaehyuk Huh. Parameter-aware I/O management for solid state disks (SSDs). *Computers, IEEE Transactions on*, 61(5):636–649, May 2012.

[Lat08] Matthieu Latapy. Main-memory triangle computations for very large (sparse (power-law)) graphs. *Theor. Comput. Sci.*, 407(1-3):458–473, November 2008.

[Les] Jure Leskovec. SNAP: Stanford large network dataset collection. `http://snap.stanford.edu/data/`. Accessed: 2014-06-03.

[LKF05] Jure Leskovec, Jon Kleinberg, and Christos Faloutsos. Graphs over time: Densification laws, shrinking diameters and possible explanations. In *Proceedings of the Eleventh ACM SIGKDD International Conference on Knowledge Discovery in Data Mining*, KDD '05, pages 177–187, New York, NY, USA, 2005. ACM.

[LLDM08] Jure Leskovec, Kevin J. Lang, Anirban Dasgupta, and Michael W. Mahoney. Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters. *CoRR*, abs/0810.1355, 2008.

[Loc] Glenn K. Lockwood. High-performance virtualization: Sr-iov and infiniband. `http://glennklockwood.blogspot.co.uk/2013_12_01_archive.html`. Published: 2013-12-14. Accessed: 2014-06-03.

[Low13] Yucheng Low. *GraphLab: A Distributed Abstraction for Large Scale Machine Learning*. PhD thesis, University of California, 2013.

[McG13] Andrew McGregor. Graph stream algorithms: A survey, 2013. `http://people.cs.umass.edu/~mcgregor/papers/13-graphsurvey.pdf`.

[Men10] Bruno Menegola. An external memory algorithm for listing triangles. Technical report, Universidade Federal do Rio Grande do Sul, 2010.

[MSOI+02] 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, October 2002.

[NL91] Bill Nitzberg and Virginia Lo. Distributed shared memory: A survey of issues and algorithms. *Computer*, 24(8):52–60, August 1991.

[NTD+09] Dushyanth Narayanan, Eno Thereska, Austin Donnelly, Sameh Elnikety, and Antony Rowstron. Migrating server storage to SSDs: Analysis of tradeoffs. In *Proceedings of the 4th ACM European Conference on Computer Systems*, EuroSys '09, pages 145–158, New York, NY, USA, 2009. ACM.

[OB14] M. Ortmann and U. Brandes. Triangle listing algorithms: Back from the diversion. In *2014 Proceedings of the Sixteenth Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 1–8, 2014.

[OP09] Tore Opsahl and Pietro Panzarasa. Clustering in weighted networks. *Social Networks*, 31(2):155 – 163, 2009.

[PC13] Ha-Myung Park and Chin-Wan Chung. An efficient MapReduce algorithm for counting triangles in a very large graph. In *Proceedings of the 22Nd ACM International Conference on Conference on Information & Knowledge Management*, CIKM '13, pages 539–548, New York, NY, USA, 2013. ACM.

[PGA10] Roger Pearce, Maya Gokhale, and Nancy M. Amato. Multithreaded asynchronous graph traversal for in-memory and semi-external memory. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pages 1–11, Washington, DC, USA, 2010. IEEE Computer Society.

[PS13] Rasmus Pagh and Francesco Silvestri. The input/output complexity of triangle enumeration. *CoRR*, abs/1312.0723, 2013.

[PT12] Rasmus Pagh and Charalampos E. Tsourakakis. Colorful triangle counting and a MapReduce implementation. *Inf. Process. Lett.*, 112(7):277–281, March 2012.

[QCK+12] Zhengping Qian, Xiuwei Chen, Nanxi Kang, Mingcheng Chen, Yuan Yu, Thomas Moscibroda, and Zheng Zhang. Madlinq: Large-scale distributed matrix computation for the cloud. In

Proceedings of the 7th ACM European Conference on Computer Systems, EuroSys '12, pages 197–210, New York, NY, USA, 2012. ACM.

[RMZ13] Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. X-Stream: Edge-centric graph processing using streaming partitions. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 472–488, New York, NY, USA, 2013. ACM.

[RW94] Chris Ruemmler and John Wilkes. An introduction to disk drive modeling. *Computer*, 27(3):17–28, March 1994.

[RWE13] Ian Robinson, Jim Webber, and Emil Eifrem. *Graph Databases*. O'Reilly, Beijing, 2013.

[Sch07] Thomas Schank. *Algorithmic Aspects of Triangle-Based Network Analysis*. PhD thesis, Universität Karlsruhe, 2007.

[SPK13] C. Seshadhri, Ali Pinar, and Tamara G. Kolda. Wedge sampling for computing clustering coefficients and triangle counts on large graphs. *CoRR*, abs/1309.3321, 2013.

[Sup] Microsoft Support. Default cluster size for NTFS, FAT, and exFAT. `https://support.microsoft.com/kb/140365`. Published: 2013-07-12. Accessed: 2014-06-03.

[Suv] Davy Suvee. Counting triangles smarter (or how to beat big data vendors at their own game). `http://datablend.be/?p=282`. Published: 2013-02-11. Accessed: 2014-06-03.

[SV11] Siddharth Suri and Sergei Vassilvitskii. Counting triangles and the curse of the last reducer. In *Proceedings of the 20th International Conference on World Wide Web*, WWW '11, pages 607–614, New York, NY, USA, 2011. ACM.

[SVP08] Daniele Paolo Scarpazza, Oreste Villa, and Fabrizio Petrini. Efficient breadth-first search on the Cell/BE processor. *IEEE Trans. Parallel Distrib. Syst.*, 19(10):1381–1395, October 2008.

[SW05] Thomas Schank and Dorothea Wagner. Finding, counting and listing all triangles in large graphs, an experimental study. In *Proceedings of the 4th International Conference on Experimental and Efficient Algorithms*, WEA'05, pages 606–609, Berlin, Heidelberg, 2005. Springer-Verlag.

[TKMF09] Charalampos E. Tsourakakis, U. Kang, Gary L. Miller, and Christos Faloutsos. DOULION: Counting triangles in massive graphs with a coin. In *Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '09, pages 837–846, New York, NY, USA, 2009. ACM.

[TMS11] Animesh Trivedi, Bernard Metzler, and Patrick Stuedi. A case for RDMA in clouds: Turning supercomputer networking into commodity. In *Proceedings of the Second Asia-Pacific Workshop on Systems*, APSys '11, pages 17:1–17:5, New York, NY, USA, 2011. ACM.

[TPT13] Kanat Tangwongsan, A. Pavan, and Srikanta Tirthapura. Parallel triangle counting in massive streaming graphs. In *Proceedings of the 22Nd ACM International Conference on Conference on Information & Knowledge Management*, CIKM '13, pages 781–786, New York, NY, USA, 2013. ACM.

[Tso08] Charalampos E. Tsourakakis. Fast counting of triangles in large real networks without counting: Algorithms and laws. In *Proceedings of the 2008 Eighth IEEE International Conference on Data Mining*, ICDM '08, pages 608–617, Washington, DC, USA, 2008. IEEE Computer Society.

[UKBM11] Johan Ugander, Brian Karrer, Lars Backstrom, and Cameron Marlow. The anatomy of the Facebook social graph. *CoRR*, abs/1111.4503, 2011.

[VS94] J.S. Vitter and E.A.M. Shriver. Algorithms for parallel memory, I: Two-level memories. *Algorithmica*, 12(2-3):110–147, 1994.

[Wal] Stephen Walkauskas. Counting triangles. `http://www.vertica.com/2011/09/21/counting-triangles/`. Published: 2011-09-21. Accessed: 2014-06-03.

[WC12] Jia Wang and James Cheng. Truss decomposition in massive networks. *Proc. VLDB Endow.*, 5(9):812–823, May 2012.

[Wil11] Virginia Vassilevska Williams. Breaking the Coppersmith-Winograd barrier. unpublished manuscript, 2011. `theory.stanford.edu/~virgi/cv-vvw.ps`.

[WS98] D.J. Watts and S.H. Strogatz. Collective dynamics of 'small-world' networks. *Nature*, (393):440–442, 1998.

[WZTT10] Nan Wang, Jingbo Zhang, Kian-Lee Tan, and Anthony K. H. Tung. On triangulation-based dense neighborhood graph discovery. *Proc. VLDB Endow.*, 4(2):58–68, November 2010.

[XGFS13] Reynold S. Xin, Joseph E. Gonzalez, Michael J. Franklin, and Ion Stoica. GraphX: A resilient distributed graph system on Spark. In *First International Workshop on Graph Data Management Experiences and Systems*, GRADES '13, pages 2:1–2:6, New York, NY, USA, 2013. ACM.

[YCH+05] Andy Yoo, Edmond Chow, Keith Henderson, William McLendon, Bruce Hendrickson, and Umit Catalyurek. A scalable distributed parallel breadth-first search algorithm on BlueGene/L. In *Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*, SC '05, pages 25–, Washington, DC, USA, 2005. IEEE Computer Society.

[YL12] Jaewon Yang and Jure Leskovec. Defining and evaluating network communities based on ground-truth. In *Proceedings of the ACM SIGKDD Workshop on Mining Data Semantics*, MDS '12, pages 3:1–3:8, New York, NY, USA, 2012. ACM.

[YNDR14] Eiko Yoneki, Karthik Nilakant, Valentin Dalibard, and Amitabha Roy. PrefEdge: SSD prefetcher for large-scale graph traversal. In *Proceedings of the 7th International Systems and Storage Conference*, SYSTOR '14, 2014.

[YWW+14] Zhi Yang, Christo Wilson, Xiao Wang, Tingting Gao, Ben Y. Zhao, and Yafei Dai. Uncovering social network sybils in the wild. *ACM Trans. Knowl. Discov. Data*, 8(1):2:1–2:29, February 2014.

[Zeh02] Norbert Ralf Zeh. *I/O-Efficient Algorithms for Shortest Path Related Problems.* PhD thesis, Carleton University, 2002.