Triangle Counting and Listing in Directed and Undirected Graphs
Using Single Machines

by

Yudi Santoso
B.Sc. (Physics), Gadjah Mada University, Indonesia, 1993
M.Sc. (Physics), Bandung Institute of Technology, Indonesia, 1996
Ph.D. (Physics), Texas A&M University, Texas, USA, 2001

A Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

in the Department of Computer Science

Triangle Counting and Listing in Directed and Undirected Graphs
Using Single Machines

by

Yudi Santoso
B.Sc. (Physics), Gadjah Mada University, Indonesia, 1993
M.Sc. (Physics), Bandung Institute of Technology, Indonesia, 1996
Ph.D. (Physics), Texas A&M University, Texas, USA, 2001

Supervisory Committee

Dr. A. Thomo, Supervisor
(Department of Computer Science)

Dr. V. Srinivasan, Co-supervisor
(Department of Computer Science)

Supervisory Committee

Dr. A. Thomo, Supervisor
(Department of Computer Science)

Dr. V. Srinivasan, Co-supervisor
(Department of Computer Science)

ABSTRACT

Triangle enumeration is an important element in graph analysis, and because of this it is a topic that has been studied extensively. Although the formulation is simple, for large networks the computation becomes challenging as we have to deal with memory limitation and efficiency. Many algorithms have been proposed to overcome these problems. Some use distributed computing, where the computation is distributed among many machines in a cluster. However, this approach has a high cost in terms of hardware resources and energy. In this thesis we studied triangle counting/listing algorithms for both directed and undirected graphs, and searched for methods to do the computation on a single machine. Through detailed analysis, we found some ways to improve the efficiency of the computation. Programs that implement the algorithms were built and tested on large networks with up to almost a billion nodes. The results were then analysed and discussed.

Contents

List of Tables

List of Figures

ACKNOWLEDGEMENTS

I would like to thank:

my family, for sharing an extraordinary journey, support, and faith.

Dr. Thomo, for giving me the opportunity to pursue this degree, and for mentoring and encouraging me throughout my study.

Dr. Srinivasan, for sharing his knowledge, mentoring, and patience.

Pooja Bhojwani, for collaboration on parts of this project, and inspiration.

University of Victoria, for funding me with a scholarship.

To become good at anything you have to know how to apply basic principles. To be great at it, you have to know when to violate those principles.
Gary Kasparov

DEDICATION

For Julie, Sarah and Lucas.

Chapter 1

Introduction

A network is a set of objects with relations or connections among them. Many systems can be viewed as a network. For example: a set of computers that are linked to each other (either wired or wireless) form a computer network; cities and roads connecting the cities form an intercity road network; molecules with physical interactions among them form a molecular network; people in a community form a social network; species in an ecosystem form a biological network; consumers, producers, distributors, and financial institutions form an economic network; and many more.

Mathematically a network is represented by a graph of nodes and edges. The nodes represent the objects, and the edges represent the relations. Two nodes are connected by an edge if they have a relation. The edges can be directed if the relations have directions, in which case we have a directed graph, or undirected if the relations are both ways, in which case the graph is undirected.

A triangle is a graph of three nodes which are pair-wise connected by edges. Given a graph, there can be triangle subgraphs inside the graph. Some graph problems require enumeration of these triangles in their solution. In this thesis, we study the computation problem of triangle counting and listing using single machines.

1.1 Motivation

The rise of social media such as Facebook [1], Twitter [2] and Instagram [3] led to the birth and growth of virtual social networks over the Internet. The relatively recent explosions of these social networks, and the Internet itself, have triggered much interests in the research on large networks.

Given a network, we would like to know about its features and underlying structures. Often, we would like to know about clusterings inside the network [11]. A cluster, or a community - following the jargon of social networks, is a set of nodes that are related more closely among themselves than to the nodes outside of the cluster. In other words, the edge density inside a cluster is significantly higher than the edge density of the whole network. There are intensive studies on how we can identify communities in a network. One indicator of a community is the existence of triangles [25].

Triangle is a primitive that is related to some characteristics of a graph, such as its connectivity [1], clustering coefficient [35], and transitivity [19]. Triangle enumeration is also used in truss decomposition [34].

We are interested in the computation problem of triangle counting and enumeration (or listing). The counting itself is not difficult. However, when the input graph is large, the computation becomes more challenging. We need to deal with limitations on memory space and disk space, and to keep the running time feasible.

There are several approaches to solve these problems. One solution is through distributed computing using, e.g., Hadoop + Map Reduce or Sparks [21, 36]. This can be done either on a local cluster or through cloud. The drawback of this solution is that it requires large investment on the infrastructure and has a high running

[1] https://www.facebook.com/
[2] https://twitter.com/
[3] https://www.instagram.com/

cost. Also, there is overhead computational cost due to the network communication and distribution that adds up to the running time. This overhead can be quite significant [17].

Another approach, which we are looking here, is to keep using a single machine. Disk space problem can be alleviated by file compression. Memory problem can be overcome by loading the data one part at a time. The WebGraph framework [6, 7] provides utilities to do both. The running time can still be challenging with this approach.

On the other hand, multicore processor has become the norm for modern computers, and the technology trend seems to say that there will be processors with even more cores in the future. Thus, we are looking at the possibilities of doing parallel computation, utilizing the multicores to speed up the running.

In this thesis we study algorithms that are suitable for this method. We explore ways of optimizing the computation. For the implementation we use Java 8. It has `parallelStream` method which is suitable for our purpose. We would like to see how much we can push the performance, and how it compares to the performance of a cluster.

The motivation for doing this is for saving cost and availability. A bit less performance can sometimes be justified if the cost is less. Not everybody has access to a computer cluster, but most institutions/companies can afford a PC or an entry level server. In addition, from the developer's point of view, debugging on a single machine is much easier than on a cluster or a cloud, hence an advantage. Thus, it is worth exploring how much can be done with a single machine.

While there are many algorithms available in the literature for triangle enumeration, they deal with undirected triangles in undirected graphs. We extend our study further by looking also into directed triangle cases: cycles and trusts.

1.2 Contributions

The contributions of this thesis are:

1. Based on previous work, we designed efficient algorithms for triangle counting and listing for both directed and undirected graphs. This includes a preprocessing algorithm for undirected graphs.

2. We provided detail analyses on the algorithms, including proof of correctness and running time analyses.

3. We built implementation of these algorithms using Java 8 and WebGraph. The iteration is parallelized using the Java 8 parallel stream method.

4. We conducted triangle counting experiments on several large networks. Directed graphs were symmetrized so that we could do both directed and undirected triangle countings.

5. We analysed the results and gave insight on how to improve the computation further.

1.3 Organization

This thesis is organized as follows:

Chapter 2 Graphs and Triangles

presents the notation and definitions on graphs and triangles that we use.

Chapter 3 Algorithms and Implementations

details each of the algorithms considered and gives algorithmic analyses.

Chapter 4 Experiments

presents our experimental settings and results.

Chapter 5 Discussions

offers discussions on our results, related work and future work.

Chapter 6 Conclusions

concludes the thesis.

Chapter 2

Graphs and Triangles

This chapter lays out the necessary background and definitions on graphs and triangles.

2.1 Graphs

Definition 1 (Graph)**.** *A graph, $G(V, E)$, is a composite mathematical object which consists of a set of nodes V and a set of edges E. Each edge in E connects a pair of nodes in V.*

A node (also known as a vertex) represents an object, and an edge represents a link, a connection, or a relationship between two objects. Pictorially, a node is drawn as a point, and an edge is drawn as a line connecting two nodes. In general, there could also be an edge from a node to itself, this is called a self-loop. If there is more than one edge from one node to another, we have a multi-edge.

If the edges have direction, $\circ\!\!\longrightarrow\!\!\circ$, the graph is said to be directed. A directed graph is also known as a digraph. If the edges have no direction, $\circ\!\!-\!\!\circ$, the graph is said to be undirected. Note that an undirected edge can also be viewed as bi-directed,

$\circ\!\!\leftrightarrow\!\!\circ$, which means that the two end nodes play the same role in the relationship. For example, friendship is mutual, if A is a friend of B, then B is a friend of A.

All the graphs that we analyze here have no multi-edge. Note that for a directed graph, edge $u \to v$ is distinct from edge $v \to u$. So we can have both and do not consider them as a multi-edge. For an undirected graph with no multi-edge, on the other hand, there can be only one edge maximum between any pair of nodes. A graph is simple if it has no multi-edge or self-loop.

In this work we consider both undirected and directed graphs. To make it clear, we denote an undirected graph by $\widetilde{G}$, and a directed graph by G (without the tilde). The notation that we use are summarized in Table 2.1.

Symbol	Definition
$\widetilde{G}(V, E)$	An undirected graph.
V_X	The set of nodes in graph X.
E_X	The set of edges in graph X.
(u, v)	An edge with end-nodes u and v.
$N(v)$	The set of neighbours of node v.
$d(v)$	The degree of node v, $d(v) = \|N(v)\|$.
(u, v, w)	A triangle with nodes u, v, and w.
Δ	The number of triangles in the graph.
$G(V, E)$	A directed graph.
$G^T(V, E^T)$	The transpose graph of $G(V, E)$, where E^T is the same set of edges as E but with the direction of every edge reversed.
$u \to v$	A directed edge from u to v.
$N^+(v)$	The set of neighbours from node v.
$N^-(v)$	The set of neighbours to node v.
$d^+(v)$	The out-degree of node v, $d^+(v) = \|N^+(v)\|$.
$d^-(v)$	The in-degree of node v, $d^-(v) = \|N^-(v)\|$.
$(u, v, w)_T$	A trust triangle with nodes u, v, and w, connected by edges $u \to v$, $v \to w$, and $u \to w$.
$(u, v, w)_C$	A cycle triangle with nodes u, v, and w, connected by edges $u \to v$, $v \to w$, and $w \to u$.
Δ_T	The number of trust triangles in the graph.
Δ_C	The number of cycle triangles in the graph.

Table 2.1: Notation used in this thesis.

Let u be a node. Any node v that is connected to u by an edge is called a neighbour of u. The number of neighbours of u is called the degree of u, $d(u)$. For directed graphs, we distinguish between in-degree and out-degree. In-degree, $d^-(u)$, is the number of nodes connected to u by edges *to* u, it is equal to the number of edges going into u. Out-degree, $d^+(u)$, is the number of nodes connected to u by edges *from* u, it is equal to the number of edges going out from u.

When we have several graphs in hand we can use different symbols, e.g., H, and use indices for V and E, such as V_H and E_H.

Definition 2 (Subgraph)**.** *$H(V_H, E_H)$ is a subgraph of $G(V_G, E_G)$ if H is a graph and $V_H \subset V_G$ and $E_H \subset E_G$.*

Definition 3 (Transpose Graph)**.** *Given a directed graph G, its transpose G^T is the same graph with all the edges reversed.*

We can symmetrise G to get an undirected graph $\widetilde{G}$ by taking the union between G and its transpose, $G \cup G^T$, and treat each pair of edges as a single undirected edge. This process is illustrated in Fig. 2.1. Note that

$$V(\widetilde{G}) = V(G) = V(G^T) \tag{2.1}$$

and

$$|E(G)| \leq 2|E(\widetilde{G})| \leq 2|E(G)| \tag{2.2}$$

Definition 4 (Complete Graph)**.** *A complete graph is a graph with all possible edges present.*

This definition implies that each node is connected to all other nodes in the graph, and for directed graphs, each connection is a pair of edges.

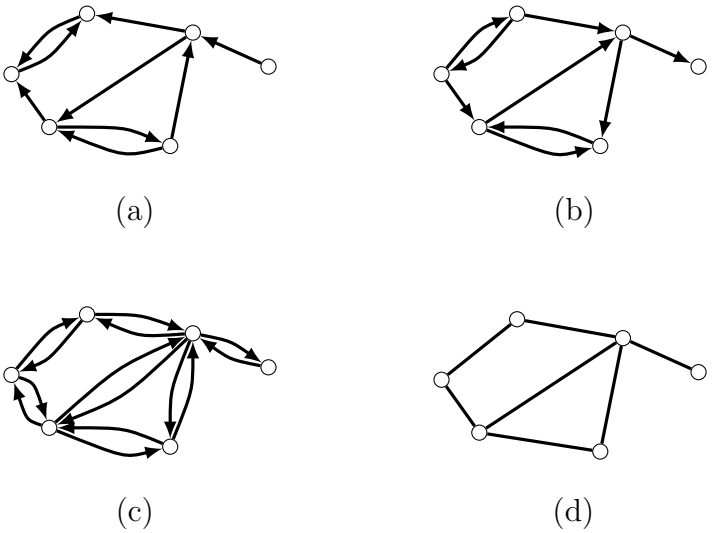

Figure 2.1: The process of symmetrising a directed graph to get an undirected graph. (a) The graph G. (b) The transpose graph G^T. (c) The union graph $G \cup G^T$. (d) The corresponding undirected graph.

2.2 Triangles

We define a triangle as follows.

Definition 5 (Triangle)**.** *A triangle is a graph of three nodes connected to each other by three edges.*

Using this definition, for directed graphs we have two kinds of triangles: trust triangles and cycle triangles. They are depicted by Fig. 2.2 (a) and (b) respectively. Here we use the name trust to reflect that if the relationship is a trust relationship, trust triangle shows propagation of trust. That is, if 1 trusts 2 and 2 trusts 3, then 1 trusts 3. Some authors use the term truss, as in truss bridge, for this type of triangle.

We can represent a triangle by its nodes, e.g., (u, v, w). In the directed case, the ordering matters. For trust triangles we start by the source node and end by the sink node so that for Fig. 2.2(a) we have $(1, 2, 3)_T$. For cycle triangles we follow the flow of the edges. To avoid double counting we start by the node with the smallest label.

For any set of three nodes (or triple), there could be between zero and six trust

(a) (b)

Figure 2.2: (a) A trust triangle, $(1, 2, 3)_T$, and (b) a cycle triangle, $(1, 2, 3)_C$.

triangles, and zero and two cycle triangles. The maximum numbers of trusts and cycles are found in triples with six edges, which is a complete graph, as shown in Fig. 2.3 and Fig. 2.4 respectively.

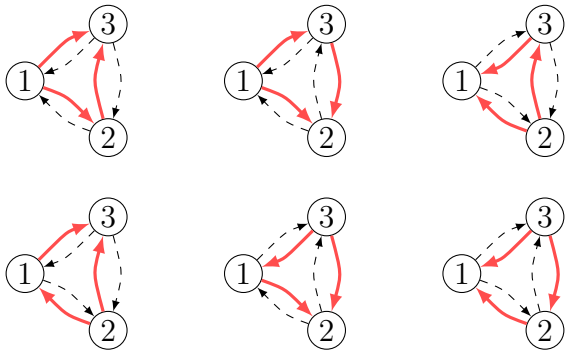

Figure 2.3: There are six trust triangles in a complete graph of three nodes, labeled as $(123)_T$, $(132)_T$, $(231)_T$, $(213)_T$, $(312)_T$, and $(321)_T$ respectively.

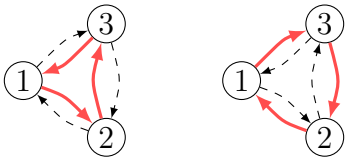

Figure 2.4: There are two cycle triangles in a complete graph of three nodes, labeled as $(123)_C$ and $(132)_C$ respectively.

For undirected graphs, there is only one type of triangles: undirected triangles. This is depicted in Fig. 2.5.

For any set of three nodes in a simple undirected graph there can only be at most one undirected triangle. For instance, $(3, 1, 2)$ is the same triangle as $(1, 2, 3)$. There are six permutations of three labels. In enumerating the triangles, this ordering

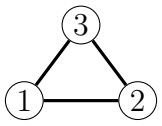

Figure 2.5: A triangle in an undirected graph, $(1, 2, 3)$.

permutation leads to multiple counting/listing that we have to take care of. We do this by requiring that the nodes in the triple are ordered according to the labels. Thus, we list $(1, 2, 3)$ but not $(3, 1, 2)$, $(2, 3, 1)$, $(1, 3, 2)$, $(3, 2, 1)$, or $(2, 1, 3)$.

In Figure 2.1(a), and similarly in Figure 2.1(b), there is one trust triangle and one cycle triangle. In Figure 2.1(c) there are six trust triangles and two cycle triangles. In Figure 2.1(d) there is one undirected triangle. In general, the number of undirected triangles (in the corresponding undirected graph) is constrained by the number of trust triangles and the number of cycle triangles

$$\max(\lceil \Delta_T/6 \rceil, \lceil \Delta_C/2 \rceil) \leq \Delta \leq \Delta_T + \Delta_C \tag{2.3}$$

The minimum value of Δ occurs when the trust and cycle triangles occupy the smallest number of triples, while the maximum occurs when every trust and cycle triangles are on different triples.

Chapter 3

Algorithms and Implementations

In this chapter we list the algorithms that we use, and give the details of the analysis toward them. In the later part, we explain our code implementation. The goal here is to count the number of triangles in a graph, and possibly also enumerate the triangles. For the directed case, the input graph contains the outgoing adjacency list and the out-degrees. We also use the corresponding transpose graph, which can be viewed as the ingoing adjacency list and the in-degrees of the original graph. The nodes in these graphs are sorted according to the labels. The adjacency lists are also sorted in the same way.

3.1 Triangles in Directed Graphs

As explained in Chapter 2, there are two types of triangles in a directed graph: trust triangles and cycle triangles. Given a directed graph (and its transpose), our goal is to count the number of triangles of each type.

3.1.1 Directed Triangles Algorithms

The algorithm that we use to count the number of trust triangles is listed in Algorithm 1. We only need the directed graph as input for this algorithm, without the transpose graph. However, note that this algorithm can also be applied to the transpose graph, without the original graph.

First, we initiate a counter for the number of the trust triangles, c, and set it to zero. We start by looking at the nodes one by one, and for each node u we iterate its neighbours, $v \in N^+(u)$. In effect, this yields iteration over the edges $u \to v$ in the graph, hence our algorithm is categorized as an edge iteration algorithm.

Once we get an edge, we try to find out if the two end points of the edge, u and v, are connected to a common neighbour, $w \in N^+(u)$ and $w \in N^+(v)$. Note that when we get a w, (u, v, w) forms a trust triangle, with edges $u \to v$, $v \to w$ and $u \to w$. Each time we find a trust triangle, we increment the counter c by one. Here, for each edge, $u \to v$, we use a temporary counter k which is then added into c. When we are done iterating over all the edges, c would give the total number of the trust triangle in the graph.

Note that we do not put any restriction on the order of the labels u, v, and w. Thus, u can be greater or smaller than v and/or w, and also v can be greater or smaller than w. Here, we assume that the graph is simple, with no multi-edge or self-loop, so $u \neq v \neq w$.

Note also that we can easily do listing/enumerating of the triangles as well by including the lines marked "For Listing". For each edge $u \to v$, S is the intersection between the neighbours of u and the neighbours of v, i.e., $S = N^+(u) \cap N^+(v)$. We can print the results onto a display or a file.

The algorithm above involves computing the intersection between two neighbour sets. The algorithm for this is given in Algorithm 2. For this algorithm the input are

Algorithm 1 Trust Triangle Counting

Input: A directed graph $G = (V, E)$
Output: The number of trust triangles in G, (optional) list of the triangles.

```
S ← ∅, k ← 0, c ← 0
for all nodes u ∈ V do
    for all nodes v ∈ N+(u) do
        (S, k) ← Intersection(N+(u), N+(v), d+(u), d+(v))
        for all nodes w ∈ S do                          ▷ For Listing
            print (u, v, w)                             ▷ For Listing
        c ← c + k
return c
```

two sorted sets A and B, as arrays, and their sizes a and b. We initiate an empty set C to contain the intersection between A and B. We use two pointers, i on A, and j on B; and a counter k to count the elements of C. We start by setting i, j, and k to zero. While i and j are not equal or greater than the size of their respective arrays, we compare $A[i]$ to $B[j]$. If they are the same, there is a match, we put the element into C, and then we increase all i, j, k by one. If $A[i] < B[j]$ we go to the next element of A by incrementing i by one. Else, $A[i]$ must be greater than $B[j]$, then we increase j by one to go to the next element of B. When $i = a$ or $j = b$ it means that we have looked over all elements of A or B respectively, and no more match is possible. Then, we return the intersection set C and the number of matches k.

The algorithm for counting cycle triangles is very similar to that for counting trust triangles. It is listed in Algorithm 3. In this case, it requires as input both the graph and its transpose. The in-going neighbour set of u is read from the out-going neighbour set of u in the transpose graph, i.e., $N^-(u, G) = N^+(u, G^T)$. Notice that in Intersection we use $N^-(u)$ as the parameter. In Cycle Triangle Counting we impose the condition that $u < v$ and $u < w$ to avoid double counting.

Algorithm 2 Intersection

Input: Two sorted sets of integers A and B, a = size of A, b = size of B
Output: The set $A \cap B$, the size of the intersection.

```
C ← ∅
i ← 0, j ← 0, k ← 0
while i < a and j < b do
    if A[i] == B[j] then
        C[k] ← A[i]
        k ← k + 1, i ← i + 1, j ← j + 1
    else if A[i] < B[j] then
        i ← i + 1
    else                                          ▷ A[i] > B[j]
        j ← j + 1
return (C, k)
```

Algorithm 3 Cycle Triangle Counting

Input: A directed graph $G = (V, E)$, its transpose $G^T = (V, E^T)$
Output: The number of cycle triangles in G, (optional) list of the triangles.

```
S ← ∅, k ← 0, c ← 0
for all nodes u ∈ V do
    for all nodes v ∈ N⁺(u) do
        if u < v then
            (S, k) ← Intersection(N⁻(u), N⁺(v), d⁻(u), d⁺(v))
            for all nodes w ∈ S do
                if u < w then
                    print (u, v, w)                ▷ For Listing
                    c ← c + 1
return c
```

3.1.2 Analysis of Directed Graph Algorithms

Now, we give a proof of the correctness of the algorithms.

Theorem 1. *The Trust Triangle Counting algorithm (Algorithm 1) correctly enumerates the trust triangles in a simple directed graph.*

Proof. The algorithm iterates over each node once, and for each node its outgoing neighbours are checked once. Thus, it goes through every edges in the graph, and each edge is checked once. Any trust triangle with two first nodes match the two

end nodes of an edge would be found by the intersection computation. Without any ordering condition on the nodes, all trust triangles found would be enumerated. □

Theorem 2. *The Cycle Triangle Counting algorithm (Algorithm 3) correctly enumerates the cycle triangles in a simple directed graph.*

Proof. A cycle triangle $(u, v, w)_C$ has three nodes u, v, w, connected by edges $u \to v$ (first edge), $v \to w$ (second edge), and $w \to u$ (third edge). When we reverse the third edge of a cycle triangle we get a trust triangle, and vice versa. Consequently, we can search for cycle triangles by using the same Intersection algorithm as in Trust Triangle Counting provided that we use in-going edge instead of out-going edge for the third edge. The in-going edges can be read from the transpose graph. Since all cycle triangles will appear like a trust triangle by this procedure, and by Theorem 1 we have shown that all trust triangles would be found, all cycle triangles would be found.

Now, notice that $(u, v, w)_C$, $(v, w, u)_C$ and $(w, u, v)_C$ are identical cycle triangles. We can avoid double counting by anchoring the first node. We do this by imposing the condition that the first node must be smaller than the other two in the triple, $u < v$ and $u < w$. Note that we do not impose any condition on the order of v and w, because $(u, v, w)_C$ and $(u, w, v)_C$ are distinct cycles and hence should be counted separately. Thus, Algorithm 3 would find all cycle triangles and enumerate each of them once. □

Next we analyse the running times. Let us first look at the Trust Triangle Counting algorithm.

Theorem 3. *The running time of the Trust Triangle Counting on graph $G(V, E)$ is bounded by $O(|V|(d^+_{\max}(G))^2)$.*

Proof. Recall that the algorithm iterates over the edges in the graph by going through the nodes and for each node its adjacent nodes. For each edge $u \to v$ the intersection computation runs in time $d^+(u)+d^+(v)$. Thus, the running time for the trust triangle counting is $\sum_{u\to v}(d^+(u) + d^+(v)) \leq \sum_u d^+(u)(d^+(u) + \max[d^+(v \in N^+(u))])$. The worst case is $2|V|(d^+_{\max}(G))^2$. □

Note that the actual running time should be less than this bound, unless the degrees are all the same and equal to $d^+_{\max}(G)$.

Theorem 4. *The running time of the Cycle Triangle Counting on graph $G(V, E)$ is bounded by $O(|V|d^+_{\max}(G)(\max[d^+_{\max}(G), d^-_{\max}(G)]))$.*

Proof. The proof is similar to the one for the trust above. The difference is that for each edge $u \to v$ here, the running time for the intersection computation is $d^-(u) + d^+(v)$. Therefore, the cycle triangle counting running time is $\sum_{u\to v}(d^-(u)+d^+(v)) \leq \sum_u d^+(u)(d^-(u) + \max[d^+(v \in N^+(u))])$. The worst case is when all the degrees are equal to the maximum, yielding running time $O(|V|d^+_{\max}(G)(\max[d^+_{\max}(G), d^-_{\max}(G)]))$. □

In practice, if the nodes with highest degrees are joined only to nodes with small degrees, i.e., $d^+(v \in N^+(u))$ are small, the bound would be $O(|V|d^+_{\max}(G)d^-_{\max}(G)])$.

At this point, we get an interesting hypothesis. First, observe that we can use either the graph or the transpose graph and get the same result.

Theorem 5. *The triangles in the transpose graph correspond one to one to the triangles in the graph. Therefore, the number of triangles in the transpose graph is equal to the number of triangles in the graph.*

Proof. By definition, transpose graph is the same graph as the original graph with the edges reversed. Therefore, if there is a triangle in the graph, there is also a triangle in the transpose graph with opposite orientation. □

Now, suppose that we have a graph where $d^+_{\max}$ is much larger than $d^-_{\max}$, then we can run the Trust Triangle Counting on the transpose graph instead, and get the answer in a shorter time. On the other hand, if $d^+_{\max}$ is much smaller than $d^-_{\max}$, we should run the Trust Triangle Counting on the graph. For the Cycle Triangle Counting, on the other hand, reversing the role between the graph and the transpose graph does not guarantee to give clear advantage in terms of the running time.

3.2 Triangles in Undirected Graphs

Next, we turn to the undirected case.

3.2.1 Undirected Triangles Algorithms

The algorithm that we use for undirected triangle counting is listed in Algorithm 4. The idea here is that an undirected edge can be viewed as a pair of opposite directed edges. Thus, an undirected triangle would have six directed edges among the nodes, and contains six trust triangles as depicted in Figure 2.3. Thus, any undirected triangle can be found by using the edge iteration algorithm same as for the Trust Triangle Counting. The only issue here is that we want to count each triangle once, instead of six times.

To avoid this multiple counting we impose a condition on the ordering of the nodes, $u < v < w$. This condition leads to an interesting consequence. Since $u < v$ and $u < w$, only the larger neighbours of u need to be considered. Similarly, since $v < w$, only the larger neighbours of v need to be considered. As a result, we can actually cut the adjacency list to only larger neighbours before the triangle counting and we obtain the same result. Since the degrees are reduced, the number of iteration is reduced, and the running time is shorter.

Algorithm 4 Undirected Triangle Counting

Input: An undirected graph $\widetilde{G} = (V, E)$
Output: The number of triangles in $\widetilde{G}$, (optional) list of the triangles.

```
S ← ∅, k ← 0, c ← 0
for all nodes u ∈ V do
    for all nodes v ∈ N(u) do
        if u < v then
            (S, k) ← Intersection(N(u), N(v), d(u), d(v))
            for all nodes w ∈ S do
                if v < w then
                    print (u, v, w)                    ▷ For Listing
                    c ← c + 1
return c
```

We can take this idea further by first ordering the nodes according to the degrees and relabel them accordingly. Then, the node with the highest degree would be labeled the highest, which after omitting the smaller neighbours would get zero neighbour. This preprocessing reduces the maximum effective degree of the graph. Following this idea, we preprocess the graph before doing the triangle counting. The algorithm for this preprocessing is listed in Algorithm 5.

First we initialize three arrays of size $n = |V|$: idx, vtx and deg. We put the labels of the nodes into idx, and the degrees into deg. Then, we sort idx based on the deg ascendingly. After sorting, we relabel the nodes and put the translation between the old and new into vtx. Then, we use vtx to translate the adjacency list. Then, we sort the list according to the new labels.

We do not need to implement the Sort function as there are well known sorting algorithms, and most modern programming languages - Java included - have a Sort function in their standard library that we can use.

Algorithm 5 Graph Sort and Cut

Input: The adjacency list $\{N(v)\}$ and degree list $\{d(v)\}$ of an undirected graph $\widetilde{G}(V, E)$, $n = |V|$.

Output: A sorted adjacency list that contains only bigger neighbours in the new labels(The nodes are relabelled after sorted.).

initialize idx, vtx, deg arrays of size n
for all $v \in \{0, \ldots, n-1\}$ **do**
  $idx[v] \leftarrow v$
  $deg[v] \leftarrow d(v)$
Sort (idx on ascending deg) ▷ Use a sort utility
for all $i \in \{0, \ldots, n-1\}$ **do**
  $vtx[idx[i]] \leftarrow i$ ▷ New label
for all $v \in \{0, \ldots, n-1\}$ **do**
  $d_1 \leftarrow d(idx[v])$
  $S_1 \leftarrow N(idx[v])$
  $d_2 \leftarrow 0$
  $S_2 \leftarrow \emptyset$
  for $i = 0$, $i < d_1 - 1$, $i{+}{+}$ **do**
    if $v < vtx[S_1[i]]$ **then** ▷ Filter out smaller neighbours
      S_2.append($vtx[S_1[i]]$)
      $d_2 \leftarrow d_2 + 1$
  Sort (S_2)
  print (d_2, S_2) ▷ Append to file

3.2.2 Analysis of Undirected Graph Algorithms

Theorem 6. *The Undirected Triangle Counting algorithm (Algorithm 4), applied on an undirected graph that is preprocessed using the Graph Sort and Cut algorithm (Algorithm 5), correctly enumerates the triangles in the input graph.*

Proof. Given an undirected graph, we can view it as a directed graph where each edge is bidirectional. Running the Trust Triangle Algorithm on this graph, a triangle with nodes u, v and w would be found as (u, v, w), (u, w, v), (v, w, u), (v, u, w), (w, u, v), and (w, v, u). That is, six permutations of three indices. One of them would be the ordered permutation, which, without loss of generality, is (u, v, w) with $u < v < w$. Thus, to enumerate the triangles we just need to find this ordered one.

Relabelling the nodes of a graph is equivalent to finding an isomorphic graph. Thus relabelling does not change the enumeration. The listing can be translated back to the old labels if needed.

Now, if an edge (u, v) is in the original graph and $u < v$, then (u, v) would also be in the preprocessed graph. Therefore, running the Triangle Counting on the preprocessed graph, the edge (u, v) would be iterated. Since $w > u$ and $w > v$, w would be in the neighbour lists of u and v, respectively, in the preprocessed graph. Therefore the triangle (u, v, w) would be found. □

The running time of the Undirected Triangle Counting (Algorithm 4) is similar to that of the Trust Triangle Counting (Algorithm 1).

Theorem 7. *The running time of the Undirected Triangle Counting on graph $\widetilde{G}(V, E)$ is bounded by $O(|V|(d_{\max}(G))^2)$.*

Proof. Recall that the algorithm iterates over the edges in the graph. For each edge (u, v) the intersection computation runs in time $d(u) + d(v)$. Thus, the triangle counting running time is $\sum_{u \to v}(d(u) + d(v)) \leq \sum_u d(u)(d(u) + \max[d(v \in N(u))])$. The worst case is $2|V|(d_{\max}(G))^2$. □

Following the same analysis, on the preprocessed graph the running time is bounded by $O(|V|(d^{\text{eff}}_{\max}(G))^2)$. However, note that after preprocessing, the effective degrees cannot all be the same as $d^{\text{eff}}_{\max}$. For one, the highest relabeled node has zero effective degree. All nodes that are relabeled higher than $n - d^{\text{eff}}_{\max}$ must have effective degree less than $d^{\text{eff}}_{\max}$. Thus, the running time must be less than the bound above.

The running time of the preprocessing (Algorithm 5) is dominated by the sorting of the whole set of nodes V. Assuming an efficient Sort algorithm such as Merge Sort, the running time is $O(n \log n)$.

3.3 Implementation

We aim at very large networks with hundred-millions of nodes and billions of edges. The file size of the datasets can easily surpass the available memory space of the computers that we use [1]. This can be aleviated by using compression. The WebGraph framework [6, 7] provides a utility to compress a graph dataset, and also to manipulate the compressed files. Furthermore, it makes it possible to load a dataset part by part, hence could reduce (up to a limit) any memory space problem.

After the data is read from the file, it has to be decompressed. There is an overhead cost for this decompression process. However, this process can be done very fast. On the other hand, an uncompressed file has larger size than the compressed one, hence using uncompressed file as input would incur larger I/O cost. Thus, foregoing compression does not necessarily make the program faster.

We use the Java 8 JDK from the Oracle for our code implementation. There are two reasons for this choice. The WebGraph libraries, `webgraph`, were written in Java. Thus, by using Java our programs will work natively with the WebGraph libraries. Java 8 also provides libraries for parallel streaming, which makes it easy to utilize the multi-thread feature of modern computers. In our case, the node iteration in the algorithms is ideal for parallelization.

Our codes are the following:

- `Java8CycleTriangle.java` - for counting cycle triangles.
- `Java8TrustTriangle.java` - for counting trust triangles.
- `Java8TriangleCounting.java` - for counting undirected triangles.
- `SortGraphAscBG.java` - for preprocessing undirected graphs.

[1] Future computers would have more memory space. However, the networks will also be larger.

- `GnuSortGraphBg.java` - for preprocessing undirected graphs using the GNU sort utility (in Linux).

These codes were tested and ran on a Ubuntu platform. They require that Java 8 and `webgraph` are installed. Our codes are available upon request.

Chapter 4

Experiments

In this chapter we describe the experiments that we did. First we describe the machines and the datasets; then we show the results for all three cases: cycle triangles, trust triangles, and undirected triangles. We will discuss the results further in the next chapter.

4.1 The Machines

We ran our experiments on two machines: a PC with Intel Core i7 processor (from here on referred as i7), and a server with Intel Xeon processor (referred as Xeon). We installed Java 8 and the WebGraph framework (`webgraph-3.6.1` and its dependencies) on the machines. The specifications are as follows.

- PC (i7):
  - Processor: Intel$^{(R)}$ Core$^{(TM)}$ i7-3770 CPU @ 3.40GHz
    4 cores - 8 threads
  - Memory: 12 GB RAM
  - OS: Ubuntu 14.04.5 LTS

  - Java: Oracle Java 8

    Java$^{(TM)}$ SE Runtime Environment (build 1.8.0_151-b12)

- Server (Xeon):

  - Processor: Intel$^{(R)}$ Xeon$^{(R)}$ CPU E5620 @ 2.40GHz

    dual processors for a total of 16 threads

  - Memory: 64 GB RAM

  - OS: Ubuntu 14.04.1 LTS

  - Java: Oracle Java 8

    Java$^{(TM)}$ SE Runtime Environment (build 1.8.0_161-b12)

4.2 The Datasets

We applied our algorithms on eleven networks in compressed webgraph format. The datasets were downloaded from the WebGraph website [4]

http://law.di.unimi.it/datasets.php

We downloaded both the graph and the transpose graph for each network, and we symmetrized them to get the corresponding undirected graph. Of the eleven graphs, two are small (with less than 100,000 nodes), two are medium (with between 100,000 and 1,000,000 nodes), six are large (with 1 million to 200 millions nodes), and one is very large (with almost 1 billion nodes). The networks are:

1. **wordassociation-2011** (abbreviated as **words**): a free word association experiment performed by six thousand participants [18].

2. **enron**: email communications among Enron employees, made public by the Federal Energy Regulatory Commission.

3. **uk-2007-05@100000** (abbreviated as **uk-2007**): a subset of 100,000 nodes taken from larger dataset uk-2007-05 which were collected by the DELIS project [5, 8].

4. **cnr-2000**: a small crawl from the Italian CNR (Consiglio Nazionale delle Ricerche).

5. **ljournal-2008** (abbreviated as **ljournal**): a snapshot from LiveJournal (https://www.livejournal.com/) in 2008.

6. **uk-2002**: a crawl of .uk domain, performed by UbiCrawler [3] in 2002.

7. **arabic-2005** (abbreviated as **arabic**): a crawl of websites that contain Arabic, performed by UbiCrawler [3] in 2005.

8. **uk-2005**: a shallow crawl of .uk, performed by UbiCrawler [3] in 2005.

9. **webbase-2001** (abbreviated as **webbase**): a crawl by the WebBase crawler [12].

10. **twitter-2010** (abbreviated as **twitter**): a snapshot of Twitter follow connections [13] in 2010.

11. **clueweb12** (abbreviated as **clueweb**): a crawl of English webpages, created by the Lemur Project (http://www.lemurproject.org/clueweb12/index.php).

For each network `basename` we downloaded `basename.graph`, `basename.properties`, `basename-t.graph`, and `basename-t.properties`. We used the WebGraph toolkit to create the offsets by the following commands

```
java it.unimi.dsi.webgraph.BVGraph -o -O -L basename
```

and

```
java it.unimi.dsi.webgraph.BVGraph -o -O -L basename-t
```

assuming that the WebGraph library is installed in the working directory.

The statistics of these datasets are listed in Table 4.1. We order them according to the number of edges. Even though *twitter* has fewer nodes than *webbase*, about 42M compared to 118M, it has more edges, 1.468B compared to 1.020B. Therefore, *twitter* is listed after *webbase*. The smallest dataset, *words*, has only 72K edges, while the largest dataset, *clueweb*, has more than 42B edges. Note that the file sizes fit easily within current typical harddisk size (of 1 TB), and with the exception of *twitter* and *clueweb* should also fit with a current PC typical memory size (of 8-12 GB). However, these are the compressed files that need to be decompressed in the memory, and also we normally need more memory space than just for the dataset. Nonetheless, the WebGraph framework enables us to load the data into the memory part by part. Using this tool, and about 10 GB memory allocated, we were able to process *clueweb* as well.

Name	$\|V\|$	$\|E\|$	Size of G (Bytes)	Size of G^T (Bytes)
words	10,617	72,172	96K	92K
enron	69,244	276,143	208K	276K
uk-2007	100,000	3,050,615	760K	536K
cnr-2000	325,557	3,216,152	1.2M	920K
ljournal	5,363,260	79,023,142	105M	105M
uk-2002	18,520,486	298,113,762	81M	61M
arabic	22,744,080	639,999,458	141M	96M
uk-2005	39,459,925	936,364,282	201M	140M
webbase	118,142,155	1,019,903,190	399M	338M
twitter	41,652,230	1,468,365,182	2.5G	2.3G
clueweb	978,408,098	42,574,107,469	12G	7.0G

Table 4.1: Dataset statistics of the directed graphs. The sizes are of the compressed webgraph files.

The degree statistics are listed in Table 4.2. We can read the maximum out-degree, the maximum in-degree, and the average degree of each graph. Keep in mind that the maximum in-degree of a graph is the same as the maximum out-degree of

the transpose graph. We see that, for all of the graphs, the average degrees are low (relative to the number of nodes), which means that these graphs are sparse. Also, the maximum degrees are larger than the average, indicating that there are dominant nodes in the graphs.

Also listed in the table is the ratio $d^-_{\max}/d^+_{\max}$. With the exception of *enron*, we see that $d^-_{\max}$ and $d^+_{\max}$ differ. This indicates the unsymmetrical nature of the graphs. We will see that this makes a difference on the running time on the graph and transpose graph. Note that *clueweb*, and to a smaller degree *uk-2005* and *webbase*, are highly unsymmetrical. Also note that *twitter* is special in that $d^-_{\max}$ is less than $d^+_{\max}$, opposite to those of the other graphs.

Name	$d^+_{\max}$	$d^-_{\max}$	d_{avg}	$d^-_{\max}/d^+_{\max}$
words	34	324	6.8	9.5
enron	1,392	1,394	4.0	1.0
uk-2007	3,753	55,252	30.5	14.7
cnr-2000	2,716	18,235	9.9	6.7
ljournal	2,469	19,409	14.7	7.9
uk-2002	2,450	194,942	16.1	79.6
arabic	9,905	575,618	28.1	58.1
uk-2005	5,213	1,776,852	23.7	340.9
webbase	3,841	816,127	8.6	212.5
twitter	2,997,469	770,155	35.3	0.26
clueweb	7,447	75,611,690	43.5	10,153.3

Table 4.2: Degrees statistics in the datasets

We symmetrized these graphs to get undirected graphs using the command

```
java it.unimi.dsi.webgraph.Transform union
    basename basename-t basename-usym
```

Here we use label `usym` for the output files, but we can choose any label we like.

The statistics of the undirected graphs are listed in Table 4.3. Note that the maximum degree $d_{\max}$ of the undirected graph is the maximum of $d^+_{\max}$ and $d^-_{\max}$ of the original directed graph. The number of nodes remain the same.

We further preprocessed these undirected graphs by sorting the nodes in ascending order according to their degree, relabelling the nodes, and recording only the connections from lower labeled nodes to higher labeled nodes in the adjacency list, according to Algorithm 5. With this preprocessing, the node with the highest degree would be labeled the highest, hence it would have an empty (effective) adjacency list. As a result, we had drastically reduced the maximum degrees by this preprocessing. The new maximum (effective) degrees are listed in the table as $d_{\max}^{\text{ascBG}}$. For *clueweb*, for example, without preprocessing the maximum degree is 75.6M, but after preprocessing it is only 4K - about eighteen thousand times smaller.

Name	$\|V\|$	$\|E\|$	$d_{\max}$	$d_{\max}^{\text{ascBG}}$
words	10,617	63,788	332	140
enron	69,244	254,449	1,634	87
uk-2007	100,000	2,779,575	55,252	140
cnr-2000	325,557	2,738,969	18,236	85
ljournal	5,363,260	49,514,271	19,432	756
uk-2002	18,520,486	261,787,258	194,955	943
arabic	22,744,080	553,903,073	575,628	3,247
uk-2005	39,459,925	783,027,125	1,776,858	592
webbase	118,142,155	854,809,761	816,127	1,527
twitter	41,652,230	1,202,513,046	2,997,487	4,102
clueweb	978,408,098	37,372,179,311	75,611,690	4,244

Table 4.3: Dataset statistics of the undirected graphs.

The file sizes of the undirected graphs before and after preprocessing are listed in Table 4.4. Interestingly, preprocessing does not always reduce the file size. This can be explained as follows. Before preprocessing, the compression is optimized by labelling related nodes close to each other. Thus, on average, the distances (in terms of the indices) in the adjacency list are not large. When we sort the nodes according to the degrees, related nodes become scattered. This leads to large average distances in the adjacency list. Thus, even though the entries are fewer (after the cut) we need more bits for the offsets. We see that now *clueweb* size has become larger, about

21.5 GB. This would increase the I/O cost of the loading process.

Name	Size Before	Size After
words	153 KB	94 KB
enron	419 KB	297 KB
uk-2007	1.1 MB	1.2 MB
cnr-2000	1.6 MB	1.6 MB
ljournal	134 MB	112 MB
uk-2002	120 MB	168 MB
arabic	208 MB	285 MB
uk-2005	287 MB	401 MB
webbase	602 MB	848 MB
twitter	4.1 GB	2.3 GB
clueweb	16.4 GB	21.5 GB

Table 4.4: The compressed file sizes of the undirected graph datasets before and after preprocessing.

4.3 Results

We ran our triangle counting programs for the cycle triangles, the trust triangles, and the undirected triangles.

4.3.1 Cycle Triangles

Let us first look at the cycle triangles. Here we ran our program (`Java8CycleTriangle`) directly on the directed graphs that were downloaded from the WebGraph website. Table 4.5 shows the results using the i7 machine [1]. We list the number of cycle triangles and the running times: on the graph (Time), on the transpose graph (Time-Tr), and on the graph using a single stream/thread (Time-1S) [2].

[1] Except for *words*, there are some self-loops present in the graphs. This causes the numbers of cycle triangles listed here to be a bit higher than the actual numbers. This problem can be corrected by preprocessing the graphs using the `webgraph` utility `it.unimi.dsi.webgraph.Transform.NO_LOOPS`.

[2] The *clueweb* requires stack memory allocation of 10 GB (`java -Xmx10g`) to run.

Name	# triangles	Time	Time-1S	Time-Tr
words	13,121	0.3	0.4	0.3
enron	275,506	1.3	1.5	1.6
uk-2007	7,172,907	3.7	10.3	3.4
cnr-2000	3,463,610	2.6	5.1	3.1
ljournal	339,359,517	174.8	473.5	186.3
uk-2002	1,494,547,662	141.8	641.0	132.5
arabic	32,924,180,892	1,362.0	4,798.4	1,370.5
uk-2005	10,266,506,653	688.5	2,978.7	717.6
webbase	6,490,805,700	452.6	2,241.8	429.7
twitter	44,738,118,599	219,197.5	413,006.9	198,520.0
clueweb	1,036,190,284,927	81,149.9	316,910.4	51,444.7

Table 4.5: The results of running cycle triangle counting on the i7 machine. The times are in seconds.

We see that Time and Time-Tr do not differ significantly, except perhaps for *clueweb*. This means that interchanging the roles between the graph and the transpose graph does not have a big impact on the running time. The comparison between Time (which uses `parallelStream`, employing all eight threads of the i7) and Time-1S (which uses `Stream`, employing only a single thread) shows the gain by parallelizing to utilize the multicores in the machine. We visualize this comparison in Fig. 4.1. For small and medium graphs, the running times are too short to definitely see the differences. For large and very large graphs, the gain using `parallelStream` is about two to five times faster. Using eight threads does not give an eight times improvement because of the overhead cost.

We also ran the cycle triangle program on the Xeon machine. The results are shown in Table 4.6. Even though the Xeon machine has more threads (16) it has a slower CPU compared to the i7 machine. We see that the Xeon machine generally gave longer running time, except for the *clueweb* (the graph). Again, we see that reversing the roles between the graph and the transpose graph does not significantly change the running time. Here, the difference between the graph and the transpose

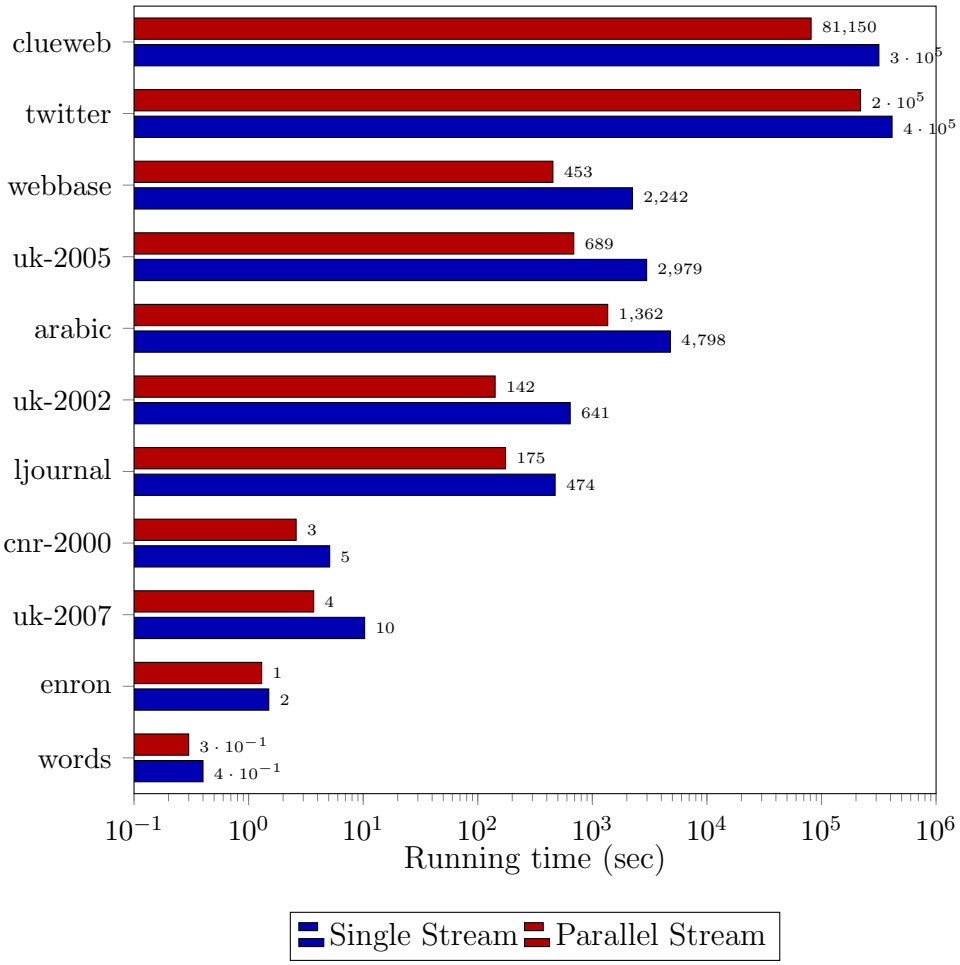

Figure 4.1: The running times of the cycle triangle counting using single thread and multi (eight) threads.

graph for *clueweb* is less compared that using the i7. It is now less significant.

4.3.2 Trust Triangles

In this section we show the results for the trust triangle counting. We ran our program (`Java8TrustTriangle`) on the directed graphs. The results from the i7 and the Xeon machines are combined into one table, Table 4.7 [3]. The Time(i7), Time-

[3]Except for *words*, there are some self-loops present in the graphs. This causes the numbers of trust triangles listed here to be a bit higher than the actual numbers. This problem can be corrected by preprocessing the graphs using the `webgraph` utility `it.unimi.dsi.webgraph.Transform.NO_LOOPS`.

Name	# triangles	Time	Time-Tr
words	13,121	0.74	0.79
enron	275,506	2.3	2.3
uk-2007	7,172,907	5.1	7.8
cnr-2000	3,463,610	3.7	4.1
ljournal	339,359,517	240.3	234.6
uk-2002	1,494,547,662	149.0	144.5
arabic	32,924,180,892	2,225.2	2,089.8
uk-2005	10,266,506,653	763.4	737.0
webbase	6,490,805,700	523.3	479.0
twitter	44,738,118,599	366,982.6	360,005.1
clueweb	1,036,190,284,927	63,429.19	57,090.61

Table 4.6: The results of running cycle triangle counting on the Xeon machine. The times are in seconds.

Tr(i7) are the running time on the graph, transpose graph respectively, using the i7; the Time(Xe), Time-Tr(Xe), are the running times on the graph, the transpose graph respectively, using the Xeon. All the time measurements are in seconds. We see that, generally, the running time is better on the PC than on the server, consistent with what we found for the cycle triangle counting.

Name	# triangles	Time(i7)	Time-Tr(i7)	Time(Xe)	Time-Tr(Xe)
words	105,525	1.1	0.3	2.0	0.7
enron	2,031,491	1.4	1.5	3.6	2.4
uk-2007	113,333,360	4.2	23.8	5.8	37.9
cnr-2000	41,706,973	2.5	6.1	3.7	22.5
ljournal	1,356,909,844	226.2	239.3	290.4	328.9
uk-2002	10,613,434,827	166.7	165.6	178.0	172.6
arabic	133,016,399,618	1,235.3	4,918.8	2,256.3	6,940.5
uk-2005	59,602,178,569	728.9	35,774.4	897.6	55,919.7
webbase	34,577,523,228	557.2	1,721.0	590.0	2,746.1
twitter	143,011,093,363	584,172.4	162,455.2	-	231,977.4
clueweb	5,508,820,034,813	57,993.3	-	68,619.1	-

Table 4.7: The results of running trust triangle counting. The running times are in seconds.

Comparing the running times on the graph and on the transpose graph, we see that for smaller graphs they do not differ significantly. For larger graphs, *arabic* and

larger, the running time on the graph is better than the running time on the transpose graph, except for *twitter*. The running-time on the *clueweb* transpose graph is greater than one million seconds. There is no point of keeping it running, so we aborted the run. Similarly, for *twitter* on the Xeon machine.

We can explain the running time difference between the graph and the transpose graph by recalling our running time analysis in Subsection 3.1.2. For the worst case, the running time for the trust triangle counting is proportional to the square of the maximum out-degree. What we have in hand here is not the worst case, but nevertheless the higher degree nodes stretch the running time longer. For example, for *arabic*, $d^-_{\max}/d^+_{\max} = 58.1$, and the running time ratio $T(G)/T(G^T)$ is between 3 and 4. For *uk-2005*, $d^-_{\max}/d^+_{\max} = 340$, and the running time ratio is more prominent, $T(G)/T(G^T)$ is about 50. For smaller graphs, the overhead could dominate the running time, hence dilutes the differences.

For *twitter*, recall that it differs from all the other graphs by the fact that its maximum out-degree is larger than its maximum in-degree (see Table 4.2), and recall that the maximum in-degree of a graph is equal to the maximum out-degree of its transpose graph. Because of this, we would expect that the running time of the trust triangle counting is longer on the graph than on the transpose graph.

We take the data for the Xeon machine and visualize the difference in the running times in Figure 4.2. It is interesting to note that the running time on the *twitter-transpose* is larger than the running time on *clueweb* eventhough *twitter* is smaller. Notice that the maximum out-degree of *clueweb* is much smaller than the maximum out-degree of *twitter-transpose*.

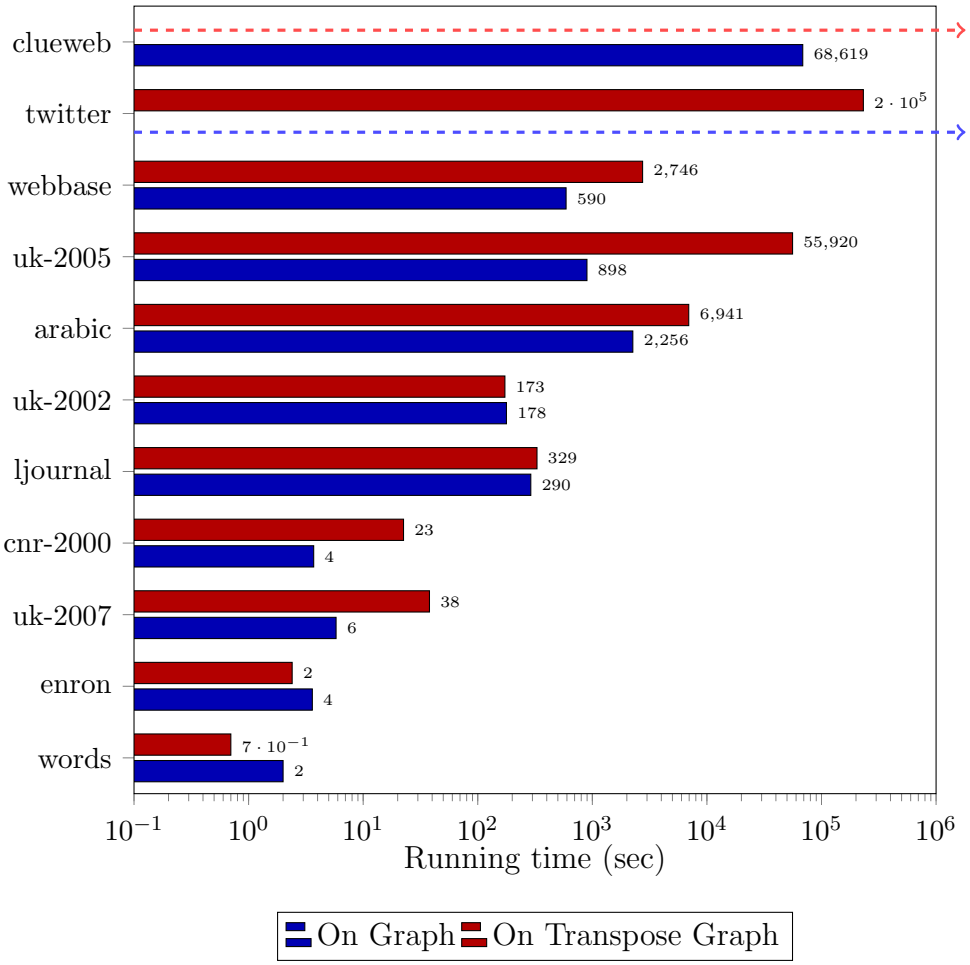

Figure 4.2: The running time of the trust triangle counting using the Xeon machine on the graphs and on the transpose graphs.

4.3.3 Undirected Triangles

Next, we ran undirected triangle counting (`Java8TriangleCounting`) on the undirected graphs. The results using the i7 machine are listed in Table 4.8, while the ones using the Xeon machine are listed in Table 4.9 [4].

The Time(i7,Xe) is the running time on the graphs without preprocessing. For *uk-2005*, *twitter*, and *clueweb* this takes longer than one million seconds, and we aborted the runs. The Time*(i7,Xe) is the running time on the graphs with preprocessing, and the PrepTime is the preprocessing time. Since Time(i7,Xe) does not involve PrepTime

[4]The presence of self-loops does not affect the countings of undirected triangles.

Name	# triangles	Time(i7)	Time*(i7)	PrepTime	TotTime*(i7)
words	61,795	1.3	0.3	0.4	0.7
enron	1,067,993	2.1	0.6	0.8	1.4
uk-2007	59,926,808	68.8	2.2	1.4	3.6
cnr-2000	20,977,629	89.7	1.6	1.6	3.2
ljournal	411,155,444	340.9	47.2	27.2	74.4
uk-2002	4,451,687,605	1,221.7	135.5	48.3	183.8
arabic	36,895,360,842	23,840.9	1,268.2	82.0	1,350.2
uk-2005	21,779,366,056	-	400.2	120.4	520.6
webbase	12,262,060,053	6,879.2	538.2	249.4	787.6
twitter	34,824,916,864	-	14,828.2	627.7	15,455.9
clueweb12	1,995,295,290,765	-	381,167.9	45,063.8	426,231.7

Table 4.8: The results of running undirected triangle counting on the i7 machine. The times are in seconds.

we should compare Time(i7,Xe) with the sum of Time*(i7,Xe) and PrepTime, which is given by TotTime*(i7,Xe).

Name	# triangles	Time(Xe)	Time*(Xe)	PrepTime	TotTime*(Xe)
words	61,795	1.1	0.5	0.5	1.0
enron	1,067,993	3.0	1.2	1.0	2.2
uk-2007	59,926,808	97.8	3.2	2.0	5.2
cnr-2000	20,977,629	100.9	2.1	2.9	5.0
ljournal	411,155,444	446.1	52.6	49.6	102.2
uk-2002	4,451,687,605	1,314.4	132.6	91.7	224.3
arabic	36,895,360,842	32,997.8	1,805.7	162.5	1,968.3
uk-2005	21,779,366,056	-	427.5	243.1	670.6
webbase	12,262,060,053	9,318.1	656.5	457.5	1,114.0
twitter	34,824,916,864	-	25,238.8	1,011.9	26,250.7
clueweb12	1,995,295,290,765	-	64,169.1	9,118.6	73,287.7

Table 4.9: The results of running undirected triangle counting on the Xeon machine. The times are in seconds.

We see that the i7 gave better performance for almost every graph. For *clueweb*, however, it is much slower than the Xeon. This could be because of the memory limit on the i7. It has only 12 GB, compared to 64 GB on the Xeon. It might also due to the different architecture between PC and Server. Servers are designed to handle larger amount of data than PCs.

Figure 4.3 shows the comparison of the times required to do triangle counting with and without the preprocessing, using the data from Xeon. We see that preprocessing consistently gives advantage over counting without preprocessing. The improvement is quite significant, especially for larger graphs. For *arabic*, for instance, it is about seventeen times better. It also enables us to run on *uk-2005*, *twitter* and *clueweb*, which previously take too long without the preprocessing.

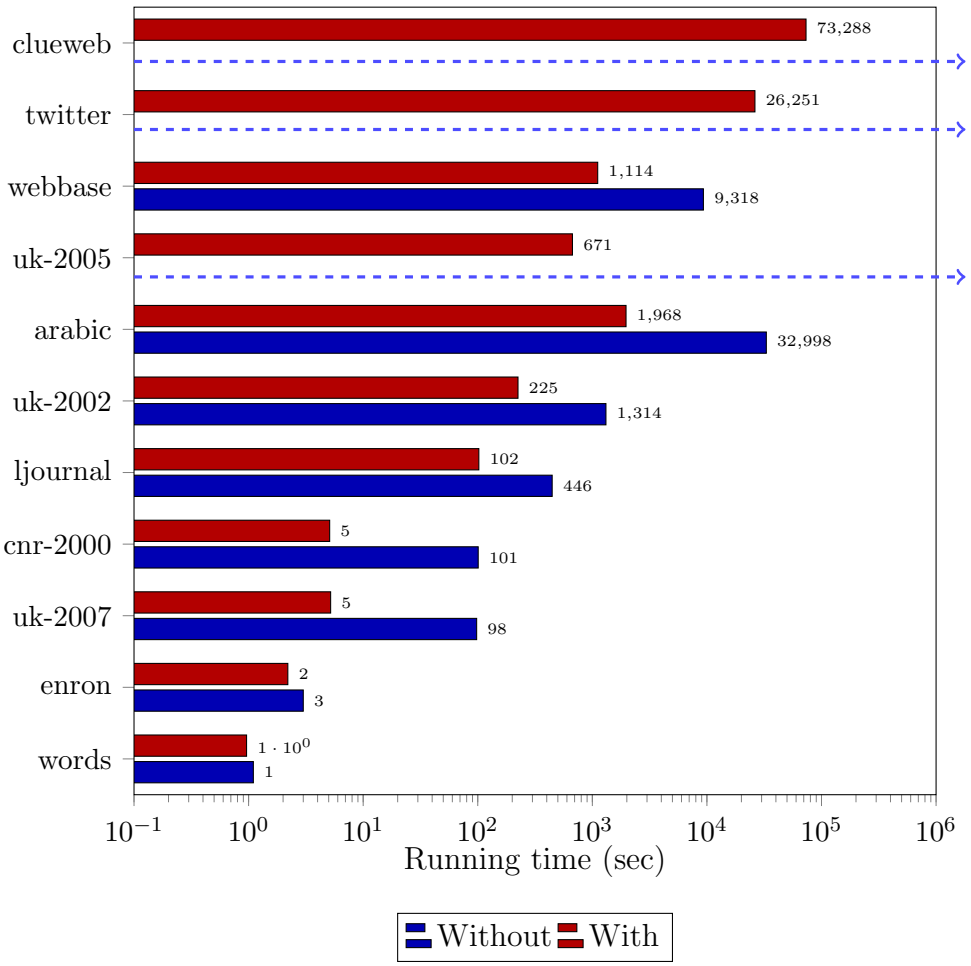

Figure 4.3: The running time on undirected graphs without and with preprocessing on the Xeon machine.

Chapter 5

Discussions

5.1 On the Results

Let us now analyse our experimental results further. First consider the number of triangles. In Figure 5.1 we plot the number of triangles versus the number of edges, normalized by the number of nodes. The plot for the undirected case is shifted to the left because the undirected graphs have smaller number of edges compared to the directed ones - we consider a bidirected pair as one undirected edge.

Note that because of this normalization, the points on each plot are not ordered by the graph size. In fact, the order from left to right is *enron*, *words*, *webbase*, *cnr-2000*, *ljournal*, *uk-2002*, *uk-2005*, *arabic*, *uk-2007*, *twitter*, and *clueweb*. The peak of trust is the *arabic*.

We see that there are more trust triangles than cycle triangles for each network. For each triple of nodes, there could be up to six trust triangles and two cycle triangles. Thus, the probability of having a trust triangle is bigger than the probability of having a cycle triangle.

The number of undirected triangles is more or less between the number of cycle

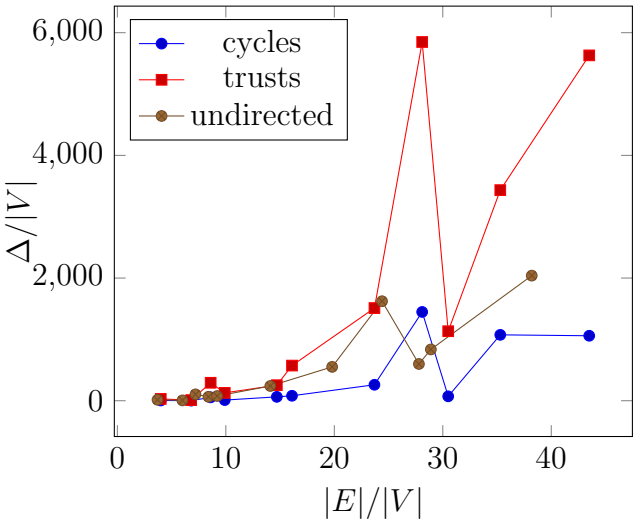

Figure 5.1: The number of triangles versus the number of edges, normalized by the number of nodes.

triangles and the number of trust triangles. The exception of this is *twitter*, where the number of undirected triangles is less than the number of cycle triangles. This is telling us that there are many triple of nodes with two cycle triangles in the *twitter* directed graph. When we get to the undirected graph those considered as just one triangle. Similarly, when the number of undirected triangles is less than the number of trust triangles, which is for all graphs that we have here, it means that there are triple of nodes with more than one trust triangles.

However, in any case, the number of undirected triangles cannot be less than one half of the number of cycle triangles and it cannot be less than one sixth of the number of trust triangles. It cannot be more than the sum of the number of cycle triangles and the number of trust triangles (see Equation 2.3).

Let $m = |E|$ and $n = |V|$. Given n nodes, the maximum number of triangles possible is $\Delta_{\max} = \begin{pmatrix} n \\ 3 \end{pmatrix}$ which is the case for a complete graph. In that case, the

number of edges is $m_{\max} = \binom{n}{2}$. Thus, the ratio is $\Delta_{\max}/m_{\max} = (n-2)/3$, which is large for large n. However, our graphs are sparse, and we get only a small number of triangles relative to the number of nodes. This can be seen in Figure 5.1. The ratio is about one to three order of magnitude.

We did not do triangle enumeration in our experiment. The reason is because we want to measure the counting time, and therefore we want to make it as efficient as possible. Another reason is, if we want to save the triangle data, we would need a huge amount of storage space, especially for the larger networks. For *clueweb*, for example, the number of undirected triangles is almost 2 trillions. With each triangle consists of three nodes, and assuming four bytes for each node, we would need 24 TB of harddisk space. Some encoding techniques or compressions can be used to reduced this number, but the required space would still be very large. Nonetheless, our programs are capable of doing triangle listing with only few lines activated (i.e., uncommented).

Next, let us look at scalability. We plot the running time of the cycle triangle counting as a function of the number of nodes in Figure 5.2, and as a number of edges in Figure 5.3. From these plots we see that the running time is more or less linear against the number of nodes or edges. Therefore, this looks promising for doing even larger networks, at least for the algorithms. In practice, there could be some limitation from the machines - recall the case of undirected triangle counting on *clueweb* using the i7 machine.

Parallelizing the counting yields some improvement. However, we are limited by the number of threads available in our machine. At the time of writing, there is Intel Core i9 CPU with 18 cores and 36 threads, and it is likely that there will be newer CPUs with even more threads in the future. Nevertheless, the limitation will still be

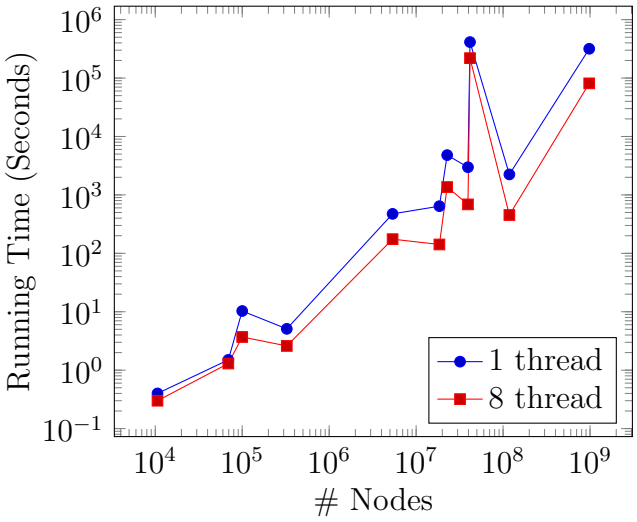

Figure 5.2: The running time of the cycle triangle counting using 1 thread (not parallelized) and 8 threads (parallel streams) on the i7 machine as a function of the number of nodes.

there.

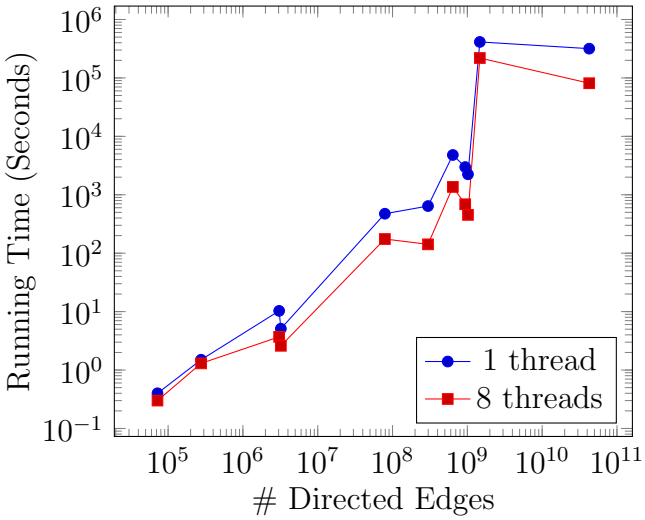

Figure 5.3: The running time of the cycle triangle counting using 1 thread (not paralleized) and 8 threads (parallel streams) as a function of the number of edges.

We found that while for *clueweb* we are able to complete the computation in slightly less than one day, for *twitter* the computation needs almost three days to complete. It might not be easy to improve this using a single machine. We should

make comparison with the performance gained by distributed computation using many machines instead. Our advantage would be on the cost, and their advantage would be on the possibility of adding more computational power (i.e., more machines). Even for distributed computing, there would be a limit on how many compute nodes can be used due to the network or the distributing program limitation. We will talk more about this comparison in the next subsection.

Figure 5.4 is a combined plot of the cycle, trust and undirected triangle countings. For the undirected case we use the running time with preprocessing. The plot shows that the running time scales similarly for all the three cases. In fact, except for *twitter*, they have about the same running times.

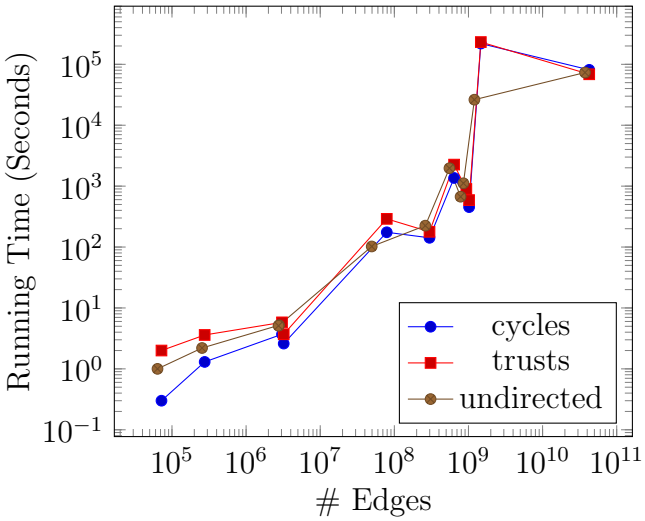

Figure 5.4: The running time of the cycle triangle counting, the trust triangle counting and the undirected triangle counting as a function of the number of edges. Note: for twitter, the trust ran on the transpose graph.

5.2 Related Work

Triangle countings can be categorized into enumeration, which yields exact counting, and estimation. In the first category, see e.g., Schank and Wagner [27], and

Latapy [15]. In the second category, see e.g., [33]. Note that our algorithms go into the enumeration category. Triangle counting/enumeration has been studied intensively, especially for the undirected case. Various algorithms had been proposed [26]. In [27] it was shown that Forward Algorithm gives good performance. It is an improvement of the Edge Iteration Algorithm with only larger neighbour nodes are iterated. Nonetheless, the algorithm is not suitable for parallelization. Namely, the node data array which is generated during the run must be available to all threads. Moreover, it must be generated sequentially according to the node index.

We take a different approach from the Forward Algorithm. The (effective) adjacency list produced by our preprocessing contains larger neighbours, with function similar to that of the node data array in the Forward Algorithm. However, it is produced before the triangle counting starts. Since it is used as input to the counting program it can be shared among the threads. Moreover, by separating the preprocessing from the counting we can sort the nodes according to degree and relabel them to get an even smaller adjacency list. As also noted by [27], our preprocessing gives most benefit when the graph has skewed degrees, i.e., there are nodes with degrees much higher than others.

Graph computation had been studied using distributed computing, in particular using Map Reduce [10]. For triangle enumeration, Suri and Vassilvitskii [31] suggested partitioning the nodes of the graph, to be distributed among the compute nodes. Park et al. had developed efficient algorithms based on this idea for use with Map Reduce [20, 22, 21]. Their latest algorithm is called PTE (Pre-partitioned Triangle Enumeration) [21]. They did experiment on a cluster with 41 machines, each with quad core Intel Xeon E3-1230v3 CPU at 3.30 GHz and 32 GB RAM. The running time on *twitter* was about 2 to 3 minutes (120 to 180 seconds).

Zhang et al. claimed that they could improve the PTE further by using bloom filter

hashing [36]. They called their algorithm FTL. Their experiment used 20 machines, each has 32 GB memory and six cores CPU of 2.6 GHz. On *twitter* their (best) running time is 210 seconds.

The PTE algorithm and code was examined by Bhojwani [2]. She extended the PTE to do triangle countings for directed graphs as well, i.e., the cycle and trust triangles. She did experiment using five compute nodes on Amazon Elastic Map Reduce Service [1]. Each node has specification of 16 vCore and 32 GB memory. The physical processor is Intel Xeon E5-2676 v3. The running time for undirected triangle counting on *ljournal* is about 200 seconds, and on *arabic* is about 4.5K seconds.

Our running times on *twitter* for undirected graph are 15.5K seconds (i7) and 26.3K seconds (Xeon). This is about two order of magnitude larger than the ones by PTE (which is about 150 seconds) and FTL (210 seconds). However, note that we used an older machine than what they used for their cluster (See Section 4.1.). The price of our machine is about two order of magnitude lower than the price of their cluster. Our running times on *ljournal* (undirected) are 74 seconds (i7) and 102 seconds (Xeon), and on *arabic* are 1.4K seconds (i7) and 2.0K seconds (Xeon). These are better than the results by Bhojwani [2].

On the other hand, there are several programs built specifically for doing graph computation on a single machine. GraphChi [14] is a disk-based system with a parallel sliding windows method. GridGraph [37] is another system with a dual sliding windows method. Mosaic [16] uses another method called Hilbert-ordered tiles for graph representation. For these, since the disk is used to store temporary data during the execution, the main overhead cost would be on the I/O cost. Mosaic was tuned to take advantage of new storage technology NVMe SSD, with the aim of reducing this I/O cost. However, these programs use vertex centered computation and need

[1] These compute nodes were not shared, but dedicated 100% to the experiment.

modification if we want to use them for triangle enumeration.

In this project we have been able to process very large graphs on a machine with limited memory by making use of compressed graph structures offered by WebGraph. This framework was also used in other works on some other graph problems, e.g., influence maximization [23, 24], 2-core [32], feedback arc set [30], clearing contamination [29], and importance-based communities [9].

5.3 Future Work

There are several possibilities to improve the performance of our programs even further. When running the experiment, we observed that towards the end of computation there are some straggler threads. These threads would still be running after the other threads had done their tasks. Sometimes it takes a very long time for the last thread to finish its job. This is most apparent on *twitter*. This problem can be alleviated if we have better divisions of tasks among the threads. So far, in our programs this dividing task was managed automatically by Java 8 through the `parallelStream` method. We would need more control on the parallelising process, but we also need to know how to estimate how much computation is needed for each group of nodes.

In this project, the largest graph that we test is *clueweb* with almost a billion nodes and about forty two billions of edges. In the future, we should run our programs on even larger graphs and test the limit. We should also make use of better technologies such as CPUs with more cores, faster storage devices (SSD), more RAM, and GPU.

Java 8 streams and Spark [2] have some similarities in their syntax. We can modify our programs so that they can be ran on a cluster using Spark. We can then test the scalability of the algorithms.

[2] https://spark.apache.org/

If we look at any triple of nodes in a directed graph they can be connected to each other by more than three edges. There are seven types of connections, which are duped as the seven types of triangles [28]. Our programs can be extended to differentiate these types in the enumeration so that we can count the number of each types in the input graph. The programs can also be developed to search for larger subgraphs, such as rectangles. In this case, one idea is to use wedge iteration, instead of edge iteration, where a wedge is three nodes connected by two edges.

Chapter 6

Conclusions

In this thesis we studied algorithms for triangle enumeration. We found that the edge iteration algorithm is suitable for our purpose. The algorithms that are available in the literature were designed for undirected graphs. We made modification, and build algorithms for directed graphs, i.e., for the trust and cycle triangle enumeration. We implements the algorithms for both directed and undirected graphs using Java 8, where the computation were parallelized to utilize the multicore CPU of a single machine. WebGraph framework were used for graph compression, and this enabled us to analyse big graphs on a machine with limited memory. The codes were tested on several large networks and succeeded on a graph with almost a billion nodes and forty two billions of edges. The codes work either on a PC or a server.

By analysing the algorithms in detail, we gained insights on how to improve the efficiency of the computation. For trust triangles we found that choosing between the graph and the transpose graph, whichever has the smaller maximum degree, can affect the running time significantly. For undirected triangles, we can reduce the running time by first preprocess the graph with our Sort and Cut algorithm.

Our running times are comparable with the performance of Map Reduce on a

cluster with few nodes. Eventhough we used older machines, our executions are at least faster than PTE on five compute nodes of Amazon Elastic Map Reduce Service. Although, distributed computing can achieve better time by using more machines, we have advantage in term of the investment cost, and communication cost.

Single machine is still a feasible option for triangle enumeration on large networks, provided that we are not aiming at getting the fastest running time. With future technologies, the performance of our codes can be made better and this option would be more plausible.

Bibliography

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

[2] Pooja Bhojwani. Triangle enumeration in massive graphs using Map Reduce. Master's thesis, University of Victoria, 2018.

[3] Paolo Boldi, Bruno Codenotti, Massimo Santini, and Sebastiano Vigna. Ubicrawler: A scalable fully distributed web crawler. *Software: Practice & Experience*, 34(8):711–726, 2004.

[4] Paolo Boldi, Marco Rosa, Massimo Santini, and Sebastiano Vigna. Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks. In Sadagopan Srinivasan, Krithi Ramamritham, Arun Kumar, M. P. Ravindra, Elisa Bertino, and Ravi Kumar, editors, *Proceedings of the 20th International Conference on World Wide Web*, pages 587–596. ACM Press, 2011.

[5] Paolo Boldi, Massimo Santini, and Sebastiano Vigna. A large time-aware graph. *SIGIR Forum*, 42(2):33–38, 2008.

[6] Paolo Boldi and Sebastiano Vigna. The Webgraph Framework I: Compression techniques. In *Proceedings of the 13th International Conference on World Wide Web*, pages 595–602, New York, NY, USA, 2004. ACM.

[7] Paolo Boldi and Sebastiano Vigna. The Webgraph Framework II: Codes for the world-wide web. In *Data Compression Conference, 2004. Proceedings. DCC 2004*, page 528. IEEE, 2004.

[8] Ilaria Bordino, Paolo Boldi, Debora Donato, Massimo Santini, and Sebastiano Vigna. Temporal evolution of the uk web. In *Data Mining Workshops, 2008. ICDMW'08. IEEE International Conference on*, pages 909–918. IEEE, 2008.

[9] Shu Chen, Ran Wei, Diana Popova, and Alex Thomo. Efficient computation of importance based communities in web-scale networks using a single machine. In *Proceedings of the 25th ACM International Conference on Information and Knowledge Management*, pages 1553–1562. ACM, 2016.

[10] Jonathan Cohen. Graph twiddling in a mapreduce world. *Computing in Science & Engineering*, 11(4):29–41, 2009.

[11] Santo Fortunato. Community detection in graphs. *Physics Reports*, 486(3-5):75–174, 2010.

[12] Jun Hirai, Sriram Raghavan, Hector Garcia-Molina, and Andreas Paepcke. WebBase: A repository of web pages. *Computer Networks*, 33(1-6):277–293, 2000.

[13] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. What is Twitter, a social network or a news media? In *Proceedings of the 19th International Conference on World Wide Web*, pages 591–600. ACM, 2010.

[14] Aapo Kyrola, Guy E Blelloch, and Carlos Guestrin. Graphchi: Large-scale graph computation on just a PC. USENIX, 2012.

[15] Matthieu Latapy. Main-memory triangle computations for very large (sparse (power-law)) graphs. *Theoretical Computer Science*, 407(1-3):458–473, 2008.

[16] Steffen Maass, Changwoo Min, Sanidhya Kashyap, Woonhak Kang, Mohan Kumar, and Taesoo Kim. Mosaic: Processing a trillion-edge graph on a single machine. In *Proceedings of the 12th European Conference on Computer Systems*, pages 527–543. ACM, 2017.

[17] Frank McSherry, Michael Isard, and Derek G Murray. Scalability! but at what COST? In *HotOS*. Citeseer, 2015.

[18] Douglas L Nelson, Cathy L McEvoy, and Thomas A Schreiber. The University of South Florida free association, rhyme, and word fragment norms. *Behavior Research Methods, Instruments, & Computers*, 36(3):402–407, 2004.

[19] Mark EJ Newman, Duncan J Watts, and Steven H Strogatz. Random graph models of social networks. *Proceedings of the National Academy of Sciences*, 99(suppl 1):2566–2572, 2002.

[20] 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 Information & Knowledge Management*, pages 539–548. ACM, 2013.

[21] Ha-Myung Park, Sung-Hyon Myaeng, and U Kang. PTE: Enumerating trillion triangles on distributed systems. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 1115–1124. ACM, 2016.

[22] Ha-Myung Park, Francesco Silvestri, U Kang, and Rasmus Pagh. MapReduce triangle enumeration with guarantees. In *Proceedings of the 23rd ACM International Conference on Conference on Information and Knowledge Management*, CIKM '14, pages 1739–1748, New York, NY, USA, 2014. ACM.

[23] Diana Popova, Akshay Khot, and Alex Thomo. Data structures for efficient computation of influence maximization and influence estimation. EDBT, 2018.

[24] Diana Popova, Naoto Ohsaka, Ken-ichi Kawarabayashi, and Alex Thomo. Nosingles: a space-efficient algorithm for influence maximization. In *Proceedings of the 30th International Conference on Scientific and Statistical Database Management*, page 18. ACM, 2018.

[25] Filippo Radicchi, Claudio Castellano, Federico Cecconi, Vittorio Loreto, and Domenico Parisi. Defining and identifying communities in networks. *Proceedings of the National Academy of Sciences of the United States of America*, 101(9):2658–2663, 2004.

[26] Thomas Schank. *Algorithmic aspects of triangle-based network analysis*. PhD thesis, University Karlsruhe, 2007.

[27] Thomas Schank and Dorothea Wagner. Finding, counting and listing all triangles in large graphs, an experimental study. In *International Workshop on Experimental and Efficient Algorithms*, pages 606–609. Springer, 2005.

[28] Comandur Seshadhri, Ali Pinar, and Tamara G Kolda. Fast triangle counting through wedge sampling. In *Proceedings of the SIAM Conference on Data Mining*, volume 4, page 5, 2013.

[29] Michael Simpson, Venkatesh Srinivasan, and Alex Thomo. Clearing contamination in large networks. *IEEE Transactions on Knowledge and Data Engineering*, 28(6):1435–1448, 2016.

[30] Michael Simpson, Venkatesh Srinivasan, and Alex Thomo. Efficient computation of feedback arc set at web-scale. *Proceedings of the VLDB Endowment*, 10(3):133–144, 2016.

[31] 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*, pages 607–614. ACM, 2011.

[32] Babak Tootoonchi, Venkatesh Srinivasan, and Alex Thomo. Efficient implementation of anchored 2-core algorithm. In *Proceedings of the 2017 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining 2017*, pages 1009–1016. ACM, 2017.

[33] 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*, pages 837–846. ACM, 2009.

[34] Jia Wang and James Cheng. Truss decomposition in massive networks. *Proceedings of the VLDB Endowment*, 5(9):812–823, 2012.

[35] Duncan J Watts and Steven H Strogatz. Collective dynamics of small-worldnetworks. *Nature*, 393(6684):440, 1998.

[36] Hao Zhang, Yuanyuan Zhu, Lu Qin, Hong Cheng, and Jeffrey Xu Yu. Efficient triangle listing for billion-scale graphs. In *Big Data (Big Data), 2016 IEEE International Conference on Big Data*, pages 813–822. IEEE, 2016.

[37] Xiaowei Zhu, Wentao Han, and Wenguang Chen. Gridgraph: Large-scale graph processing on a single machine using 2-level hierarchical partitioning. In *USENIX Annual Technical Conference*, pages 375–386, 2015.