Fast In-Memory Triangle Listing for Large Real-World Graphs

Martin Sevenich
Oracle Labs
martin.sevenich@oracle.com

Sungpack Hong
Oracle Labs
sungpack.hong@oracle.com

Adam Welc
Oracle Labs
adam.welc@oracle.com

Hassan Chafi
Oracle Labs
hassan.chafi@oracle.com

ABSTRACT

Triangle listing, or identifying all the triangles in an undirected graph, is a very important graph problem that serves as a building block of many other graph algorithms. The compute-intensive nature of the problem, however, necessitates an efficient method to solve this problem, especially for large real-world graphs. In this paper we propose a fast and precise in-memory solution for the triangle listing problem. Our solution includes fast common neighborhoods finding methods that consider power law degree distribution of real-word graphs. We prove how theoretic lower bound can be achieved by sorting the nodes in the graph by their degree and applying pruning. We explain how our techniques can be applied automatically by an optimizing DSL compiler. Our experiments show that hundreds of billions of triangles in a five billion edge graph can be enumerated in about a minute with a single server-class machine.

1. INTRODUCTION

Triangle listing, or identifying all the triangles in an undirected graph, is a very important graph problem as it serves as a building block for many other graph analysis algorithms. For example, counting the number of triangles is an essential step in computing global and local clustering coefficients [27] as well as graph transitivity [20], all of which are important metrics in social network analysis. Triangles are also a frequent basic pattern in graph mining applications [13, 19]. For instance, Becchetti et al. demonstrated that triangle patterns can be used to detect spamming activities [8].

Noticeably, the triangle listing problem is heavily computation intensive. It takes a long time to solve this problem with conventional algorithms, especially on large real-world graph instances. Consequently, several distributed processing methods [12, 23, 26] have been proposed in order to solve the problem more quickly. There have even been attempts to obtain only an approximate number of triangles [21, 25] for the sake of fast execution.

In this paper, we present an exact, fast, parallel, *in-memory* method for solving the triangle listing problem. Our solution is exact in that we iterate all the triangles in the graph. Our methods performs such iteration in a very fast, parallel manner, as long as the graph instance fits in the main memory of a single machine. We argue that this in-memory assumption is reasonable even for processing fairly large graph instances, considering the memory capacity of contemporary server-class machines. For example, Twitter has been using an in-memory graph analysis framework which is able to handle their massive follower graph and largely responsible for their who-to-follow recommendation service [14]. Similarly, SAP HANA provides a successful off-the-shelf in-memory graph analysis option [11]. Indeed, using an efficient representation, a large graph having 5 billion undirected edges consumes about 40 GB of main memory, while modern analytic servers are easily equipped with hundreds of gigabytes of DRAM.

The contributions of our paper can be summarized as follows:

- We present a novel fast implementation of the common neighbor iterator that considers the power-law degree distribution of large real-world graph instances. (Section 3)
- We propose novel extensions to the conventional in-memory triangle counting algorithm and explain how their combined effect achieves an asymptotic lower bound on the triangle counting algorithm's execution time. (Section 4)
- We show how an optimizing compiler can automatically apply our techniques to a high-level Domain Specific Language program (Section 5)
- We present an experimental evaluation of our implementation demonstrating that it outperforms existing state-of-the-art triangle counting solutions. We also provide an in-depth analysis of the effects of all the optimizations we introduced. (Section 6)

2. BACKGROUND

In this section we provide some background information on the nature of the triangle counting problem, the conventional algorithm that is used to solve this problem, and the infrastructure we used to develop our solution.

2.1 Triangle Listing and Counting Problem

Formally, the triangle listing problem and its variant, the triangle counting problem, are defined on *simple, undirected graphs* as follows:

PROBLEM 1. *Let $G = (V, E)$ be an undirected, simple graph, where V is the set of nodes and E is the set of edges in G. The* Triangle Listing *problem is to identify set $T = \{(u, v, w) : u, v, w \in V, (u, v), (v, w), (w, u) \in E$, where $u \neq v \wedge v \neq w \wedge w \neq u\}$. In*

SNAKDD'14, August 24 2014, New York City, NY, USA

http://dx.doi.org/10.1145/2659480.2659494.

```
Procedure EdgeIterator(G: Graph) : Long {
  Long T = 0;
  // iterate over all edges of graph G
  Foreach(e: G.Edges) {
    Node u = e.FromNode(); // one end of edge e
    Node v = e.ToNode();   // the other end of edge e
    // iterate over each common neighbor of u and v
    Foreach(w: CommonNbrs(u,v)) {
        T++;
  } }
  Return T/3;  // eliminate repeated triangles
}
```

Figure 1: EdgeIterator algorithm

```
Procedure NodeIteratorPlusPlus(G: Graph) : Long {
  Long T = 0;
  // iterate over all nodes of graph G
  Foreach(u: G.Nodes) {
    // iterate over neighbors of v such that v>u
    Foreach(v: u.Nbrs) (v>u) {
      // iterate over neighbors of w such that w>v
      Foreach(w: u.Nbrs) (w>v) {
        If (w.IsNbr(v))
          T++;
  } } }
  Return T;
}
```

Figure 2: NodeIterator++ algorithm

other words, T is the set of three-cliques *in graph G. The* Triangle Counting *problem is to identify $|T|$, that is the size of T.*

The triangle *counting* problem is also considered interesting because there are cases when only the number of triangles is required but not the actual list. Note that solving the triangle listing problem trivially solves the triangle counting problem, as one can simply enumerate all the triangles without producing the list. In this paper, we present a solution for the triangle listing problem but use it to solve the triangle counting problem – i.e. we omit the output list construction.

2.2 Baseline Algorithm

Our baseline algorithm is a combination of two existing algorithms described in the literature, the `EdgeIterator` [22] algorithm (Figure 1) and the `NodeIterator++` algorithm [23] (Figure 2).

The `EdgeIterator` algorithm [22] presented in Figure 1 iterates over all edges of a given graph and identifies triangles by consulting a common neighbor iterator to determine all common neighbors of two end-point of each graph edge. The common neighbor iteration is implemented by comparing the adjacency data structures of the two edge end-points, or with a hashed container.

The `NodeIterator++` algorithm [23] presented in Figure 2 iterates over all nodes in the graph and for each node `u` identifies a pair of its neighborhood nodes, `v` and `w`, to check if they in turn are connected with each other by an edge. Unlike its original unoptimized version, the `NodeIterator++` algorithm assumes that there exists a total ordering among all nodes in the graph, which is used to avoid repeated counting as shown in Figure 2.

Our baseline algorithm, combining the `EdgeIterator` and the `NodeIterator++` algorithms is presented in Figure 2.2 – similarly to the `NodeIterator++` algorithm, it iterates over all nodes in the graph and takes advantage of total node ordering to avoid repeated counting, but similarly to the `EdgeIterator` algorithm it utilizes the common neighbor iterator. The motivation standing behind this `CombinedIterator` algorithm is that the in-memory graph representation used for our algorithm implementation (described in Section 2.3) supports very efficient implementation of both all-node iteration and common-neighbor iteration.

```
Procedure CombinedIterator(G: Graph) : Long {
  Long T = 0;
  Foreach(u: G.Nodes) {
    Foreach(v: u.Nbrs) (u>v) {
      Foreach(w: CommonNbrs(u,v)) (w>u) {
        T++;
  } } }
  Return T;
}
```

Figure 3: CombinedIterator Algorithm

The algorithm descriptions in this section are all expressed in Green-Marl [15], a Domain Specific Language(DSL) for graph analysis. The semantics of the language in these examples are largely self-explanatory, with additional clarifications provided in the comments. In Section 3 and Section 4, we will discuss our enhancements over the baseline algorithm in Figure 3 both in algorithmic way and in its implementation. Although these enhancements do not require the use of a DSL, in Section 5 we will show how these enhancements can be automatically applied to the high-level programs as Figure 3 or Figure 2 by the DSL compiler.

2.3 In-memory Graph Representation

We adopt the conventional Compressed Sparse Row (CSR) format as in-memory graph representation. In this representation, node v in the graph is represented as a unique identifier (`nid`(v)), an integer between 0 and `N-1`, where `N` is the number of nodes in the given graph. However, in the remainder of the paper, we use `nid`(v) and v interchangeably. Note that this representation naturally imposes a total ordering required for `NodeIterator++` and `CombinedIterator` algorithm in Section 2.2.

The edge structure of the graph is encoded in two arrays:

- `edge` – concatenation of the neighborhood lists of all the nodes in the graph in ascending `nid` order.
- `begin` – an array of indices which points to the beginning of the neighborhood list of the corresponding `nid`. This array maintains one sentry node at the end.

Moreover, each neighborhood list is sorted by values (i.e. `nid` of destination nodes). Therefore, for each node v, $Adj(v)$ is a sorted array, unless $Adj(v)$ is empty. This invariant is sometimes referred to as the *semi-sorted* property in literature, in order to distinguish it from the concept of sorting and numbering the nodes by their degree values (Section 4).

Although the CSR format is originally designed to represent directed graphs, it can be used for undirected graphs as well via duplication. That is, if there is an edge (u, v) in the given graph, we put v in $Adj(u)$ as well as put u in $Adj(v)$. Therefore the size of the array `edge` in CSR representation becomes $2 * E$, where E is the number of edges in the undirected graph.

The CSR encoding allows for even large graphs to be represented efficiently in terms of memory utilization. As an example, let us consider the Twitter-2010 graph instance [17]. It has been considered large as the undirected version has $\sim$ 41.7 million nodes and $\sim$2.4 billion edges (details see Table 1). However the CSR representation of this graph is relatively small. If we use 32-bit integer as `nid` (i.e. vales stored in `edge` array) and 64-bit integer as indices (i.e. values stored in `begin` array), the CSR representation of the graph will then occupy a little less than 19 GB.

2.4 Characteristics of Real-world Graphs

The focus of our work is on large real-world graphs that draw huge interest in academia and industry these days, such as social graphs, web graphs, or biological networks. Recent studies [7, 27] have revealed that the vast majority of these large real-world graphs

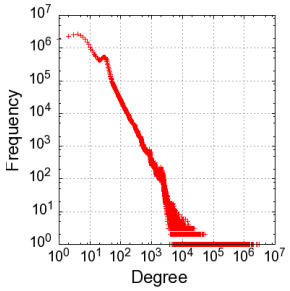


Figure 4: Degree Distribution of Twitter-2010 graph (undirected)

share certain interesting characteristics. One such characteristic is scale-freeness which states that the degree distribution of nodes in such graphs is far from uniform but follows a power law instead. In other words, in such graphs there exist a small number of nodes that are connected to an extremely large number of other nodes, often as many as O(N), in contrast to most of the nodes which have only a small number of neighbors.

As an example, consider the undirected version of the Twitter-2010 graph that we introduced in Section 2.3. Figure 4 illustrates the node degree distribution for the Twitter-2010 graph by showing how often (`y` axis) a node with a given degree (`x` axis) appears in the graph. As one can observe, millions of nodes in the Twitter-2010 graph have degree smaller than 10 and only small number of nodes have degree higher than 10000. Moreover, there exist a few nodes which have millions of neighbors.

The high-degree nodes in a scale-free graph can have a detrimental effect on the performance of a triangle counting algorithm – Section 3 and Section 4 discuss how our solution alleviates this issue.

3. FAST COMMON NEIGHBOR ITERATION

The `CombinedIterator` source code in Figure 3 reveals that the main part of the triangle counting solution is to repeatedly search for common neighbors of different pairs of nodes. Therefore the key to a fast triangle counting implementation is to have a fast method for identifying common neighbors of two nodes.

In order to identify common neighbors in a large real-world graphs, however, one has to consider the power-law degree distribution of such graphs. Although there exist only a handful of nodes whose degree is very large, it takes a long time to identify common neighborhoods between such a high degree node and another node. Conversely, the high degree nodes have a lot of two-hop neighbors and need to be processed many times for identifying common neighborhoods with those many small-degree nodes.

In the remainder of the section, we propose two methods for alleviating these problems, by taking advantage of some nice characteristics of our CSR graph representation described in Section 2.3.

3.1 Hybrid Linear Scan and Binary Splitting

In our CSR representation, neighborhood information for each node is stored in a sorted adjacency array. Therefore the common neighborhood finding problem is reduced into finding common elements in two sorted arrays, $Adj(u)$ and $Adj(v)$. Since these two arrays are already sorted, all the common elements can be identified with a single linear scan on both arrays, as in merge-sort. This takes only O(d_u + d_v). The original `EdgeIterator` implementation by Schank and Wagner took this approach.

However, this naive approach performs badly when there is a large difference between sizes of the two adjacency arrays, i.e., when identifying common neighbors between a high-degree node and a low-degree node. This is because the larger array may need to be scanned entirely, no matter how small the other array is.

```
void do_hybrid(node_t adj_v[], node_t adj_u[]) {
 // let adj_v be the larger array
 if (adj_v.size() < adj_u.size())
     swap(adj_u, adj_v);

 if (adj_u.size() <= 0) return;

 // switch to linear search
 if ((adj_v.size() < α*adj_u.size()) ||
     (adj_v.size() < β))
     linear_scan(adj_v, adj_u); return;
 }
 // pick middle element in one array
 // do binary search in the other
 size_t idx_v = adj_v.middle();
 size_t idx_u = binary_search(adj_u,adj_v[idx_v]);
 if (adj_v[idx_v] == adj_u[idx_u]) {
    found_common_element(adj_v[idx_v]);
    idx_u++;
 }
 idx_v++;
 // recurse
 do_hybrid(adj_v.left_to(idx_v-1), // exclusive
           adj_u.left_to(idx_u-1));
 do_hybrid(adj_v.right_from(idx_v), // inclusive
           adj_u.right_from(idx_u));
}
```

Figure 5: Hybrid linear scan and binary splitting common neighbor iterator implementation

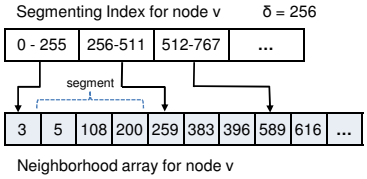


Figure 6: Segmenting Index

In order to address such cases, we adopted another method, namely *binary splitting*. In this method, the search for a common neighbor starts with choosing the middle element of the larger array and doing a binary search on the smaller array. This splits both arrays in two and the procedure is then recursively applied to each of the sub-arrays.

Binary splitting is efficient when the size of one array is significantly larger than the size of the other array, as it can prevent some array elements from being read altogether. On the contrary, when two arrays are similar in size, this technique may result in reading certain array elements multiple times and thus execution time becoming higher than that of a linear scan.

Consequently, we end up with a hybrid solution combining the best features of both basic techniques, as presented in in Figure 5. There are two control parameters that affect the behavior of the hybrid technique: α controls to what extent a difference between two adjacency array sizes affects selection of one of the basic techniques and β controls how the selection is affected by the absolute array size. That is, if the larger of the two arrays is less than α times bigger than the smaller, we do linear scan, concluding that they are similar in size. If the size of the larger array is smaller than β, we also do the linear scan, since simpler instruction flow of linear scan tends to work better when both arrays are very small.

3.2 Segmenting Index

It has also been proposed that one can build up a hashed container for every adjacency array, supporting O(1) node lookup time in the container [22]. This way, when computing common neighbors of two nodes, one can iterate over the adjacency array of one node and look up visited nodes in the hashed container. This bounds the time to perform common neighbor search for nodes u

```
void common_nbr(node_t v, node_t u) {
  if (v.degree()<u.degree()) swap(v,u);

  if (v.degree()<α*u.degree() ||  // similar size
      v.degree()<β)
     linear_scan(v.nbrs(), u.nbrs());
  else if (v.degree()>γ)
     search_by_index(v.get_index(),u.nbrs());
  else
     do_hybrid(v.nbrs(),u.nbrs());
}
```

Figure 7: Common neighbor iterator implementation with segmenting index

and v to O($min(d_u, d_v)$), where d_u and d_v are degrees of nodes u and v, respectively. However, it requires huge amounts of memory to build up hashed container for every adjacency list in a large graph.

To address this issue, we adopted two simple approaches.

First, we take into consideration the power law degree distribution of the real-world graphs and we construct the index only for nodes whose degree is larger than a certain threshold specified as the control parameter γ. We identify these high-degree nodes and build the index structure only once at the beginning of the algorithm.

Second, as for the implementation of the index structure, we use a memory-efficient data structure rather than using a general hash table (e.g. `unordered_map` from C++ standard library). Specifically, we use *segmenting index* (Figure 6) which takes advantage of the fact that the neighborhood array is already sorted and that the value range of the array is fixed as $0 - N$.

These two observations allow us to segment the adjacency array of v such that each segment contains elements within a certain range specified by the control parameter δ. We then create an additional array structure of size ($\lceil N/\delta \rceil + 1$) that points to the beginning of each segment (Figure 6). The look-up time using this index is then bound to O($log\delta$). In Section 6 we will demonstrate that this index structure indeed improves performance while using much less memory than using a general `unordered_map` from the C++ standard library.

Figure 7 presents the final implementation of our common neighborhood iteration method, which combines both the hybrid solution and the segmenting index. If the adjacency arrays of the two nodes are similar in size, even when any of the two nodes has an index structure, we *do* linear scan. If that is not the case, we use index structure of the larger array if present, or revert to the hybrid search scheme otherwise.

As a note, the `degree()` method in the figure is retuning the size of the adjacency array of the node after the *pruning* optimization is applied – the *pruning* optimization will be described in Section 4.

4. ALGORITHMIC EXTENSIONS

In this section we present two extensions to the baseline `CombinedIterator` algorithm (Section 2.2) and explain how these extensions achieve the theoretic lower bound of the algorithm's complexity.

The first extension is to perform early *pruning* of nodes which have no chance of forming a triangle. Please note that nodes visited by the `CombinedIterator` algorithm conform to the following invariant: $w > u > v$ (Figure 3). Also note that in our CSR representation the adjacency list of each source node is sorted by `nid` of the destination nodes (Section 2.3). Consequently, when searching for common neighbors of nodes u and v, we do not consider nodes whose `nid` is smaller than or equal to the `nid` of node u at all. Instead, we prune such nodes from both arrays at the beginning of the search. And because, adjacency lists are stored in sorted arrays, such pruning can be completed in O($log\ d_u$ + $log\ d_v$).

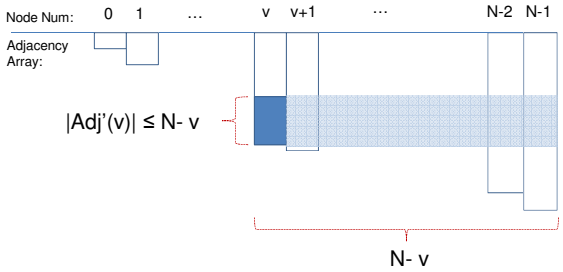


Figure 8: Visualization of the proof of Theorem 4: Note that $|Adj'(v)|(N - v)$ = (the shaded area) $\leq$ (Sum of white boxes' area) = E

The second extension is to *sort* the nodes of the graph by their degree in ascending order. In terms of our CSR representation, we reassign the `nid`s such that for all nodes u' and v' in the sorted graph, if $u' \leq v'$ then $|Adj(u')| \leq |Adj(v')|$. Thus after the sorting, the node with `nid` 0 has the lowest degree, and `nid` $N - 1$ the highest.

The actual implementation of this procedure involves not only sorting of the `begin` array but also re-sorting each of the adjacency list segment in the `edge` array, because the `nid` assignment has been changed. The time complexity for these operations is O($N \log N$) and O($E \log E$), respectively. Although this sorting operation may look expensive at first, it turns out that the sorting leads to dramatic performance improvement, when combined with early-pruning. In fact, we can prove that they lead to the computational lower-bound of triangle listing problem, through the following theorem:

THEOREM 1. *Let, G be a sorted graph, v a node in G, $Adj'(v)$ the adjacency array of v after applying early-pruning, and $|Adj'(v)|$ the length of $Adj'(v)$.*

Then, $|Adj'(v)| < \sqrt{E}$, $\forall v \in G$, where E is the number of edges in G

PROOF. We use Figure 8 as a visual help for carrying out the proof. The figure presents a degree-sorted (from left to right) graph where each node's neighborhood data is represented as a white vertical box depicting a `nid`-sorted adjacency array Please note that E, that is the number of edges in the graph, can be visualized as the sum of all the white boxes.

Let us consider node v in the graph. By the pruning invariant $w > u > v$, the pruned adjacency array $Adj'(v)$ contains only nodes with `nid`s greater than v. Thus it contains at most $N - v$ elements ($|Adj'(v)| < N - v$), since there are at most $N - v$ nodes whose `nid` is greater than v, and there can be at most one edge between each pair of nodes in a simple graph. Furthermore, since the graph is sorted, there exists $N - v$ arrays for all nodes n whose `nid`s are greater than v, such that $|Adj'(v)| \leq |Adj(v)| \leq |Adj(n)|$

Therefore:

$$|Adj'(v)|^2 \leq (N - v)|Adj'(v)| \leq \sum_{n=v}^{N-1} |Adj(n)| \leq E$$

□

Consequently, the total computation time of the `CombinedIterator` algorithm (even with Linear-Scan) becomes $O(E\sqrt{E}) = O(E^{3/2})$, reaching the theoretical lower bound of triangle listing problem [22].

Note that applying early pruning on a sorted real-world graph has the same effect of eliminating nodes with the highest degree skew

(i.e nodes having $O(N)$ neighbors). More importantly, real-world large-scale graphs are typically very *sparse*, that is $E = d_{avg}N$, where d_{avg}, the average degree of the graph, is some small number (e.g. 100). In such a case, the above bound of triangle counting becomes $O(N^{3/2})$, which is even lower than the complexity of counting triangles without explicit listing: $O(N^{2.3729})$ (Section 7).

5. COMPILER-ENABLED OPTIMIZATION AUTOMATION

In this section, we discuss how the algorithmic extensions described in Section 4 can be automatically applied by an optimizing compiler. For this purpose, we use the Green-Marl DSL [15] to express the triangle counting algorithms, and rely on our Green-Marl compiler to generate an efficient C++ implementation of the given algorithm.

First, Green-Marl's compiler automatically transforms the `NodeIterator++` algorithm (see Figure 2) into the `CombinedIterator` algorithm (see Figure 3) by identifying the following pattern:

```
Foreach(x:y.Nbrs) {
  If (x.IsNbr(z)) { // z and x can be exchanged
    // body statements
} }
```

and converting it to use the common neighbor iterator:

```
Foreach(x:CommonNbrs(y,z)) {
    // body statements
}
```

The compiler can also identify the following edge iteration pattern:

```
Foreach(e:G.Edges) {
  Node x = e.FromNode();
  Node y = e.ToNode();
  // body statements
}
```

and transforms it into the node iteration pattern:

```
Foreach(x:G.Nodes) {
  Foreach(y:x.Nbrs) {
    Edge e = y.ToEdge(); // edge from x to y
    // body statements
} }
```

Consequently, both `EdgeIterator` (with proper filtering expressions for total ordering) and `NodeIterator++` algorithms can be automatically transformed into the `CombinedIterator` algorithm, which can take advantage of our fast common neighbor iterator (Section 3).

Next, the Green-Marl compiler automatically parallelizes the code of the `CombinedIterator` (Figure 3). In particular, the compiler parallelizes the outermost loop and generates a C++ program utilizing the OpenMP [9] `parallel for` pragma with chunking. See [15] for further information about parallelization strategies in Green-Marl.

During code generation, The compiler maps the `CommonNbrs` iteration into our backend C++ library which implements our common neighbor identifying method described in Section 3. The statements inside the `CommonNbrs` loop become a lambda function that is inlined with our C++ common neighbor iterator implementation.

In addition, the compiler also automatically applies the *pruning* optimization (Section 4). Specifically, the compiler identifies common neighbor iteration routines with filter expressions:

```
// EXP: any expr, not-dependent on w or loop-body
Foreach(w:CommonNbrs(u,v))(w > EXP){
    // loop-body
}
```

The compiler then checks if the filter expression dictates the minimum (or maximum) value of the common neighbor. If so, the compiler inserts the early pruning operation in the generated code:

```
// the generated C++ code
commonNbrIterator I(u,v);//an encapsulation of Sec. 3
I.setMinValue(EXP); // early pruning
I.iterate(
   // loop-body becomes a lamdba function
);
```

Finally, the compiler automatically inserts code that checks the graph to see if there is already an index structure for common neighbor identification and builds one if it doesn't exist. However, the compiler does not automatically insert code for sorting the graph. Instead it relies on the user to trigger sorting - see Section 6 for the motivation of this decision.

6. EXPERIMENTS

In this section, after discussing the experimental methodology, we present performance comparisons between different triangle counting solutions. Then we provide an in-depth analysis regarding the performance impact of each of our techniques explained in Section 3 and Section 4.

6.1 Methodology

We ran all our experiments on machines featuring two different types of architectures, x86 and SPARC. The numbers for the SPARC architecture were generated on a T5-8 machine (8x16 3.6 GHz cores with 8 virtual threads each, 4TB of RAM) running Solaris 11. The numbers for the x86 architecture were generated on a cluster of 16 Intel Xeon E5-2660 (Sandy Bridge) machines (2x 8 hyper-threaded 2.2GHz cores, 264GB of RAM) running 64-bit SMP Linux 2.6.32.

We ran our experiments on a variety of large real-world graphs. Table 1 describes a short description of each graph along with its origin and some characteristics. Prior to running experiments, all graphs have been converted to their undirected versions as described in Section 2.3.

6.2 Performance Comparisons

The main goal of our experimental analysis was to compare the performance of our graph counting implementation with the performance of the existing state-of-the-art solutions. The experimental data we collected is presented in Table 2. The execution times presented in the table are split into three categories:

- **Loading** – time to load the graph fully into memory (applicable only to solutions supporting in-memory analysis)
- **Pre-proc.** – time to perform solution-specific graph preprocessing tasks (see below for details)
- **Computing** – time to run the actual algorithm

In the first two data columns, we present execution times for our implementation of the `CombinedIterator` algorithm, on a single node of the x86 cluster and on the T5-8 SPARC machine, respectively, featuring the early-pruning and degree-sorting optimizations. In case of our implementation, graph loading time includes conversion to the undirected graph and pre-processing time includes sorting graph nodes by their degree. The reported execution times are an average from three runs – the observed variance between runs was below 2%.

Graph Name	Number of Nodes	Number of Directed Edges	Number of Undirected Edges	Number of Triangles	Description	Data Source
Patents	3,774,768	16,518,948	33,037,895	7,515,023	citation graph for U.S. patents	[5]
LiveJournal	4,848,571	68,993,773	86,220,856	285,730,264	social relations in an on-line community	[5]
Wikipedia	15,172,740	130,166,252	247,079,325	358,442,188	links in English Wikipedia	[3]
Twitter-2010	41,652,230	1,468,365,182	2,405,206,390	34,824,916,864	Twitter user profiles and social relations	[17]
UK-2006-05	77,741,046	2,965,197,340	5,285,526,791	363,786,546,359	monthly snapshot of the .uk domain	[4]

Table 1: Real-world graph datasets used in the experiments.

Noticeably, all of our execution times shown in the table are extremely short. Even with the largest UK-2006-05 instance, a single x86 execution took only 64 seconds to count 363 billions triangles, with additional 18 seconds spent for preprocessing. With a T5 machine that features many more (but weaker) cores, the computation time is even shorter: 26 seconds. Overall, T5 execution is 1.14x $\sim$ 6.97x faster than x86 execution for computation phase. This is mainly due to massive amount of cores of the T5. On the other hand, preprocessing time on T5 has been increased since our sorting algorithm is not fully parallelized; single T5 core performed less than x86 core despite of its higher clock frequency.

Next, we compare our performance with GraphLab2 [12], an open-source distributed graph processing engine. To the best of our knowledge, the triangle counting implementation provided in GraphLab2's built-in package, has been known as the fastest (distributed or single-machine) *in-memory* implementation.

In the middle three data columns (grouped together) of Table 2, we present execution times of the GraphLab2's implementation. We downloaded the source code of GraphLab2 from their web site [1], compiled it using a script provided with the engine distribution (using g++ 4.8.1), and ran the built-in triangle counting algorithm with all the graphs using 1, 2 and 16 machines of our x86 cluster, exploiting all the cores of all the machines. We followed the recommendations of the GraphLab2 authors with respect to the remaining parameters settings. Especially, when running on a single machine, all the cores in the system were wholly dedicated to computation.

In the following discussion regarding GraphLab2's performance, we only consider time for *computing* phase for the sake of fair comparison. Note that this consideration is in favor of GraphLab2. As for *loading* time, we used a custom binary file format for our system, but a text-based one (i.e. *edgelist*) for GraphLab2 due to compatibility issues. As for *preprocessing* time, while we counted their *finalizing* step as preprocessing, GraphLab2 may consider issues related to distributed computation, which we do not.

We first compare the single machine performance by looking at the first and the third data column in Table 2. The numbers show that our implementation is 2.95x to 10.95x faster than GraphLab2's. Note that GraphLab2's performance becomes relatively worse for smaller instances; GraphLab2, being a general framework, has a certain amount of fixed overheads which become harder to compensate with smaller instances. That said, GraphLab2 failed to process the UK-2006-05 instance on a single x86 machine, running out of memory during its finalizing phase.

Next, we consider the cases when multiple machines were used for GraphLab2 execution. Our single x86 execution was still faster than 16 machine execution of GraphLab2 for all graphs other than Twitter-2010 instance. As for the case of Twitter-2010 instance, the break-even point was around 8 machines. Note that the execution time of GraphLab2 with two machines is longer than GraphLab2 with a single machine, as communication overheads are introduced when using multiple machines; it needs several machines to compensate such overhead. Finally, our SPARC execution time are even faster than 16 machine execution of GraphLab2 for all graph instances, including Twitter-2010.

Finally, we consider other triangle counting methods that adopt various out-of-memory techniques [18, 16, 23]. We simply cite the execution times of these methods from their original publications, since these numbers are not for direct comparison, but for general illustration of performance benefit when loading the whole graph into memory.

GraphChi [18] and MGT [16] are external solutions designed for small systems, where only a portion of the graph can be present in-memory at any given time. The authors of MGT [16] proved their optimality of their external algorithm. Still, there is a large inherent performance gap between in-memory solutions versus external solutions. Finally, the Hadoop implementation [23] are significantly slower than those of the other methods, due to the inherent overhead of yet scalable Hadoop framework.

6.3 Performance Analysis

In this section we present an in-depth analysis of the overheads related to graph loading, of parallelization impact, as well as of the implications of applying different optimizations and varying parameters of our methods. For this purpose we focus on the Twitter-2010 graph instance throughout this section.

6.3.1 Graph Loading and Pre-processing

Table 3 shows a break-down of Twitter-2010 loading and preprocessing time. The original graph is directed and thus loading includes both the time it takes to read the data from disk (from a simple binary format which is essentially a disk snapshot of the CSR representation) and the time to create an undirected version of the graph. The loading time is reported for reference purposes only, as it may vary greatly between different on-disk representations. In particular, if the graph is already stored on-disk as an undirected graph, the explicit *undirecting* step would be omitted.

Table 3 also shows the breakdown of preprocessing time as index creation time and graph sorting time. Noticeably most of the preprocessing time is spent on sorting the graph by degree (Section 4). Also note that SPARC takes longer than x86 for the preprocessing phase; sorting and index building have been parallelized only partially while a single SPARC core is weaker than a single x86 core. We are currently improving parallel implementation of these steps.

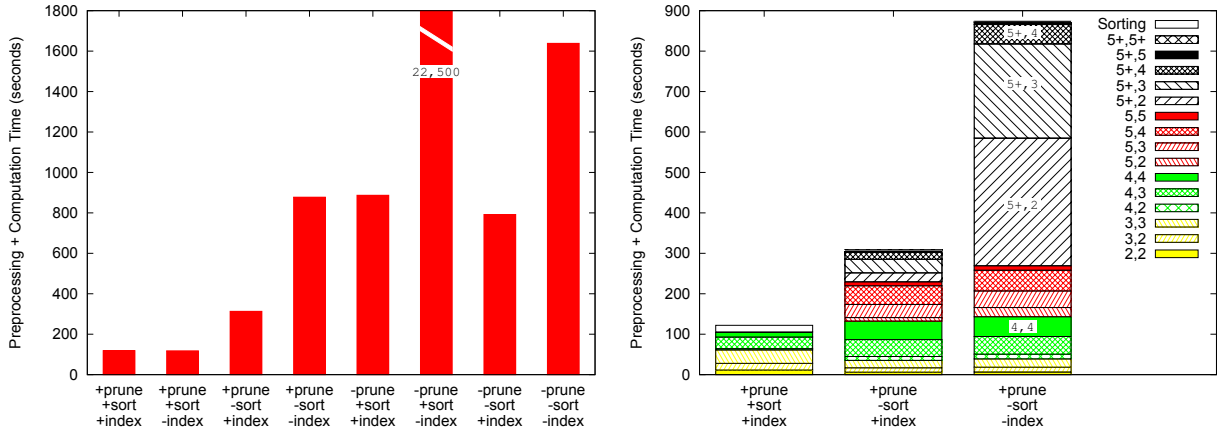
6.3.2 Impact of Optimizations

In Figure 9(a) we present execution times (preprocessing time plus computation time) on a single node of the x86 cluster when only some combinations of the three optimizations described in Section 3 and Section 4, that is *pruning*, *sorting* and *index*, are enabled. For example, the combination `+prune,-sort,+index` means *pruning* and *index* are applied but *sorting* is not.

When comparing `+prune` and `-prune` configurations, one can observe that applying *pruning* was always beneficial. Similarly, adding *index* optimization also improved execution time except for when *sorting* and *pruning* was already applied, which overall formed the best performing configuration, with no performance impact from the *index* optimization in this case. On the other hand, applying *sorting* was beneficial only when it was applied together with *prun-*

		Ours (X86)†	Ours (SPARC)†	GraphLab2 [12]†			GraphChi [18]	MGT [16]	Hadoop [23]
Environment	CPU type	Intel Xeon	SPARC	Intel Xeon			Intel i5	Intel	N/A
	(Model)	E5-2660	T5-8	E5-2660			N/A	N/A	N/A
	CPU Freq.	2.2GHz	3.6GHz	2.2 GHz			2.5GHz	3 GHz	N/A
	Cores x HTs	16*2	128*8	16*2			2*1	N/A	N/A
	DRAM	264 GB	4 TB	264 GB			N/A	N/A	N/A
	# Machines	1	1	1	2	16	1	1	**1636**
Patents (304MB)	Loading	0.69	0.93	7.54	7.74	7.83	-	-	-
	Pre-proc.	1.35	0.55	7.38	6.85	3.25	-	-	-
	Computing	0.16	0.11	1.75	3.25	2.04	-	-	-
LiveJournal (736MB)	Loading	1.30	1.78	17.66	18.04	18.10	-	(included)	(included)
	Pre-proc.	1.47	1.17	13.81	11.04	4.20	-	(included)	(included)
	Computing	0.76	0.26	3.34	5.67	3.24	-	20	319.8
Wikipedia (2.06GB)	Loading	2.82	4.50	54.27	54.91	55.14	-	-	-
	Pre-proc.	3.73	2.23	42.95	35.96	12.02	-	-	-
	Computing	2.48	1.19	10.46	13.31	6.81	-	-	-
Twitter-2010 (18.6GB)	Loading	26.8	41.77	469.77	477.08	479.31	(included)	-	(included)
	Pre-proc.	16.7	35.19	469.79	238.03	51.92	600	-	(included)
	Computing	101.3	14.54	299.28	220.61	62.50	3,600	-	25,380
UK-2006-05 (40.5GB)	Loading	30.1	60.37	N/A	1059.49	1054.71	-	(included)	-
	Pre-proc.	18.8	28.12	N/A	492.27	80.54	-	(included)	-
	Computing	63.8	25.37	N/A	348.04	124.09	-	1000	-

Table 2: Execution times (in seconds) for different triangle counting implementations: columns marked with † represent execution times we measured ourselves, while other numbers are cited from their original publications. The size of graph when expressed in CSR format with 4B `nid`, is shown under each graph name.



(a) (b)

Figure 9: Effect of various optimizations (a) and breakdown of execution time per adjacency arrays sizes (b)

		x86	SPARC
Loading	File Reading	3.6	26.8
	Undirecting	23.2	14.9
Pre-proc.	Index Creation	0.3	7.35
	Sorting	16.4	27.84

Table 3: Loading and Pre-processing time (in seconds)

ing – otherwise it was detrimental to performance. Figure 9(b) provides some additional insight into this performance phenomenon.

Figure 9(b) presents an alternative way of viewing the optimizations impact. Similarly to Figure 9(a), it shows breakdown of the execution time when counting Twitter-2010 graph triangles on a single x86 machine, but each of the three total execution times depicted in the figure is broken down into segments representing the execution time it took to find a common neighbor (and thus a triangle) by analyzing pairs of pruned adjacency arrays of a given size. For example, in the $(4, 3)$ segment, number 3 denotes the size of one adjacency array to be in the $10^2 - 10^3$ range and number 4 denotes the size of the other adjacency array to be in the $10^3 - 10^4$ range (number 2 corresponds to sizes smaller than 100 elements and $5+$ corresponds to sizes larger than 10^5).

Let us first consider the case when only the pruning optimization was applied (the `+prune,-sort,-index` configuration). Clearly, most of the execution time is spent when one array is large (e.g. $5+$) and the other is small (e.g. 2). As one can observe, adding the index structure dramatically reduces the execution time in this case (`+prune,-sort,+index`). Adding the sorting optimization further reduces the execution time (`+prune,+sort,+index`), as all large arrays are then pruned to $O(\sqrt{E})$ size which makes their size drop below the value of the γ control parameter (see Section 3.2) and the index never be used in this configuration. We have also observed, that applying sorting without applying pruning can harm overall performance by causing poor load balancing between multiple cores, which effectively negates all benefits of the parallel execution.

Note that the `-prune,-sort,-index` configuration would correspond to a parallel version of the baseline `EdgeIterator` algorithm (no hash) in Schank and Wagner's [22]. Similarly, the `-prune,-sort,+index` configuration would correspond to a parallel version of the `EdgeIteratorHashed` by the same authors. These figures thereby show that our contribution is not simply providing a parallel implementation of conventional algorithms but adding a significant algorithmic improvement as well. [1]

[1]Even for implementation, a naive parallel implementation of `EdgeIterator` and `EdgeIteratorHashed` would perform worse

6.3.3 Parameter Sensitivity Analysis

As described in Section 3, our implementation of the common neighbor iterator uses four control parameters: α, β, γ, and δ. In Figure 10 we illustrate the impact of choosing different parameter values on the performance of the triangle counting algorithm running on a single node of the x86 cluster. The figure marks with circles the values of all the parameters that we ultimately used as default in our implementation (and used when running all other experiments described in this section): 40 for α, 256 for β, 1024 for γ and 65536 for δ.

Figures 10(a)-(b) show the effect of different values of the α and β parameters on the `+prune,-sort,-index` configuration. These parameters control when the system chooses binary splitting over linear scan when processing adjacency arrays during common neighbor iteration. When values of α or β are too large, the system starts favoring linear scan over binary splitting even when processing a small array with a really large array and thus leads to significantly increased execution time. At the same time, if the values of these parameters are too small then binary splitting is invoked even in cases when its higher constant cost negates all possible advantages over linear scan.

Figures 10(c)-(d) show the effect of different values of the γ and δ parameters on the `+prune,-sort,+index` configuration. These parameters control for which nodes the system will construct an index (γ) and what the size of the index segments will be (δ), and different values of these parameters can affect not only the execution time of the algorithm (pre-processing time and computation time) but also the amount of memory consumed by the index structure.

Figure 10(d) demonstrates that smaller values of δ lead to execution time improvements at the cost of increased memory usage. However, the total amount of memory consumption can be limited by choosing the value of γ appropriately, as illustrated in Figure 10(c).

In Figure 10(c) we also show the execution time and memory consumption when using `unordered_map` from the C++ standard library as the index structure instead of the segmenting index. As one can observe, the segmenting index (with δ = 1024) consumes 10x less memory while still offering performance superior to that of the `unordered_map`.

6.3.4 On Graph Indexing

In Section 6.3.2 we concluded that the index structure is not being used when pruning and sorting optimizations are already applied, due to (pruned) adjacency list sizes dropping to below a threshold enabling use of the index. For certain situation it still makes sense to create an index, e.g. to solve the local triangle counting problem, a variant of the local triangle counting problem where the task is to only count triangles which include a given node. In this case, we cannot rely on the fact that the same triangle has been already counted when processing lower-numbered nodes.

7. RELATED WORK

Over the years, many variants of triangle counting and listing algorithms have been developed. Schank and Wagner [22] surveyed and compared several of the most popular ones, including the `NodeIterator` and `EdgeIterator` algorithms described in Section 2.2. They also proposed `Forward` algorithm that achieves O($E^{3/2}$) (where E is the number of edges in the graph) lower bound. However, their experiments showed that it did not performed as well as the `EdgeIterator` algorithm for large graphs, since it relies on a dynamic data structure modified during the algorithm run, which leads to a relatively high constant overhead. Moreover, such a single, dynamic data structure does not suit for parallel execution. To the contrary, our method achieves the O($E^{3/2}$) lower bound with small constant factors and with parallelism trivially enabled.

Theoretically, it is possible to solve the triangle counting problem without solving the triangle listing problem [6]. That is, the number of triangles can be computed by multiplying the adjacency matrix of an input graph (by itself) two times, summing up the diagonal elements of the result, and dividing the resulting number by two. This method provides theoretic bound of O($N^{2.3729}$) (where N the number of nodes), same as that of matrix multiplication. Although this bound seems lower than aforementioned O($E^{3/2}$) bound, it is actually higher for the case of large real-world graphs where O(E) equals to O(N). Moreover, fast matrix multiplication typically requires O(N^2) memory, which is impossibly huge even with moderate size graph instances.

The triangle counting problem can also be solved in a distributed setting [23, 26, 12]. Suri and Vassilvitskii used Hadoop to count triangles in the Twitter-2010 graph but their implementation suffered from a very high overhead, as reported in Section 6. Walkauskas [26] demonstrates that the HP Vertica distributed database system [2] can count the number of triangles in the LiveJournal graph in 90 seconds on four 12-core machines. It takes our implementation less than 5 seconds to perform the same task (including loading time, pre-processing time and computation time). GraphLab2 (a.k.a. PowerGraph) is a recently developed distributed graph processing engine [12]. Although GraphLab2 can solve the triangle counting problem much faster than other distributed frameworks, it is still outperformed by our in-memory solution, as we demonstrated in Section 6.

Several external (where the input graph is not fully loaded into memory at any instance of time) algorithms to solve the triangle counting problem have also been proposed in the literature [10, 16, 18]. Chu and Cheng presented two fast external algorithms [10] then followed by Hu et al. which presented an external algorithm that achieves the asymptotic optimal lower-bound [16]. Kyrola et al. also showed that their GraphChi graph processing framework can solve the triangle counting problem on the large Twitter-2010 graph instance with a small PC [18]. In contrast to these approaches, we focus on in-memory graph processing, as it delivers performance unmatched by the external algorithms while still easily handling large graphs with billions of edges.

There also exist approaches that estimate the number of triangles in a given graph rather than providing the exact count [24, 25, 21]. Although these methods can achieve fairly high accuracy, they clearly cannot be used if the exact number of triangles is the desired output and, perhaps more importantly, they cannot be used to solve the triangle listing problem. Note that the execution time to compute the exact answer with our method for the twitter graph is less than the time reported in a parallel, approximate method [21].

8. CONCLUSION

In this paper, we presented a fast, exact, parallel, in-memory solution for the triangle listing and thereby the counting problem. We prove that our technique achieves the known computation bound. Our experiments confirm that our implementation is faster than any other existing in-memory, distributed, or external methods. We believe that our findings can be used to enhance distributed solutions as well.

than our `-prune,-sort,-index` and `-prune,-sort,+index` configuration, respectively, since they would not get benefits from our fast common neighborhood iterator implementation in Section 3.1. Moreover, naive implementation of hash collection would blow up the memory for large graph instances. See our parameter sensitivity analysis in 6.3.3 for details.

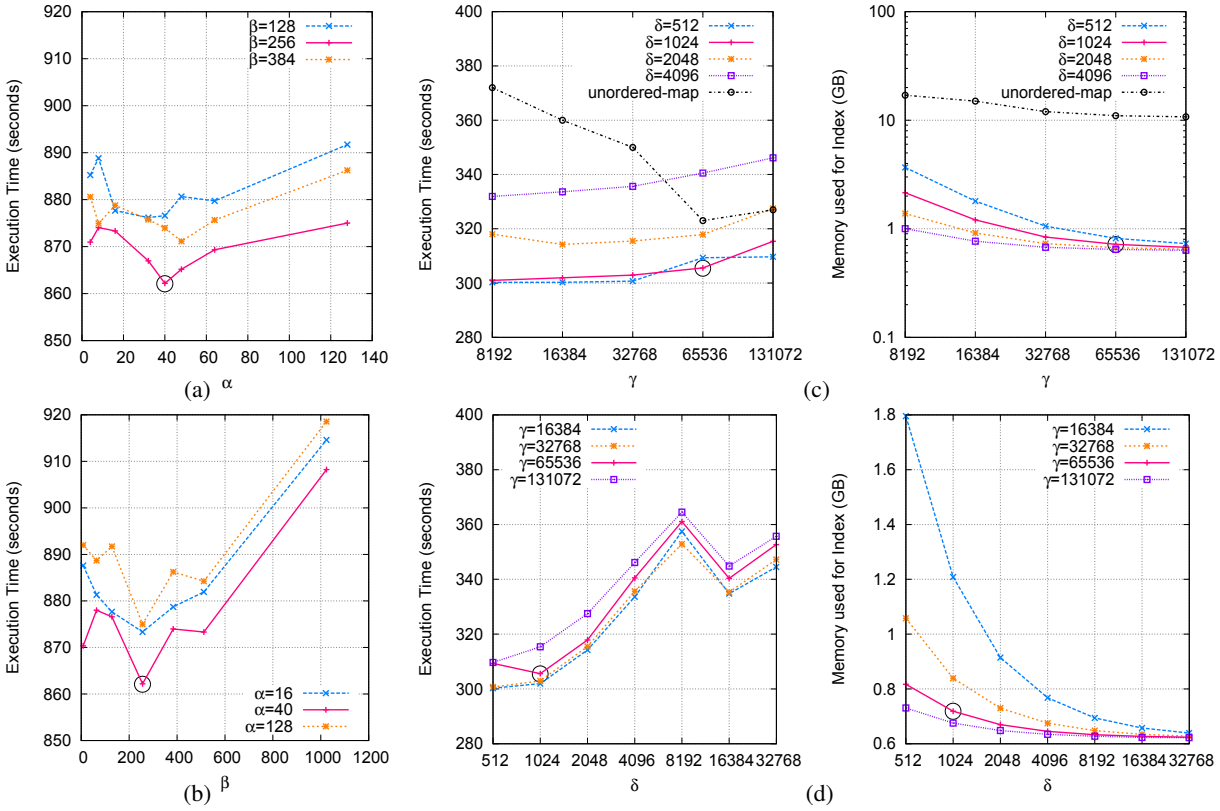


Figure 10: Parameter sensitivity analysis – a circle marks the parameter value we have ultimately chosen

9. REFERENCES

[1] Graphlab. http://graphlab.org.

[2] HP Vertica. http://www.vertica.com.

[3] Koblenz network collection. http://konect.uni-koblenz.de.

[4] Laboratory for web algorithmics. http://law.di.unimi.it/datasets.php.

[5] Stanford network analysis library. http://snap.stanford.edu/snap.

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

[7] A.-L. Barabási and R. Albert. Emergence of scaling in random networks. *science*, 286(5439):509–512, 1999.

[8] L. Becchetti, P. Boldi, C. Castillo, and A. 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*, 2008.

[9] O. A. R. Board. OpenMP application program interface. `http://www.openmp.org/mp-documents/OpenMP_4.0_RC2.pdf`, 2013.

[10] S. Chu and J. Cheng. Triangle listing in massive networks and its applications. In *Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 672–680. ACM, 2011.

[11] F. Färber, S. K. Cha, J. Primsch, C. Bornhövd, S. Sigg, and W. Lehner. Sap hana database: data management for modern business applications. *ACM Sigmod Record*, 40(4):45–51, 2012.

[12] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2012.

[13] Y. Guo, Z. Pan, and J. Heflin. Lubm: A benchmark for owl knowledge base systems. *Web Semantics: Science, Services and Agents on the World Wide Web*, 3(2):158–182, 2005.

[14] P. Gupta, A. Goel, J. Lin, A. Sharma, D. Wang, and R. Zadeh. Wtf: The who to follow service at twitter. In *Proceedings of the 22nd international conference on World Wide Web*, pages 505–514, 2013.

[15] S. Hong, H. Chafi, E. Sedlar, and K. Olukotun. Green-Marl: A DSL for Easy and Ffficient Graph Analysis. In *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2012.

[16] X. Hu, Y. Tao, and C.-W. Chung. Massive graph triangulation. In *Proceedings of the 2013 ACM SIGMOD international conference on Management of data*, 2013.

[17] H. Kwak, C. Lee, H. Park, and S. Moon. What is Twitter, a social network or a news media? In *Proceedings of the 19th international conference on World wide web*, 2010.

[18] A. Kyrola, G. Blelloch, and C. Guestrin. Graphchi: Large-scale graph computation on just a pc. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 31–46, 2012.

[19] R. Milo, S. Shen-Orr, S. Itzkovitz, N. Kashtan, D. Chklovskii, and U. Alon. Network motifs: simple building blocks of complex networks. *Science*, 298(5594):824–827, 2002.

[20] M. E. Newman. The structure and function of complex networks. *SIAM review*, 45(2):167–256, 2003.

[21] A. Pavan, K. Tangwongan, and S. Tirthapura. Parallel and distributed triangle counting on graph streams. Technical Report RC25352, IBM Research, 2013.

[22] T. Schank and D. Wagner. Finding, counting and listing all triangles in large graphs, an experimental study. In *Experimental and Efficient Algorithms*, pages 606–609. Springer, 2005.

[23] S. Suri and S. Vassilvitskii. Counting triangles and the curse of the last reducer. In *Proceedings of the 20th international conference on World wide web*. ACM, 2011.

[24] C. E. Tsourakakis. Fast counting of triangles in large real networks without counting: Algorithms and laws. In *Data Mining, 2008. ICDM'08. Eighth IEEE International Conference on*, 2008.

[25] C. E. Tsourakakis, U. Kang, G. L. Miller, and C. 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.

[26] S. Walkauskas. Counting triangles. http://www.vertica.com/2011/09/21/counting-triangles/.

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