Multicore Triangle Computations Without Tuning

Julian Shun[1], Kanat Tangwongsan[2]
[1]*Computer Science Department, Carnegie Mellon University, USA*
[2]*Computer Science Program, Mahidol University International College, Thailand*
jshun@cs.cmu.edu, kanat.tan@mahidol.edu

Types of thread: main, ad-hoc, pooled

***Abstract*—Triangle counting and enumeration has emerged as a basic tool in large-scale network analysis, fueling the development of algorithms that scale to massive graphs. Most of the existing algorithms, however, are designed for the distributed-memory setting or the external-memory setting, and cannot take full advantage of a multicore machine, whose capacity has grown to accommodate even the largest of real-world graphs.**

This paper describes the design and implementation of simple and fast multicore parallel algorithms for exact, as well as approximate, triangle counting and other triangle computations that scale to billions of nodes and edges. Our algorithms are provably cache-friendly, easy to implement in a language that supports dynamic parallelism, such as Cilk Plus or OpenMP, and do not require parameter tuning.

On a 40-core machine with two-way hyper-threading, our parallel exact global and local triangle counting algorithms obtain speedups of 17–50x on a set of real-world and synthetic graphs, and are faster than previous parallel exact triangle counting algorithms. We can compute the exact triangle count of the Yahoo Web graph (over 6 billion edges) in under 1.5 minutes. In addition, for approximate triangle counting, we are able to approximate the count for the Yahoo graph to within 99.6% accuracy in under 10 seconds, and for a given accuracy we are much faster than existing parallel approximate triangle counting implementations.

I. Introduction

As graphs are increasingly used to model and study interactions in a variety of contexts, there is a growing need for graph analytics to process massive graphs quickly and accurately. Among various metrics of interest, the triangle count and related measures have attracted a lot of recent attention because they reveal important structural information about the network being studied. Unsurprisingly, triangle counting and enumeration has seen applications in the study of social networks [37], identifying thematic structures of networks [18], spam and fraud detection [6], link classification and recommendation [58], joining three relations in a database [38, 40], database query optimization [5]—with further examples discussed in [7, 25, 40].

Driven by such applications, several algorithms have been proposed for the distributed setting (e.g., [2, 16, 21, 42, 53, 59]) and the external-memory setting (e.g., [15, 17, 25, 28, 33, 36, 40]) as graphs of interest were deemed too big to keep in the main memory of a single computer. The distributed algorithms are not tailored for a multicore machine, and the external-memory algorithms typically do not support parallelism (with the exception of [28, 33]). Today, however, a single multicore machine can have tens of cores and can support several terabytes of memory[1]—capable of storing graphs with tens or even hundreds of billions of edges. Compared to distributed-memory systems, communication costs are much cheaper in multicore systems, leading to performance benefits with a proper design. Moreover, for graph algorithms, multicores are known to be more efficient per core and per watt than an equivalent distributed system.

In this paper, we develop fast and simple in-memory parallel algorithms for triangle computations. Specifically, we address the following question: *How can we engineer fast multicore algorithms for triangle counting and enumeration that require no parameter tuning?*

Traditionally, shared-memory parallel programs are written with explicitly assigning tasks to threads (e.g., using `pthreads`). However, for programs where there is no clear way to evenly split the work among threads, scheduling for good performance is a big challenge. Triangle computations fall into this category of programs because the work among vertices is often highly skewed due to the large variation in the number of potential triangles incident on each vertex. Therefore, writing efficient triangle computation code with explicit task management is generally very difficult. For these reasons, direct adaption of existing distributed algorithms to the shared-memory setting is unable to take full advantage of a multicore machine.

Another method for writing shared-memory programs is to use simple constructs that indicate which parts of the program are safe to run in parallel, and allow a run-time scheduler to assign work to threads and perform load balancing on-the-fly. This approach is known as *dynamic multithreading*. Using languages such as Cilk Plus, OpenMP, and Intel Threading Building Blocks that support dynamic multithreading, one can write clean programs while letting the run-time scheduler perform the work allocation and load balancing. With advances in scheduling, it is now possible to write a wide class of parallel programs in this framework that are efficient, both in theory and in practice [12], without having to tune the program to achieve balanced workloads.

Recognizing these benefits, we design our algorithms in the dynamic multithreading framework. Here, a parallel computation can be thought of as a directed acyclic graph (DAG), where vertices represent instructions and edges represent dependencies between instructions. This gives an abstraction in which an algorithm's description exposes the logical parallelism without reference to work allocation or threads. In this setting, the theoretical cost of a program is captured by the total number of operations performed (known as *work* or W) and the longest chain of dependencies (known as *depth* or D), where W/D indicates the amount of parallelism available.

In addition to parallelism, the cache behavior of programs has a significant impact on performance. Writing parallel programs with good cache behavior has often required expertise. Because machines differ, this often requires fine-tuning code

[1]For example, the Intel Sandy Bridge-based Dell R920 can be configured with up to 40 cores and 6 Terabytes of memory.

Algorithm	Work	Depth	Cache Complexity
TC-Merge	$O(E^{3/2})$	$O(\log^{3/2} E)$	$O(E + E^{3/2}/B)$
TC-Hash	$O(V \log V + \alpha E)$	$O(\log^{3/2} E)$	$O(\text{sort}(V) + \alpha E)$
Parallel-PS	$O(E^{3/2})$	$O(\log^{5/2} E)$	$O(E^{3/2}/(\sqrt{M}B))$

TABLE I: Complexity bounds for triangle counting algorithms, where V = number of vertices, E = number of edges, α is arboricity of the graph, M = cache size, B = cache line size, and $\text{sort}(n) = O((n/B)\log_{M/B}(n/B))$.

or parameters for each individual machine. Even then, it is still difficult to achieve good cache performance because the memory system of a modern machine has become highly sophisticated, consisting of multiple levels of caches and layers of indirection.

To sidestep this complex issue, we design algorithms that make efficient use of caches without needing to know the specific memory/cache parameters (e.g., cache size, cache line size). Such parallel algorithms are known as *parallel cache-oblivious* algorithms, as they are oblivious to cache parameters [9, 20, 52]. Parallel cache-oblivious algorithms free the programmer from optimizing the cache parameters for specific machines, as they run efficiently on all shared-memory machines. These algorithms are analyzed for parallel cache complexity as a function of the problem size n, the cache size M, and the cache line size B.

Overall, we aim for algorithms that have plenty of parallelism (low depth) and also low cache complexity, as well as having a work complexity matching that of the sequential work complexity (i.e., be *work-efficient*). We would also like the good theoretical guarantees of the algorithms to translate into good performance and scalability in practice.

Contributions. This paper presents fast and simple shared-memory parallel algorithms for triangle counting, both exact and approximate, that are able to scale to billions of nodes and edges. The algorithms take full advantage of parallelism in a multicore system via dynamic multithreading and are optimized for the memory hierarchy by being cache-oblivious. Our main contributions are as follows:

—*Parallel Algorithms.* We design parallel algorithms for triangle counting, one which uses merging for intersecting adjacency lists (TC-Merge) and one which uses hashing for intersection (TC-Hash). Our algorithms are based on Latapy's sequential algorithm [34]. We show that the algorithms have good theoretical bounds in the Parallel Cache Complexity (PCC) model [9, 52]. The work, depth, and cache complexity bounds are shown in Table I. We extend our algorithms to approximate triangle counting, directed triangle counting, triangle enumeration, local triangle counting, and computing clustering coefficients. The algorithms are easy to implement and do not require parameter tuning. We also discuss a parallelization of the recent sequential cache-oblivious triangle enumeration algorithm of Pagh and Silvestri [40] (Parallel-PS), obtaining the complexity bounds shown in Table I, which may be of independent interest.

—*Performance Evaluation.* We conduct extensive empirical evaluation on a 40-core Intel machine with two-way hyper-threading as well as a 64-core AMD machine. Our Cilk Plus implementations of the parallel exact global and local triangle counting algorithms achieve speedups of 17–50x and outperform previous algorithms for the same task. On the large-scale Yahoo Web graph (with over 6 billion edges), our algorithm computes the triangle count in under 1.5 minutes. For approximate triangle counting, our parallel implementation approximates the triangle count for the Yahoo graph to within 99.6% accuracy in under 10 seconds, and is much faster than existing parallel approximate triangle counting implementations for a given accuracy.

—*Analysis of Cache Behavior.* To further understand how these performance benefits come about, we analyze the cache performance of our implementations on several graphs, showing that cache performance is consistent with the theory and that cache efficiency is crucial for performance.

II. Background and Preliminaries

Let $G = (V, E)$ be a simple, undirected graph. A *triangle* is a set of three vertices $v_1, v_2, v_3 \in V$ such that the undirected edges (v_1, v_2), (v_2, v_3), and (v_1, v_3) are present in E. The *triangle counting* problem takes an undirected graph G and returns a count of the number of triangles in G. For *triangle listing*, all of the triangles in the graph are output. The *triangle enumeration* problem takes an `emit` function that is called on each triangle discovery (hence, each triangle must appear in memory). Algorithms for *local triangle counting/listing* return the count/list of triangles incident on each vertex $v \in V$.

Define $d(v)$ to be the degree of vertex v and denote by $N(v)$ the set of neighbors of v. When clear from context, we also use V and E to refer to the number of vertices and the number of edges, respectively, in G.

We represent graphs using the *adjacency list* format, which stores for each vertex an array of indices of other vertices that it has an edge to as well as the vertex's degree. We assume that the arrays are stored consecutively in memory. This representation requires $O(V+E)$ space. We assume, without loss of generality, that the graph does not have any isolated vertices (they can be removed within the complexity bounds of the algorithms we describe). The *arboricity* α of a graph is the minimum number of forests its edges can be partitioned into (hence, $\alpha \geq 1$). This is upper bounded by $O(\sqrt{E})$ for general graphs and $O(1)$ for planar graphs [14]. Furthermore, it is known that $\sum_{(u,v)\in E} \min\{d(u), d(v)\} = O(\alpha E)$.

Parallel Computation. The *dynamic multithreading* model represents a parallel computation as a directed acyclic graph (DAG), where vertices represent instructions and edges represent dependencies between instructions. In particular, if there is an edge between vertices u and v, then instruction u must be executed before instruction v in the computation. The runtime scheduler has freedom to execute the computation in any manner as long as it respects the dependencies. To create parallel tasks in a program, one uses the *spawn* procedure (`cilk_spawn` in Cilk Plus), and to synchronize parallel tasks, one uses the *sync* procedure (`cilk_sync` in Cilk Plus). A *parallel for-loop* (`cilk_for` in Cilk Plus and **parfor** in the pseudo-code) indicates that all iterates of the for loop may execute in parallel (it can be implemented using spawn and sync procedures). Cilk Plus uses a randomized work-stealing scheduler [12], where processors that run out of tasks to perform steal tasks from another randomly chosen thread.

Parallel Cost Model. Algorithms expressed in the dynamic multithreading model can be analyzed for theoretical efficiency in terms of work, depth, and cache complexity. The *work* W is equal to the number of operations required (the number of vertices in the computation DAG) and the *depth* D is equal to the number of time steps required on the critical path (the longest directed path in the computation DAG). Then, if P processors are available, using Brent's scheduling theorem [11]

we can bound the running time by $O(W/P + D)$, which can be realized using a randomized work-stealing scheduler [12].

For cache complexity analysis, we use the parallel cache complexity (PCC) model [9, 52], a parallel variant of the cache-oblivious model [20]. A cache-oblivious algorithm has the advantage of being able to make efficient use of the memory hierarchy without knowing the specific cache parameters (e.g. cache size, cache line size). In the PCC model, the cache complexity of an algorithm is given as a function of cache size M and cache line size B, assuming the optimal offline replacement policy. This function reflects how the algorithm behaves for a particular cache/line size, although this information is unknown to the algorithm. For a parallel machine, it represents the number of cache misses across all processors for a particular level (e.g., L2, L3, etc.). An algorithm is analyzed assuming a single level of cache, but since the algorithm is oblivious to the cache parameters, the bounds simultaneously hold across all levels of the memory hierarchy, which can contain both private and shared caches.

Parallel Primitives. We will make use of the basic parallel primitives, prefix sum (scan), filter and merge [11]. *Prefix sum (scan)* takes an sequence A of length n, an associative binary operator $\oplus$, and an identity element $\perp$ such that $\perp \oplus a = a$ for any a, and returns the sequence $(\perp, \perp \oplus A[0], \perp \oplus A[0] \oplus A[1], \ldots, \perp \oplus A[0] \oplus A[1] \oplus \ldots \oplus A[n-1])$. *Filter* takes a sequence A of length n, and a predicate function f, and returns a sequence A' of length n' containing the elements in $a \in A$ such that $f(a)$ returns true, in the same order that they appear in A. Filter can be implemented using prefix sum, and both require $O(n)$ work and $O(\log n)$ depth [11]. *Merge* takes sorted sequences A and B of lengths n and m, respectively, and returns a sorted sequence containing the union of the elements in A and B. It can be implemented in $O(n + m)$ work and $O(\log(n + m))$ depth [11]. Merge can be modified to return the intersection of the elements of two sorted sequences in the same complexity bounds, and this is the version we use in the paper.

We use scan(n) and sort(n) to denote the cache complexity of scanning (prefix sum) and sorting, respectively, on an input of size n. In the PCC model, it has been shown that scan(n) = $O(n/B)$ and sort(n) = $O((n/B)\log_{M/B}(n/B))$, under the standard assumption $M = \Omega(B^2)$, which is readily met in practice. In the PCC model, scan requires $O(n)$ work and $O(\log n)$ depth, and sort requires $O(n \log n)$ work and $O(\log^{3/2} n)$ depth with high probability[2] (or $O(\log^2 n)$ depth deterministically) [9, 10, 52]. Merging two sorted sequences of lengths n and m requires $O(n + m)$ work, $O(\log(n + m))$ depth and a cache complexity of scan($n + m$) [9, 10, 52].

III. Triangle Counting

This section describes a conceptual algorithm for triangle counting that exposes substantial parallelism. In later sections, we describe how to derive efficient implementations for it.

Our conceptual algorithm follows Latapy's sequential *compact-forward* algorithm [34] for triangle counting. We extend Latapy's algorithm because it was shown to perform well sequentially, and we observe that it is amenable to parallelization. To count the number of triangles in a graph, the algorithm performs two main steps, as shown in Algorithm 1.

[2]We use "with high probability" (whp.) to mean with probability at least $1 - 1/n^c$ for any constant $c > 0$.

Algorithm 1 High-level parallel triangle counting algorithm

```
1:  procedure Rank-By-Degree(G = (V, E))
2:      Compute an array R such that if R[v] < R[w] then d(v) ≤ d(w)
3:      parfor v ∈ V do
4:          A⁺[v] = {w ∈ N(v) | R[v] < R[w]}
5:      return A⁺
6:  procedure TC(A⁺)
7:      Allocate an array C of size Σ_{v∈V} |A⁺[v]|
8:      parfor v ∈ V do
9:          parfor w ∈ A⁺[v] do
10:             I = intersect(A⁺[v], A⁺[w])
11:             C[ρ(v, w)] = |I|        ▷ ρ(·) gives a unique index in C
12:     count = sum of values in C
13:     return count
```

Step 1: *Ranking*—form a directed graph where each undirected input edge gives rise to exactly one directed edge. The ranking helps to improve the asymptotic performance and ensures each triangle is counted only once.

Step 2: *Counting*—count triangles of a particular form in the directed graph formed in the previous step.

For the ranking step, the Rank-By-Degree function on Lines 1–5 takes an undirected graph G, and computes a rank array R ordering the vertices by non-decreasing degree.[3] R contains unique integers, and for any two vertices v and w, if $R[v] < R[w]$ then $d(v) \leq d(w)$. On Lines 3–4, it goes over the vertices of G in parallel, storing for each vertex v, the higher-ranked neighbors of v in $A^+[v]$. Finally, it returns the *ranked adjacency list* A^+.

For the counting step, the triangle counting function TC on Lines 6–13 takes as input a ranked adjacency list A^+. An array C of size equal to the number of directed edges ($\sum_{v \in V} |A^+[v]|$) is initialized on Line 7. Each edge (v, w) is assigned a unique location in C, denoted by $\rho(v, w)$. On Lines 8–11, all vertices are processed, and for each vertex v, its neighbors w in $A^+[v]$ are inspected, and the intersection between $A^+[v]$ and $A^+[w]$ is computed. Each common out-neighbor u corresponds to a triangle (v, w, u) where $R[v] < R[w] < R[u]$. The count of triangles incident on (v, w) is thus set to the size of the intersection (Line 11). In Line 12, the individual counts are summed, and finally returned on Line 13.

Two observations are in order: First, because of the ranking step, all triangles will be counted exactly once. Second, since the intersection can be computed on all directed (v, w) pairs in parallel, this algorithm already has abundant parallelism.

We now illustrate these steps with an example (Fig. 1). Notice the degree of parallelism the algorithm obtains (Fig. 2).

Example. In Figure 1, we show an example graph and the graph after ranking by degree, which contains directed edges from lower to higher-ranked vertices. The rank of the vertices are stored in an array R:

Vertex	0	1	2	3	4
R	1	4	0	3	2

Figure 1 (right) shows the edges after running Rank-By-Degree. Then, running TC on this graph will compute the set intersections of multiple pairs, as shown in Fig. 2. Notice that at this point, the algorithm indicates that these intersections are parallel

[3] Various ranking functions can be used, but ordering by degree in the original graph has it has been shown to perform the best in practice if both ranking and triangle counting times are included [39]. This ordering heuristic also leads to good theoretical guarantees for triangle counting [34].

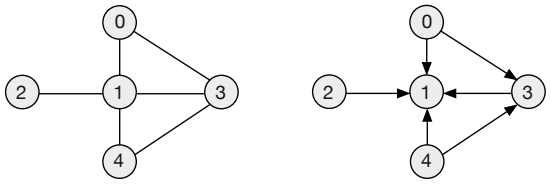


Fig. 1: Example of a graph (**left**) and its directed edges after ranking by degree (**right**). The contents of A^+ are $A^+[0] = \{1, 3\}$, $A^+[1] = \{\}$, $A^+[2] = \{1\}$, $A^+[3] = \{1\}$, and $A^+[4] = \{1, 3\}$. The triangles found are $(0, 3, 1)$ and $(4, 3, 1)$, discovered by $\texttt{intersect}(A^+[0], A^+[3])$ and $\texttt{intersect}(A^+[4], A^+[3])$.

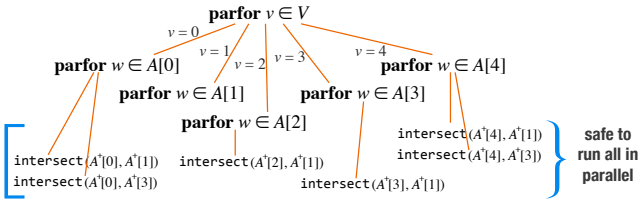


Fig. 2: Example of how the parallel triangle counting algorithm performs in action.

tasks; however, in the context of dynamic multithreading, the exact combination of tasks that will be run simultaneously depends on the scheduler. Subsequently, for each of these pairs, the size of the intersection is recorded in C (e.g., $C[\rho(0, 3)] = 1$ as $|A^+[0] \cap A^+[3]| = 1$ and $C[\rho(4, 3)] = 1$ as $|A^+[4] \cap A^+[3]| = 1$).

IV. Exact Triangle Counting

This section describes efficient parallel algorithms for exact triangle counting based on the conceptual algorithm in the previous section. In particular, we discuss how the ranking and counting steps are implemented.

A. Ranking

To implement the ranking step, we begin by constructing a rank array R. Assume that we have the degrees of the vertices stored in an array D of size V in order of vertex ID (i.e., $D[i]$ is the degree of the i^{th} vertex). By sorting the vertices by degree and breaking ties by ID, we can convert this into an array R such that R contains unique integers, and if $R[u] < R[v]$ then $D[u] \leq D[v]$. The sort requires $O(V \log V)$ work, $O(\log^{3/2} V)$ depth and $O(\text{sort}(V))$ cache misses whp., as mentioned in Section II.

Then, given the ranking array R, we look up the rank for each endpoint of every edge and choose which direction to retain. In particular, each vertex looks up the rank of each of its neighbors and applies a parallel filter, keeping only the higher-ranked neighbors. Each vertex will incur a cache miss to access the start of its adjacency list, for a total of $O(V)$ cache misses. The filters require $O(E)$ work, $O(\log E)$ depth and $\text{scan}(E)$ cache misses overall. Looking up the rank of the neighbors requires $O(E)$ work, $O(1)$ depth and $O(E)$ cache misses overall (since the neighbors can appear anywhere in R). The following lemma summarizes the complexity of ranking:

Lemma 1 Rank-By-Degree *can be implemented in* $O(V \log V + E)$ *work,* $O(\log^{3/2} V)$ *depth and* $O(\text{sort}(V) + E)$ *cache misses whp.*

We note that we can obtain a cache complexity $O(\text{sort}(E))$ whp. for ranking (while increasing the work to $O(E \log E)$) by using sorting routines. However we found this to be more expensive in practice, and furthermore it does not improve the overall complexity of triangle counting, so we do not elaborate on the approach here.

B. Counting

We now describe the counting algorithm TC assuming that the ranked adjacency list A^+ has already been computed. The size of C and the unique locations $\rho(v, w)$ in C for each directed edge (v, w) can be computed with a parallel scan over the directed edges. In particular, each vertex v writes the length of $A^+[v]$ into a shared array at location v, and then a scan with the $+$ operator is applied to generate the starting offset o_v for each vertex. The offset for element i in $A^+[v]$ is computed as $o_v + i$. The result of the scan also gives the size of C. This requires $O(E)$ work, $O(\log E)$ depth and $O(\text{scan}(E))$ cache misses. On Line 12, the individual counts in C are added together using a prefix sum. We now describe two implementations of Lines 10–11, differing in how the `intersect` function is implemented.

Algorithm I: Merge-based Algorithm: The first algorithm, called ***TC-Merge***, implements Line 10 by using a merge on the directed adjacency lists of v and w. It requires sorting the adjacency lists as a preprocessing step, which requires $O(E \log E)$ work, $O(\log^{3/2} E)$ depth and $O(\text{sort}(E) + V)$ cache misses whp. Merging the sorted lists gives the intersection and its size, requiring work linear in the size of the two lists. Sequentially, the total amount of work done in merging has been shown to be $O(E^{3/2})$ [34], and since the merge is done in the same asymptotic work in parallel, the bound is the same (hence, it is work-efficient). The depth for merging is $O(\log(E^{3/2})) = O(\log E)$ and cache complexity is $O(\text{scan}(E^{3/2})) = O(E^{3/2}/B)$ (note that this dominates the cache complexity of sorting). Accessing the adjacency list for each edge involves a random access, adding a total of $O(E)$ cache misses. The complexity of counting dominates the complexity of ranking, and we have the following theorem:

Theorem 2 *Ranking and triangle counting using TC-Merge can be performed in* $O(E^{3/2})$ *work,* $O(\log^{3/2} E)$ *depth and* $O(E + E^{3/2}/B)$ *cache misses whp.*

We point out that if $B = O(\sqrt{E})$, then $E^{3/2}/B$ is the dominant term in the cache complexity. This condition is readily met in practice for in-memory algorithms since a typical cache line is 64 bytes, which holds at most 16 edges in a standard implementation,[4] and typical graphs of interest have at least tens of thousands of edges (the graphs we use have tens of millions to billions of edges). We conjecture that this condition will continue to hold in the future when analyzing large graphs (i.e. graph sizes in terms of number of edges will grow faster than $\Omega(B^2)$). The situation may be different for the external-memory setting, as was pointed out in [25].

Algorithm II: Hash-based Algorithm: Our second algorithm, ***TC-Hash***, uses a hash table storing the edges of A^+ to compute the intersection on Line 10 of Algorithm 1. A hash table can be implemented in parallel to support worst-case $O(1)$ work and depth queries [35], and so Line 8 can be implemented in $O(\min\{A^+[v], A^+[w]\})$ work by looping over the smaller of $A^+[v]$ and $A^+[w]$ and querying the hash table of the other vertex. Insertion of the edges into the hash tables can be done in $O(E)$ work, $O(\log E)$ depth and $O(E)$ cache misses whp. [35].

Since $|A^+[v]| \leq d(v)$ for all v, this gives an overall work bound of $O(\alpha E)$ for TC-Hash, where $\alpha \geq 1$ denotes the arboricity of the graph (recall from Section II that

[4] Compression techniques could be used to store edges more compactly, however $B = O(\sqrt{E})$ still holds for large graphs.

$\sum_{(u,v)\in E} \min\{d(u), d(v)\} = O(\alpha E)$). Note that since $\alpha = O(\sqrt{E})$, this bound is tighter than $O(E^{3/2})$ and is in fact optimal. However, each hash table look-up incurs $O(1)$ cache misses, leading to $O(\alpha E)$ total cache misses. Looking up the adjacency list of each edge involves a random access, leading to $O(E)$ cache misses. Computing the size of each intersection can be done work-efficiently with a parallel scan. Hence, this can be implemented in $O(\alpha E)$ work, $O(\log E)$ depth and a parallel cache complexity of $O(\alpha E)$.

By putting together the bounds for ranking and counting, we obtain the following theorem:

Theorem 3 *Ranking and triangle counting using TC-Hash that performs $O(V \log V + \alpha E)$ work, $O(\log^{3/2} E)$ depth and $O(\text{sort}(V) + \alpha E)$ cache misses whp.*

V. Approximate Triangle Counting

If some amount of error can be tolerated, the running time of triangle counting can be reduced using approximate counting algorithms. In this section, we extend the parallel algorithms for exact triangle counting to approximate triangle counting. As will be discussed in Section IX, many approximate triangle counting schemes have been proposed [3, 41, 49, 56, 57, 58, 59], and the recent colorful triangle counting scheme of Pagh and Tsourakakis (PT) [41] is one of the most efficient. We will use the PT colorful triangle counting scheme to develop parallel and cache-oblivious approximate triangle counting algorithms.

Algorithm 2 Pagh-Tsourakakis Sampling

Input: a graph $G = (V, E)$ and a parameter $0 < p \leq 1$
Output: a sampled subgraph $H = (V_H, E_H)$ of G.
1: Assign a random color $c(v) \in \{1, \ldots, C\}$ to every vertex v, where $C = \lceil 1/p \rceil$.
2: Construct $E_H = \{(u, v) \in E \mid c(u) = c(v)\}$ and $V_H \subseteq V$ if the vertex has at least one neighbor.
3: Return $H = (V_H, E_H)$.

The PT algorithm works by first sampling edges from the input graph using Algorithm 2. An exact triangle counting algorithm is then run on the subgraph. If the exact triangle counting algorithm reports T triangles, then the PT algorithm reports an estimate of T/p^2 triangles.

Pagh and Tsourakakis [41] show that the estimate T/p^2 is an unbiased estimate (i.e., its expectation equals the true triangle count) as each triangle is included in the subgraph with probability p^2 (if two edges in a triangle are present in the subgraph, then the third edge must also be present). They also prove that the estimate is tightly concentrated around its mean for appropriate values of p. Note that a larger p value leads to higher quality estimates and vice versa.

Using TC-Merge after sampling gives the following lemma:

Lemma 4 *For a parameter $0 < p \leq 1$, approximating the number of triangles in a graph can be done in $O(E + (pE)^{3/2})$ work, $O(\log^{3/2} E)$ depth and a parallel cache complexity of $O(\text{scan}(E) + pE + (pE)^{3/2}/B)$ in expectation.*

Proof: To form the subgraph, we first convert the adjacency list representation of the graph to an *edge array* representation, which is an array of length E storing pairs of vertices that have an edge between them. Since the adjacency list representation stores the neighbor arrays contiguously in memory, the conversion can be done using a scan. We then apply a parallel filter to the edge set keeping only edges with both endpoints having the same color. We assume that the color of a vertex can be computed with a hash function, and so does not involve a memory access. The scan and filter can be done in $O(E)$ work, $O(\log E)$ depth and $O(\text{scan}(E))$ cache misses. We then remove any singleton (isolated) vertices and relabel the remaining vertices and the edges so that the vertex ID's are in a consecutive range. This "packing" step can be done using standard techniques involving prefix sums in $O(pE)$ work and cache misses. Afterward, we convert the edge array back to the adjacency list representation using prefix sums. Using TC-Merge on the subgraph and applying Theorem 2 on a subgraph with an expected number of edges equal to pE proves the lemma. ■

The following lemma can be obtained by using TC-Hash instead of TC-Merge on the subgraph, where V_H is the number of vertices in the subgraph ($V_H = O(pE)$ in expectation, since we remove singleton vertices) and $\alpha_H \geq 1$ is the arboricity of the subgraph.

Lemma 5 *For a parameter $0 < p \leq 1$, approximating the number of triangles in a graph can be done in $O(E + V_H \log V_H + p\alpha_H E)$ work, $O(\log^{3/2} E)$ depth and a parallel cache complexity of $O(\text{scan}(E) + \text{sort}(V_H) + p\alpha_H E)$ in expectation.*

VI. Extensions

Triangle Enumeration. To adapt TC-Merge and TC-Hash for triangle enumeration, we only need to modify the implementation of Line 10 of Algorithm 1 so that `emit` is called whenever a triangle is present in memory. Note that since the `emit` function may be called in parallel, one must ensure that any modifications to shared structures are atomic.

For example, to list all the triangles in the graph, we can initialize a concurrent hash table, and have the `emit` function add the triangle to the hash table when it finds one.[5] With a good hash function and large enough hash table, the probability that two triangles hash to the same location is small, and hence memory contention will be small. After all triangles are added, then one can write out the contents of the hash table.

Without accounting for the cost of `emit` (consistent with the analysis in [40]), which varies with the application, the complexity is the same as that of exact triangle counting.

Directed Triangle Counting and Enumeration. Triangle computations on directed graphs have also attracted recent interest [21, 48]. The goal is to count triangles of different configurations of directed edges. For example, the GraphLab directed triangle counting implementation [21] counts four types of triangles: in-, out-, through and cycle triangles. If a vertex v with two incoming edges participates in a triangle, it is said to be an *in-triangle* incident on v. If a vertex v with two outgoing edges participates in a triangle, it is said to be an *out-triangle* incident on v. Finally, if a vertex v with one incoming edge and one outgoing edge participates in a triangle, and the final triangle edge forms a cycle, then the triangle is a *cycle triangle* incident on v; otherwise it is said to participate in a *through triangle*.

We describe how to modify TC-Merge and TC-Hash using

[5] If threads are explicitly managed then we can initialize a list for each thread, and whenever a thread finds a triangle it simply adds the triangle to its list. The lists are then be joined at the end. This approach, however, breaks the dynamic multithreading abstraction.

Algorithm 1 to count the 4 types of directed triangles described above. When we symmetrize the graph for the ranking phase, we also store additional information indicating which direction(s) the edge appears in the original graph. The array of counts C on Line 7 is modified to store 4-tuples per entry, where $C[\rho(u,w)]$ stores the count of each type of triangle incident on edge (u,w). Then on Lines 10–11, we count the number of each type of directed triangle to store into $C[\rho(u,w)]$. The type(s) of each triangle can be computed locally with constant work/depth and no memory accesses. Finally, to sum the counts on Line 12, we perform element-wise sums of the 4-tuples of C using a prefix sum, and return a single 4-tuple. If the enumeration variant is instead desired, we can modify `emit` to take additional information about the orientation of edges in the triangle. The work, depth and cache complexity bounds of Theorems 2 and 3 are preserved for directed triangle counting.

Local Triangle Counting. The *local triangle counting* problem takes a graph and returns for each vertex, the number of triangles incident on it. TC-Merge and TC-Hash as described in Section IV only count each triangle once, instead of 3 times, since the ranking phase keeps each edge in only one direction. So just returning the array of counts C in Algorithm 1 would not produce the correct answer. Therefore, we first store all of the triangles in an array using a triangle enumeration algorithm. To obtain the local counts, we sort the array of triangles, using the first endpoint of the triangle as the key. After the sort, the triangles sharing the first endpoint will be in consecutive order. We can then use standard techniques involving prefix sum operations to compute the partial local counts per vertex. We then repeat this procedure (sorting and computing partial local counts) on the second and third endpoints of the triangles, and the result will be the local triangle counts for each vertex. The cost of this method is dominated by sorting the triangles, and since there are $O(\alpha E)$ triangles, the work is $O(\alpha E \log E)$, depth is $O(\log^{3/2} E)$ and cache complexity is $O(\text{sort}(\alpha E))$ whp. Including the cost of triangle enumeration using TC-Hash increases the cache complexity to $O(\alpha E + \text{sort}(\alpha E))$ whp. If TC-Merge is used, then the work becomes $O(E^{3/2} + \alpha E \log E)$ and cache complexity becomes $O(E + E^{3/2}/B + \text{sort}(\alpha E))$ whp.

If we assume that an atomic increment operation is supported with $O(1)$ work and depth, then the bounds can be improved with the following scheme. In practice, this assumption can be met, for example, by using `x86`'s atomic add instructions and controlling contention at each location. We create an array of size V to store the local count of each vertex (initialized to 0). Whenever a triangle is identified in the triangle counting algorithm, an atomic increment is performed on the locations in the array corresponding to each of the three triangle endpoints. Since these locations can be anywhere, each triangle found causes $O(1)$ cache misses. The total number of triangles is bounded by $O(\alpha E)$ so if we use TC-Hash for counting, we obtain an algorithm with $O(V \log V + \alpha E)$ work, $O(\log^{3/2} E)$ depth and $O(\text{sort}(V) + \alpha E)$ cache misses whp. In our experiments, we use TC-Merge for counting as we found it to perform better in practice, although the theoretical bounds of local triangle counting become weaker—the work bound increases to $O(E^{3/2})$ and cache complexity increases to $O(E^{3/2}/B + \alpha E)$.

Local triangle counting also works for the directed setting. In the first method, we can use a directed triangle enumeration algorithm which gives the type of each triangle. After each sort, which groups the triangles by a certain endpoint, we can sort within the groups by triangle type. We can then compute the sizes of these subgroups as well as the groups using prefix sums. For the second method, we can store 4-tuples in the global array of local counts, and atomically increment the appropriate element(s) in the tuples based on the triangle type(s).

Clustering Coefficients and Transitivity Ratio. The *local clustering coefficient* [60] for a vertex v is defined to be the number of triangles incident on v divided by $d(v)(d(v)-1)/2$ (the number of potential triangles incident on v). The *global clustering coefficient* is the average over all local clustering coefficients. Both quantities can be computed using the algorithms for local triangle counting.

The *transitivity ratio* of a graph is defined to be the ratio of 3 times the number of triangles to the number of length-2 paths (wedges), which can be computed as $\sum_{v \in V}(d(v)(d(v)-1)/2)$. The number of triangles is already returned by TC-Merge and TC-Hash and the number of wedges can be computed with a prefix sum. Hence, the bounds for computing the transitivity ratio are the same as in Theorems 2 and 3.

VII. Evaluation

We experimentally evaluate how our algorithms perform in practice, specifically how well they scale with the number of threads, how fast they are compared to existing alternatives, and whether they are cache-effective. To this end, we report and discuss the running times, parallel speedups, and cache misses for our exact algorithms, as well as the accuracy of our approximation algorithm versus its running time. Overall, the results indicate that *our algorithms are very fast in practice, scaling well with the number of cores.*

Data. Our input graphs include a variety of real-world networks from the Stanford Network Analysis Project (SNAP),[6] and several synthetic graphs generated from the Problem Based Benchmark Suite (PBBS) [51]. We also use the Twitter graph [32] and the Yahoo Web graph.[7] These graphs are drawn from many fields and have different characteristics, and many are graphs stemming from social media, where triangle computations often see applications. The graph sizes and triangle counts are shown in Table II. We report the number of undirected edges (i.e., an edge between u and v is counted once), but our implementations store, in the intermediate representation, each edge in both directions, so store twice as many edges. Therefore, we effectively symmetrized all of the graphs. We also preprocess these graphs to remove self-loops and duplicate edges.

Input Graph	Num. Vertices	Num. Edges*	Num. Triangles
random	100,000,000	491,001,390	24,899,692
rMat	134,217,728	498,586,618	539914
3D-grid	99,897,344	299,692,032	0
soc-LJ	4,847,571	42,851,237	285,730,264
Patents	3,774,768	16,518,947	7,515,023
com-LJ	3,997,962	34,681,189	177,820,130
Orkut	3,072,441	117,185,083	627,584,181
Twitter	41,652,231	1,202,513,046	34,824,916,864
Yahoo	1,413,511,391	6,434,561,035	85,782,928,684

TABLE II: Graph inputs. *Number of unique undirected edges.

Environment. We run our experiments on two machines: (1) a 40-core (with two-way hyper-threading) Intel machine with 4×2.4GHz 10-core E7-8870 Xeon processors (with a 1066MHz

[6] http://snap.stanford.edu
[7] http://webscope.sandbox.yahoo.com/catalog.php?datatype=g

Algorithm		random	rMat	3D-grid	soc-LJ	Patents	com-LJ	Orkut	Twitter	Yahoo
serial-OB	T_1	278	298	133	24.52	6.23	18.15	95.4	–	–
Green et al.	T_{40h}	6.92	9.54	3.66	2.55	0.31	1.61	17.98	–	–
GraphLab	T_{40h}	58.0	56.1	51.3	3.45	1.7	2.33	5.7	178.7	–
TC-Merge	T_1	106	155	60.4	15.2	3.22	10.7	94.1	2680	1740
	T_{40h}	3.13	3.89	1.75	0.49	0.079	0.389	1.92	55.9	77.7
	T_1/T_{40h}	33.9	39.8	34.5	31.0	40.8	27.5	49.0	47.9	22.4
TC-Hash	T_1	193	279	107	27.5	6.92	19.5	158	4850	2960
	T_{40h}	5.33	7.21	3.25	0.931	0.198	0.723	3.3	93	104
	T_1/T_{40h}	36.2	38.7	32.9	29.5	34.9	27.0	47.9	50.2	28.5
TC-Local	T_1	119	166	64.9	17.3	3.72	12.2	101	2900	2090
	T_{40h}	3.28	3.99	1.76	0.639	0.088	0.397	2.09	163	90.7
	T_1/T_{40h}	36.3	41.6	36.9	27.1	42.3	30.7	48.3	17.8	23.0

TABLE III: Triangle counting times (seconds) on the Intel machine: T_1 is single-thread time; T_{40h} the time on 40 cores with hyper-threading; and T_1/T_{40h} the parallel speedup.

bus and 30MB L3 cache) and 256GB of main memory, and (2) a 64-core AMD machine with 4×2.4GHz 16-core 6278 Opteron processors (with a 1600MHz bus and 16MB L3 cache) and 188GB of main memory. Most of the reported results are obtained from the Intel machine. We also report some results on the AMD machine, showing that our algorithms exhibit the same performance trends on different machines. The codes use Cilk Plus to express parallelism. We compile all of our code with the `g++` compiler version 4.8.0 (which supports Cilk Plus) with the `-O2` flag.

A. Implementation

In our implementations, we use the parallel primitives prefix sum, filter and sort, from the Problem Based Benchmark Suite (PBBS) [51], which are all cache-oblivious. The parallel sort is an implementation of the sample sort described in [10]. All of our implementations are lock-free.

In our implementations of Algorithm 1, the for-loop on Line 3 and nested parallel for-loops on Lines 8 and 9 use the `cilk_for` construct. Note that already, the counting code has abundant parallelism (a lot more than the number of processors available) because all of the `intersect` calls are made in parallel (Lines 10–11 of Algorithm 1). Consequently, for TC-Merge, it suffices for each `intersect` to use a sequential merge; making the merge parallel would not improve the speedup as we have experimentally confirmed. Each merge terminates when one of the lists has been fully traversed. For TC-Hash, we use a lock-free concurrent hash table based on linear probing [50]. Before counting, each vertex creates a hash table of its neighbors in $A^+[v]$. During counting, which intersects $A^+[v]$ and $A^+[w]$ for each directed edge (v, w), we loop through the smaller adjacency list and query the table of the vertex with the larger adjacency list. Again, due to abundant parallelism in the nested parallel for-loop, we perform the hash table look-ups for each `intersect` sequentially, as we did not get any speedup from parallelizing it.

B. Exact Triangle Counting

The first set of experiments is concerned with exact triangle counting. The times on the Intel machine are shown in Table III, and the times on the AMD machine are shown in Table IV. The times include both ranking and counting, and are based on a median of three trials. We also report the parallel speedup by dividing the time on a single thread by the parallel time (40 cores with hyper-threading for the Intel machine and 64 cores for the AMD machine). For some graphs, we obtain a speedup factor of over 40 on the Intel machine due to the effects of hyper-threading. Overall, the times on the Intel machine are faster than on the AMD machine, but the parallel speedups are comparable. Later, we discuss parallel performance of our algorithms compared with recent parallel/distributed algorithms.

Algorithm		random	rMat	3D-grid	soc-LJ	Patents	com-LJ	Orkut	Twitter	Yahoo
TC-Merge	T_1	188	283	72.4	20.2	4.29	14.3	122	3730	2420
	T_{64}	4.93	6.18	2.68	0.81	0.155	0.623	2.67	78.9	100
	T_1/T_{64}	38.1	45.8	27.0	24.9	27.7	23.0	45.7	47.3	24.2
TC-Hash	T_1	274	416	184	33.4	11.1	23.8	173	6050	4340
	T_{64}	8.26	12.0	4.95	1.39	0.321	1.12	4.24	133	183
	T_1/T_{64}	33.2	34.7	37.2	24.0	34.6	21.3	40.8	45.5	23.7
TC-Local	T_1	168	268	79.1	24.5	5.3	17.2	134	4100	4060
	T_{64}	5.29	6.26	2.65	0.886	0.172	0.628	3.15	164	152
	T_1/T_{64}	31.8	42.8	29.8	27.7	30.8	27.4	42.5	25.0	26.7

TABLE IV: Triangle counting times (seconds) on the AMD machine: T_1 is single-thread time; T_{64} the time on 64 cores; and T_1/T_{64} is the speedup.

Several things are worth discussing: First, *our single-threaded performance is competitive with existing implementations*. To see whether our implementations incur high overhead due to parallelization, we run the Ortmann and Brandes serial implementations (***serial-OB***) [39] on the same set of graphs and report the running time for their best implementation on each input on the Intel machine (Table III). We do not have their times on the Twitter and Yahoo graphs, as we could not run their implementations on them. When running single-threaded, our implementation is faster than theirs, and their fastest implementation is also a variant of the compact-forward algorithm. Their paper includes a comprehensive evaluation of other serial algorithms, which are described in Section IX.

Second, *both TC-Merge and TC-Hash obtain very good speedups on all graphs, between 22–50x on 40 hyper-threaded cores, with TC-Merge having an edge over TC-Hash*. For further detail, Figure 3 shows the running time versus the number of threads for several graphs on the Intel machine. We see that both implementations scale well as the number of threads is increased. We also observe that TC-Merge is faster than TC-Hash for all thread counts (by a factor of 1.3–2.5x). The trends are similar on the AMD machine, with TC-Merge again being faster than TC-Hash, although the absolute running times are slower than on the Intel machine.

We had difficulty isolating the benefit of hyper-threading on the Intel machine as we did not have root access to disable hyper-threading. We did run the experiments using only 40 threads; however, this does not guarantee that the threads are assigned to distinct cores. We found the hyper-threaded running times to be about 40–50% faster than these times on average over all inputs.

Third, *for both implementations, usually the majority of the time is spent inside counting*. We show the breakdown of the parallel running times for the two implementations on the Intel machine in Figure 4. We observe that ranking usually takes a small fraction of the total time. For most of the real-world graphs (except Patents), the time for ranking in TC-Merge is at most 10%, although it is higher for the synthetic graphs (as high as 48% for the 3D-grid graph). This is because the number of potential triangles is much lower in our synthetic graphs, so the fraction of time spent in the counting portion of the computation is lower. For TC-Hash, at most 25% of the time is spent in ranking. We also measure the time for inserting

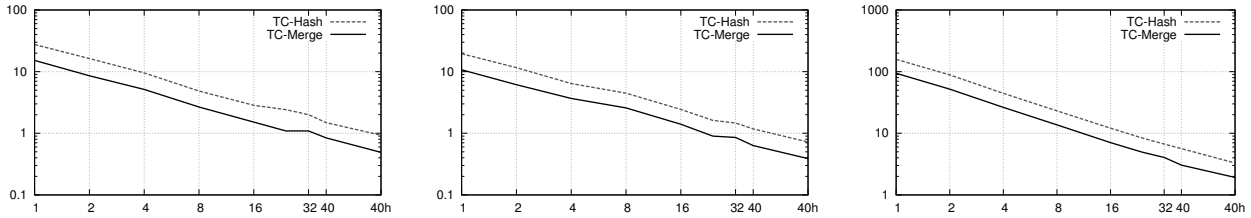


Fig. 3: Times in seconds (vertical axis) for exact triangle counting (TC-Merge and TC-Hash) on soc-LJ (**left**), com-LJ (**center**), and Orkut (**right**) as the number of threads varies (horizontal axis) on a log-log scale. "40h" indicates 80 hyper-threads.

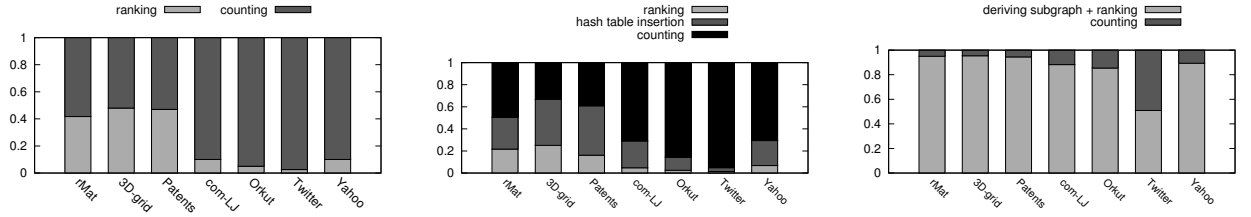


Fig. 4: Breakdown of times on 40 cores with hyper-threading on various graphs, reporting on the vertical axis the fraction of time spent on each algorithmic component for TC-Merge (**left**), TC-Hash (**center**), and TC-Approx with $p = 1/25$ (**right**).

Algorithm	soc-LJ	Patents	com-LJ	Orkut
TC-Merge (L3 misses)	126M	58M	87M	762M
TC-Hash (L3 misses)	217M	90M	150M	1.2B
TC-Merge (L2 misses)	301M	134M	215M	1.4B
TC-Hash (L2 misses)	432M	182M	314M	1.8B
TC-Merge (ops)	2.54B	153M	1.7B	15.8B
TC-Hash (ops)	2.58B	164M	1.7B	18.4B

TABLE V: L2 and L3 cache misses and work for intersection (ops).

the edges into the hash tables, and observe that for most of the real-world graphs this step takes longer than ranking, but less time than counting. For most of the real-world graphs (except Patents), this step also takes at most 25% of the total time.

Fourth, *despite the bounds, in practice, TC-Hash performs about the same amount of work as TC-Merge—but, as predicted from the theoretical bounds, TC-Hash incurs many more cache misses than the TC-Merge.* We show the number of L2 and L3 cache misses for our two algorithms on several input graphs in Table V. The numbers are collected from a 32-core (with two-way hyper-threading) Intel Nehalem machine with 8×2.27GHz Intel X7560 Xeon processors (with a 1066 MHz bus and 24MB L3 cache, and 256KB L2 cache per core), since we did not have root access on the 40-core Intel machine. The cache misses reported are for an execution using all hyper-threads; however, we found the cache misses for all thread counts to be similar.

We also report the total number of operations inside `intersect` for each implementation. For TC-Merge, the number of operations is computed by the number of comparisons done in the merge between elements in the adjacency lists. For TC-Hash, the number of operations is computed by the number of locations inspected in the hash table, for both insertions and finds. We observe that TC-Hash performs about the same amount of work as TC-Merge, but the key differentiating factor is the number of cache misses. This confirms that cache efficiency is crucial for algorithm performance.

Parallel Pagh-Silvestri Algorithm. Pagh and Silvestri (PS) [40] recently present a sequential cache-oblivious algorithm, which we parallelize and experiment with (more details appear in Section VIII). We find that our parallel PS implementation achieves reasonable parallel self-relative speedup; however, it is orders of magnitude slower than TC-Merge and TC-Hash. When run sequentially, we also found it to be orders of magnitude slower than other sequential triangle counting algorithms. This is because the PS algorithm makes many more passes over the edges of the graph, and does many sorts, which makes it expensive in practice. As far as we know, there is no public implementation of the PS algorithm available. Engineering the algorithm to run fast in practice, both sequentially and in parallel, would be an interesting direction for future work.

Comparison with other work: Several parallel triangle counting algorithms for distributed-memory have been proposed, and run on recent machines with comparable specifications to ours. Arifuzzaman et al. [2] propose PATRIC, which is an MPI-based parallel algorithm based on a variant of the node-iterator algorithm. Using 200 processors, they require 9.4 minutes to process the Twitter graph. Park and Chung [42] propose a MapReduce algorithm for counting triangles, which requires 213 minutes to process the Twitter graph on a cluster server with 47 nodes. They show that their algorithm outperforms the MapReduce algorithms of Cohen [16] and Suri and Vassilvitskii [53]. The MapReduce triangle enumeration algorithm of Park et al. [43] takes several hours on the Twitter graph, although they are solving the more expensive task of enumerating all triangles instead of just counting them. GraphLab implements triangle counting using MPI, and achieves better performance than the other algorithms—they process the Twitter graph in 1.5 minutes using 64 16-core machines [21]. In contrast to the distributed-memory algorithms, we are able to process Twitter in under a minute using TC-Merge on a single 40-core machine. We note that while our algorithm is much faster than the distributed-memory algorithms, ours is constrained to graphs that fit in the memory of a single machine.

We also compare with the implementations of Green et al. [23], the fastest in-memory implementations of triangle counting that we are aware of. We report the parallel time on the Intel machine for their fastest implementation per graph in Table III for the graphs which we could run their implementations on. We note that their times do not include the time for sorting the edges per vertex (required for merging), although this would be a small fraction of the total time for most graphs. In parallel, our algorithm TC-Merge is 2–9 times faster than their fastest algorithm. Their algorithms are parallel versions of the node iterator algorithm without any ordering heuristic, and uses merging for intersection. Therefore their

algorithms take $O(\sum_{v \in V}(d(v)^2 + \sum_{w \in N(v)} d(w)))$ work, which in general is higher than the work of our algorithms. We believe the difference in empirical performance between their algorithms and ours is largely due to the algorithmic difference. They also perform load balancing by estimating the work per vertex and dividing vertices and edges among threads appropriately, whereas we take the simpler approach of leaving the scheduling to the run-time system, which we found to work well in practice. In addition, we compare with running GraphLab on a single machine (the 40-core Intel machine) and report the times in Table III for all of the input graphs except Yahoo, which caused their program to thrash. We found it to be several times slower than our implementations as well as the implementations of Green et al. [23], as the GraphLab implementation is designed for distributed-memory using MPI, which has additional overheads when run on a single machine.

We note that there has been recent work showing that hash-based joins are usually better than sort-merge-based joins on multicores [4]. However, the setting of this work is that only two tables are joined and hence only a single join needs to be performed. Thus, the cost for sorting and hash table insertions dominate the cost. In contrast, in triangle computations each vertex participates in many intersections, but the sorting and hash table insertions for each vertex only needs to be done once, so this preprocessing cost is amortized over all of the subsequent intersections. Another difference is that for a single hash-based join, the elements of the smaller set are inserted into a hash table, with the elements from the larger set querying it, while to obtain good complexity bounds for triangle computations, the elements from the smaller adjacency list are queried in the hash table of the vertex with a larger adjacency list. Therefore, the conclusion of [4] does not directly apply to our context.

C. Approximate Triangle Counting

The previous section showed that TC-Merge is fast and scales well with the number of threads. In this section, building on TC-Merge, we study our parallel approximate triangle counting implementation, which sparsifies the input graph using the colorful triangle counting scheme of Pagh and Tsourakakis [41], and applies TC-Merge on the sampled subgraph. We refer to this algorithm as ***TC-Approx***. In our implementation, we combine the ranking step with the subgraph creation step to improve overall performance. In addition, we operate directly on the adjacency list representation, and have each vertex separately apply a filter on its edges, instead of converting to the edge array representation and back as described in Section V. While this adds an extra $O(V)$ term to the cache complexity, it performs better in practice as less work is performed.

The times on the 40-core Intel machine for $p = 1/25$ and $p = 1/10$ are shown in Table VI. The times include sampling edges from the original graph, and performing ranking and counting on the sampled subgraph. The reported times are based on an average of 10 trials, and the average error and variance of the estimates are also reported. We observe that the times are much lower than those for exact triangle counting, and the error and variance of the estimates are very small and well-controlled. For graphs where the number of edges is much larger than the number of vertices, the speedup of TC-Approx over TC-Merge in parallel is significant (28.7x for Orkut and 23.3x for Twitter with $p = 1/25$), although for sparser graphs the savings is not as high. For the real-world

TC-Approx		random	rMat	3D-grid	soc-LJ	Patents	com-LJ	Orkut	Twitter	Yahoo
$p = 1/25$	T_1	43.5	47.8	30.1	1.39	1.05	1.11	2.64	42.4	300
	T_{40h}	1.38	1.54	0.95	0.04	0.031	0.033	0.067	2.4	9.1
	Err.(%)	0.48	3.06	0.0	0.31	0.99	0.48	0.23	0.1	0.39
	σ^2	0.003	0.11	0.0	0.001	0.014	0.003	0.0	0.0	0.002
$p = 1/10$	T_1	56.5	62.3	40.1	1.77	1.22	1.39	4.05	79.4	350
	T_{40h}	1.6	1.77	1.11	0.05	0.036	0.042	0.1	5.88	14.5
	Err.(%)	0.19	0.8	0.0	0.34	0.38	0.4	0.17	0.12	0.18
	σ^2	0.0	0.007	0.0	0.002	0.003	0.001	0.0	0.0	0.0

TABLE VI: Times (seconds) and accuracy for approximate triangle counting on the Intel machine for $p = 1/25$ (**top**) and $p = 1/10$ (**bottom**). T_1 indicates single-thread time, and T_{40h} indicates the time on 40 cores with hyper-threading.

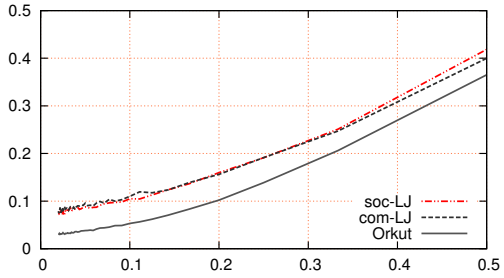


Fig. 5: The fraction of time taken by TC-Approx relative to TC-Merge without sampling (vertical axis) as the sampling rate p (horizontal axis) varies, on the input graphs soc-LJ, com-LJ, and Orkut.

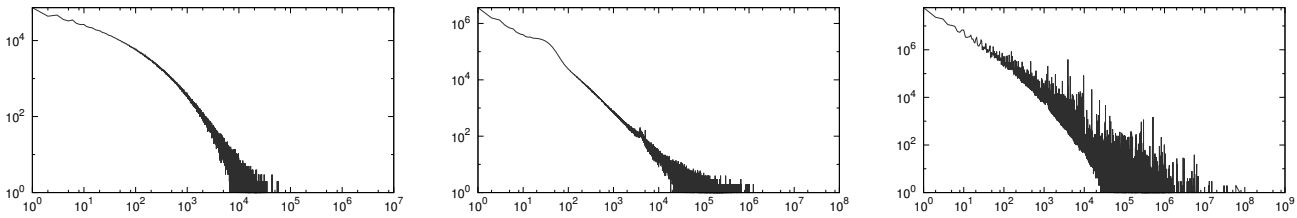
graphs, the average error is less than 1% for a sampling factor of $p = 1/25$.

Figure 4 (right) shows the breakdown of the parallel running time on the Intel machine of TC-Approx for $p = 1/25$. The time spent on computing the subgraph and ranking is a large fraction of the total time (at least 80% for all graphs except for the Twitter graph) because all of the edges are inspected. In contrast, the time spent on counting is a small fraction of the overall time for most graphs because there are much fewer edges in the sampled subgraph than in the original graph.

Figure 5 shows the parallel running time of TC-Approx relative to TC-Merge on the Intel machine as a function of the parameter p for several graphs. Overall, the time goes up as p increases, as this corresponds to a larger sample of edges.

Comparison with other work: Our algorithm is much faster than the multicore algorithm for approximate triangle counting by Rahman and Al Hasan [45]. We tested our algorithm on the Wikipedia-2007-02-06 graph[8] that they report times for (which has 3.566 million vertices and 42.375 million undirected edges), and on 16 threads our approximate counting algorithm obtains a 99.5% accuracy in 0.13 seconds (for $p = 1/10$), while they require 10.68 seconds to achieve 99.07% accuracy using 16 threads. We also note that our exact algorithm runs in 1.45 seconds using 16 threads on the same graph, faster than their approximate algorithm. The machines used in both cases are comparable, but even after adjusting for any small differences, we would still be significantly faster. Recent work has extended wedge sampling to the MapReduce setting [29]. Their experiments use 32 4-core machines with hyper-threading, and they show that the overhead of MapReduce in their algorithm is already 225 seconds, and require about 10 minutes on the Twitter graph, which is slower than our parallel times for exact counting using 40 cores shown in Table III. Papers for other approximate algorithms [41, 57] do not have parallel running times, so we are unable to perform a comparison.

[8] http://www.cise.ufl.edu/research/sparse/matrices/



(a) Orkut (b) Twitter (c) Yahoo

Fig. 6: Distribution of local triangle counts (log-log scale), showing local triangle count (horizontal axis) vs. the number of vertices with that count (vertical axis).

D. Local Triangle Counting

We have also implemented a parallel algorithm for local triangle counting (***TC-Local***). For this algorithm, we modify TC-Merge to keep a count for every vertex in the graph. To keep the algorithm lock-free while respecting the dynamic multithreading abstraction, we use an atomic add (using the `x86` atomic instruction `xadd`) to a global array of local counts when a triangle is found. We use the following optimization to reduce work/contention: for a triangle discovered by looping over vertex v and vertex $w \in A^+[v]$ with a third vertex being $u \in A^+[v] \cap A^+[w]$, we atomically increment the count of u when it is discovered; for the second endpoint (w), we atomically add the count (if nonzero) after the intersection $A^+[v] \cap A^+[w]$ is finished, and for the first endpoint (v), we atomically add the count (if nonzero) after all merges with neighbors are finished.

Our experiments show that *TC-Local also scales well with the number of cores*. Tables III and IV show the times for local triangle counting (TC-Local) on the Intel machine and the AMD machine, respectively. As expected, it is slightly slower than global triangle counting because whenever a triangle is found, an atomic increment to a global array is performed (which likely involves a cache miss). Compared to TC-Merge, on 40 cores with hyper-threading on the Intel machine TC-Local is at most 30% slower for most graphs, but almost 3 times slower for the Twitter graph possibly due to contention with the atomic increment (Twitter has many high-degree vertices). TC-Local achieves 17–48x speedup over all of the inputs. The trends on the AMD machine are similar, although the absolute running times are slower.

As a simple application, we extend the analysis of Tsourakakis [55] to much larger graphs. Tsourakakis observes that in real-world graphs the relationship between local triangle count and the number of vertices with such a count follows a power law [55], though the graphs used were much smaller than our input graphs. In Figure 6, we plot the relationship in log-log scale for the larger real-world input graphs and confirm that this relationship does indeed quite closely resemble a power law. Due to the efficiency of our algorithm, we are able to generate such plots for some of the largest publicly-available graphs in just a few minutes.

VIII. Parallelization of the Pagh-Silvestri Algorithm

Pagh and Silvestri [40] recently describe a sequential cache-oblivious algorithm for triangle enumeration with an expected cache complexity of $O(E^{3/2}/(\sqrt{M}B))$. In this section, we review their sequential algorithm and then show how to parallelize it. Their algorithm uses the edge array representation of the graph, which uses an array of length E storing pairs of vertices that have an edge between them.

Pagh and Silvestri first show that enumerating all triangles containing a given vertex v can be done with $O(\text{sort}(E))$ cache misses. They do this by (1) finding all of v's neighbors via a scan and sorting them lexicographically, (2) sorting the edge array by the source vertex and intersecting it with v's neighbors to get the outgoing edges of v's neighbors, and (3) sorting the result of step 2 by target vertex and intersecting it with v's neighbors to get all edges with both endpoints in v's neighbor set. The result of this is all the triangles incident on v. Since these operations are known to be implementable in a parallel and cache-oblivious manner, we have the following lemma:

Lemma 6 *There is an algorithm for enumerating all triangles incident on a vertex v that requires $O(E \log E)$ work, $O(\log^{3/2} E)$ depth, and $O(\text{sort}(E))$ cache misses whp.*

However, naively using this for each vertex is too costly, and hence their algorithm only uses this step for high-degree vertices and then uses a novel coloring scheme to recursively solve the problem on subgraphs. Using their definitions, a triangle (u, v, w) satisfies the (c_0, c_1, c_2) coloring if $c(u) = c_0$, $c(v) = c_1$ and $c(w) = c_2$ where c is the coloring function. An edge (u, v) is *compatible* with a coloring (c_0, c_1, c_2) if $(c(u), c(v)) \in \{(c_0, c_1), (c_1, c_2), (c_0, c_2)\}$. The ***Pagh-Silvestri (PS) algorithm*** is a recursive algorithm with 3 steps:

Algorithm 3 Pagh-Silvestri (PS) algorithm

procedure PS-Enum($G = (V, E), (c_0, c_1, c_2)$)

(1) For each high-degree vertex (degree at least $E/8$), enumerate all triangles satisfying the (c_0, c_1, c_2) coloring, and construct G' by removing these high-degree vertices.

(2) On G', assign new colorings to the vertices by adding a random bit to its least significant position in its current coloring.

(3) Recursively call PS-Enum on G' on the 8 colorings in $(c_0', c_1', c_2') \in \{2c_0-1, 2c_0\} \times \{2c_1-1, 2c_1\} \times \{2c_2-1, 2c_2\}$, where each subproblem contains only compatible edges.

The algorithm is initially called on the original edge set E with a coloring $(1, 1, 1)$, and all vertices assigned a color of 1.

Step 1 applies the subroutine described above to at most 16 vertices, and so requires $O(\text{sort}(E))$ cache misses. Step 2 requires $O(\text{scan}(V))$ cache misses. Pagh and Silvestri show that each subproblem in step 3 contains at most $E/4$ edges in expectation and uses this to show an expected cache complexity of $O(E^{3/2}/(\sqrt{M}B))$. The work of their algorithm is $O(E^{3/2})$.

We show how to parallelize each of the three steps of the PS algorithm. Step 1 requires at most 16 calls to the subroutine that finds all triangles incident on a vertex, hence can be done in the bounds stated in Lemma 6. Step 2 can be implemented with a parallel scan in $O(V)$ work, $O(\log V)$ depth and $O(\text{scan}(V))$ cache misses. The new colors of the endpoints of the edges can be computed by sorting the edges by the first endpoint, merging with the array of colors, then sorting by the second endpoint and doing the same. For each subproblem in Step 3, generating the subset of edges belonging to the subproblem can be done with a parallel filter in $O(E)$

work, $O(\log E)$ depth and $O(\text{scan}(E))$ cache misses. As the expected size of each subproblem is at most $E/4$, there are $O(\log E)$ levels of recursion whp. This gives an overall depth of $O(\log^{5/2} E)$ whp. The parallel algorithm requires $O(E^{3/2})$ work since every sequential routine that we replace with a parallel routine has the same asymptotic work bound. The parallel cache complexity is $O(E^{3/2}/(\sqrt{M}B))$ in expectation as the cache complexity of the parallel routines match those of the sequential routines. This gives us the following theorem:

Theorem 7 *A parallel version of the PS algorithm can be implemented in $O(E^{3/2})$ work, $O(\log^{5/2} E)$ depth and a parallel cache complexity of $O(E^{3/2}/(\sqrt{M}B))$ in expectation.*

While the cache complexity of the parallel PS algorithm is better than that of TC-Merge and TC-Hash, in practice we found it to be much slower due to large constants in the bounds, as discussed in Section VII.

IX. Prior and Related Work

Exact sequential algorithms. Sequential algorithms for exact triangle counting and enumeration have a long history (see, e.g., [26, 34, 39, 46, 47]). For sparse graphs, of particular interest is the line of work starting from Schank and Wagner [47], who describe an algorithm, called *forward*, that achieves a work bound of $O(E^{3/2})$ with a space bound of $O(V+E)$. The algorithm, like our algorithms, ranks the vertices in order of non-decreasing degree, but it populates the neighborhood A^+ sequentially while computing the intersection. Improving upon the constants in the space bounds, Latapy [34] describes an algorithm *compact-forward*, on which our algorithms are based. Both algorithms are sequential and require $O(E^{3/2})$ work and $O(V+E)$ space. By using hash tables for intersection, the work of both algorithms can be improved to $O(\alpha E)$ [14]. Experimentally, Latapy shows that forward and compact-forward yield the best running time with compact-forward consuming less space [34], consistent with Schank's findings [46].

The *node-iterator* algorithm [46] iterates over all vertices $v \in V$, and intersects the adjacency lists of each pair of v's neighbors. This algorithm requires $O(\sum_{v\in V}(d(v)^2 + \sum_{w\in N(v)} d(w))) = O(Ed_{max})$ work and $O(V+E)$ space. Green and Bader describe an optimization to this algorithm using vertex covers, which improves its performance in practice [22]. The *edge-iterator* algorithm [26] iterates over the edges instead of the vertices. For each edge, it intersects the adjacency lists of the two endpoints.

Ortmann and Brandes [39] describe a framework for designing triangle listing algorithms and explore many variations of the previous algorithms. They show that a variant of forward and compact-forward performs the best in practice.

For a graph with Δ triangles, Bjorklund et al. [8] give the best work bounds for triangle listing, requiring roughly $O(V^\omega + V^{\frac{3(\omega-1)}{5-\omega}}\Delta^{\frac{2(3-\omega)}{5-\omega}})$ work for dense graphs, and $O(E^{\frac{2\omega}{\omega+1}} + E^{\frac{3(\omega-1)}{\omega+1}}\Delta^{\frac{3-\omega}{\omega+1}})$ work for sparse graphs, where ω is the matrix multiplication exponent ($\omega \approx 2.3729$, using the current-best algorithm [61]).

Triangle counting, but not listing, can also be solved using matrix multiplication in $O(V^\omega)$ work [26]. For sparse graphs, this can be improved to $O(E^{\frac{2\omega}{\omega+1}})$ [1]. Other algorithms and variants can be found in [34, 39, 46] and the references therein.

Exact parallel algorithms. There has been recent work on adapting sequential triangle counting/listing/enumeration algorithms to the parallel setting. Several algorithms have been designed for distributed-memory using MapReduce [16, 42, 43, 53, 59]. Arifuzzaman et al. describe a distributed-memory algorithm using MPI [2], and GraphLab also contains an MPI implementation [21]. A multicore implementation of the node-iterator algorithm is presented by Green et al. [23]. Triangle counting has also been implemented on the GPU [24, 62].

I/O complexity of triangle computations. Various triangle counting/listing/enumeration algorithms have been designed for I/O efficiency, either in terms of disk accesses or cache misses. Triangle enumeration can be computed by using a natural join of three relations using $O(E^3/(M^2B))$ I/O's [40]. An external-memory version of compact-forward was described by Menegola [36], requiring $O(E + E^{3/2}/B)$ I/O's. An external-memory version of node-iterator was described by Dementiev [17], requiring $O((E^{3/2}/B)\log_{M/B}(E/B))$ I/O's. Chu and Cheng [15] describe an algorithm using graph partitioning with an I/O complexity $O(E^2/(MB) + \Delta/B)$, where Δ is the number of triangles in the graph. Their algorithm requires that each partitions fits in memory, that $V \le M$, and that $M = \Omega(\sqrt{EB})$. Later, Hu et al. [25] describe an algorithm achieving the same I/O complexity of $O(E^2/(MB) + \Delta/B)$, without the restrictions of the previous algorithm. These algorithms are designed for the external-memory model, where the algorithm must be tuned for the parameters M and B of the specific machine. Recently, Pagh and Silvestri [40] describe a cache-oblivious algorithm requiring $O(E^{3/2}/(\sqrt{M}B))$ expected I/O's (cache misses), which we describe in Section VIII. They also describe a deterministic cache-aware algorithm requiring $O(E^{3/2}/(\sqrt{M}B))$ I/O's (cache misses) with the requirement $M \ge E^{\Omega(1)}$ [40]. None of the above algorithms have been parallelized. Kyrola et al. [33] and Kim et al. [28] present parallel disk-based triangle counting implementations, which require parameter tuning.

Approximate counting schemes. To speed up triangle counting, many approximation schemes have been proposed. These do not work for triangle listing/enumeration, as not all triangles are even generated. DOULION is among the first approximation schemes proposed [57]. Pagh and Tsourakakis [41] later give a more accurate scheme that improves upon DOULION, called colorful triangle counting, which we describe in Section V. A recent scheme based on sampling wedges was presented by Seshadri et al. [49]. Hadoop implementations have been described for some of these schemes (e.g., [29, 41, 59]). Several other approximation schemes have been proposed based on computing eigenvalues of the graph [3, 56, 58]. The performance of these methods depend on the spectrum of the graphs. Rahman and Al Hasan recently present approximate counting algorithms for multicores based on the edge-iterator algorithm [45], which we compare with in Section VII.

Streaming algorithms. Triangle counting has also been studied in streaming settings as an alternative means to processing massive graphs (see, e.g., [5, 6, 13, 19, 27, 30, 31, 44, 54] among many others).

X. Conclusion

We have described fast parallel cache-oblivious algorithms for triangle computations, proven complexity bounds for the algorithms in the Parallel Cache Complexity model, and shown that they are efficient in practice. We believe that with the rapid growth in capacity of shared-memory machines, our fast algorithms will continue to be very useful in the analysis of triangle structures in large graphs.

Acknowledgments. This work is supported by a Facebook Graduate Fellowship, the National Science Foundation under grant number CCF-1314590, and the Intel Labs Academic Research Office for the Parallel Algorithms for Non-Numeric Computing Program. We thank Oded Green for providing and helping us with the code of [23], and Mark Ortmann for providing and assisting us with the code of [39]. We thank Rasmus Pagh and Francesco Silvestri for discussions about their algorithm [40]. We are grateful to Harsha Simhadri for discussions about the PCC model and helping us with cache performance tools.

References

[1] N. Alon, R. Yuster, and U. Zwick, "Finding and counting given length cycles," *Algorithmica*, vol. 17, no. 3, 1997.
[2] S. Arifuzzaman, M. Khan, and M. Marathe, "PATRIC: A parallel algorithm for counting triangles in massive networks," in *CIKM*, 2013.
[3] H. Avron, "Counting triangles in large graphs using randomized matrix trace estimation," in *KDD-LDMTA*, 2010.
[4] C. Balkesen, G. Alonso, J. Teubner, and M. T. Özsu, "Multi-core, main-memory joins: Sort vs. hash revisited," *PVLDB*, vol. 7, no. 1, 2013.
[5] Z. Bar-Yossef, R. Kumar, and D. Sivakumar, "Reductions in streaming algorithms, with an application to counting triangles in graphs," in *SODA*, 2002.
[6] L. Becchetti, P. Boldi, C. Castillo, and A. Gionis, "Efficient semi-streaming algorithms for local triangle counting in massive graphs," in *KDD*, 2008.
[7] J. W. Berry, L. K. Fostvedt, D. J. Nordman, C. A. Phillips, C. Seshadhri, and A. G. Wilson, "Why do simple algorithms for triangle enumeration work in the real world?" in *ITCS*, 2014.
[8] A. Bjorklund, R. Pagh, V. V. Williams, and U. Zwick, "Listing triangles," in *ICALP*, 2014.
[9] G. E. Blelloch, J. T. Fineman, P. B. Gibbons, and H. V. Simhadri, "Scheduling irregular parallel computations on hierarchical caches," in *SPAA*, 2011.
[10] G. E. Blelloch, P. B. Gibbons, and H. V. Simhadri, "Low-depth cache oblivious algorithms," in *SPAA*, 2010.
[11] G. E. Blelloch and B. M. Maggs, "Parallel algorithms," in *The Computer Science and Engineering Handbook*, 1997.
[12] R. D. Blumofe and C. E. Leiserson, "Scheduling multithreaded computations by work stealing," *J. ACM*, vol. 46, no. 5, Sep. 1999.
[13] L. S. Buriol, G. Frahling, S. Leonardi, A. Marchetti-Spaccamela, and C. Sohler, "Counting triangles in data streams," in *PODS*, 2006.
[14] N. Chiba and T. Nishizeki, "Arboricity and subgraph listing algorithms," *SIAM J. Comput.*, vol. 14, no. 1, Feb. 1985.
[15] S. Chu and J. Cheng, "Triangle listing in massive networks," *Trans. Knowl. Discov. Data*, vol. 6, no. 4, Dec. 2012.
[16] J. Cohen, "Graph twiddling in a MapReduce world," *Computing in Science and Eng.*, vol. 11, no. 4, Jul. 2009.
[17] R. Dementiev, "Algorithm engineering for large data sets," *PhD Thesis, Saarland University*, 2006.
[18] J.-P. Eckmann and E. Moses, "Curvature of co-links uncovers hidden thematic layers in the world wide web," *PNAS*, vol. 99, no. 9, 2002.
[19] D. Ediger, K. Jiang, J. Riedy, and D. A. Bader, "Massive streaming data analytics: A case study with clustering coefficients," in *MTAAP*, 2010.
[20] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran, "Cache-oblivious algorithms," in *FOCS*, 1999.
[21] J. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "PowerGraph: Distributed graph-parallel computation on natural graphs," in *OSDI*, 2012.
[22] O. Green and D. A. Bader, "Faster clustering coefficient using vertex covers," in *SocialCom*, 2013.
[23] O. Green, L. M. Munguia, and D. A. Bader, "Load balanced clustering coefficients," in *PPAA*, 2014.
[24] O. Green, P. Yalamanchili, and L. M. Munguia, "Fast triangle counting on the GPU," in *IAAA*, 2015.
[25] X. Hu, Y. Tao, and C.-W. Chung, "Massive graph triangulation," in *SIGMOD*, 2013.
[26] A. Itai and M. Rodeh, "Finding a minimum circuit in a graph," in *STOC*, 1977.
[27] M. Jha, C. Seshadhri, and A. Pinar, "A space efficient streaming algorithm for triangle counting using the birthday paradox," in *KDD*, 2013.
[28] J. Kim, W.-S. Han, S. Lee, K. Park, and H. Yu, "OPT: A new framework for overlapped and parallel triangulation in large-scale graphs," in *SIGMOD*, 2014.
[29] T. G. Kolda, A. Pinar, T. Plantenga, C. Seshadhri, and C. Task, "Counting triangles in massive graphs with MapReduce," *SIAM Journal on Scientific Computing*, vol. 36, no. 5, 2014.
[30] M. N. Kolountzakis, G. L. Miller, R. Peng, and C. E. Tsourakakis, "Efficient triangle counting in large graphs via degree-based vertex partitioning," *Internet Mathematics*, vol. 8, no. 1-2, 2012.
[31] K. Kutzkov and R. Pagh, "Triangle counting in dynamic graph streams," in *SWAT*, 2014.
[32] H. Kwak, C. Lee, H. Park, and S. Moon, "What is Twitter, a social network or a news media?" in *WWW*, 2010.
[33] A. Kyrola, G. Blelloch, and C. Guestrin, "GraphChi: Large-scale graph computation on just a PC," in *OSDI*, 2012.
[34] M. Latapy, "Main-memory triangle computations for very large (sparse (power-law)) graphs," *Theor. Comput. Sci.*, 2008.
[35] Y. Matias and U. Vishkin, "On parallel hashing and integer sorting," *Journal of Algorithms*, vol. 12, no. 4, 1991.
[36] B. Menegola, "An external memory algorithm for listing triangles," *Tech. report, Universidade Federal do Rio Grande do Sul*, 2010.
[37] M. E. J. Newman, "The structure and function of complex networks," *SIAM REVIEW*, vol. 45, 2003.
[38] H. Q. Ngo, C. Ré, and A. Rudra, "Skew strikes back: New developments in the theory of join algorithms," *SIGMOD Rec.*, vol. 42, no. 4, Feb. 2014.
[39] M. Ortmann and U. Brandes, "Triangle listing algorithms: Back from the diversion," in *ALENEX*, 2014.
[40] R. Pagh and F. Silvestri, "The input/output complexity of triangle enumeration," in *PODS*, 2014.
[41] R. Pagh and C. E. Tsourakakis, "Colorful triangle counting and a MapReduce implementation," *Inf. Process. Lett.*, 2012.
[42] H.-M. Park and C.-W. Chung, "An efficient MapReduce algorithm for counting triangles in a very large graph," in *CIKM*, 2013.
[43] H.-M. Park, F. Silvestri, U. Kang, and R. Pagh, "MapReduce triangle enumeration with guarantees," in *CIKM*, 2014.
[44] A. Pavan, K. Tangwongsan, S. Tirthapura, and K.-L. Wu, "Counting and sampling triangles from a graph stream," *PVLDB*, vol. 6, no. 14, 2013.
[45] M. Rahman and M. Al Hasan, "Approximate triangle counting algorithms on multi-cores," in *Big Data*, 2013.
[46] T. Schank, "Algorithmic aspects of triangle-based network analysis," *PhD Thesis, Universitat Karlsruhe*, 2007.
[47] T. Schank and D. Wagner, "Finding, counting and listing all triangles in large graphs, an experimental study," in *WEA*, 2005.
[48] C. Seshadhri, A. Pinar, N. Durak, and T. G. Kolda, "The importance of directed triangles with reciprocity: patterns and algorithms," *CoRR*, vol. abs/1302.6220, 2013.
[49] C. Seshadri, A. Pinar, and T. G. Kolda, "Triadic measures on graphs: The power of wedge sampling," in *SDM*, 2013.
[50] J. Shun and G. E. Blelloch, "Phase-concurrent hash tables for determinism," in *SPAA*, 2014.
[51] J. Shun, G. E. Blelloch, J. T. Fineman, P. B. Gibbons, A. Kyrola, H. V. Simhadri, and K. Tangwongsan, "Brief announcement: the Problem Based Benchmark Suite," in *SPAA*, 2012.
[52] H. V. Simhadri, "Program-centric cost models for locality and parallelism," *PhD Thesis, Carnegie Mellon University*, 2013.
[53] S. Suri and S. Vassilvitskii, "Counting triangles and the curse of the last reducer," in *WWW*, 2011.
[54] K. Tangwongsan, A. Pavan, and S. Tirthapura, "Parallel triangle counting in massive streaming graphs," in *CIKM*, 2013.
[55] C. E. Tsourakakis, "Fast counting of triangles in large real networks without counting: Algorithms and laws," in *ICDM*, 2008.
[56] ——, "Counting triangles in real-world networks using projections," *Knowl. Inf. Syst.*, vol. 26, no. 3, 2011.
[57] C. E. Tsourakakis, U. Kang, G. L. Miller, and C. Faloutsos, "DOULION: Counting triangles in massive graphs with a coin," in *KDD*, 2009.
[58] C. Tsourakakis, P. Drineas, E. Michelakis, I. Koutis, and C. Faloutsos, "Spectral counting of triangles via element-wise sparsification and triangle-based link recommendation," *SNAM*, vol. 1, no. 2, 2011.
[59] W. Wang, Y. Gu, Z. Wang, and G. Yu, "Parallel triangle counting over large graphs," in *DASFAA*, 2013.
[60] D. J. Watts and S. H. Strogatz, "Collective dynamics of 'small-world' networks," *Nature*, vol. 393, no. 6684, 1998.
[61] V. V. Williams, "Multiplying matrices faster than Coppersmith-Winograd," in *STOC*, 2012.
[62] H. Wu, D. Zinn, M. Aref, and S. Yalamanchili, "Multipredicate join algorithms for accelerating relational graph processing on GPUs," in *ADMS*, 2014.