Triangle Counting Through Cover-Edges

David A. Bader*, Fuhuan Li, Anya Ganeshan+, Ahmet Gundogdu#, Jason Lew, Oliver Alvarado Rodriguez, Zhihui Du
Department of Data Science
New Jersey Institute of Technology
Newark, New Jersey, USA
{bader,fl28,jl247,oaa9,zhihui.du}@njit.edu

***Abstract*—Counting and finding triangles in graphs is often used in real-world analytics to characterize cohesiveness and identify communities in graphs. In this paper, we propose the novel concept of a cover-edge set that can be used to find triangles more efficiently. We use a breadth-first search (BFS) to quickly generate a compact cover-edge set. Novel sequential and parallel triangle counting algorithms are presented that employ cover-edge sets. The sequential algorithm avoids unnecessary triangle-checking operations, and the parallel algorithm is communication-efficient. The parallel algorithm can asymptotically reduce communication on massive graphs such as from real social networks and synthetic graphs from the Graph500 Benchmark. In our estimate from massive-scale Graph500 graphs, our new parallel algorithm can reduce the communication on a scale 36 graph by 1156x and on a scale 42 graph by 2368x.**

***Index Terms*—Graph Algorithms, Triangle Counting, Parallel Algorithms, High Performance Data Analytics**

I. Introduction

Triangle counting [1] is a fundamental problem in graph analytics, which involves finding the number of unique triangles in a graph. It plays a crucial role in various graph analysis techniques such as clustering coefficients [2], k-truss [3], and triangle centrality [4]. The significance of triangle counting is evident in its application in high-performance computing benchmarks like Graph500 [5] and the MIT/Amazon/IEEE Graph Challenge [6], as well as in the design of future architecture systems (e.g., IARPA AGILE [7]).

Both sequential and parallel triangle counting algorithms have been studied extensively since 1977 [8]. Latapy [9] provides a comprehensive overview of sequential triangle counting and various finding algorithms. Existing techniques, including list intersection, matrix multiplication, and subgraph matching [10], are techniques used to count triangles.

To enhance the performance of triangle counting, Cohen [11] introduced a novel map-reduce parallelization technique that generates *open wedges* between triples of vertices in the graph. It determines whether a closing edge exists to complete a triangle, thus avoiding the redundant counting of the same triangle while maintaining load balancing. Many parallel approaches for triangle counting [12], [13] partition the sparse graph data structure across multiple compute nodes and adopt the strategy of generating open wedges, which are sent to other compute nodes to determine the presence of a closing edge. Consequently, the communication time for these open wedges often dominates the running time of parallel triangle counting.

In traditional edge-based triangle counting methods, all triangles are identified by accumulating the sizes of intersections between pairs of endpoints for each edge. *Direction-oriented* approaches can avoid counting the same triangle multiple times. However, in this paper, we propose a novel approach that efficiently identifies all triangles using a reduced set of edges known as a cover-edge set. By leveraging the cover-edge-based triangle counting method, unnecessary edge checks can be skipped while ensuring that no triangles are missed. This significantly reduces the number of computational operations compared to existing methods. Furthermore, for distributed parallel algorithms, the cover-edge-based method can greatly reduce overall communication requirements. As a result, our proposed method offers improved efficiency and scalability for triangle counting.

Our contributions include:

- A novel concept, *Cover-Edge Set*, is proposed to support efficient triangle counting. The essential idea is that we can identify all triangles from a significantly reduced cover-edge set instead of the complete edge set. A simple breadth-first search (BFS) is used to orient the graph's vertices into levels and to generate the cover-edge set.
- A novel triangle counting and finding algorithm, *CETC*, is developed based on the concept of *Cover-Edge Set*. *CETC* runs in $\mathcal{O}(m \cdot d_{\max})$ time and $\mathcal{O}(n + m)$ space, where $d_{\max}$ is the maximal degree of a vertex $v \in V$.
- A novel communication-efficient distributed parallel algorithm for triangle counting and finding, *Comm-CETC*, is also developed based on the concept of *Cover-Edge Set*. *Comm-CETC* can asymptotically reduce the communication to improve total performance.

The remainder of the paper is organized as follows. Section II presents our new approach for triangle counting. In Section III, we employ our idea in distributed parallel triangle counting to reduce communication. Section V discusses related work. Lastly, in Section VI, we conclude the paper.

+Anya Ganeshan attends Bergen County Academies, Hackensack, NJ

#Ahmet Gundogdu attends Paramus High School, Paramus, NJ

*This research was partially supported by NSF grant number CCF-2109988.

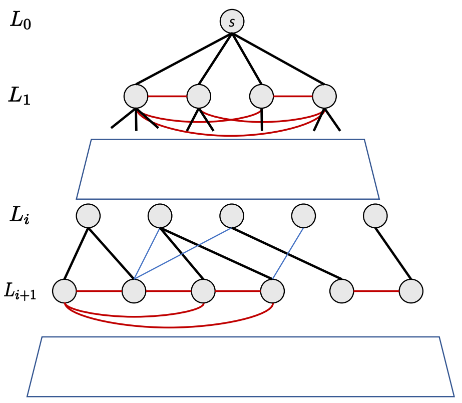

Fig. 1. Example BFS tree of Graph G. The tree-edges are black, strut-edges are blue, and horizontal-edges are red.

II. Cover-Edge Set Based Triangle Counting

A. Notations and Basic Idea

Let $G = (V, E)$ be an undirected graph with $n = |V|$ vertices and $m = |E|$ edges. A *triangle* in the graph is a set of three vertices $\{v_a, v_b, v_c\} \subseteq V$ such that $\{(v_a, v_b), (v_a, v_c), (v_b, v_c)\} \subseteq E$. We will use $N(v) = \{u | u \in V \wedge ((v, u) \in E)\}$ to denote the *neighbor set* of vertex $v \in V$. The degree of vertex $v \in V$ is $d(v) = |N(v)|$, and $d_{\max}$ is the maximal degree of a vertex in graph G.

Definition 1 (Cover-Edge and Cover-Edge Set)**.** *For any edge e of a triangle Δ in graph G, e is referred to as a cover-edge of Δ. For a given graph G, an edge set $S \subseteq E$ is called a cover-edge set if it contains at least one cover-edge for every triangle in G.*

Based on the given definition, it is evident that the entire edge set E can serve as a cover-edge set S for graph G. However, our proposed method aims to efficiently count all triangles using a smaller subset of edges instead of E. Thus, the primary challenge lies in generating a compact cover-edge set, which forms the initial problem to be addressed in our approach. Let $k = |S|/|E|$. Our goal is to identify cover-edge sets with the smallest k. In this paper, we propose using breadth-first search (BFS) to generate a compact cover-edge set.

Definition 2 (BFS-Edge)**.** *Let r be the root vertex of an undirected graph G. The level $L(v)$ of a vertex v is defined as the shortest distance from r to v obtained through a breadth-first search (BFS). From the BFS, we classify the edges into three types:*

- Tree-Edges*: These edges belong to the BFS tree.*
- Strut-Edges*: These are non-tree edges with endpoints on two adjacent levels in the BFS traversal.*
- Horizontal-Edges*: These are non-tree edges with endpoints on the same level in the BFS traversal.*

Fig. 1 gives an example of these different edge types.

Lemma 1. *Each triangle $\{u, v, w\}$ in a graph contains at least one horizontal-edge.*

Proof. (Proof by contradiction) A triangle is a path of length 3 that starts and ends at the same vertex. Suppose there are no horizontal-edges in the triangle. In that case, every edge in the path (i.e., a tree-edge or strut-edge) either increases or decreases the level by one.

Since the path must end on the same level as the starting vertex, the number of edges in the path that decrease the level must be equal to the number of edges that increase the level. Consequently, the length of the path must be even to maintain level parity. However, this contradicts the fact that a triangle has an odd path length of 3.

Therefore, we conclude that there must be at least one horizontal-edge in every triangle. □

Theorem 3 (Cover-Edge Set Generation)**.** *All horizontal-edges form a valid cover-edge set.*

Proof. According to Definition 1, for any triangle Δ in graph G, we can always find at least one horizontal-edge that serves as a cover-edge for Δ. Thus, the set of all horizontal-edges constitutes a cover-edge set. □

Therefore, we can construct a cover-edge set, denoted as *BFS-CES*, by selecting all the horizontal-edges obtained during a breadth-first search (*BFS*). It is evident that *BFS-CES* is a subset of E and is typically much smaller than the complete edge set E.

B. Cover-Edge based Triangle Counting

In this subsection, we provide a comprehensive description of the process involved in identifying all triangles using a cover-edge set generated through a breadth-first search.

Lemma 2. *Each triangle $\{u, v, w\}$ must contain either one or three horizontal-edges.*

Proof. By referring to the proof of Lemma 1, we know that the path corresponding to the triangle's three edges consists of an even number of tree-edges and strut-edges. This implies that there can be either 0 or 2 tree- or strut-edges within each triangle.

In the case where there are 0 tree- or strut-edges, all three edges of the triangle must be horizontal-edges. This is because the absence of tree- or strut-edges implies that the entire path is composed of horizontal-edges.

In the case where there are 2 tree- or strut-edges, the triangle contains exactly one horizontal-edge. This is because having two tree- or strut-edges in the path means that there is one horizontal-edge connecting the remaining two vertices.

Therefore, we conclude that each triangle $\{u, v, w\}$ must contain either one or three horizontal-edges. □

Our triangle counting approach, described in Alg. 1, efficiently counts triangles using a cover-edge set. In line 1, we initialize the counter T to 0, which will store the total number of triangles. To generate the cover-edge set, we perform a breadth-first search (BFS) starting from any unvisited vertex, identifying the level ($L(v)$) of each vertex v in its respective component, as shown in lines 2 to 3. In lines 4 to 8 the algorithm iterates over each edge, selecting the cover-set of horizontal edges (u, v) in a direction-oriented fashion in line 5. For each vertex w in the intersection of u and v's neighborhoods (line 6), we check the following two conditions to determine if (u, v, w) is a unique triangle to be counted (line 7). If $L(u) \neq L(w)$ then the edge (u, v) is the only horizontal-edge in the triangle (u, v, w). If $L(u) \equiv L(w)$, then the edge (u, v) is one of three horizontal-edges in the triangle (u, v, w). To ensure uniqueness, the algorithm then checks the added constraint that $v < w$. If the constraints are satisfied, we increment the triangle counter T in line 8.

This approach effectively counts the triangles in the graph while avoiding redundant counting.

Algorithm 1 CETC:Cover-Edge Triangle Counting

```
Input: Graph $G = (V, E)$
Output: Triangle Count $T$
1: $T \leftarrow 0$
2: $\forall v \in V$
3:     if $v$ unvisited, then BFS($G$, $v$)
4: $\forall (u, v) \in E$
5:     if $(L(u) \equiv L(v)) \wedge (u < v)$                    ▷ $(u, v)$ is horizontal
6:         $\forall w \in N(u) \cap N(v)$
7:             if $(L(u) \neq L(w)) \vee ((L(u) \equiv L(w)) \wedge (v < w))$ then
8:                 $T \leftarrow T + 1$
```

Theorem 4 (Correctness)**.** *Alg. 1 can accurately count all triangles in a graph G.*

Proof. Lemma 2 establishes that a triangle in the graph falls into one of two cases: 1) the two endpoint vertices of the horizontal-edge are on the same level while the apex vertex is on a different level, or 2) all three vertices of the triangle are at the same level.

Consider a triangle $\{v_a, v_b, v_c\}$ in G. Without loss of generality, assume that (v_a, v_b) is a horizontal-edge, implying $L(v_a) \equiv L(v_b)$. Let v_c be the apex vertex. The two cases can be distinguished as follows:

For the first case, each triangle is uniquely defined by a horizontal-edge and an apex vertex from the common neighbors of the horizontal-edge's endpoint vertices. Whenever Alg. 1 identifies such a triangle $\{v_a, v_b, v_c\}$, it increments the total triangle count T by 1.

In the second case, where all three vertices are at the same level ($L(v_c) \equiv L(v_a) \equiv L(v_b)$), Alg. 1 ensures that T is increased by 1 only when $v_a < v_b < v_c$. This condition ensures that triangle $\{v_a, v_b, v_c\}$ is counted only once, preventing triple-counting and ensuring the correctness of the triangle count.

Hence, Alg. 1 is proven to accurately count all triangles in the graph G. □

The time complexity of Alg. 1 can be analyzed as follows. The computation of breadth-first search, including determining the level of each vertex and marking horizontal-edges, requires $\mathcal{O}(n + m)$ time.

Since there are at most $\mathcal{O}(m)$ horizontal-edges, finding the common neighbors of each horizontal-edge individually can be done in $\mathcal{O}(d_{\max})$ time. Here, $d_{\max}$ represents the maximal degree of a vertex in the graph.

Therefore, the overall time complexity of Alg. 1 is $\mathcal{O}(m \cdot d_{\max})$.

III. Communication Efficient Triangle Counting Algorithm

This section presents our communication-efficient parallel algorithm for counting triangles in massive graphs on a p-processor distributed-memory parallel computer. We will take advantage of the concept of *Cover-Edge Set* to significantly improve the communication performance of our triangle counting method. Since distributed triangle counting is communication-bound [12], this algorithm is expected to improve the overall running time. The input graph G is stored in a compressed sparse row (CSR) format. The vertices are partitioned non-uniformly to the p processors such that each processor stores approximately $2m/p$ edge endpoints. This graph input follows the format used by the majority of parallel graph algorithm implementations and benchmarks such as Graph500 and Graph Challenge.

A. Parallel Algorithm Description

Our communication-efficient parallel algorithm (see Alg. 2) is based on the same cover-edge approach proposed in section II. The binary operator $\oplus$ used in line 9 is bitwise exclusive OR (XOR).

Algorithm 2 Comm-CETC: Communication Efficient Triangle Counting

```
Input: Graph $G = (V, E)$
Output: Triangle Count $T$
 1: Run parallel BFS($G$) and build partial cover-edge set $S_i$ on $p_i$
 2: For all $p_i, i \in \{0 \ldots p-1\}$ in parallel do:
 3:     $t_i \leftarrow 0$
 4:     $\forall (u, v) \in S_i$ with $u < v$ on $p_i$
 5:         $\forall w \in V_i$ such that $w \in N(u), N(v)$
 6:             if $(L(u) \neq L(w)) \vee ((L(u) \equiv L(w)) \wedge (v < w))$ then
 7:                 $t_i = t_i + 1$
 8:     For $j \leftarrow 1$ to $p - 1$ do:
 9:         Processors $i$ and $i \oplus j$ swap edge sets $S_i$ and $S_j$.
10:         $\forall (u, v) \in S_j$ with $u < v$ on $p_i$
11:             $\forall w \in V_i$ such that $w \in N(u), N(v)$
12:                 if $(L(u) \neq L(w)) \vee ((L(u) \equiv L(w)) \wedge (v < w))$ then
13:                     $t_i = t_i + 1$
14: $T \leftarrow$ Reduce($t_i$, +)
```

Similar to the baseline *CETC* algorithm, the cover-edge set $S = \cup_{i=0}^{p-1} S_i$ is determined in line 1 by labeling the horizontal edges from a parallel BFS.

Each processor runs lines 2 to 13 in parallel that consists of two main substeps. Local triangles are counted in lines 4 to 7 and a total exchange of cover-edges between each pair of processors to count triangles is performed in lines 8 to 13. Note at the end of each iteration of the *for* loop, processor p_i can discard the cover-edge set S_j. In lines 5 and 11, processor p_i determines for each cover edge (u, v) all the apex vertices w held locally that are adjacent to both u and v. The logic for counting triangles in lines 6 and 12 is similar to Alg. 1 as to only count unique triangles. Finally, a reduction operation in line 14 calculates the total number of triangles by accumulating the triangle counters across the system, i.e., $T = \sum_{i=0}^{p-1} t_i$.

B. Cost Analysis

1) Space: In addition to the input graph data structure, an additional bit is needed per edge (for marking a horizontal-edge) and $\mathcal{O}(\lceil \log D \rceil)$ bits per vertex to store its level, where D is the diameter of the graph. This is a total of at most $m + n\lceil \log D \rceil$ bits across the p processors. Preserving the graph requires additional $\mathcal{O}(n+m)$ space for the graph.

2) Compute: The BFS costs $\mathcal{O}((n+m)/p)$ [14], the modified neighbor sets take $\mathcal{O}(m/p)$. The search corresponding to one cover-edge in a vertex's adjacency list takes at most $\mathcal{O}(\log(d_{max}))$ time using binary search, and only $\mathcal{O}(1)$ expected time using a hash table. Let d_i be the degree of vertex v_i where $0 \le i < n$. Searching km edges in all vertices' adjacency lists takes $\mathcal{O}(km \sum_{i=0}^{n-1} \log(d_i)) = \mathcal{O}(km \log(\Pi_{i=0}^{n-1} d_i))$ time. Since $\sum_{i=0}^{n-1} d_i = 2m$, we know that $\log(\Pi_{i=0}^{n-1} d_i)$ reaches its maximum value when $d_i = 2m/n$ for $0 \le i < n$. Thus, $\mathcal{O}(km \log(\Pi_{i=0}^{n-1} d_i)) \le \mathcal{O}(km \log((2m/n)^n)) \le \mathcal{O}(kmn \log(2n^2/n)) = \mathcal{O}(mn \log(n))$.

3) Total Communication: In our analysis of communication cost for BFS, we measure the total communication volume independent of the number of processors. Thus, this is a conservative overestimate of communication since a fraction (e.g., $1/p$) of accesses will be on the same compute node versus message traffic between nodes. At the same time, we do not consider the savings from overlapping with the computation cost.

The cost of the breadth-first search is m edge traversals with $\lceil \log D \rceil + 3\lceil \log n \rceil$ bits communicated per edge traversal for the level information, pair of vertex ids, and vertex degree, yielding $m \cdot (\lceil \log D \rceil + 3\lceil \log n \rceil)$ bits for the BFS. Transferring km horizontal-edges requires $kmp\lceil \log n \rceil$ bits, where p is the number of processors. The final reduction to find the total number of triangles requires $(p-1)\lceil \log n \rceil$ bits.

Hence, the total communication volume is $m \cdot (\lceil \log D \rceil + 3\lceil \log n \rceil) + kmp\lceil \log n \rceil + (p-1)\lceil \log n \rceil = m \cdot (\lceil \log D \rceil + (kp+3)\lceil \log n \rceil) + (p-1)\lceil \log n \rceil$ bits. Hence, since the word size is $\Theta(\log n)$ and $D \le n$, the communication is $\mathcal{O}(pm)$ words.

IV. Communication Analysis on Real and Synthetic Graphs

In this section, we analyze the performance of the parallel triangle counting algorithm on both real and synthetic graphs. We implemented our new triangle counting algorithm using Python to accurately compute the exact communication volume and determine an analytic model based on the size of the graph and number of processors, and the ratio or percentage (k) of cover-edges from the BFS. The results given in Table I are exact communication volumes from our new algorithm on

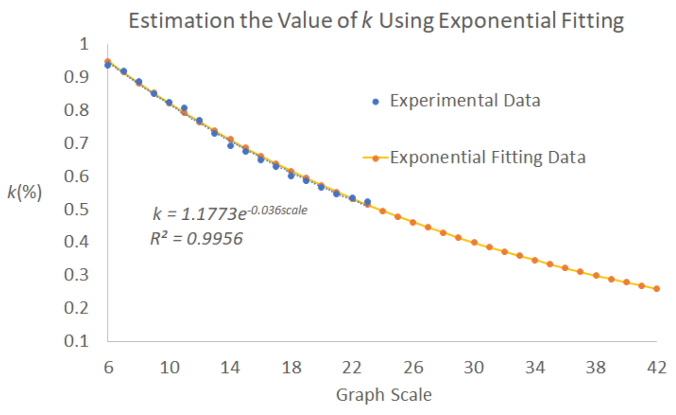

Fig. 2. Estimate of k using an exponential model, based on observations of k for RMAT graph scale 6 to 23 graphs.

all of the graphs except the two large RMAT graphs where we compute the communication volume from the validated analytic model. For the comparison with prior approaches [15]–[17], we estimate the communication volume from the number of wedges which is exact for all graphs other than the last two large RMAT graphs where we estimate the number of wedges using graph theory.

For the real graphs, we find the actual value of k, the percentage of graph edges that are cover-edges, for an arbitrary breadth-first search, and set the number p of processors to a reasonable number given the size of the graph. For the synthetic graphs, we use large Graph500 RMAT graphs [18] with parameters $a = 0.57$, $b = 0.19$, $c = 0.19$, and $d = 0.05$, for scale 36 and 42 with $n = 2^{\text{scale}}$ and $m = 16n$, similar with the IARPA AGILE benchmark graphs, and set p according to estimates of potential system sizes with sufficient memory to hold these large instances.

For comparison, most prior parallel algorithms for triangle counting operate on the graph as follows. A parallel loop over the vertices $v \in V$ produces all 2-paths (*wedges*) where $(v, v_1), (v, v_2) \in E$ and (w.l.o.g.) $v_1 < v_2$. The processor that produces this wedge will send an open wedge query message containing the vertex ids of v_1 and v_2 to the processor that owns vertex v_1. If the consumer processor that receives this query message finds an edge $(v_1, v_2) \in E$, then a local triangle counter is incremented. After producers and consumers complete all work, a global reduction over the p triangle counts computes the total number of triangles in G.

A. Graph500 RMAT Graphs

For the large Graph500 RMAT graphs, the number of triangles is estimated from our model based on the number of triangles found in RMAT graphs up to scale 29 in the literature [18]–[21]. The fitting equation is #Triangles $= 77.422n^{1.125}$ with $R^2 = 1.0$, where n is the total number of vertices. The number of triangles estimated for scale 36 and 42 RMAT graphs are 1.20×10^{14} and 1.30×10^{16}, respectively.

We estimate the number of wedges for the scale 36 and 42 Graph500 RMAT graphs based on the theorem given by

TABLE I
COMMUNICATION COSTS FOR REAL AND SYNTHETIC GRAPH. THE SYNTHETIC GRAPHS ARE GRAPH500 RMAT GRAPHS OF SCALE 36 AND 42. THE COLUMN **'PREVIOUS'** REPRESENTS THE COMMUNICATION VOLUME OF THE BEST PRIOR PARALLEL ALGORITHMS [15]–[17], THAT USE WEDGE-CHECKING BASED ALGORITHMS AND **'THIS PAPER'** REPRESENTS THE COMMUNICATION COST OF OUR NEW APPROACH. **'REDUCTION'** REPRESENTS THE COMMUNICATION REDUCTION BETWEEN THESE TWO, AND THUS, THE EXPECTED SPEEDUP OF THE PARALLEL ALGORITHM. ENTRIES IN *italics* ARE ESTIMATED VALUES.

Graph	**n**	**m**	**# Triangles**	**# Wedges**	k	p	**Previous**	**This paper**	**Reduction**
ca-GrQc	5242	14484	48260	165798	0.522	4	526KB	122KB	4.31
ca-HepTh	9877	25973	28339	277389	0.423	4	948KB	218KB	4.35
as-caida20071105	26475	53381	36365	776895	0.225	4	2.78MB	401KB	7.10
facebook_combined	4039	88234	1612010	17051688	0.914	4	48.8MB	893KB	56.0
ca-CondMat	23133	93439	173361	1567373	0.511	4	5.61MB	897KB	6.40
ca-HepPh	12008	118489	3358499	5081984	0.621	4	17.0MB	1.13MB	15.1
email-Enron	36692	183831	727044	5933045	0.478	4	22.6MB	1.79MB	12.7
ca-AstroPh	18772	198050	1351441	8451765	0.667	4	30.2MB	2.08MB	14.6
loc-brightkite_edges	58228	214078	494728	6956250	0.441	4	26.5MB	2.02MB	20.4
soc-Epinions1	75879	405740	1624481	21377935	0.498	4	86.7MB	4.25MB	10.7
amazon0601	403394	2443408	3986507	96348699	0.529	8	436MB	40.9MB	10.7
com-Youtube	1134890	2987624	3056386	209811585	0.347	8	1.03GB	44.3MB	23.7
RMAT-36	68719476736	1099511627776	*1.2E+14*	*2.73E+16*	*0.311*	128	218PB	*192TB*	*1156*
RMAT-42	4398046511104	70368744177664	*1.3E+16*	*5.79E+18*	*0.260*	256	52.8EB	*22.8PB*	*2368*

Seshadhri *et al.* in [22]. According to their formula, we can estimate the expected number of vertices $N(d)$ for a given out-degree d. The number of wedges that can be formed by vertices with such a degree is calculated as $\binom{d}{2} \times N(d)$, where $\binom{d}{2}$ means choosing two from d.

By summing all such wedges generated from the minimum ($e \ln n$) to the maximum degree ($\sqrt{n}$), which is the assumption of the formula, we can approximate the total number of wedges in the given graph, where n is the total number of vertices. This is a conservative estimate because it only considers the out-degree instead of the sum of out and in-degrees. Employing the formula, we calculate the number of wedges to be 2.73×10^{16} for scale 36 and 5.8×10^{18} for scale 42. With $2 \log n$ bits/wedge, the total volume of wedge checks is 218PB and 52.8EB for RMAT graphs of scales 36 and 42, respectively[1].

Beamer *et al.* [23] find a typical BFS on a scale 27 Graph500 RMAT graph has 7 levels, so 4 bits is a reasonable estimate for $\log D$ in our analyses of scale 36 and 42 graphs.

The methodology for estimating the value of k for RMAT graphs is as follows. RMAT graphs from scale 6 to 23 are generated, and the exact value of k is determined for each by counting the horizontal-edges after a breadth-first search. The data fit to an exponential model $k = 1.1773e^{-0.036 \cdot \text{scale}}$ with very high $R^2 = 0.9956$ (see Fig. 2). For scale 36, k is estimated to be 0.311 and for scale 42, k is estimated to be 0.260.

In our new approach for scale 36, where the communication cost is $m \cdot (\lceil \log D \rceil + (kp+3) \lceil \log n \rceil) + (p-1) \lceil \log n \rceil$ bits. With $\lceil \log D \rceil = 4$, and assuming $p = 128$ processors, we have a total communication volume of 192TB, for a communication reduction of $1156\times$. For scale 42, and assuming $p = 256$ processors, we estimate the communication of our new triangle counting algorithm as 22.8PB, for a communication reduction of $2368\times$.

[1]Throughout this paper, a petabyte (PB) is 2^{50} bytes and an exabyte (EB) is 2^{60} bytes.

V. RELATED WORK

A. *Sequential Algorithms*

The naïve approach for triangle counting uses brute-force: find all the triplets $\{v_a, v_b, v_c\}$, that is, permutations of three arbitrary vertices in the graph, and check whether each edge in the triplet exists. The time complexity is $\Omega(n^3)$. Latapy [9] and Schank and Wagner [24] provide surveys of faster sequential algorithms. Triangle counting generally can be formulated as three kinds of problems: set (list) intersection, matrix multiplication and subgraph (cycle) query.

The three main **intersection-based** triangle counting algorithms are: 1) the node-iterator algorithm iterates over all vertices and tests for each pair of neighbors whether they are connected by an edge, 2) the edge-iterator algorithm iterates over all edges and searches for common neighbors of the two endpoints of each edge, and 3) the forward algorithm is a refinement of the edge-iterator algorithm that computes the intersection of a subset of neighborhoods by using an orientation of the graph. The time complexity of node-iterator and edge-iterator are both $\mathcal{O}(m \cdot d_{\max})$ and the forward algorithm is $\mathcal{O}\left(m^{\frac{3}{2}}\right)$, which has significantly better performance when $d_{\max} \gg \sqrt{m}$ [9].

When performing the intersection of two lists, the commonly used techniques are *merge-path*, *binary search* and *hashing-based* algorithms.

Merge-path algorithms (e.g., [25], [26]) use two pointers to scan through neighbor lists of two endpoints from beginning to end in order to find the list intersection. During the scan, the pointer that points to a smaller value will be incremented. A triangle is enumerated if both pointers are incremented (i.e., they both point to the same vertex). Binary-search algorithms (e.g., [19], [27]) organize the longer list as a binary tree and use the shorter list as search keys. For each search key, it descends through the binary-search tree in order to find the equal entry, which is a triangle. Hashing-based algorithms (e.g., [10], [26]) construct a hash table for one list and use the other list as search keys to find the common elements in the hash table. The hash table is used here to find the intersection of two adjacency lists, so it is not necessary to sort all the adjacency lists to find all the triangles. The running time is proportional to the size of the two adjacency lists.

Triangle counting using **matrix multiplication** [28] relies on a linear algebra formulation for triangle counting. This approach can be optimized [29] using matrix decomposition by decomposing A into lower and upper triangular matrices L and U, and then computing $(L \times U) \odot L$, or $(L \times L) \odot L$ to determine the number of triangles. The binary operator $\odot$ denotes the Hadamard product.

A **subgraph-based** approach for triangle counting searches for all occurrences of a query graph, which is a triangle, in the input graph. Wang and Owens [30] use breadth-first search to update the subgraph matching approach by pruning more invalid vertices based on neighborhood encoding information, and using optimizations like k-step look-ahead to reduce unwanted intermediate results. Alon *et al.* [31] proposed a $\mathcal{O}\left(m^{1.41}\right)$ algorithm to find length 3 cycles (triangle) in a graph, which is an improvement over the Itai and Rodeh sequential $\mathcal{O}\left(m^{\frac{3}{2}}\right)$ algorithm [8].

B. Parallel Algorithms

Map-reduce is a standard platform for large scale distributed computation. Cohen [11] first demonstrated the capability of map-reduce to solve triangle counting in an approach that generates *open wedges* between triples of vertices in the graph and determines if a closing edge exists that completes a triangle. Suri *et al.* [32] implemented triangle counting using map-reduce that ranks vertices by degree and distributes them across hosts. Pearce [12] developed an algorithm that is based on creating an augmented degree-ordered directed graph, where the original undirected edges are directed from low-degree to high degree, and implemented this approach in the distributed asynchronous graph processing framework HavoqGT. DistTC [19] is a distributed triangle counting implementation for multiple machines that uses mirror proxy on each partition to eliminate almost all the inner-host communication. TriCore [27] partitions the graph held in a compressed-sparse row (CSR) data structure for multiple GPUs and uses stream buffers to load edge lists from CPU memory to GPU memory on-the-fly and then uses binary search to find the intersection. Hu *et al.* [33] employed a "copy-synchronize-search" pattern to improve the parallel threads efficiency of GPU and mixed the computing and memory intensive workloads together to improve the resource efficiency. Pandey *et al.* [10] employed an vertex-centric hash-based design to scale triangle counting to over 1,000 GPUs. TriC [13] exploits the vertex-based distributed triangle counting and sends vertices rather than edges (vertex pairs), and then the remote processor could translate the sequence of vertex IDs to correct combination of vertices as edges to reduce communication. An enhancement is then presented to TriC [34] that added a user-defined buffer to improve the flexibility of controlling the memory usage for large data sets and used a probabilistic data structure to optimize the edge lookups by trading off the accuracy. Strausz *et al.* [35] use CLaMPI, a software caching layer that caches data retrieved through MPI remote memory access operations, to reduce the overall communication cost. Zeng *et al.* [36] proposed a triangle counting algorithm that adaptively selects vertex-parallel and edge-parallel paradigm.

Panduranga *et al.* [37] and Dolev *et al.* [15]'s work focused on the communication cost. Compared with our work, there are two major differences. First, they use the number of communication rounds to measure the total communication with a bandwidth restriction. However, we use the total volume of messages to evaluate the communication. Second, they are probabilistic algorithms, but our algorithm is a deterministic algorithm (Dolev *et al.* [15] also contains a deterministic version). Probabilistic methods cannot be used under scenarios with an exact result requirement. Uhl [17], [38] also focuses on reducing the communication cost of triangle counting. The paper's basic idea is only requiring communication for counting triangles consisting of cut edges. If the partition generates many cut edges, the proposed method cannot significantly reduce communication. In contrast, our method identifies a subset of the total set of edges independent of the partitioning and only transfers this smaller set of edges during the triangle counting to significantly reduce the total communication.

VI. Conclusions

In this paper, we present novel sequential and parallel algorithms for counting and finding triangles in graphs based on a compact cover-edge set. The parallel algorithm is the first communication-efficient triangle counting algorithm by exploiting BFS horizontal-edges to significantly reduce the communication volume on massive graphs of practical interest. Our approach uses the breadth-first search to significantly reduce the number of edges examined and minimize the communication required for triangle checking. The parallel algorithm achieves an order of magnitude or more reduction of communication volume for large graphs as communication is the main bottleneck for triangle counting on distributed memory systems.

VII. Reproducibility

The sequential triangle counting source code and the Python code for determining the communication volume of the parallel algorithm are open source and available on GitHub at https://github.com/Bader-Research/triangle-counting. The input graphs are from the Stanford Network Analysis Project (SNAP) available from http://snap.stanford.edu/.

References

[1] M. Al Hasan and V. S. Dave, "Triangle counting in large networks: a review," *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, vol. 8, no. 2, p. e1226, 2018.

[2] D. J. Watts and S. H. Strogatz, "Collective dynamics of 'small-world' networks," *Nature*, vol. 393, no. 6684, pp. 440–442, 1998.

[3] J. Cohen, "Trusses: Cohesive subgraphs for social network analysis," *National Security Agency Technical Report*, vol. 16, no. 3.1, 2008.

[4] P. Burkhardt, "Triangle centrality," *CoRR*, vol. abs/2105.00110, 2021. [Online]. Available: https://arxiv.org/abs/2105.00110

[5] Graph 500 Steering Committee, "The Graph500 benchmark," 2010. [Online]. Available: https://www.graph500.org

[6] S. Samsi, V. Gadepally, M. Hurley, M. Jones, E. Kao, S. Mohindra, P. Monticciolo, A. Reuther, S. Smith, W. Song, D. Staheli, and J. Kepner, "GraphChallenge.org: Raising the bar on graph analytic performance," in *2018 IEEE High Performance Extreme Computing Conference (HPEC)*, 2018, pp. 1–7.

[7] W. Harrod, "Advanced graphical intelligence logical computing environment (AGILE)," https://www.iarpa.gov/images/PropsersDayPDFs/AGILE/AGILE_Proposers_Day_sm.pdf, 2020, accessed: 2022–05-24.

[8] A. Itai and M. Rodeh, "Finding a minimum circuit in a graph," in *Proceedings of the Ninth Annual ACM Symposium on Theory of Computing*, ser. STOC '77. New York, NY, USA: Association for Computing Machinery, 1977, p. 1–10.

[9] M. Latapy, "Main-memory triangle computations for very large (sparse (power-law)) graphs," *Theoretical Computer Science*, vol. 407, no. 1, pp. 458–473, 2008.

[10] S. Pandey, Z. Wang, S. Zhong, C. Tian, B. Zheng, X. Li, L. Li, A. Hoisie, C. Ding, D. Li, and H. Liu, "Trust: Triangle counting reloaded on GPUs," *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 11, pp. 2646–2660, 2021.

[11] J. Cohen, "Graph twiddling in a MapReduce world," *Computing in Science & Engineering*, vol. 11, no. 4, pp. 29–41, 2009.

[12] R. Pearce, "Triangle counting for scale-free graphs at scale in distributed memory," in *IEEE High Performance Extreme Computing Conference (HPEC)*, 2017, pp. 1–4.

[13] S. Ghosh and M. Halappanavar, "TriC: Distributed-memory triangle counting by exploiting the graph structure," in *IEEE High Performance Extreme Computing Conference (HPEC)*, 2020, pp. 1–6.

[14] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms, Fourth Edition*. MIT Press, 2022.

[15] D. Dolev, C. Lenzen, and S. Peled, "'Tri, tri again': finding triangles and small subgraphs in a distributed setting," in *International Symposium on Distributed Computing*. Springer, 2012, pp. 195–209.

[16] R. Pearce and G. Sanders, "K-truss decomposition for scale-free graphs at scale in distributed memory," in *2018 IEEE high performance extreme computing conference (HPEC)*. IEEE, 2018, pp. 1–6.

[17] P. Sanders and T. N. Uhl, "Engineering a distributed-memory triangle counting algorithm," *arXiv preprint arXiv:2302.11443*, 2023.

[18] D. Chakrabarti, Y. Zhan, and C. Faloutsos, "R-MAT: A recursive model for graph mining," in *Proceedings of the 2004 SIAM International Conference on Data Mining*. SIAM, 2004, pp. 442–446.

[19] L. Hoang, V. Jatala, X. Chen, U. Agarwal, R. Dathathri, G. Gill, and K. Pingali, "DistTC: High performance distributed triangle counting," in *IEEE High Performance Extreme Computing Conference (HPEC)*, 2019, pp. 1–7.

[20] I. Giechaskiel, G. Panagopoulos, and E. Yoneki, "PDTL: Parallel and distributed triangle listing for massive graphs," in *2015 44th International Conference on Parallel Processing*. IEEE, 2015, pp. 370–379.

[21] P. Burkhardt, "Graphing trillions of triangles," *Information Visualization*, vol. 16, no. 3, pp. 157–166, 2017.

[22] C. Seshadhri, A. Pinar, and T. G. Kolda, "A hitchhiker's guide to choosing parameters of stochastic kronecker graphs," *CoRR, abs/1102.5046*, vol. 1, 2011.

[23] S. Beamer, K. Asanovic, and D. Patterson, "Searching for a parent instead of fighting over children: A fast breadth-first search implementation for Graph500," *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2011-117*, 2011.

[24] T. Schank and D. Wagner, "Finding, counting and listing all triangles in large graphs, an experimental study," in *International workshop on experimental and efficient algorithms*. Springer, 2005, pp. 606–609.

[25] O. Green, P. Yalamanchili, and L.-M. Munguía, "Fast triangle counting on the GPU," in *Proceedings of the 4th Workshop on Irregular Applications: Architectures and Algorithms*, 2014, pp. 1–8.

[26] J. Shun and K. Tangwongsan, "Multicore triangle computations without tuning," in *2015 IEEE 31st International Conference on Data Engineering*. IEEE, 2015, pp. 149–160.

[27] Y. Hu, H. Liu, and H. H. Huang, "TriCore: Parallel triangle counting on GPUs," in *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2018, pp. 171–182.

[28] A. Azad, A. Buluç, and J. Gilbert, "Parallel triangle counting and enumeration using matrix algebra," in *2015 IEEE International Parallel and Distributed Processing Symposium Workshop*. IEEE, 2015, pp. 804–811.

[29] S. Acer, A. Yaşar, S. Rajamanickam, M. Wolf, and Ü. V. Catalyürek, "Scalable triangle counting on distributed-memory systems," in *IEEE High Performance Extreme Computing Conference (HPEC)*, 2019, pp. 1–5.

[30] L. Wang and J. D. Owens, "Fast BFS-based triangle counting on GPUs," in *IEEE High Performance Extreme Computing Conference (HPEC)*, 2019, pp. 1–6.

[31] N. Alon, R. Yuster, and U. Zwick, "Finding and counting given length cycles," *Algorithmica*, vol. 17, no. 3, pp. 209–223, 1997.

[32] 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*, 2011, pp. 607–614.

[33] L. Hu, L. Zou, and Y. Liu, "Accelerating triangle counting on GPU," in *Proceedings of the 2021 International Conference on Management of Data*, 2021, pp. 736–748.

[34] S. Ghosh, "Improved distributed-memory triangle counting by exploiting the graph structure," in *IEEE High Performance Extreme Computing Conference (HPEC)*, 2022, pp. 1–6.

[35] A. Strausz, F. Vella, S. Di Girolamo, M. Besta, and T. Hoefler, "Asynchronous distributed-memory triangle counting and LCC with RMA caching," in *36th IEEE International Parallel and Distributed Processing Symposium, IPDPS*. IEEE, 2022, pp. 291–301.

[36] L. Zeng, K. Yang, H. Cai, J. Zhou, R. Zhao, and X. Chen, "HTC: Hybrid vertex-parallel and edge-parallel triangle counting," in *IEEE High Performance Extreme Computing Conference (HPEC)*, 2022, pp. 1–7.

[37] G. Pandurangan, P. Robinson, and M. Scquizzato, "On the distributed complexity of large-scale graph computations," *ACM Transactions on Parallel Computing (TOPC)*, vol. 8, no. 2, pp. 1–28, 2021.

[38] T. N. Uhl, "Communication efficient triangle counting," Ph.D. dissertation, Karlsruhe Institute of Technology, 2021.