A Novel Parallel Triangle Counting Algorithm with Reduced Communication

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,ooa9,zhihui.du}@njit.edu

Abstract—Counting and finding triangles in graphs is often used in real-world analytics for characterizing the cohesiveness and identifying communities in graphs. In this paper, we present novel sequential and parallel triangle counting algorithms based on identifying horizontal-edges in a breadth-first search (BFS) traversal of the graph. The BFS allows our algorithm to drastically reduce the number of edges examined for set intersections. Our new approach is that rather than send all open wedges in the graph edge is sent repeatedly, the repetition factor being the degree of one of its endpoints. One way to view our new algorithm uses breadth-first search to identify horizontal-edge to determine triangles.

Triangle counting algorithms are often based on techniques such as list intersection, matrix multiplication and subgraph matching. The three main list intersection based triangle counting algorithms are summarized as: the node iterator, the edge iterator and the forward algorithm (detailed in Section III).

Cohen [9] designed a novel map-reduce parallelization of triangle counting that generates open wedges between triples of vertices in the graph, and determines if a closing edge exists that completes a triangle. Most parallel approaches for triangle counting [10], [11] partition the sparse graph data structure across the compute nodes, and follow this strategy of generating open wedges that are sent to other compute nodes to find whether or not a closing edge exists. As such, the running time of parallel triangle counting is often dominated by the communication time for these open wedges.

In fact, most parallel triangle counting algorithms are based on generating open wedges (also known as 2-paths) and checking whether or not a closing edge exists. This approach results in an overabundance of communication because each graph edge is sent repeatedly, the repetition factor being the degree of one of its endpoints. One way to view our new approach is that rather than send all open wedges in the graph, we reverse this and send potential closing edges once, resulting in asymptotically less communication. In this paper, we present novel sequential and parallel triangle counting algorithms. Our contributions include:

- A new sequential triangle counting and finding algorithm that runs in \(O(m^2)\) time and \(O(n^2)\) space, which is cost-prohibitive for practical use on large graphs.

- Triangle counting algorithms are often based on techniques such as list intersection, matrix multiplication and subgraph matching. The three main list intersection based triangle counting algorithms are summarized as: the node iterator, the edge iterator and the forward algorithm (detailed in Section III).

- Cohen [9] designed a novel map-reduce parallelization of triangle counting that generates open wedges between triples of vertices in the graph, and determines if a closing edge exists that completes a triangle. Most parallel approaches for triangle counting [10], [11] partition the sparse graph data structure across the compute nodes, and follow this strategy of generating open wedges that are sent to other compute nodes to find whether or not a closing edge exists. As such, the running time of parallel triangle counting is often dominated by the communication time for these open wedges.

In fact, most parallel triangle counting algorithms are based on generating open wedges (also known as 2-paths) and checking whether or not a closing edge exists. This approach results in an overabundance of communication because each graph edge is sent repeatedly, the repetition factor being the degree of one of its endpoints. One way to view our new approach is that rather than send all open wedges in the graph, we reverse this and send potential closing edges once, resulting in asymptotically less communication. In this paper, we present novel sequential and parallel triangle counting algorithms. Our contributions include:

- A new sequential triangle counting and finding algorithm that runs in \(O(m \cdot d_{\text{max}})\) time and \(O(n + m)\) space, where \(d_{\text{max}}\) is the maximal degree of a vertex \(v \in V\). The algorithm uses breadth-first search to identify horizontal-edges, and uses the intersection of the endpoints of each horizontal-edge to determine triangles.

- A novel communication-optimal parallel algorithm for triangle counting and finding that asymptotically reduces the communication from all prior parallel approaches. The new approach has a total communication volume of \(O(m)\) and is the first parallel algorithm to achieve this.
Section II gives additional notation. Section III discusses related work. Section IV presents our new approach for triangle counting. In Section V, we present our new parallel triangle counting. Lastly, in Section VI we conclude the paper.

II. NOTATION

Let \( G = (V, E) \) be an undirected graph with \( n = |V| \) vertices and \( m = |E| \) edges. We will use \( N(v) = \{ u \in V : (v, u) \in E \} \) to denote the neighborhood of vertex \( v \in V \). The degree of vertex \( v \in V \) is \( d(v) = |N(v)| \), and \( d_{\text{max}} \) is the maximal degree in graph \( G \). Given a source vertex \( s \), the level \( L(v) \) of a vertex \( v \) is the distance from \( s \) to \( v \). After breadth-first search (BFS) of an undirected graph, edges can be classified as 1) tree-edges: in the BFS-tree, 2) strut-edges: non-tree edges whose endpoints are on two adjacent levels, and 3) horizontal-edges: non-tree edges whose endpoints are on the same level. See Fig. 1 for an example.

III. 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 [6] and Schank and Wagner [12] provide surveys of faster sequential algorithms. Triangle counting algorithms generally fall into three approaches: list intersection, matrix multiplication and subgraph matching.

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 \( O(m \cdot d_{\text{max}}) \) and the forward algorithm is \( O(m^{3/2}) \), which is significantly better performance [6].

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., [13], [14]) 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., [15], [16]) 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., [8], [14]) construct a hash table for one list and use the other list as search keys to find the common elements in the hash table. We use a hash table 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 [17] relies on a linear algebra formulation for triangle counting. For the graph’s adjacency matrix \( A \), the approach performs \( B = A \times A \), which counts the number of wedges, then the element-wise multiplication \( A \odot B \) determines whether the wedge is closed. The method finds the triangle count after scaling. This approach can be optimized [18] 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 triangle.

A matching-based approach for triangle counting searches for all occurrences of a query graph, which is a triangle, in the input graph. Wang and Owens [19] 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.

B. Parallel Algorithms

Map-reduce is a standard platform for large scale distributed computation. Cohen [9] 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. [20] implemented triangle counting using map-reduce that ranks vertices by degree and distributes

---

Fig. 1. Example BFS tree of Graph \( G \). The tree-edges are black, strut-edges are blue, and horizontal-edges are red.
them across hosts. Pearce [10] 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 HavocGT. DistTC [15] 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 [11] 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. TriC [11] exploits the vertex-based distributed triangle counting and sends vertices rather than the 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 [21] 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. Strauss et al. [22] 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. [23] proposed a triangle counting algorithm that adaptively selects vertex-parallel and edge-parallel paradigm.

IV. NEW APPROACH

We observe the following about a triangle \{v_a, v_b, v_c\} in an undirected graph \(G = (V, E)\). (We assume \(G\) is connected. If not, it is trivial to extend this approach to each component.)

Given an arbitrary vertex \(s \in V\), the shortest path distance from \(s\) to \(v_a\), \(d(s, v_a)\), and the distances to the other two triangle vertices, \(d(s, v_b)\) and \(d(s, v_c)\), differ by at most 1.

(A simple proof by contradiction is left to the reader.) This motivates our new sequential approach given in Alg. 1.

Algorithm 1 Triangle Counting

**Input:** Graph \(G = (V, E)\)

**Output:** Triangle Count \(T\)

1. Select an arbitrary vertex \(s \in V\)
2. Compute the breadth-first search of \(G\) from \(s\) and label each edge \(e \in E\) as a tree-, strut- or horizontal-edge
3. for all \(v \in V\) do
4. \(L(v) \leftarrow d(s, v)\)
5. end for
6. for all horizontal-edges \((u, w)\) do
7. \(I \leftarrow N(u) \cap N(w)\)
8. for all \(v \in I\) do
9. if \(L(v) \neq L(u)\) then \(c_1 \leftarrow c_1 + 1\)
10. else \(c_2 \leftarrow c_2 + 1\)
11. end if
12. end for
13. end for
14. \(T \leftarrow c_1 + c_2/3\)

Lemma 1. Each triangle \(\{u, v, w\}\) must contain at least one horizontal-edge.

**Proof.** (By contradiction.) A triangle is a path of length 3 that has the same beginning and ending vertex. If no horizontal-edges are in the triangle, then each edge in the path (i.e., a tree- or strut-edge) increases or decreases the level by one. Since the path must end on the same level as the starting vertex, there must be the same number of edges in the path decreasing the level as there are edges in the path increasing the level. This requires an even path length. However, this is a contradiction since a triangle has an odd path length of 3.

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

**Proof.** It follows from the proof of Lemma 1 that the path must include an even number of tree- and strut-edges; hence, 0 or 2 tree- or strut-edges in each triangle. In the first case with 0 tree- or strut-edges, all three triangle edges must be horizontal-edges. In the second case with 2 tree- or strut-edges, then the triangle contains exactly one horizontal-edge.

Theorem 1 (Novel Counting Method). The triangle count of a graph \(G\) is the sum of the number of triangle apex vertices on a different level from the horizontal-ends endpoints and one-third of the number of that are on the same level.

**Proof.** Lemma 2 proves that for a triangle either 1) the two endpoint vertices of the horizontal-edge are on the same level and the apex vertex is on a different level, or 2) all three triangle vertices are in the same level. For the triangle \(\{v_a, v_b, v_c\}\), assume (w.l.o.g.) that \((v_a, v_b)\) is a horizontal-edge (so \(L(v_a) \equiv L(v_b)\)) and that \(v_c\) is the apex vertex. If \(L(v_c) \neq L(v_a)\), then we have the first case; otherwise \(L(v_c) \equiv L(v_a) \equiv L(v_b)\) and we have the second case. Each unique triangle is defined by a horizontal-edge and an apex vertex from the list intersection of the horizontal-edge’s endpoint vertices. A triangle is triply-counted when the second case occurs: once for each of the triangle’s three horizontal-edges. Hence, the total number of triangles is the sum of the number of the apex vertices in the first case and one-third the number of apex vertices in the second case.

A. Time Complexity

Computing breadth-first search, the level of each vertex, and marking horizontal-edges, takes \(O(n + m)\) time. There are at most \(O(m)\) horizontal-edges, and the time complexity for finding the list intersection for each is \(O(m \cdot d_{max})\). Thus, the time complexity is \(O(m \cdot d_{max})\).
V. PARALLEL ALGORITHM

In this section, we present our communication-optimal parallel algorithm for counting triangles in massive graph on a p-processor distributed-memory parallel computer. Similar with prior approaches, the input graph $G$ is stored in 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.

Our parallel algorithm (see Alg. 2) is based on the horizontal-edge approach of our sequential algorithm. This is the first triangle counting algorithm to our knowledge that uses the breadth-first search followed by a transpose of the sparse edge arrays to significantly reduce the communication for set intersections.

We illustrate the parallel triangle counting algorithm using a real dataset of a very unusual social community of bottlenose dolphins living in a fjord, a geographically-isolated environment at the southern most extreme of the species’ range. A research team systematically surveyed the social interactions of this animal community for seven years, from November 1994 to November 2001 in Doubtful Sound, Fiordland, New Zealand [24]. In Figure 2 we show 62 individual dolphins from this survey and their interactions in an undirected social network graph.

The algorithm first runs parallel breadth-first search (BFS) (line 3) on the graph $G$ and assigns a level $L(v)$ to each $v \in V$. If the diameter of the graph is $D$, then $\lceil \log D \rceil$ bits are needed per vertex to store its level. During the BFS, for each edge in the graph we assign a bit as to whether the edge is a horizontal-edge or not. Figure 3 gives an example of the breadth-first search and identifying horizontal-edges in the dolphin social network graph. The horizontal-edges from this example are given in Figure 4.

Earlier we defined $\hat{N}(v) = \{u \in V \mid \langle v, u \rangle \in E\}$ to denote the neighborhood of vertex $v \in V$. Let the modified neighborhood of $v$ be $\tilde{N}(v) = \hat{N}(v) - \{w \in V \mid \langle v, w \rangle \leftarrow \text{horizontal-edge}, v < w\}$. That is, we remove all vertices adjacent to $v$ along a horizontal-edge where $v$’s label is less than $w$’s. The modified neighborhoods break symmetry and will be used to prevent triple-counting of triangles comprised of three horizontal-edges in the graph. In line 4 of Alg. 2 the modified neighborhoods are created. In Figure 5 we list the modified neighborhoods of the dolphin social network graph and their assignments to four processors.

The approach performs a sparse matrix transpose of the modified neighborhoods (lines 24 to 28). This transpose allows intersections of the respective sublists to occur locally without further communication. The transpose must partition the $n$ vertices to the $p$ processors. Simply assigning blocks of $n/p$ vertices to each processor; that is, $i \cdot n/p$ to $(i+1)n/p - 1$ to processor $p_i$, for $0 \leq i \leq p - 1$, potentially results in a load imbalance. Instead, our approach uses techniques first developed for parallel sorting (e.g., [25]–[27]) to ensure that each processor receives at most twice the average ($2 \cdot (2 - k)m/p$) number of elements. By oversampling, we

### Algorithm 2 Parallel Triangle Counting

** Input: ** Graph $G = (V,E)$

** Output: ** Triangle Count $T$

1. Select an arbitrary vertex $s \in V$
2. Compute the breadth-first search in parallel of $G$ from $s$ and set $X(\langle v, w \rangle)$ if $\langle v, w \rangle$ is a horizontal-edge
3. for all $v \in V$ in parallel do
   4. $\tilde{N}(v) \leftarrow N(v) - \{w \in V \mid X(\langle v, w \rangle), v < w\}$
5. end for
6. for all $i \in 0 \ldots p - 1$ in parallel do
   7. $z_i = \Sigma |\tilde{N}(v)|$
8. end for
9. for all $i \in 0 \ldots p - 1$ in parallel do
   10. On $p_i$ perform a multi-way merge of $\tilde{N}(v)$ and select elements at positions $[j \cdot z_i/(p + 1)]$ for $1 \leq j \leq p$ as samples.
11. end for
12. for all $i \in 0 \ldots p - 1$ in parallel do
   13. $w_i$ sends its $p$ samples to $p_0$
14. end for
15. $p_0$ creates a multi-way merge of the $p$ sorted lists, and selects elements at positions $j \cdot p$ for $1 \leq j \leq (p - 1)$ as the $p - 1$ splitters.
16. $p_0$ broadcasts the $p - 1$ splitters
17. $p_i$ sends its $p$ samples to $p_0$
18. end for
19. for all $i \in 0 \ldots p - 1$ in parallel do
   20. $p_i$ sends $p_i \oplus j$ all $\tilde{N}(v)$ sublists with values $> Splitter[i \oplus j]$ and $\leq Splitter[i \oplus j] + 1$
21. and receives its sublists from $p_i \oplus j$
22. end for
23. for $j = 0$ to $p - 1$ do
   24. $p_i$ sends $p_i \oplus j$ all $\tilde{N}(v)$ sublists with values $> Splitter[i \oplus j]$ and $\leq Splitter[i \oplus j] + 1$
25. and receives its sublists from $p_i \oplus j$
26. end for
27. end for
28. end for
29. for all $i \in 0 \ldots p - 1$ in parallel do
   30. $t_i \leftarrow 0$
31. end for
32. for all $i \in 0 \ldots p - 1$ in parallel do
   33. for each horizontal-edge $(v, w)$ with $v < w$ on $p_i$ do
   34. $t_i = t_i + |\tilde{N}(v) \cap \tilde{N}(w)|$
35. end for
36. for $j = 1$ to $p - 1$ do
   37. Processors $p_i$ and $p_i \oplus j$ swap
   38. horizontal-edges $(v, w)$ where $v < w$
   39. for each edge $(v, w)$ on $p_i$ do
   40. $t_i = t_i + |\tilde{N}(v) \cap \tilde{N}(w)|$
   41. end for
42. end for
43. end for
44. $T \leftarrow \text{Reduce}(t_i)$
Fig. 2. Interactions of 62 bottlenose dolphins represented by an undirected graph.

Fig. 3. Breadth-first search of the dolphin social network graph. The vertices are organized by level with tree-edges in black, strut-edges in blue, and horizontal-edges in red.

Fig. 4. The horizontal-edges in the breadth-first search of the dolphin social network graph, organized by level.

Fig. 5. The horizontal-edges in the breadth-first search of the dolphin social network graph, organized by level.

Fig. 6. The horizontal-edges in the breadth-first search of the dolphin social network graph, organized by level.

Fig. 7. The horizontal-edges in the breadth-first search of the dolphin social network graph, organized by level.

Fig. 8. The horizontal-edges in the breadth-first search of the dolphin social network graph, organized by level.

Fig. 9. The horizontal-edges in the breadth-first search of the dolphin social network graph, organized by level.

Fig. 10. The horizontal-edges in the breadth-first search of the dolphin social network graph, organized by level.

Fig. 11. The horizontal-edges in the breadth-first search of the dolphin social network graph, organized by level.

Fig. 12. The horizontal-edges in the breadth-first search of the dolphin social network graph, organized by level.

Fig. 13. The horizontal-edges in the breadth-first search of the dolphin social network graph, organized by level.

Fig. 14. The horizontal-edges in the breadth-first search of the dolphin social network graph, organized by level.

Fig. 15. The horizontal-edges in the breadth-first search of the dolphin social network graph, organized by level.

Fig. 16. The horizontal-edges in the breadth-first search of the dolphin social network graph, organized by level.

Fig. 17. The horizontal-edges in the breadth-first search of the dolphin social network graph, organized by level.

Fig. 18. The horizontal-edges in the breadth-first search of the dolphin social network graph, organized by level.

Fig. 19. The horizontal-edges in the breadth-first search of the dolphin social network graph, organized by level.

Fig. 20. The horizontal-edges in the breadth-first search of the dolphin social network graph, organized by level.

Fig. 21. The horizontal-edges in the breadth-first search of the dolphin social network graph, organized by level.

Fig. 22. The horizontal-edges in the breadth-first search of the dolphin social network graph, organized by level.

Fig. 23. The horizontal-edges in the breadth-first search of the dolphin social network graph, organized by level.

Fig. 24. The horizontal-edges in the breadth-first search of the dolphin social network graph, organized by level.

Fig. 25. The horizontal-edges in the breadth-first search of the dolphin social network graph, organized by level.

Fig. 26. The horizontal-edges in the breadth-first search of the dolphin social network graph, organized by level.

Fig. 27. The horizontal-edges in the breadth-first search of the dolphin social network graph, organized by level.

Fig. 28. The horizontal-edges in the breadth-first search of the dolphin social network graph, organized by level.

Fig. 29. The horizontal-edges in the breadth-first search of the dolphin social network graph, organized by level.

Fig. 30. The horizontal-edges in the breadth-first search of the dolphin social network graph, organized by level.

Fig. 31. The horizontal-edges in the breadth-first search of the dolphin social network graph, organized by level.

Fig. 32. The horizontal-edges in the breadth-first search of the dolphin social network graph, organized by level.

Fig. 33. The horizontal-edges in the breadth-first search of the dolphin social network graph, organized by level.

Fig. 34. The horizontal-edges in the breadth-first search of the dolphin social network graph, organized by level.

Fig. 35. The horizontal-edges in the breadth-first search of the dolphin social network graph, organized by level.

Fig. 36. The horizontal-edges in the breadth-first search of the dolphin social network graph, organized by level.

Fig. 37. The horizontal-edges in the breadth-first search of the dolphin social network graph, organized by level.

Fig. 38. The horizontal-edges in the breadth-first search of the dolphin social network graph, organized by level.

Fig. 39. The horizontal-edges in the breadth-first search of the dolphin social network graph, organized by level.

Fig. 40. The horizontal-edges in the breadth-first search of the dolphin social network graph, organized by level.

Fig. 41. The horizontal-edges in the breadth-first search of the dolphin social network graph, organized by level.

Fig. 42. The horizontal-edges in the breadth-first search of the dolphin social network graph, organized by level.

Fig. 43. The horizontal-edges in the breadth-first search of the dolphin social network graph, organized by level.
system is performed and the total number of triangles is $T = \sum_{i=0}^{p-1} t_i$. 

Fig. 5. The modified neighborhoods of the 62 vertices of the dolphin social network (left), and the processor assignments for these modified neighborhoods (right).
Fig. 6. Selection of the 16 samples from the modified neighbors of the dolphin graph.

Fig. 7. Finding the 3 pivots for $p = 4$ processors from the samples.

Fig. 8. The partitioning of the dolphin graph’s modified neighborhoods on each processor using the three pivots.
Fig. 9. The transposed modified neighborhood sets of the dolphin graph on $p = 4$ processors.
A. 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 $O([log \ D])$ bits per vertex to store its level. This is a total of at most $m + n[log \ D]$ bits across the $p$ processors, which is negligible in practice. Preserving the graph requires additional $O(n + m)$ space for the graph, otherwise the space can be reclaimed during the transpose of the modified neighborhoods. Note that the horizontal-edges that are swapped between each of the processors are used in local computation and do not need to be stored.

2) Communication: In our analysis for communication cost for BFS and transpose, we measure the total communication volume independent of the number of processors. Thus, this is a conservative over-estimate of communication since a fraction (e.g., $1/p$) of accesses will be on the same compute node versus message traffic between nodes.

The cost of breadth-first search is $m$ edge traversals with $[log \ D] + 3[log \ n]$ bits communicated per edge traversal for the level information, pair of vertex ids, and vertex degree, yielding $m \cdot ([log \ D] + 3[log \ n])$ bits for the BFS.

Determining the $p-1$ splitters requires $p^2$ samples to be sent and for $p_0$ to broadcast the $p-1$ splitters. The communication cost is $(p^2 + (p-1)p)[log \ n] = (2p^2 - p)[log \ n]$ bits.

Due to the symmetry-breaking of the horizontal-edges, $km$ vertices, one for each horizontal-edge, are removed from the neighborhoods $N(*)$ to form the modified neighborhoods $N(\ast)$. Hence, the total size of the modified neighborhoods is $(2 - k)m$ vertices. Thus, the transpose of the modified neighborhoods requires the communication of $(2 - k)m[log \ n]$ bits.

Swapping the $km$ horizontal-edges requires $kmp[log \ n]$ bits, where $p$ is the number of processors.

The final reduction to find the total number of triangles requires $(p - 1)[log \ n]$ bits.

Hence, the total communication volume is $m \cdot ([log \ D] + 3[log \ n]) + (2p^2-p)[log \ n] + (2-k)m[log \ n] + kmp[log \ n] + (p - 1)[log \ n] = m \cdot ([log \ D] + (kp - k + 5)[log \ n]) + (2p^2 - 1)[log \ n]$ bits. Hence, on a given parallel machine, the communication is $O(m)$ words, which is communication-optimal.

B. 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. The summary is given in Table 1. For the real graphs, we find the actual value of $k$, the percentage of graph edges that are horizontal-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[28] with parameters $a = 0.57$, $b = 0.19$, $c = 0.19$, and $d = 0.05$, for scale 36 and 42 with $n = 2^{scale}$ 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 these graphs,

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 a 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 $G$.

C. Graph500 RMAT Graphs

Pearce[10] shows that for large Graph500 graphs, the total running time closely tracks the wedge checking time. Their implementation for scale 36 takes 3960s on 1.5M CPUs of IBM BG/Q to count triangles. The result finds $2.7 \times 10^{13}$ triangles and checks $1.05 \times 10^{15}$ wedges. Since each wedge requires 72 bits, the total data volume of checks is $8.32PB$.

We estimate the number of triangles and wedges for the scale 42 graph by extrapolating from counts up to scale 34 in[10] and set triangles to $2^{22-36}$ #Triangles[36] and wedges to $2^{42-36+2}$ #Wedges[36], resulting in the scale 42 estimate of $8.64 \times 10^{14}$ triangles and $1.08 \times 10^{18}$ wedges. With 84 bits/wedge, the total volume of wedge checks is 9.84EB.

Beamer et al.[29] 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. In our experimental analyses of these RMAT graphs for scales from 10 to 24, we found the fraction $k$ of horizontal-edges to be approximately 0.65.

In our new approach for scale 36, where the communication cost is $m \cdot ([log \ D] + (kp - k + 5)[log \ n]) + (2p^2 - 1)[log \ n]$ bits. With $[log \ D] = 4$, and assuming $p = 128$ processors, we have a total communication volume of 394TB, for a communication reduction of 21.8×.

For scale 42, and assuming $p = 256$ processors, we estimate the communication of our new triangle counting algorithm as 56.1PB, for a communication reduction of 180×.

VI. CONCLUSIONS

In this paper, we present novel sequential and parallel algorithms for counting and finding triangles in graphs. The parallel algorithm is the first communication-optimal triangle counting algorithm and is an asymptotic improvement upon all prior approaches and significantly reduces the communication volume on massive graphs of practical interest. Our approach uses the breadth-first search to drastically reduce the number of edges examined and a sparse transpose in the parallel algorithm that minimizes the communication required for set intersections. The parallel algorithm achieves an order of magnitude or more of speedup for large graphs as communication is the main bottleneck for triangle counting on distributed memory systems.

1Throughout this paper, a petabyte (PB) is $2^{50}$ bytes and an exabyte (EB) is $2^{60}$ bytes.
| Graph   | n    | m    | # Triangles | # Wedges | k   | p    | Previous | This paper | Speedup |
|---------|------|------|-------------|----------|-----|------|----------|------------|---------|
| ca-GrQc | 5242 | 14484| 48260       | 165798   | 0.522| 4    | 514KB    | 156KB      | 3.37    |
| ca-HepTh| 9877 | 25973| 28339       | 217389   | 0.423| 4    | 926KB    | 288KB      | 3.29    |
| as-caida20071105 | 26475 | 53381 | 36565 | 776895 | 0.225 | 4 | 2.78MB | 374KB | 4.96 |
| Facebook_combined | 4039 | 88234 | 1612010 | 17051688 | 0.914 | 4 | 48.3MB | 1.01MB | 48.4 |
| ca-CoraMat | 23133 | 93439 | 175361 | 1367573 | 0.311 | 4 | 5.61MB | 1.13MB | 4.98 |
| ca-HepPh | 12008 | 118491 | 3356499 | 5081984 | 0.621 | 4 | 17.0MB | 1.40MB | 12.1 |
| email-Enron | 36692 | 183831 | 727044 | 5933045 | 0.478 | 4 | 22.6MB | 2.32MB | 9.75 |
| ca-AstroPh | 18772 | 19050 | 1351441 | 8451765 | 0.667 | 4 | 30.2MB | 2.55MB | 11.9 |
| loc-brightkite_edges | 58228 | 214078 | 494728 | 6956250 | 0.441 | 4 | 26.5MB | 2.66MB | 9.98 |
| soc-EpinionsI | 75879 | 405740 | 1624481 | 2137935 | 0.498 | 4 | 86.7MB | 5.49MB | 15.8 |
| amazon0601 | 403394 | 2443408 | 1733617 | 2443408 | 0.529 | 4 | 436MB | 49.0MB | 8.90 |
| com-Youtube | 1134890 | 2987624 | 3056386 | 20981158 | 0.347 | 4 | 103GB | 66.5MB | 18.6 |
| RMAT-36 | 68719476736 | 1.09951E+12 | 2.7E+13 | 1.05E+15 | 0.65 | 4 | 128 | 394TB | 21.8 |
| RMAT-42 | 4.93905E+12 | 7.03658E+13 | 8.6E+14 | 1.0E+18 | 0.65 | 4 | 256 | 59.4MB | 160.0 |

Communication costs for real and synthetic graphs. 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 that use wedge-checking based algorithms and THIS PAPER represents the communication cost of our new approach. Speedup represents the communication reduction between these two, and thus, the expected speedup of the parallel algorithm.

REFERENCES

[1] D. J. Watts and S. H. Strogatz, “Collective dynamics of ‘small-world’ networks,” Nature, vol. 393, no. 6684, pp. 440–442, 1998.
[2] J. Cohen, “Trusses: Cohesive subgraphs for social network analysis,” National Security Agency Technical Report, vol. 16, no. 3, 2011.
[3] P. Burkhardt, “Triangular centrality,” CoRR, abs/2105.00110, 2021. [Online]. Available: https://arxiv.org/abs/2105.00110
[4] Graph 500 Steering Committee, “The Graph500 benchmark,” 2010. [Online]. Available: https://www.graph500.org
[5] W. Harrod, “Advanced graphical intelligence logical computing environment (AGILE),” https://www.iarpa.gov/images/PropsersDayPDFs/AGILEAGILE_ProposersDay_sm.pdf, 2020, accessed: 2022-05-24.
[6] J. Cohen, “Graph twiddling in a MapReduce world,” Computing in Science & Engineering, vol. 11, no. 4, pp. 29–41, 2009.
[7] R. Pearce, “Triangle counting for scale-free graphs at scale in distributed memory,” in IEEE High Performance Extreme Computing Conference (HPEC), 2021.
[8] A. Itai and M. Rodeh, “Finding a minimum circuit in a graph,” SIAM Journal on Computing, vol. 7, no. 4, pp. 413–423, 1978.
[9] S. Pandey, Z. Wang, S. Zhong, C. Tian, B. Zheng, X. Li, L. Li, A. Hoisie, C. Ding, D. Li et al., “Trust: Triangle counting reloaded on GPUs,” IEEE Transactions on Parallel and Distributed Systems, vol. 32, no. 11, pp. 2646–2660, 2021.
[10] S. Ghosh, “Improved distributed-memory triangle counting by exploiting graph structure,” in IEEE High Performance Extreme Computing Conference (HPEC), 2022, pp. 1–6.
[11] D. Lusseau, K. Schneider, O. J. Boisseau, P. Haase, E. Slooten, and J. M. Dawson, “The bottlenose dolphin community of Doubtful Sound features a large proportion of long-lasting associations,” Behavioral Ecology and Sociobiology, vol. 54, no. 4, pp. 396–405, 2003. [Online]. Available: https://doi.org/10.1006/jbem.2003.1051
[12] J. J. Keller, S. Rajamanickam, M. Wolf, and U. V. Catalyurek, “Scalable triangle counting on distributed-memory systems,” in IEEE High Performance Extreme Computing Conference (HPEC), 2019, pp. 1–5.
[13] 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. 1–6.
[14] A. Ashrafi, S. D. Girolamo, M. Besta, and T. Hoeffer, “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.
[15] 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. 1–6.
[16] J. Cohen, “Graph twiddling in a MapReduce world,” Computing in Science & Engineering, vol. 11, no. 4, pp. 29–41, 2009.
[17] 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.
[18] 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.
[19] 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.
[20] O. Green, P. Yalamanchili, and L.-M. Munguia, “Fast triangle counting on the GPU,” in Proceedings of the 4th Workshop on Irregular Applications: Architectures and Algorithms, 2014, pp. 1–8.
[21] J. Shi and K. Tangwongsan, “Multicore triangle computations without tuning,” in 2015 IEEE 31st International Conference on Data Engineering. IEEE, 2015, pp. 149–160.
[22] Y. Hu, H. Liu, and H. H. Huang, “TriCore: Parallel triangle counting on GPU,” in SC15: International Conference for High Performance Computing, Networking, Storage and Analysis. IEEE, 2018, pp. 171–182.
[23] L. Hoang, V. Jataja, X. Chen, U. Agarwal, R. Dathathri, G. Gill, and K. Pingali, “DistHC: High performance distributed triangle counting,” in IEEE High Performance Extreme Computing Conference (HPEC), 2019, pp. 1–7.
[24] 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.