IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received January 8, 2019, accepted January 15, 2019, date of publication February 22, 2019, date of current version March 12, 2019.

Digital Object Identifier 10.1109/ACCESS.2019.2901025

LiteTE: Lightweight, Communication-Efficient Distributed-Memory Triangle Enumerating

YONGXUAN ZHANG[1], HONG JIANG[2], (Fellow, IEEE), FANG WANG[1], YU HUA[1], (Member, IEEE), DAN FENG[1], (Member, IEEE), AND XIANGHAO XU[1]

[1]Wuhan National Laboratory for Optoelectronics, Key Laboratory of Information Storage System, Ministry of Education of China, School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan 430074, China
[2]Department of Computer Science and Engineering, The University of Texas at Arlington, Arlington, TX 76019, USA

Corresponding author: Fang Wang (wangfang@hust.edu.cn)

This work was supported in part by the National High Technology Research and Development Program (863 Program) of China under Grant 2013AA013203, Grant 2015AA016701, and Grant 2015AA015301, in part by the National Basic Research 973 Program of China under Grant 2011CB302301, in part by the Key Laboratory of Information Storage System, Ministry of Education, China, in part by the NSFC under Grant 61502190, and in part by the CERNET Innovation Project under Grant NGII20170120.

ABSTRACT Distributed-memory triangle enumerating has attracted considerable interests due to its potential capability to process huge graphs quickly. However, existing algorithms suffer from low speed due to high communication cost and load imbalance. To solve the problems, we propose LiteTE, a lightweight, communication-efficient triangle enumerating scheme. To reduce communication cost, LiteTE proposes several techniques, including a graph partitioning method to fully leverage the large memory of commodity servers and the high bandwidth of modern networks and a fast broadcast algorithm to effectively utilize the bidirectional bandwidth of cables and the aggregate bandwidth of clusters. To reduce load imbalance, LiteTE proposes three-level techniques, including a codesign technique of graph partitioning and partition-level load balance, a decentralized dynamic node-level load balance technique, and a chunk-based lock-free work-stealing technique, all of which are lightweight and incur no or hardly any communication cost. The experimental results show that LiteTE reduces communication cost and load imbalance considerably and achieves much better performance in metrics, such as setup time, runtime, scalability, and load balance than the state-of-the-art algorithms. On a small-scale cluster, LiteTE enumerates the 15 trillion triangles in a graph of 92 billion edges in 15 min, while other algorithms fail to complete.

INDEX TERMS Triangle enumerating, triangle computation, graph processing, distributed computing, parallel processing.

I. INTRODUCTION

A triangle, i.e., a subgraph of three vertices pairwise connected, is an important concept in the structural analysis of graphs. Triangle Computation (TC), which obtains the number of triangles or enumerates every triangle in a graph, is a fundamental tool in graph processing to compute important graph properties such as clustering coefficient and transitivity ratio [1]. TC has been widely used in real-world applications, such as detecting spamming activity, assessing content quality in social networks [2], and optimizing query plans in databases [3]. Therefore, TC has been extensively studied [3]–[10]. Due to the potential capability to process huge graphs quickly, distributed-memory TC algorithms have attracted considerable interests. However, existing solutions suffer from low speed due to high communication cost and load imbalance [8], [11]–[17].

To simplify discussions, we assume that the number of processes (or threads) is equal to the total number of CPU cores in the cluster so that each core is dedicated to running one process (or thread) to get maximum computing power. We also assume that the aggregate available memory capacity of the cluster is not smaller than the storage sizes of graphs. The two assumptions are also made by all distributed-memory TC algorithms [8], [11], [12], [14]–[17]. In this paper, **setup time** means the time after input finished and before triangle computing actually starts, and **runtime** means the time during which triangle computing is actually executed.

The associate editor coordinating the review of this manuscript and approving it for publication was Wei Wang.

Existing distributed-memory TC solutions are generally divided into two categories: range-partitioning scheme and vertex-centric scheme. The former usually use adjacency list to store graphs and divide graphs into partitions including consecutive vertices (range-partitioning). Because they support direct two-hop neighbor access and are efficient to implement state-of-the-art TC algorithms [11], [12], [15]. However, due to the inefficiency of load balance techniques and massive messages incurred in setup time and runtime, these solutions usually suffer from long processing times. The latter employ fine-grained load balance techniques and vertex-centric computation mode in which each vertex need to send messages to all its neighbors. Hence, they usually achieve good load balance effect while still suffer from low speed due to high communication cost from massive messages [8], [14]. Furthermore, because a vertex cannot directly send messages to its two-hop neighbors in vertex-centric mode, each vertex has to send its entire neighbor list to all its neighbors, which results in prohibitive memory footprint and causes vertex-centric algorithms cannot process huge graphs.

Due to the inherent limitations of vertex-centric mode, tailored TC algorithms usually employ the range-partitioning scheme [11], [12], [15]. This paper proposes another range-partitioning TC algorithms called LiteTE. As we have mentioned, the two major problems to be solved in range-partitioning TC are 1) the high communication cost due to massive messages and long network latency compared with local memory access [13]; 2) the high load imbalance due to the power-law degree distribution ubiquitous in large-scale graphs (i.e., large-scale graphs usually have a small number of huge-degree vertices and a vast majority of low-degree vertices [14]). LiteTE solves the two problems by proposing several lightweight but effective techniques and achieves much better performance. Our contributions are as follows:

A. DRAMATIC REDUCTION OF COMMUNICATION COST

First, existing solutions often divide graphs into as many partitions as the number of processes and result in a large number of small partitions, which incurs massive messages during runtime [11], [12], [15], [17]. To address the problem, we leverage the big memory of modern servers and high bandwidth of modern networks. Specifically, we divide graphs into appropriate huge partitions and judiciously copy each partition to multiple nodes (called a **nodegroup**) that share the processing of the partition. This helps to reduce runtime messages by dozens of times. Second, we further employ message coalescing and communication-computation overlapping to further reduce runtime messages significantly and hide communication cost effectively. Third, when copying huge graph partitions (in GBs) to multiple nodes, we find that the built-in broadcast algorithm in MPI is inefficient, and propose a fast broadcast algorithm called FastBC. FastBC can effectively utilize the bidirectional bandwidth of cables and the aggregate bandwidth of clusters, and achieve much faster speed than the MPI built-in algorithm.

B. EFFICIENT THREE-LEVEL LOAD BALANCING

Existing solutions usually try to partition graphs as evenly as possible based on complex load metrics whose computing incurs massive messages and causes long setup times. We argue that the loss of partitioning graph based on complex load metrics outweighs the gain, and show that simple metrics such as storage size and the number of nodes can be used by lightweight but effective load balance techniques to achieve better load balance. To better balance load on partition-, node- and thread-level, we propose a codesign technique of graph partitioning and partition-level load balance, a decentralized dynamic node-level load balance technique and a chunk-based lock-free work-stealing technique respectively. These techniques incur few messages or avoid messages and hence incur negligible overhead. Furthermore, due to better load balance, these techniques help to achieve much faster processing speed.

C. EXTENSIVE EVALUATIONS

We implement different versions of LiteTE to evaluate the effect of our techniques. Extensive evaluations are executed with eight real-world graphs and six synthetic graphs on a 12-node cluster. The results show that, compared with the state-of-the-art distributed algorithms, LiteTE reduces communication cost dramatically and achieves much better load balancing, and gains speedups of $9\times$ to $202\times$ with an average of $49\times$ for runtime. For setup time, scalability and load balance, the performance of LiteTE is also much better. In the small-scale cluster (only having 144 cores and 384 GB aggregate RAM), LiteTE can enumerate the 15 trillion triangles of a graph with 92 billion edges in 15 minutes, which all competing algorithms fail to complete due to massive messages or intermediate data.

The rest of the paper is organized as follows. §II presents preliminaries. The techniques of LiteTE are described and analyzed in §III. §IV shows and analyzes experimental results. In §V, existing solutions and related works are reviewed. Finally, §VI concludes the paper with remarks on future work. For ease of reference, Table 1 lists frequently used notations.

TABLE 1. Notation.

Notation	Description
N_u	The neighbor list of a vertex u
S	The number of partitions of a graph
N	The number of computing nodes in a homogeneous cluster
c	The number of cores in each computing node
P	The number of processes/threads/cores in a distributed system, i.e., $P = c * N$
M	The available memory capacity of each computing node

II. PRELIMINARIES

In this section, we first provide a primer on TC and introduce the common degree-based orientation technique used by most recent TC algorithms and also adopted in LiteTE. Then we introduce a distributed-memory TC algorithm called Surrogate, a range-partitioning MPI-implemented algorithm that inspires our work.

A. TRIANGLE COMPUTATION

Triangle Computation (TC) includes triangle counting and triangle enumerating (listing). The former obtains the number of triangles of the entire graph (global triangle counting) or of every vertex (local triangle counting), and the latter enumerates (lists) every triangle in a graph. The two problems are similar, and the only difference between them is that the former increments a counter while the latter enumerates the triangle after each triangle is found. This paper focuses on triangle enumerating and our conclusions are also tenable for triangle counting. As all previous works have done, we do not actually output each triangle and just call a function **enum** with the triangle as its parameter after a triangle is found [18]–[20].

B. ORIENTATION HEURISTICS

Most recent TC algorithms leverage an efficient degree-based orientation heuristics [4], [5], [8], [11], [12], [15], [16], [18], [21], which is based on a total order $\prec$ of vertices and defined as follows [18]:

$$u \prec v \Longleftrightarrow d_u < d_v \; or \; (d_u = d_v \; and \; u < v)$$

where d_u is the degree of a vertex with ID u. For an undirected graph, the orientation heuristics can be executed by directing each edge from its lower ranked vertex to the other, i.e., after orientation every edge (u, v) satisfies $u \prec v$.

With orientation executed, each triangle (u, v, w) is enumerated/counted exactly once when $u \prec v \prec w$, while without orientation each triangle will be enumerated/counted six times (all permutations of the three vertices). More importantly, orientation helps to considerably speedup TC algorithms by reducing time complexity [11]. LiteTE also leverages the orientation heuristics.

C. SURROGATE AND MESSAGE INVERSION

PATRIC [11] and Surrogate [12] (detailed later) are MPI-implemented distributed-memory TC algorithms with Surrogate being an updated version of PATRIC. To run Surrogate, a graph is range-partitioned: The vertex set V of a graph is divided into S non-overlapping subsets $V_0, V_1, \ldots, V_{S-1}$, each containing consecutive vertices, where $S = P = N * c$, i.e., the number of cores/processes. The edge set E is also divided into S non-overlapping subsets $E_0, E_1, \ldots, E_{S-1}$ and any E_i contains all edges (u, v) such that $u \in V_i$ and $v \in N_u$, i.e., all edges incident to the vertices in V_i. A partition s_i consists of V_i and E_i. Each process p_i runs Surrogate on s_i and is responsible for counting triangles incident to each v in s_i.

The key operation in most recent TC algorithms including Surrogate is intersection [4], [5], [8]–[12], [14]–[16], [18], [21]. For each edge (u, v), all triangles containing the edge will be found by doing $N_u \cap N_v$, as illustrated in Fig. 1(a). To make Surrogate understandable, we simplify it in Fig. 1(b). For each edge (u, v) in s_i (line 1), different actions are taken depending on whether v is also in s_i or not. If v is also in s_i (say v_0 in Fig. 2), $N_u \cap N_v$ is done without message passing (line 3).

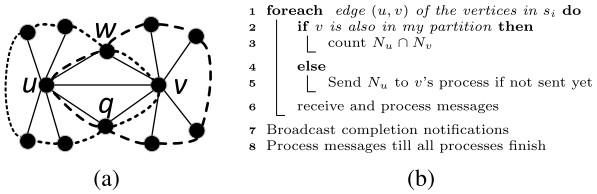

FIGURE 1. Intersection operation in TC and the Surrogate algorithm. (a) All triangles containing an edge (u, v) are found by doing $N_u \bigcap n_v$. (b) The algorithm is run by each process p_i.

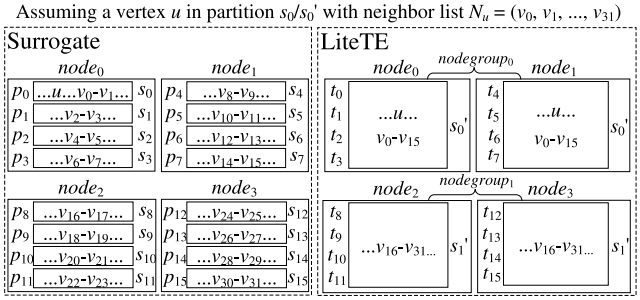

FIGURE 2. Running example.

If v is in another partition s_j (say v_2 or v_8 in Fig. 2), a naive method is to request N_v from p_j and $N_u \cap N_v$ is done by p_i. However, this method usually causes N_v to be sent to p_i multiple times because v is probably the neighbors of multiple vertices in s_i.

Instead of the naive method, Surrogate proposes a **message inversion** technique. Whenever v is in another partition s_j, p_i sends N_u to p_j (line 5) and $N_u \cap N_v$ is done by p_j (line 8). N_u only needs to be sent once to p_j, because once N_u arrives, for all $w \in N_u \cap s_j$, $N_u \cap N_w$ is done by p_j. We take the edge (u, v_8) of s_0 in Fig. 2 as an example to show the details. We assume that the process p_0 is processing u and vertices in N_u are evenly distributed across partitions. Instead of requesting N_{v_8} from p_4, p_0 sends N_u to p_4. After N_u arrives, $N_u \cap N_{v_8}$ and $N_u \cap N_{v_9}$ both are done. In the naive method, p_0 requests N_{v_8} and does $N_u \cap N_{v_8}$. Because v_8 may probably be the neighbor of multiple vertices in s_0, and thus N_{v_8} may probably be sent to p_0 multiple times.

Due to message inversion, Surrogate reduces runtime messages significantly. However, we observe that Surrogate still generates massive messages during both setup time and runtime. Our proposed scheme LiteTE, which also uses message inversion, reduces runtime messages further by at least orders-of-magnitude times and avoid setup-time messages completely.

III. LiteTE

In this section, we first present our graph partitioning technique, and then present the message coalescing and communication-computation overlapping techniques, both of which are used to reduce the runtime communication cost; Next, we show the fast broadcast algorithm for huge messages, which is used to speed up the copy of huge

graph partitions and reduce setup time significantly; Finally, we present our techniques to solve the load imbalance problem, i.e., the three-level load balance techniques. The main framework of LiteTE is shown in Fig. 3.

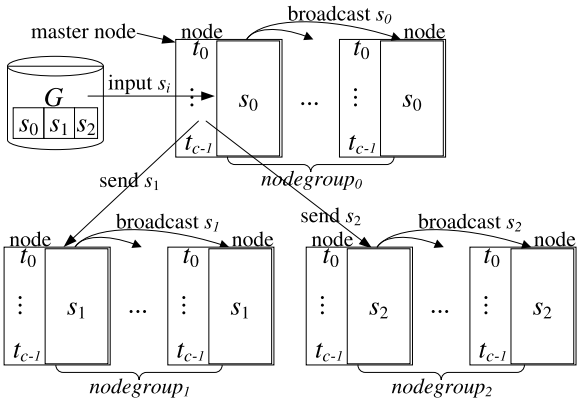

FIGURE 3. LiteTE framework. Assuming the graph G should be divided into three partitions (s_0 to s_2). The workflow is as follows: 1) the master node compute the minimum number of partitions S based on the graph size and M §(III-D; 2) the master node divides nodes into S nodegroups as evenly as possible §(III-D; 3) the master node computes graph partitioning scheme based on the graph size, M and N §(III-D; 3) the master node inputs and sends each partition to its nodegroup, and each partition is broadcasted in its nodegroup §(III-A; 4) each node enumerates triangles and balances load §(III-D to §(III-F).

A. GRAPH PARTITIONING

LiteTE also use range-partitioning scheme. Like in Surrogate, a partition s_i consists of a vertex subset V_i containing consecutive vertices and an edge subset E_i containing every edge (u, v) such that $u \in V_i$ and $v \in N_u$, i.e., all edges incident to the vertices in V_i. The partitioning methods of existing works usually try to divide graphs into partitions as many as the number of cores, which results in massive runtime messages hence long runtimes [11], [12], [14], [22], [23]. To solve the problem, we divide graphs into appropriate huge partitions up to M to fully utilize the large memory and high network bandwidth, and leave load balance to other techniques (§III-D to §III-F). "Appropriate huge" means that the partition sizes are not the huger the better due to imbalance issues. The optimal partition sizes should be determined by comprehensively considering the graph size, M and N, which is detailed in §III-D.

Because S is usually much smaller than N (in contrast, $S = N * c$ for Surrogate), a partition is usually judiciously copied to multiple nodes (called a **nodegroup**). All nodes in a nodegroup share the processing of the partition and do not need to pass messages one another. Furthermore, by using the message inversion technique, messages across nodegroups are also reduced dramatically. The detailed reasons behind the dramatic reduction of messages are shown through the example in Fig. 2. 1) More intersections are done without message passing, including intersections between N_u and N_{v_0} to $N_{v_{15}}$ (for Surrogate, only N_{v_0} and N_{v_1}); 2) more intersections are done after an inverted message N_u arrives at $node_2$ or $node_3$, including intersections between N_u and $N_{v_{16}}$ to $N_{v_{31}}$ (for Surrogate, only two intersections); 3) in LiteTE, nodes in a nodegroup, say $node_2$ and $node_3$, need not pass messages one another because each node has the same partition (s'_1) stored in a share-memory area. In practice, the message reduction is more significant due to more nodes and more cores per node. Besides the dramatically reduced messages, another notable advantage of huge partition is that the partition-size imblance is reduced significantly from existing work, which helps to process much larger graphs (detailed in §IV-B2).

Algorithm 1 Graph Partitioning and Partition-Level Load Balance

```
S ← ceiling(graph_size/M)    // the number of partitions
if S > N then
    print "Too big graph to be processed"
    return
Divide all the N nodes into S nodegroups in a round-robin fashion
Set ideal per-node size ideal_ps = graph_size/N
Set the initial size of each partition to #node_of_my_nodegroup * ideal_ps
if N MOD S = 0 then
    return
S_l ← the number of larger nodegroups
n_l ← #node in a larger nodegroup
S_s ← S − S_l    // the number of smaller nodegroups
if ideal_ps * n_l > M then
    the size of partitions of larger nodegroups is set to M
    the size of partitions of smaller nodegroups is set to (graph_size − M * S_l)/S_s
```

The partitioning algorithm (Algorithm 1, detailed in §III-D) only computes partitioning scheme, and the actual partitioning work is done during input. To input and partition a graph, the master node inputs and sends the partitions one by one in reverse order. Because the vertices in partitions are consecutive, border vertices are determined by simply scanning vertices during input, which hardly incurs any cost. For each partition, the master node inputs and sends the partition to the first node in the nodegroup, and the first node in turn broadcasts the partition to the remaining nodes in the nodegroup (if any), as shown in Fig. 3.

Assume that the number of vertices in each partition is equal. For any edge (u, v) in a partition s_i, u is in s_i according to partitioning method (§II-C). The probability that v is also in s_i is $1/S$, and thus $N_u \cap N_v$ is done without message passing with probability of $1/S$. For Surrogate, S is determined by the number of cores (i.e., $S = P = c * N$) and usually in hundreds or more. Thus, $1/S$ is small and the vast majority of computing work is done via message passing. Because most real-world large-scale graphs can be fit in the memory

of one node [24], i.e., $S = 1$ for LiteTE, message passing is avoided for these graphs. However, when process huge graphs, $1/S$ can be insignificant even for LiteTE (say, when $S > 10$), and most of the work still must be done via message passing. In this case, the advantage of LiteTE is mainly shown by dramatically increasing the efficacy of message inversion.

Theorem 1: For both Surrogate and LiteTE, the total number of messages is $O(n(S-1))$, and the total size of messages is $O(m(S-1))$, where n and m are the numbers of vertices and edges in a graph respectively.

Proof: Assume that the number of vertices in each partition is equal. For any partition s_i and any vertex u in s_i, a message N_u should be sent once to each of the other $S-1$ partitions in the worst case, i.e., for each partition, the total number of messages sent is $O(n/S*(S-1))$. Hence, the total number of messages for all partitions is $O(S*n/S*(S-1)) = O(n(S-1))$. Assume that the average vertex degree is $\bar{d}$ and each edge is stored with one unit of storage. The total size of messages is $O(n(S-1))*\bar{d} = O(n\bar{d}(S-1)) = O(m(S-1))$, in which the last $=$ is because $n\bar{d}$ is just the number of edges in a graph, i.e., m. □

For Surrogate, $S = c*N$, i.e., the number cores (processes), and $O(n(S-1)) = O(n(cN-1)) = O(cnN) = O(N)$, i.e., the total number of messages is linear to N. For LiteTE, S is determined by the graph size and M, and is independent of N (detailed in §III-D). The discussion related to the total size of messages is similar.

Theorem 2: In terms of the number of messages, LiteTE is more than c times fewer than Surrogate.

Proof: Given a graph and assume that the number of partitions in Surrogate and LiteTE are S_s and S_l respectively. The per-process number of messages in Surrogate is

$$n(S_s - 1)/P = n(P-1)/P \tag{1}$$

because $P = c*N = S_s$. The per-process number of messages in LiteTE is

$$n(S_l - 1)/P \tag{2}$$

where $P = c*N > S_l$. Thus, the ratio of (1) to (2) is

$$\frac{(1)}{(2)} = \frac{n(P-1)/P}{n(S_l-1)/P} = \frac{P-1}{S_l-1} > \frac{P}{S_l} = \frac{cN}{S_l} \geq c \tag{3}$$

The $>$ in (3) is because $P > S_l \geq 1$. The $\geq$ in (3) is because $N \geq S_l$, i.e., in LiteTE the number of partitions S_l is no larger than the number of nodes N, or else the graph cannot be fit in the aggregate available memory of the cluster. □

Current commodity servers equipped with dozens of cores are commonplace, i.e., c is usually in dozens. This means that, even in the worst case, i.e., the number partitions in LiteTE is equal to N, the number of messages in LiteTE is still reduced by more than dozens of times. For the vast majority of graphs, N is much larger than S_l and the reduction is more significant (When $S_l = 1$, it is ∞).

B. MESSAGE COALESCING AND COMMUNICATION-COMPUTATION OVERLAPPING

In TC, for any edge (u, v), $N_u \cap N_v$ needs to be computed. Thus, TC algorithms are usually more compute-intensive and generate more short messages than common graph algorithms such as BFS and PageRank (hence much longer runtimes [14], [25]). Message Coalescing (MC) usually achieves better speedup when massive short messages are generated [13]. Thus, TC algorithms are amenable to MC. If an algorithm are compute-intensive and generates massive long messages, Communication-Computation Overlapping (CCO) usually achieves reasonable speedup [13]. By utilizing MC, TC algorithms will meet the requirements. Hereafter, the two techniques are collectively called **MC3O**. Based on the discussions, we employ MC3O in LiteTE.

In each process, $S-1$ pairs of sending buffers, i.e., one pair for each of the other $S-1$ partitions, and one pair of receiving buffers are created. Of the two buffers in a sending pair, one is used to send a (coalesced) message while simultaneously the other is used to collect and coalesce messages generated during computing. Once the message sending is completed and the other buffer is full, the two buffers are swapped. The usage of the pair of receiving buffers is similar. The MC3O techniques help to achieve higher speed and the capability of processing larger graphs as detailed in §IV-B2.

In Thm. 2, we have proved that in terms of the number of messages, LiteTE is more than c times fewer than Surrogate, and c is usually in dozens. By using message coalescing in LiteTE, assuming the message buffer size is B and the average degree is $\bar{d}$, the number of messages is further reduced by a factor of $B/\bar{d}$. When B is in KBs, $B/\bar{d}$ is usually in dozens. Hence, the number of messages in LiteTE is reduced by more than orders-of-magnitude times from Surrogate (dozens × dozens). Due to the common elementary operations, i.e., for each edge (u, v), $N_u \cap N_v$ is computed, the computing complexity of LiteTE are the same to most of the state-of-the-art TC algorithms including Surrogate, i.e., $O(E^{3/2})$ [3]–[10].

C. FAST BROADCAST ALGORITHM

When broadcast huge partitions, we find that the built-in algorithm of MPI is inefficient [26]. In big data era, broadcasting huge chunks of data in clusters is a common operation. For example, to support this operation, Hadoop introduces Distributed Cache and the size is set to 10 GB by default [27]. However, existing broadcast algorithms are usually designed for messages no larger than hundreds of MBs, and they usually use tree-based logic topology and small-chunk-based pipelining techniques which cause the contention of the bandwidth of a physical link by multiple logic links and high chunking overhead [26], [28], [29]. To fully utilize each cable's bidirectional bandwidth and the aggregate bandwidth of clusters, and avoid the chunking overhead of pipelining, we propose a Fast BroadCast algorithm (FastBC).

In FastBC, the master node divides a huge message into N-1 equal-sized chunks (seed chunks) and sends one chunk to each slave node. Then, each slave node sends its seed chunk to

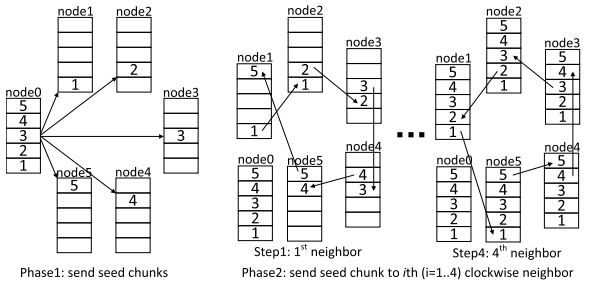

FIGURE 4. Fast BroadCast algorithm (FastBC).

other slave nodes one by one. The details are shown in Fig. 4. Node0 is the master. In Phase1, node0 sends each chunk to a slave node; In the ith step of Phase2, each slave node sends its seed chunk to its ith ($i = 1..4$) clockwise neighbor. Specifically, in Step1, each slave node sends its seed chunk to its first clockwise neighbor; in Step2, each slave node send its seed chunk to its second clockwise neighbor, etc. Obviously, in FastBC there are no contention of a single physical link by more than one logic link (i.e., an arrow, and a cable contains two reciprocal physical links), and the aggregate bandwidth of the cluster is more effectively utilized. Furthermore, FastBC is simple and easy to be implemented. In our 12-node cluster, in which all nodes are connected to a switch, the speed of FastBC when broadcast messages of GBs is more than 5× faster than the MPI built-in algorithm. We leave the discussions of the efficiency of FastBC on other network topologies to future work.

D. CODESIGN OF GRAPH PARTITIONING AND PARTITION-LEVEL LOAD BALANCE

To avoid the complex load metrics whose computing incurs massive messages, metrics that incur no message such as storage size and the number of vertices are used. We will show that, by combining these simple metrics and lightweight but effective load balance techniques, dramatic speedups and better load balance can be achieved.

As discussed in §III-A, the larger the partition is, the fewer messages are induced. An intuitive partitioning method is to divide graphs into partitions as large as M (the last one is usually smaller than M). However, the intuitive method may result in high load imbalance and hurt runtime performance. For example, we assume that the size of a graph is 100, $M = 47$ and $N = 4$. By using the intuitive method, the graph are divided into three partitions with sizes of 47, 47 and 6. The nodes should be divided into three nodegroups containing 2, 1 and 1 node respectively. To measure imbalance, we define **Imbalance Ratio** (**IR**) as the ratio of the maximum of a metric to the minimum, i.e., $IR = max/min$. By using the intuitive method, the per-node load IR across nodegroups is as high as 7.8 (47/6).

To fully utilize the big memory of modern servers and avoid high load imbalance, we propose a codesign algorithm of graph partitioning and partition-level load balance, as shown in Algorithm 1 (examplified later). The main idea is as follows. S is minimized by being set to $ceiling(graph\_size/M)$ (line 1). Nodes are divided into S nodegroups as evenly as possible by being simply assigned to each nodegroup in a round-robin fashion (line 5), which results in no more than two types of nodegroups with the larger one contains one more nodes than the smaller one. Each partition gets an initial size based on the ideal per-node size and the number of nodes in its nodegroup (line 6-7). If the initial partition size of larger nodegroups are larger than M, we adjust the size to M and adjust the size of the smaller nodegroups' partitions accordingly (line 13-15).

To exemplify the algorithm, we assume that the size of a graph is 100, $M = 47$ and $N = 4$, and hence the three nodegroups contains 2, 1 and 1 node respectively. The ideal per-node size is 25 (100/4) and each partition gets an initial (ideal) size of 50, 25 and 25 respectively. Because $M = 47 < 50$, the final partition sizes should be adjusted to 47, 26.5 and 26.5 respectively. Hence, the per-node load IR is reduce to 1.1 (26.5/23.5) and the maximum per-node load is reduced from 47 in the intuitive method to 26.5.

We now analyze the algorithm. The main conclusion is that the per-node load $IR \leq 2$. The algorithm can be analyzed in the following two cases. Case1: When N is divisible by S, $IR = 1$. Case2: When N is not divisible by S, n_l and n_s denote the numbers of nodes for larger and smaller nodegroups respectively. Then, we have $n_l = n_s + 1$. s_l and s_s denote the partition sizes for larger and smaller nodegroups respectively, and then $s_s \leq s_l$. $ideal\_ps$ denotes the ideal per-node size. If $ideal\_ps * n_l \leq M$, we need not adjust the initial partition sizes and the per-node load $IR = 1$. If $ideal\_ps * n_l > M$, we have to adjust the initial partition sizes by decreasing s_l and increasing s_s. The per-node load $IR = \frac{s_s/n_s}{s_l/n_l} = \frac{s_s * n_l}{s_l * n_s} = \frac{s_s}{s_l}(1 + \frac{1}{n_s})$. Because $\frac{s_s}{s_l} \leq 1$ and $1 + \frac{1}{n_s} \leq 2$ ($n_s \geq 1$), and we have $IR \leq 2$.

E. DECENTRALIZED DYNAMIC NODE-LEVEL LOAD BALANCE

After nodes are divided into nodegroups, we need to consider how to balance the load of a partition across the nodes of its nodegroup. Static techniques are usually cost-effective while cause considerable load imbalance [4], [11]; Centralized dynamic techniques usually gain good balancing effect while cause considerable overhead [30]. We propose a technique called Decentralized dynamic Node-level Load Balance (DNLB), which is lightweight hence easy to be implemented and only incurs few messages.

In DNLB, the vertices are processed by each node in a round-robin fashion at starting point. If a node is not the last one finishes its load, it sends idle notifications to the intra-nodegroup nodes still working on their own load to request load sharing. The nodes that have received idle notifications respond with need-not-help message if they only have little load left or designate an appropriate amount of remaining load to the idle node (Data migration is not needed

because each node in the same nodegroup has the same partition). When nodes finish the designated load and become idle again, they broadcast completion notifications to all nodes in the cluster.

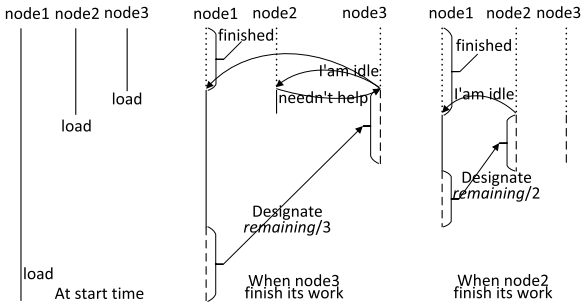

FIGURE 5. Decentralized dynamic node-level load balance working on a threenode nodegroup.

We take a three-node nodegroup as an example to detail DNLB, as shown in Fig. 5. When triangle enumerating starts, each node starts to process its load. After node3 finishes its load, it sends idle messages (''I'am idle'') to nodes still processing their load. After node2 receives the message, because its remaining load is less than a threshold (we test and set the threshold to 1024 vertices, because balancing of too little load cannot gain performance improvement), node2 responds with a need-not-help message; After node1 receives the message, it designates 1/3 of remaining load to node3, where 3 is the number of nodes whose idle notifications have not been received plus one. When node2 finishes its load, the operations are similar. When node1 finishes its load or node2 (node3) finishes its designated load, they broadcast completion notifications to all nodes in the cluster and continue to process messages from other nodes until all nodes finish their work. The triangle enumerating algorithm of LiteTE after incorporating DNLB is shown in Algorithm 2. DNLB reduces the intra-nodegroup runtime IR from around 1.8 when only use round-robin load balance to around 1.1.

Algorithm 2 Triangle Enumerating in LiteTE

```
1  foreach edge (u, v) of the vertices in my load parallel do
2      if v is also in my partition then
3          enumerate triangles by doing N_u ∩ N_v
4      else
5          Send N_u to one of nodes containing v if not sent yet
6      while exist pending messages do
7          receive a message
8          case
9              message is neighbor list
10         enumerate triangles case
11             message is idle notification from node_j
12         if my remaining #vertices > 1024 then
               // r is #nodes whose idle notifications haven't been received
13             designate 1/(r + 1) of my remaining work to node_j
14         else
15             send need-not-help message to node_j
16         case
17             message is completion notification
18         increment completion counter
19     if I'm not the last node becoming idle then
20         send idle notification to intra-nodegroup nodes whose idle notifications haven't been received
21         In parallel process messages and designated load until designated load done
22 Broadcast completion notifications
23 Process messages till have received N − 1 completion notifications
```

F. CHUNK-BASED LOCK-FREE WORK-STEALING FOR THREAD-LEVEL LOAD BALANCE

In each node, we need to balance load among threads (cores). Because threads can interact with each other through lightweight share-memory access, we need not resort to heavyweight message passing as inter-node load balance. Though pure static methods such as round-robin incurs hardly any overhead, they usually cause remarkable imbalance (e.g. the runtime IR, i.e., *max_runtime*/*min_runtime*, of round-robin is averagely 1.8) [4]. Though fine-grained dynamic techniques, such as work-stealing in edges or vertices can achieve ideal load balance effect, our evaluations show that they incur remarkable overhead. We employ a chunk-based lock-free work-stealing technique to solve the problem. Vertices in the load of the node are divided into chunks containing 128 vertices each, and threads process these chunks in round-robin fashion at starting point. When a thread finishes its own work, it try to steal and process chunks from other threads until all work done. The stealing operation is implemented through the atomic CAS instruction [31] and the use of locks is avoided. Smaller chunk sizes cause too frequent use of atomic instruction and hurt parallelism; Larger chunk sizes may cause measurable load imbalance and hurt runtime. Our evaluations show that the chunk size of 128 vertices achieves a good compromise. After employing this technique, the load imbalance among threads becomes negligible, i.e., the intra-node runtime IR is reduced to close to 1.

IV. EVALUATION

In this section, we experimentally evaluate LiteTE and compare it with state-of-the-art TC algorithms. We first introduce the setup, including the competitor algorithms, the running environment and the datasets used, and then evaluate and

compare LiteTE with competitor algorithms in terms of setup time, runtime, scalability, the number of messages and load imbalance.

A. EXPERIMENTAL SETUP

1) COMPETITOR ALGORITHMS

We mainly compare LiteTE with three state-of-the-art distributed TC algorithms including Surrogate,[1] HavoqGT[2] and PTE.[3] HavoqGT [8] is an vertex-centric TC algorithms originated from PowerGraph [14]; PTE [18] is the fastest MapReduce TC algorithm running on Hadoop, and its authors only provide the binary code.

Parallel shared-memory graph algorithms running on a high-end server with big-memory and dozens of cores are usually significantly faster than the distributed algorithms running on small-scale clusters because shared-memory accesses are much faster than inter-node massage passing. However, because LiteTE reduces messages from previous works by at least orders-of-magnitude and causes message passing to no longer be bottleneck, LiteTE achieves significant speedups than state-of-the-art shared-memory algorithms. The performance improvement over share-memory algorithms will be shown by comparing LiteTE with GBBS[4] and TRICORE.[5] The former is a CPU algorithm and the latter is a GPU algorithm.

2) RUNNING ENVIRONMENT

All distributed algorithms are run in a cluster of 12 nodes connected via QDR Infiniband Network (40 Gbps), with each node equipped with 12 cores and 32 GB of memory. Shared-memory algorithms are run on a high-end server with 256 GB of memory, 32 cores and two K40C GPUs. Though no more nodes available, LiteTE should gain better performance improvements in larger scale clusters given its better scalability (§IV-C). Programs are coded in C++, MPI and Pthreads, and are compiled using G++ 4.9.2 with -O3 option.

Unless otherwise stated, all algorithms are run with the maximum computing power. Specifically, LiteTE, Surrogate and HavoqGT are run with all the 12 nodes and 12 processes/threads per node; For PTE, all the 12 nodes is used, and in each node we run 11 Reducers because it gives the best results. When only part of the nodes are used, programs are also run with the possible maximum computing power. All times are the average of three runs. In LiteTE evaluations, M is set to 26 GB to leave enough RAM to MPI and OS. When the size of the data collected in a buffer reaches a sending threshold of 4 KB, we consider the buffer full.

[1]https://github.com/cbaziotis/patric-triangles
[2]https://github.com/LLNL/havoqgt
[3]https://datalab.snu.ac.kr/ ukang/
[4]https://github.com/ldhulipala/gbbs
[5]https://github.com/huyang1988/TC/

3) DATASETS

The information of datasets is in Table 2. The upper eight datasets are real-world graphs, and the rest are synthetic RMAT graphs [32] generated with a tool from Ligra.[6] In the real-world graphs, soc-lj,[7] twitter[7] and friendster[8] are social networks; arabic,[9] IRLhost,[10] gsh,[9] uk[9] and eu[9] are web graphs. Orientation can be executed in preprocessing [11], [20], [21], [33] or on the fly [4], [5], [19]. Because the binary code of PTE only accepts graphs with orientation executed, the orientation of all datasets is uniformly done in preprocessing for fair comparisons.

TABLE 2. Dataset.

Dataset	#Vertices	#Edges	#Triangles	Dataset	#Vertices	#Edges	#Triangles
soc-lj	4.8M	69M	286M	**IRLhost**	642M	6.4B	438B
arabic	23M	640M	37B	**gsh**	988M	34B	910B
twitter	42M	1.5B	35B	**uk**	788M	48B	7.9T
friendster	125M	2.6B	4.2B	**eu**	1.1B	92B	15T
RMAT2	67M	2.1B	169M	**RMAT8**	268M	8.6B	377M
RMAT4	134M	4.3B	253M	**RMAT10**	537M	11B	148M
RMAT6	268M	6.4B	169M	**RMAT12**	537M	13B	248M

B. SETUP TIME AND RUNTIME

Because LiteTE sends a large graph partition to each node during setup, a natural question arises: whether the sending of these large partitions results in longer overall processing time or not? We first answer the question.

1) SETUP TIME

For LiteTE, Surrogate and HavoqGT, the setup time is the time before triangle computing actually start with input time subtracted. For PTE, the setup time is the pre-partitioning time. To evaluate the effect of FastBC, we use two versions of LiteTE, i.e., one using MPI built-in broadcast algorithm (LiteTE^b) and the other using FastBC (LiteTE). The results are shown in Fig. 6(a). The speedups of LiteTE range from 6.1× to 162× with an average of 58×.

The reasons for the much faster setup speed of LiteTE are 3-fold. First, the partitioning method of LiteTE is lightweight and avoids message passing. In contrast, the setup of competitor algorithms generate massive messages or huge amounts of intermediate data. Second, LiteTE only needs to send some large partitions. Due to high network bandwidth but long latency, long messages can be sent with much higher bps than short messages [13]. Third, the FastBC algorithm more effectively utilize the bidirectional bandwidth of cables and the aggregate bandwidth of the cluster. Compared with LiteTE$_b$, the speedups of LiteTE range from 2.6 to 5.5 with an average of 4.1. For the graphs from soc-LJ to friendster that are divided into only a single partition (i.e., they can be fit in M), the setup time is just the broadcast time and the speedups are higher (from 3.4 to 5.5). For IRLhost and gsh,

[6]https://github.com/jshun/ligra
[7]http://snap.stanford.edu/
[8]https://archive.org/download/friendster-dataset-201107
[9]http://law.di.unimi.it/
[10]http://irl.cs.tamu.edu/projects/motifs/index.asp

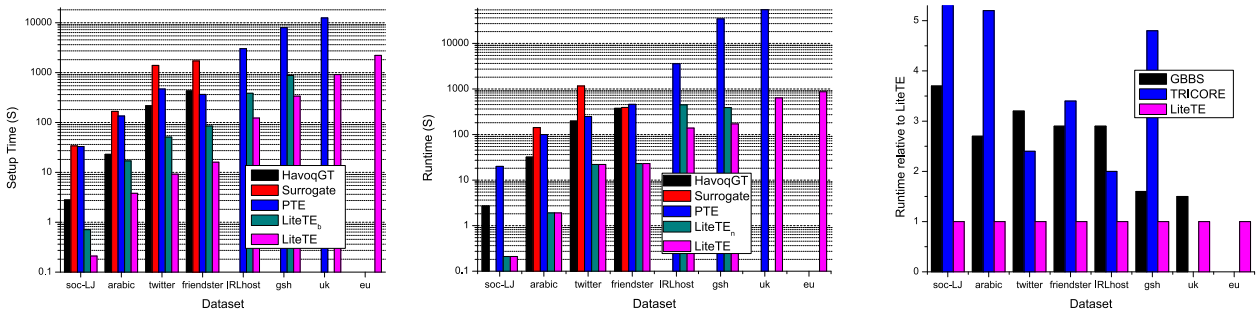

FIGURE 6. Setup time and runtime. (a) LiteTE achieves setup speedups of 6.1× to 162× with an average of 58×. (b) LiteTE achieves speedups of 9× to 202× with an average of 49×. (c) LiteTE achieves speedups of 1.5× to 5.3× with an average of 3.2×. The absent bars indicate processing failures.

the broadcast in some nodegroups is overlapped with partition sending and the speedups are relative low (3.1× and 2.6×).

2) RUNTIME

a: COMPARED WITH DISTRIBUTED ALGORITHMS

To evaluate the effect of MC3O techniques, we use two versions of LiteTE, including LiteTE without MC3O (LiteTE$_n$) and LiteTE with MC3O (LiteTE). The results are shown in Fig. 6(b). The speedups range from 9× to 202× with an average of 49×. Due to the dramatically reduced messages (detailed in §IV-D) and the good effect of the three-level load balance techniques (§III-D to §III-F), LiteTE can successfully process all the graphs with high speed while competing algorithms fail to process at least one.

MPI usually needs to buffer large amounts of messages for speed, and when buffer small messages more auxiliary memory is consumed protect [34]. Surrogate generates at least dozens-of-times more (uncoalesced) small messages than LiteTE$_n$; The partitioning metrics of Surrogate only considers computation load and results in severe size imbalance across partitions; Large-size partitions usually not only take up more memory but also generates more messages. All these factors result in out of memory error (OOM) in the nodes housing large partitions during Surrogate's processing of graphs from IRLhost to eu. LiteTE$_n$ generates dozens-of-times more messages than LiteTE and the messages are small (uncoalesced). Thus, much more auxiliary memory needed to buffer the messages, which results in OOM when LiteTE$_n$ processes uk and eu. For LiteTE, messages are coalesced and at least dozens-of-times fewer than those in LiteTE$_n$. Thus, much less auxiliary memory needed and LiteTE can process all graphs. When processing eu, PTE fails to complete in 48 hours due to huge amount of intermediate data.

Because the four graphs from soc-LJ to friendster can be fit in the memory of a single node and message passing is avoided, which explains why the runtimes of LiteTE$_n$ are identical to those of LiteTE. For IRLhost and gsh, due to the MC3O technique, LiteTE is significantly faster than LiteTE$_n$ (the speedups are 3.2 and 2.3).

b: COMPARED WITH SHARED-MEMORY ALGORITHMS

As we have mentioned, shared-memory graph algorithms on high-end servers are usually significantly faster than the distributed algorithms on small-scale clusters due to the much faster shared-memory accesses than inter-node massage passing. However, LiteTE overcomes this deficiency. We show this by comparing LiteTE with two state-of-the-art shared-memory algorithms including GBBS, a CPU algorithm, and TRICORE, a GPU algorithm, and the results are shown in Fig. 6(c). The speedups range from 1.5× to 5.3× with an average of 3.2×. The significant speedups are achieved by LiteTE because communication and load imbalance overheads are no longer bottlenecks and the processing capacity of CPUs is effectively utilized (computation accounts for more than 70% of runtime in LiteTE. In contrast, in the three competing distributed algorithms, computation accounts for no more than 10% of runtime).

C. SCALABILITY

Both strong scalability, i.e., how runtime varies with the number of nodes for a fixed total problem size, and weak scalability, i.e., how runtime varies with the number of nodes for a fixed problem size per node [35], are evaluated. The strong scalability is shown in Fig. 7. With the number of nodes varies from 1 to 12, the runtime of LiteTE decreases steadily and quickly, showing superior scalability on every graph to other algorithms. As we have proved in Thm. 1, the better strong scalability of LiteTE is because the total number of messages keep constant with the increasing of the number of nodes for a given graph. Because Hadoop must be run on at least three nodes, PTE fails to run on one node. For Surrogate and HavoqGT, the runtimes on a single node are usually shorter than those on 3 nodes because intra-node message passing is implemented through shared-memory copying, which is much faster than inter-node message passing. When HavoqGT processes arabic on one node, the huge number of messages passed along each edge result in heavy usage of swap areas and hence severe memory thrashing, which explains the abnormally long processing time (2156 S). The other missing data points indicate the failures of processing due to OOM.

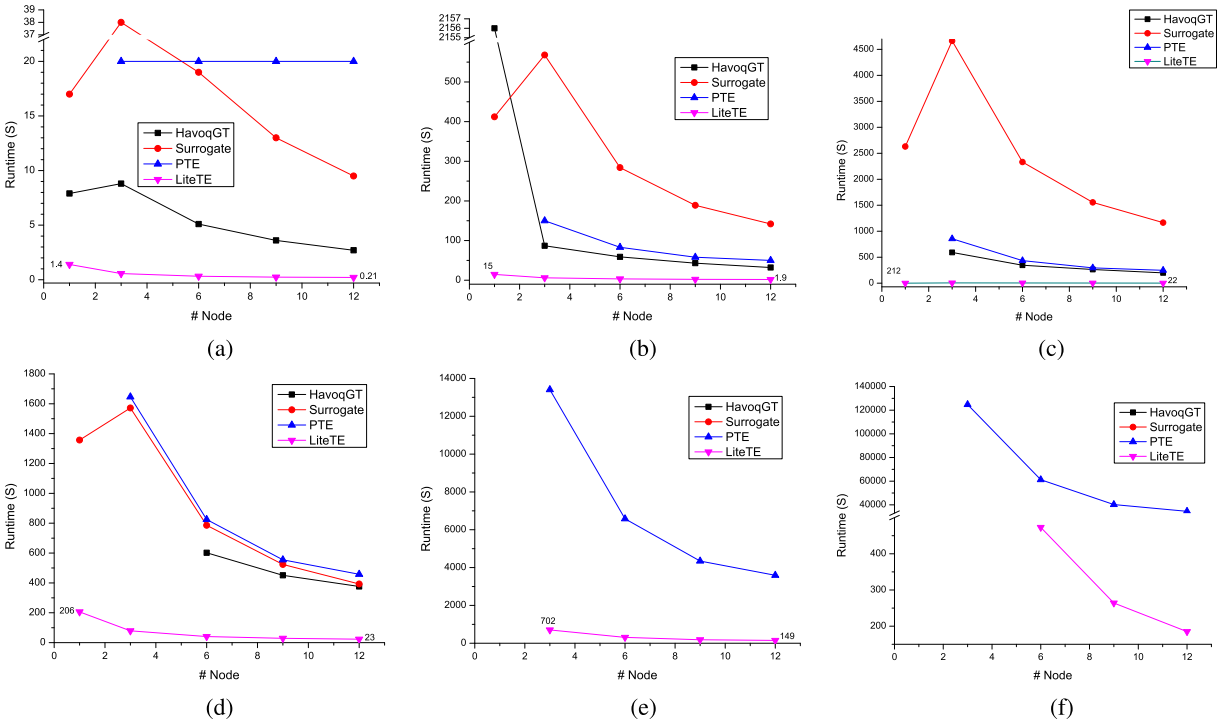

FIGURE 7. Strong scalability. The missing data points indicate the failures of processing (for LiteTE, due to OOM). Note that, there are breaks in some vertical axes. (a) soc-LJ. (b) Arabic. (c) Twitter. (d) Friendster. (e) IRLhost. (f) Gsh.

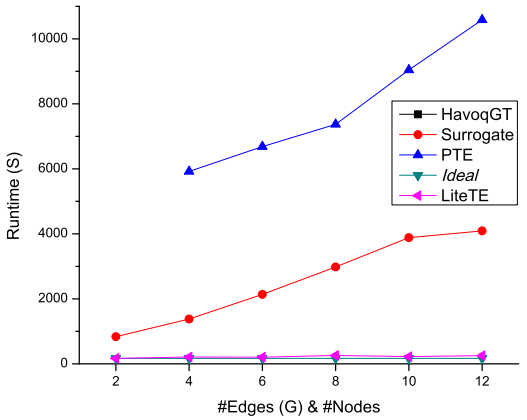

FIGURE 8. Weak scalability. HavoqGT fails to process any graph.

We use synthetic graphs to evaluate weak scalability. When use two nodes, graph RMAT2 is processed; when use four nodes, graph RMAT4 is processed, etc. As shown in Fig. 8, LiteTE shows nearly ideal and much better scalability than other algorithms. The better results are because its novel design reduces messages dramatically and balances load better, which causes communication and imbalance to no longer be bottlenecks. PTE fails to run on two nodes because Hadoop must be run on at least three nodes. HavoqGT fails to process any graph due to OOM caused by huge amounts of messages.

D. THE NUMBER OF MESSAGES

In our analysis in §III-A, we predict that LiteTE should induce at least orders-of-magnitude fewer messages than Surrogate. We will validate the prediction in this section. The number of messages when processing real-world graphs are shown in the upper part of Table 3. For LiteTE, the graphs from soc-LJ to friendster can be fit in the memory of a single node and message passing is avoided. For the rest real-world graphs, only LiteTE can process all of them and Surrogate fails to process any of them. Thus, we cannot directly compare LiteTE with Surrogate on the same graphs when LiteTE needs message passing. Nevertheless, the prediction in §III-A can be indirectly validated by the results of soc-LJ vs. those of from gsh to eu. Though the latter are orders-of-magnitude times larger than the latter, the numbers of messages of gsh to eu in LiteTE (21M, 12M, 65M) are comparable to that of soc-LJ in Surrogate (27M).

TABLE 3. The number of messages.

Dataset	Algorithm	#msg.($\times 10^6$)	Dataset	Algorithm	#msg.($\times 10^6$)
soc-LJ	LiteTE	0	IRLhost	LiteTE	5.1
	Surrogate	27		Surrogate	-
arabic	LiteTE	0	gsh	LiteTE	21
	Surrogate	11		Surrogate	-
twitter	LiteTE	0	uk	LiteTE	12
	Surrogate	385		Surrogate	-
friendster	LiteTE	0	eu	LiteTE	65
	Surrogate	1127		Surrogate	-
RMAT2	LiteTE	0	RMAT8	LiteTE	7
	Surrogate	1153		Surrogate	4595
RMAT4	LiteTE	0	RMAT10	LiteTE	11
	Surrogate	2345		Surrogate	6492
RMAT6	LiteTE	0	RMAT12	LiteTE	14
	Surrogate	3826		Surrogate	7530

To further compare the number of messages, we experiment with RMAT graphs and the results are shown in the lower part of Table 3. For the graphs from

RMAT8 to RMAT10, compared with those of Surrogate, the numbers of messages in LiteTE are reduced by 656×, 590× and 538× respectively, which validates our prediction, i.e. messages are reduced by more than orders-of-magnitude times. The dramatic message reduction stems from our graph partitioning algorithm and the MC3O techniques.

As mentioned in §IV-A, the sending threshold of buffer is set to 4 KB, i.e., once the total size of data in a sending buffer is not smaller than 4 KB, we consider the buffer is full. Our evaluations show that when the threshold goes larger than 4 KB, runtimes no longer decrease obviously.

E. LOAD IMBALANCE

To demonstrate that simple load metrics such as storage sizes and the number of vertices combined with lightweight but effective load balance techniques achieve better load balance than the complex metrics of Surrogate, we further compare the communication and computation imbalance of LiteTE with those of Surrogate.

1) COMMUNICATION IMBALANCE

The IRs (Imbalance Ratio) of the number of messages are shown in Table 4. For LiteTE, the graphs from soc-LJ to friendster can be fit in the memory of a single node and message passing is avoided, and hence the IRs do not exist theoretically. Nevertheless, because we use IR to measure the communication imbalance and communication imbalance does not exist for these graphs, and thus we consider these IRs to be 1, the ideal IR.

TABLE 4. Load imbalance.

Dataset	Communication IR		Computation IR	
	Surrogate	LiteTE	Surrogate	LiteTE
soc-LJ	3.34	1	2.82	1.05
arabic	4.80	1	3.09	1.08
twitter	5.48	1	4.47	1.06
friendster	1.31	1	1.38	1.21
IRLhost	-	1.73	-	1.66
gsh-2015	-	1.12	-	1.64
uk-2014	-	1.26	-	1.70
eu-2015	-	1.13	-	1.49

IR - Imbalance Ratio, i.e., max/min, and 1 is ideal. The smaller the IR is, the better. Dash (-) indicates the failures of processing.

The average IRs of LiteTE and Surrogate are 1.16 and 3.73. Thus, the communication in LiteTE is much better balanced. The better balance comes from the three-level load balance techniques (§III-D to §III-F). The partitioning method of Surrogate only takes computation into consideration and results in significant partition-size imbalance. Large partitions usually generate more messages and cause high communication imbalance. Furthermore, the IRs of 1 for the graphs from soc-LJ to friendster significantly contributes to the overall better balance of LiteTE.

2) COMPUTATION IMBALANCE

The computation load is defined as the runtime of the code snippet after setup is finished and before completion notifications are broadcasted, i.e., line 1 to 6 in Fig. 1(b) or line 1 to 19 in Algorithm 2, and the IRs are shown in Table 4. The average IRs of LiteTE and Surrogate are 1.36 and 2.94. Thus, the computation of LiteTE is better balanced. The better balance of LiteTE comes from the three-level load balance techniques (§III-D to §III-F). When LiteTE processes graphs that can be fit in the memory of a single node (from soc-LJ to friendster), the low average IR of 1.10 shows the high efficiency of the decentralized dynamic node-level load balance technique (III-E) and chunk-based lock-free work-stealing technique (III-F) because the partition-level load balance technique does not take effect for these graphs.

V. RELATED WORK

To support TC in various settings, a large body of works propose different techniques. They are generally classified into four categories and briefly reviewed in this section.

A. SINGLE NODE OUT-OF-CORE TC

These algorithms usually focus on TC in a PC. Because large-scale graphs usually cannot be fit in the memory of a PC and must be processed out of core. These works usually introduce efficient I/O techniques to leverage the relatively high sequential bandwidth of secondary storage devices. Hu *et al.* [19] present MGT, which buffers a specific number of edges in memory and search all the triangles containing one of the in-memory edges by traversing every vertex. Cui *et al.* [20] propose Trigon, which creates a scheme based on the lessons learned from previous works and allows balance between the I/O and CPU costs in order to achieve high speed.

Despite highly cost effective, these algorithms usually cannot be scaled out to process huge graphs and suffer from long processing times due to the limited resources of a PC. Furthermore, these algorithms usually cannot output all the triangles of huge graphs. Taking the eu graph in our datasets for example, it contains 15 trillion triangles taking up 180 TB of storage space if all are output.

B. SHARED-MEMORY TC

Because graphs with billions of edges can usually be fit and efficiently processed in the memory of a commodity server equipped with hundreds of GB memory and dozens of cores, several share-memory TC algorithms are proposed. Except the competitor algorithms introduced in §IV-A, Latapy [36] proposes an algorithm to solve the load imbalance problem of TC. Shun and Tangwongsan [5] present a high-speed cache-oblivious algorithm, which can be easily implemented with dynamic multithreading libraries such as Cilk Plus and OpenMP.

Shared-memory TC in a commodity machine is usually more efficient than in a cluster in terms of cost effectiveness, ease of programming and runtime. However, due to the limited resources of a single machine, these algorithms usually cannot be scaled out to process huge graphs with dozens of billions of or more edges. Furthermore, our proposed algorithm LiteTE has shown that distributed algorithms can be

much faster, which seems to diminish the speed superiority of these algorithms.

C. DISTRIBUTED TC

Except the competing algorithms introduced in §IV-A, there are many other distributed works. Zhu *et al.* [37] propose a MapReduce algorithm called FTL, which utilizes a light-weight data structure to reduce intermediate data and leverage multiple-round techniques to reduce the usage of memory and network bandwidth. Giechaskiel *et al.* [4] propose PDTL, which copies the entire graph to the secondary storage of every node and processes the graph in a static load balance way. Wang *et al.* [16] propose a hybrid TC algorithm called SEN-Iterator that uses the bulk synchronous parallel (BSP) mode.

MapReduce allows for simplified programming on clusters of commodity machines, and MapReduce programs can be easily scaled out to a large number of machines. However, MapReduce TC algorithms usually suffer from long processing times due to the huge amount of intermediate data exchanged through the network. Because the entire graph needs to be copied to every machine, PDTL cannot be scaled out to process huge graphs. Algorithms based on the BSP mode usually generate massive messages and hence are slow.

VI. CONCLUSION AND FUTURE WORK

To speed up distributed-memory TC, we propose LiteTE in this paper, which divides graphs into appropriate huge partitions to dramatically reduce messages during runtime, and introduces three-level load balance techniques to better balance load across partitions, nodes and threads with hardly any message passing. Evaluations show that LiteTE achieve much better performance than previous algorithms in runtime, setup time and other metrics. Future work can focus on expanding the work to other graph algorithms.

REFERENCES

[1] M. E. Newman, "The structure and function of complex networks," *SIAM Rev.*, vol. 45, no. 2, pp. 167–256, 2003.

[2] L. Becchetti, P. Boldi, C. Castillo, and A. Gionis, "Efficient semi-streaming algorithms for local triangle counting in massive graphs," in *Proc. SIGKDD*, 2008, pp. 16–24.

[3] Z. Bar-Yossef, R. Kumar, and D. Sivakumar, "Reductions in streaming algorithms, with an application to counting triangles in graphs," in *Proc. SODA*, 2002, pp. 623–632.

[4] I. Giechaskiel, G. Panagopoulos, and E. Yoneki, "PDTL: Parallel and distributed triangle listing for massive graphs," in *Proc. ICPP*, Sep. 2015, pp. 370–379.

[5] J. Shun and K. Tangwongsan, "Multicore triangle computations without tuning," in *Proc. ICDE*, Apr. 2015, pp. 149–160.

[6] A. Azad, A. Buluç, and J. Gilbert, "Parallel triangle counting and enumeration using matrix algebra," in *Proc. IPDPS*, May 2015, pp. 804–811.

[7] D. Ediger, J. Riedy, D. A. Bader, and H. Meyerhenke, "Tracking structure of streaming social networks," in *Proc. IPDPS*, May 2011, pp. 1691–1699.

[8] R. Pearce, "Triangle counting for scale-free graphs at scale in distributed memory," in *Proc. HPEC*, Sep. 2017, pp. 1–4.

[9] L. Dhulipala, G. E. Blelloch, and J. Shun, "Theoretically efficient parallel graph algorithms can be fast and scalable," in *Proc. SPAA*, 2018, pp. 393–404.

[10] Y. Hu, H. Liu, and H. H. Huang, "Tricore: Parallel triangle counting on GPUs," in *Proc. SC*, 2018, pp. 171–182.

[11] S. Arifuzzaman, M. Khan, and M. Marathe, "PATRIC: A parallel algorithm for counting triangles in massive networks," in *Proc. SIGKDD*, 2013, pp. 529–538.

[12] S. Arifuzzaman, M. Khan, and M. Marathe, "A space-efficient parallel algorithm for counting exact triangles in massive networks," in *Proc. HPCC*, Aug. 2015, pp. 527–534.

[13] D. A. Patterson, "Latency lags bandwith," *Commun ACM*, vol. 47, no. 10, pp. 71–75, 2004.

[14] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "Powergraph: Distributed graph-parallel computation on natural graphs," in *Proc. OSDI*, 2012, pp. 17–30.

[15] S. Arifuzzaman, M. Khan, and M. Marathe. (2014). "Parallel algorithms for counting triangles in networks with large degrees." [Online]. Available: https://arxiv.org/abs/1406.5687

[16] W. Wang, Y. Gu, Z. Wang, and G. Yu, "Parallel triangle counting over large graphs," in *Proc. DASFAA*, 2013, pp. 301–308.

[17] A. S. Tom *et al.*, "Exploring optimizations on shared-memory platforms for parallel triangle counting algorithms," in *Proc. HPEC*, Sep. 2017, pp. 1–7.

[18] H.-M. Park, S.-H. Myaeng, and U. Kang, "PTE: Enumerating trillion triangles on distributed systems," in *Proc. SIGKDD*, 2016, pp. 1115–1124.

[19] X. Hu, Y. Tao, and C.-W. Chung, "Massive graph triangulation," in *Proc. SIGMOD*, 2013, pp. 325–336.

[20] Y. Cui, D. Xiao, D. B. H. Cline, and D. Loguinov, "Improving I/O complexity of triangle enumeration," in *Proc. ICDM*, Nov. 2017, pp. 61–70.

[21] Y. Cui, D. Xiao, and D. Loguinov, "On efficient external-memory triangle listing," in *Proc. ICDM*, Dec. 2016, pp. 101–110.

[22] H. Das and S. Kumar, "A parallel TSP-based algorithm for balanced graph partitioning," in *Proc. ICPP*, Aug. 2017, pp. 563–570.

[23] D. Margo and M. Seltzer, "A scalable distributed graph partitioner," *Proc. VLDB Endowment*, vol. 8, no. 12, pp. 1478–1489, 2015.

[24] Y. Perez *et al.*, "Ringo: Interactive graph analytics on big-memory machines," in *Proc. SIGMOD*, 2015, pp. 1105–1110.

[25] A. Buluç and K. Madduri, "Parallel breadth-first search on distributed memory systems," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.*, 2011, p. 65.

[26] R. Thakur, R. Rabenseifner, and W. Gropp, "Optimization of collective communication operations in MPICH," *Int. J. High Perform. Comput. Appl.*, vol. 19, no. 1, pp. 49–66, 2005.

[27] T. White, *Hadoop—The Definitive Guide*. Sebastopol, CA, USA: O'Reilly Media, 2015.

[28] J. L. Träff and A. Ripke, "Optimal broadcast for fully connected processor-node networks," *J. Parallel Distrib. Comput.*, vol. 68, no. 7, pp. 887–901, 2008.

[29] A. A. Awan, C.-H. Chu, H. Subramoni, and D. K. Panda. (2017). "Optimized broadcast for deep learning workloads on dense-GPU infiniband clusters: MPI or NCCL?" [Online]. Available: https://arxiv.org/abs/1707.09414

[30] Z. Khayyat, K. Awara, A. Alonazi, H. Jamjoom, D. Williams, and P. Kalnis, "Mizan: A system for dynamic load balancing in large-scale graph processing," in *Proc. EurSys*, 2013, pp. 169–182.

[31] G. Marçais and C. Kingsford, "A fast, lock-free approach for efficient parallel counting of occurrences of k-mers," *Bioinformatics*, vol. 27, no. 6, pp. 764–770, 2011.

[32] D. Chakrabarti, Y. Zhan, and C. Faloutsos, "R-MAT: A recursive model for graph mining," in *Proc. SODA*, 2004, pp. 442–446.

[33] 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 *Proc. SIGMOD*, 2014, pp. 637–648.

[34] M. Snir, S. Otto, S. Huss-Lederman, J. Dongarra, and D. Walker, *MPI: The Complete Reference: The MPI Core*. Cambridge, MA, USA: MIT Press, 1998.

[35] P. Pacheco, *An Introduction to Parallel Programming*. Burlington, MA, USA: Elsevier, 2011.

[36] M. Latapy, "Main-memory triangle computations for very large (sparse (power-law)) graphs," *Theor. Comput. Sci.*, vol. 407, pp. 458–473, Nov. 2008.

[37] Y. Zhu, H. Zhang, L. Qin, and H. Cheng, "Efficient mapreduce algorithms for triangle listing in billion-scale graphs," *Distrib. Parallel Databases*, vol. 35, no. 2, pp. 149–176, 2017.

YONGXUAN ZHANG received the B.E. degree in computer science and technology from Nanchang Hangkong University, China, in 2005. He is currently pursuing the Ph.D. degree in computer science and technology with the Huazhong University of Science and Technology, Wuhan, China. His current research interests include graph processing and parallel/distributed processing.

HONG JIANG (F'14) received the B.E. degree from the Huazhong University of Science and Technology, Wuhan, China, in 1982, the M.A.Sc. degree from the University of Toronto, Canada, in 1987, and the Ph.D. degree from Texas A&M University, College Station, in 1991. He is currently the Wendell H. Nedderman Endowed Professor and Chair of the Department of Computer Science and Engineering, The University of Texas at Arlington. He has over 200 publications in major journals and international conferences in these areas, including the IEEE TPDS, IEEE TC, ACM TOS, ACM TACO, JPDC, ISCA, MICRO, FAST, USENIX ATC, USENIX LISA, SIGMETRICS, MIDDLEWARE, ICDCS, IPDPS, OOPLAS, ECOOP, SC, ICS, HPDC, and ICPP. His research has been supported by the NSF, DOD, and the State of Nebraska. His research interests include computer architecture, computer storage systems, and parallel/distributed computing. He serves as an Associate Editor for the IEEE TRANSACTIONS ON PARALLEL AND DISTRIBUTED SYSTEMS.

FANG WANG received the B.E. and master's degrees in computer science and the Ph.D. degree in computer architecture from the Huazhong University of Science and Technology, China, in 1994, 1997, and 2001, respectively, where she is currently a Professor of computer science and engineering. She has more than 50 publications in major journals and international conferences, including FGCS, ACM TACO, *Science China Information Sciences*, the *Chinese Journal of Computers*, HiPC, ICDCS, HPDC, and ICPP. Her research interests include distributed file systems, parallel I/O storage systems, and graph processing systems.

YU HUA (M'13) received the B.E. and Ph.D. degrees in computer science from Wuhan University, China, in 2001 and 2005, respectively. He is currently a Professor with the Huazhong University of Science and Technology, China. He has more than 80 papers to his credit in major journals and international conferences, including IEEE the TRANSACTIONS ON COMPUTERS, the IEEE TRANSACTIONS ON PARALLEL AND DISTRIBUTED SYSTEMS, USENIX ATC, USENIX FAST, INFOCOM, SC, ICDCS, ICPP, and MASCOTS. His research interests include computer architecture, cloud computing, and network storage. He is a Senior Member of CCF and a member of ACM and USENIX. He has been on the organizing and program committees of multiple international conferences, including INFOCOM, ICDCS, ICPP, RTSS, and IWQoS.

DAN FENG (M'12) received the B.E., M.E., and Ph.D. degrees in computer science and technology from the Huazhong University of Science and Technology, China, in 1991, 1994, and 1997, respectively, where she is currently a Professor and the Vice Dean of the School of Computer Science and Technology. She has more than 100 publications in major journals and international conferences, including the IEEE TC, IEEE TPDS, ACM-TOS, JCST, FAST, USENIX ATC, ICDCS, HPDC, SC, ICS, IPDPS, and ICPP. Her research interests include computer architecture, massive storage systems, and parallel file systems. She is a member of ACM. She serves on the program committees of multiple international conferences, including SC 2011, SC 2013, and MSST 2012.

XIANGHAO XU received the B.E. degree in computer science and technology from Liaoning University, Shenyang, China, in 2015. He is currently pursuing the Ph.D. degree in computer architecture with the Huazhong University of Science and Technology, Wuhan, China. His current research interests include computer architecture and graph processing.