Distributed Algorithms for k-truss Decomposition

Pei-Ling Chen*, Chung-Kuang Chou* and Ming-Syan Chen†

*†Department of Electrical Engineering, National Taiwan University, Taipei, Taiwan
†Research Center of Information Technology Innovation, Academia Sinica, Taipei, Taiwan
{plchen, ckchou}@arbor.ee.ntu.edu.tw, mschen@cc.ee.ntu.edu.tw

***Abstract*—k-truss, a type of cohesive subgraphs of a network, is an important measure for a social network graph. However, with the emergence of large online social networks, the running time of the traditional batch algorithms for k-truss decomposition is usually prohibitively long on such a graph with billions of edges and millions of vertices. Moreover, the size of a graph becomes too large to load into the main memory of a single machine. Currently, cloud computing has become an imperative way to process the big data. Thus, our aim is to design a scalable algorithm of k-truss decomposition in the scenario of cloud computing. In this paper, we first improve the existing distributed k-truss decomposition in the MapReduce framework. We then propose a theoretical basis for k-truss and use it to design an algorithm based on graph-parallel abstractions. Our experiment results show that our method in the graph-parallel abstraction significantly outperforms the methods based on MapReduce in terms of running time and disk usage.**

Keywords*—*Large graph processing, k-truss decomposition, Parallel processing

I. Introduction

Graph measures or methods for describing the characteristic of a vertex or capturing the structure of a network, such as cliques, betweennesses, closenesses, k-cores, k-trusses, and k-plexes, play an important role in network analysis. For example, some measures, e.g., betweennesses, could be used to find relatively important vertices in a large graph, which is valuable for social applications. In viral marketing, the important vertices could be used to facilitate or block a diffusion process [1], [2], such as promoting products or blocking opinions of another party. Other social applications, e.g., solving the problem of group formation for impromptu activities [3], [4], adopt k-plexes to capture the structure property of a social network. Moreover, in bioinformatics, clique detection could be used to identify the topological features in the complex structures of protein networks [5].

Among these measures, the k-truss [6], a type of cohesive subgraphs of a network, draws researchers' attention recently. The k-truss is one of the relaxed versions of clique by weakening the requirement of reachability (the diameter of a clique is 1). k-trusses are similar to k-cores in terms that they both represent the cores of a network at different levels. In fact, the k-truss is conceptually more rigorous than the k-core since the k-truss is defined on the basis of triangles. In addition, Wang and Cheng [7] construct an illustrated example to claim that the k-truss may be more suitable than the k-core in applications, such as visualization of large-scale networks and analysis of network connectivity.

Large online social networks, e.g., Facebook and Twitter, usually have millions of vertices, i.e., users, and billions of edges, i.e., friendships or followerships, and their sizes still keep growing. The running time of the traditional batch algorithms for graph measures would be too long on such *big graphs*, and therefore these algorithms are not suitable for real-time applications. Moreover, such a big graph would be too big to be accommodated by the main memory of a single machine with commodity hardware. Thus, designing algorithms to find graph measures from the aspect of distributed computing on big graphs [8] has become an important direction in the big data era.

MapReduce [9], proposed by Google, is the mainstream model for distributed computing, which is widely used in companies and could be accessed easily from Amazon Elastic MapReduce, Google Cloud Platform, and Microsoft Azure as PaaS in the scenario of cloud computing. Basically, MapReduce is a programming model that abstracts data into key-value pairs used as both input and output of map and reduce functions. However, this abstraction is not suitable for graph algorithms and MapReduce suffers from I/O overhead for iterative algorithms. Thus, Google further proposes Pregel [10], inspired by the Bulk Synchronous Parallel (BSP) model [11], as a new computing model for iterative computations over graph.

In this paper, our aim is to design an efficient and scalable algorithm for k-truss decomposition in the distributed system by embracing the new paradigm aroused by Google for graph computing. For the purpose, we derive a theoretical basis for k-truss decomposition to design a distributed algorithm in the new computing model. For comparison, we also propose an improved algorithm in the MapReduce framework, based on the previous work [8]. Finally, we conduct comprehensive experiments on both synthetic and real datasets and the results show that our proposed distributed algorithm in the new computing model scales up well and much efficient than the version in the MapReduce framework. Our contributions are summarized as follows.

- We derive a theoretical basis in the new computational aspect for k-truss decomposition on information exchange between an edge e and its edge neighbors, i.e., the edges sharing a common vertex with e. The number of iterations of our algorithm is much smaller than the one of the traditional batch algorithm.
- To efficiently run our algorithm in the graph parallel abstraction, we derive a theorem and propose a pruning strategy to reduce the size of the line graph, transforming the original input graph into an edge-centric view, by removing useless edges in the line graph. Thus, the

978-1-4799-5666-1/14/$31.00 ©2014 IEEE

Authorized licensed use limited to: INDIAN INSTITUTE OF TECHNOLOGY DELHI. Downloaded on March 14,2023 at 17:31:35 UTC from IEEE Xplore. Restrictions apply.

amount of communication in a single iteration is decreased, leading to prominent performance improvement by our approach.

The remaining of this paper is organized as follows. We summarize the related work in Section II. We introduce the definition and background of k-truss decomposition in Section III. Our improved distributed k-truss decomposition based on an existing MapReduce version is presented in Section IV. We further provide a theoretical basis and design an efficient distributed algorithm in the new programming model for iterative graph computing in Section V. Section VI shows our experiment results on both synthetic and real datasets. Finally, we conclude in Section VII.

II. Related Work

In this section, we briefly introduce the graph parallel abstraction first and then we review the related works on computing social measures in large graphs.

A. Graph-Parallel Abstraction

The *graph-parallel abstraction* is the new paradigm for graph algorithms since the abstraction of MapReduce is not suitable for graph algorithms and MapReduce suffers from I/O overhead for iterative algorithms. A graph-parallel abstraction comprises a graph and a vertex-program executed in parallel on every vertex in the graph. A vertex-program can interact neighbors of the vertex. Pregel [10] is a well-known abstraction based on the Bulk Synchronous Parallel model in which a vertex-program passes messages to other neighbors in a sequence of supersteps. Barrier synchronization is introduced in the end of a superstep to ensure the synchronization. Both Apache Hama [12] and Apache Giraph[1] are the open source counterparts to Pregel. In contrast to Pregel, GraphLab [13] is an asynchronous graph-parallel abstraction without the barrier synchronization in the end of a superstep. Moreover, PowerGraph [14] is proposed to consider the imbalanced computing problem of partitioning graphs with power-law distributions in GraphLab.

B. Computing Social Measures in Large Graphs

Due to the importance of graph measures and the emergence of large graphs, many researchers have focused on designing distributed or streaming algorithms for them. Cheng et al. [15] address the problem of processing large graphs for cliques with limited memory and proposes an efficient partition-based algorithm for maximal clique enumeration. Fan et al. [16] propose distributed evaluation algorithms for three classes of reachability queries. Chu and Cheng [17] provide an I/O efficient algorithm for triangle listing. In addition, for the k-core, Li et al. [18] deal with the problem of k-core maintenance in large dynamic graphs and Montresor et al. [19] propose a new distributed algorithm for the k-core decomposition. Cheng et al. [20] further propose an algorithm for the k-core decomposition, which is both CPU-efficient and I/O efficient. Moreover, Saríyüce et al. [21] first discuss about the k-core decomposition in the streaming scenario.

[1] http://incubator.apache.org/giraph/

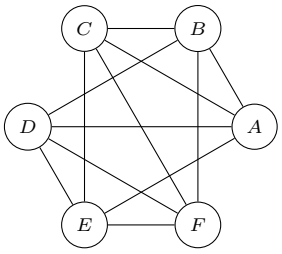


Fig. 1. A 4-truss with the support of each edge equal to 2.

For the k-truss decomposition in large graphs, Wang and Cheng [7] propose two I/O efficient algorithms, which are designed to conquer the problem of limited memory of a single machine. In contrast, our aim is to find the solution as soon as possible via utilizing the cloud computing. In addition, a heuristic distributed k-truss decomposition in the MapReduce framework has been proposed in [8]. Therefore, the algorithm suffers from IO overhead between MapReduce jobs since MapReduce is not designed for iterative algorithms. Instead of using the traditional MapReduce framework, we adopt the most recent graph computing model and provide a rigorous theoretical basis to propose an algorithm of efficient and scalable k-truss decomposition.

III. Preliminaries

In this section, we first introduce the k-truss and then describe the traditional batch algorithm for the k-truss decomposition.

A. k-Truss

A k-truss is a connected subgraph of a graph in which every edge is contained in at least $(k-2)$ triangles [6]. For the graph in Fig. 1, since each edge is contained in 2 triangles, the graph is a k-truss with $k = 2 + 2 = 4$. It is clear to see that the structure of a k-truss is very affinitive. Thus, k-truss is useful to identify smaller but more important areas of a massive network. Before the truss decomposition, we first give necessary definitions as follows.

Let $G = \{V_G, E_G\}$ be a simple graph, where V_G and E_G represent the vertex set and the edge set of G. We sometimes omit subscript G when the corresponding graph is clear in the context. Let $nb(v)$ be the set of neighbors of a vertex v, i.e., $nb(v) = \{u : (u, v) \in E_G\}$. We then define $nb_i(v)$ as the i-th element in $nb(v)$, where $i = 1, 2, 3, ..., |nb(v)|$, and the degree of v in G is denoted by $deg(v) = |nb(v)|$. Thus, the definition of *support* [7] is defined as follows.

Definition 1 (Support)**.** The support of an edge $e = (u, v) \in E_G$, denoted by $sup(e, G)$, is defined as $|nb(u) \cap nb(v)|$. Similarly, when G is clear in the context, we replace $sup(e, G)$ by $sup(e)$.

$|nb(u) \cap nb(v)|$ represents the number of common neighbors of vertices u and v, and further indicates the number of triangles in G that contain the edge $e = (u, v)$. Now, the notion of k-truss [6] is defined as follows.

Definition 2 (k-Truss)**.** A k-truss R_k of G, where $k \geq 2$, is defined as a connected subgraph such that each $sup(e, R_k) \geq k - 2$ for all $e \in R_k$.

Authorized licensed use limited to: INDIAN INSTITUTE OF TECHNOLOGY DELHI. Downloaded on March 14,2023 at 17:31:35 UTC from IEEE Xplore. Restrictions apply.

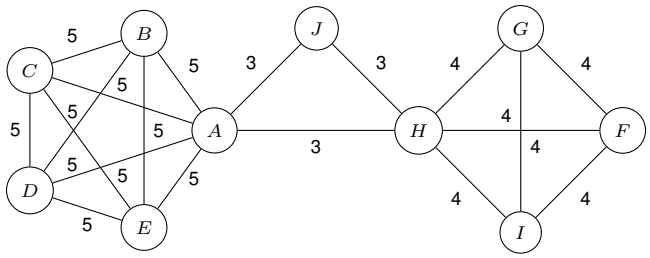


Fig. 2. A graph with trussnesses labeled on its edges.

We further define a T_k as the subgraph of the union of all k-trusses, that is, $T_k = \bigcup_i R_k^i$, where R_k^i is the i-th k-truss in G.

Definition 3 (Trussness)**.** The trussness of an edge e in G, denoted by $\phi(e) = k$, is the maximal k such that e is contained in R_k.

Now, we show how to find T_k in Fig. 2, where trussnesses are labeled on edges. To find T_4, each edge e with $\phi(e) \geq 4$ is selected, i.e., $E_{T_4} = \{AB, AC, AD, AE, BC, BD, BE, CD, CE, DE, GH, GI, GF, HI, IF, HF\}$. T_5 with $E_{T_5} = \{AB, AC, AD, AE, BC, BD, BE, CD, CE, DE\}$ can be obtained. Both T_3 and T_2 are equal to the whole graph. Note that to find k-trusses, an additional depth-first search algorithm could be applied to separating T_k into connected subgraphs. For example, T_4 can be separated into two maximal R_4s: $R_4^1 = \{AB, AC, AD, AE, BC, BD, BE, CD, CE, DE\}$ and $R_4^2 = \{GH, GI, GF, HI, IF, HF\}$. A maximal k-truss mentioned above is a k-truss that is not a proper subgraph of another k-truss. Since the minimal 4-truss contains 4 vertices, there are total $\binom{5}{4} + \binom{5}{5} + \binom{4}{4} = 7$ R_4s, where $\binom{5}{4} + \binom{5}{5}$ are subgraphs of R_4^1 and $\binom{4}{4}$ is R_4^2.

With the definitions above, given a graph G, the truss decomposition in G is to find all R_ks for $2 \leq k \leq k_{max}$. This problem is equivalent to obtaining the trussness of each edge because T_k can be simply obtained by unionizing each edge e with $\phi(e) \geq k$, and for separating each T_k into R_k's, a depth-first search can be applied trivially as mentioned above. Therefore, we address the problem of truss decomposition as *finding the trussness $\phi(e)$ of each edge e*[2].

B. k-Truss Decomposition

The traditional batch k-truss decomposition algorithm is proposed in [6]. This algorithm runs iteratively and each iteration contains three main phases: *1*) Count the support $\mathrm{sup}(e)$ of each edge $e \in E_G$; *2*) Remove edges with the smallest support; *3*) Compute the trussness $\phi(e)$, the smallest support sup_{min} plus two, of each removed edge $e \in E$ in the iteration. Fig. 3 is an illustrated example of the algorithm. Supports of each edges are computed first in Fig. 3a. Because of $sup_{min} = 1$ in this iteration, CD and DE are removed in Fig. 3b, and then $sup(CE)$ is decreased to 2. The next iteration starts in Fig. 3c with $sup_{min} = 2$. Since there are no edges with supports larger than 2, all edges are deleted in this iteration and the algorithm terminates. The trussness $\phi(e)$ of each edge $e \in E_G$ is computed as $\phi(e) = sup_{min}^i + 2$ where sup_{min}^i is the smallest support of the corresponding iteration i in which the edge is removed. The result, i.e., the trussness $\phi(e)$ of each edge e, is shown as Fig. 3d. Then, it is clear to see that the graph contains T_2 and T_3 equal to the whole graph, and $T_4 = \{AB, AC, AE, BC, BE, CE\}$. Thus, R_ks can be obtained.

The time complexity of the algorithm requires $O(\sum_{v \in V_G}(deg(v))^2)$ [7]. The running time will be long if there are many vertices of high degree. Moreover, when the scale of data becomes larger, the algorithm is impractical due to the limited memory of a single machine.

IV. Distributed k-Truss Decomposition in MapReduce Framework

In this section, we introduce how to design a MapReduce algorithm for the k-truss decomposition. We first discuss a basic version of the decomposition in MapReduce framework proposed in [8]. We then propose an improved version based on the basic algorithm. Our improved algorithm is advantageous in terms of IO operations and running time.

A. Basic Version

A basic algorithm for distributed k-truss decomposition in the MapReduce framework is proposed in [8]. We use MRTruss to represent the algorithm afterwards for the ease to mention it. Because MRTruss follows the same procedure in the traditional batch algorithm, the number of iterations to terminate for MRTruss is the same. Before introducing the details of MRTruss, we first provide two necessary definitions.

Definition 4 (Edge Neighbors)**.** For a simple graph $G = (V_G, E_G)$, the neighbors of an edge $e = (u, v)$ are defined as $enb(e) = \{d = (s, t) : |\{s, t\} \bigcap \{u, v\}| = 1, d \in E_G\}$, that is, d shares a common vertex with e. We further define $enb_i(e)$ as the i-th element in $enb(e)$ where $i = 1, 2, 3, ..., |enb(e)|$.

Definition 5 (Edge Triangle Relationship)**.** The edge triangle relationship of an edge e is a set of edges $\mathcal{R}(e) = \{r(e) : r(e) \in enb(e), r(e)$ forms a triangle with e and another $r'(e) \in enb(e)\}$. We further define $r_i(e)$ as the i-th element in $\mathcal{R}(e)$ where $i = 1, 2, 3, ..., |\mathcal{R}(e)|$.

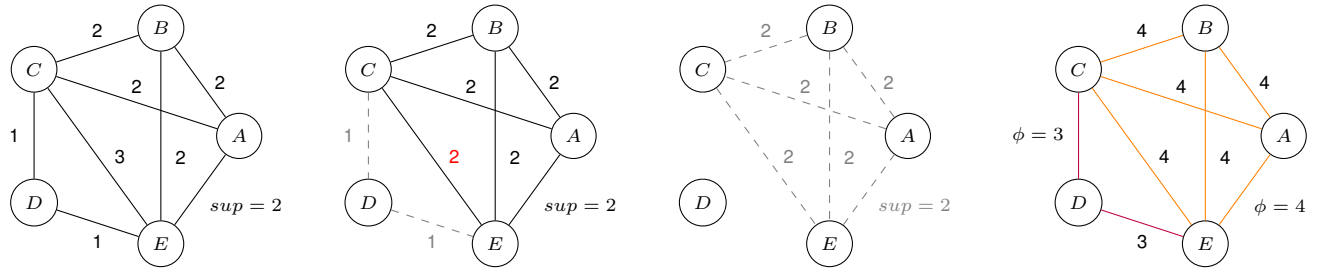
With the above definitions, there are three tasks in each iteration of MRTruss: *1*) For each pair of edges with a common vertex, i.e., an *open triad*, generate a record with a triad as a value and a potential *closure* which is an edge closing this triad as a triangle and the existing of such an edge is unknown in this task, as a key; *2*) Check whether a closure specified in a key exists or not, and output existing triangles; *3*) Count $sup(e)$ for each edge e, and delete edges with the smallest support. Note that each task represents a MapReduce job in MRTruss and the edge triangle relationship $\mathcal{R}(e)$ of each edge e is always recalculated in each iteration, which generates many redundant jobs, incurring many disk I/O operations. We therefore propose an improved version next.

B. Improved Version

The main issue in MRTruss is that edge triangle relationships for the input graph are not preserved in each iteration. We

[2]A similar transformation of the decomposition problem in the graph is k-core decomposition [19].

Authorized licensed use limited to: INDIAN INSTITUTE OF TECHNOLOGY DELHI. Downloaded on March 14,2023 at 17:31:35 UTC from IEEE Xplore. Restrictions apply.



(a) Graph $G = (V, E)$ with $sup(e)$ for $e \in E$

(b) Edge removal with the lowest support

(c) Edge removal with the lowest support (again)

(d) The resulted graph with trussness $\phi(e)$ for $e \in E$

Fig. 3. An example of k-truss decomposition.

Algorithm I. I-MRTRUSS

Input: $G = (V, E)$
Output: Records with $(e, \phi(e))$
1: run Procedure I: Triangle Finding
2: $c \leftarrow 2$
3: **repeat**
4: run Procedure II: Trussness Counting
5: **until** no potential trussnesses of edges are changed
6: **if** $\exists e \in E$, $truss(e) > c$ **then**
7: $c \leftarrow (c + 1)$, **goto** Step 4

Procedure I. TRIANGLE FINDING

Input: The set of each vertex $v \in V$ and each $nb_i(v) \in nb(v)$: $I_1 = \{v, nb_1(v), nb_2(v), ..., nb_{|nb(v)|}(v)\}$.
Output: The set of each edge $e \in E$, the potential trussness $truss(e)$, and each edge $r_i(e) \in \mathcal{R}(e)$: $O_1 = \{e, truss(e), r_1(e), r_2(e), ..., r_{|\mathcal{R}(e)|}(e)\}$.
Map(Records of Type I_1):
1: emit $k = v$ and $val = (nb_1(v), ..., nb_{|nb(v)|}(v), v)$;
2: **for** each $nb_i(v)$ **do**
3: emit $k = nb_i(v)$ and $val = (nb_1(v), ..., nb_{|nb(v)|}(v), v)$;
Reduce(Key $k = v_m$, Value $val[1, ..., r]$):
4: $H \leftarrow$ new table
5: **for** each $val \in val[1, ..., r]$ **do**
6: $L_u \leftarrow (nb_1(u), ..., nb_n(u))$ for each u in the last cell of val;
7: insert (u, L_u) to H;
8: **for** each element (u, L_u) in H **do**
9: $B \leftarrow \{b : b \in L_u \cap L_{v_m}\}$ and b_i is the i-th element in B where $i = 1, 2, 3, ..., |B|$;
10: Output$(e, truss(e), r_1(e), r_2(e), ..., r_{|\mathcal{R}(e)|}(e)\}$ where $e = (v_m, u)$, $truss(e) = |B|+2$, and $r_i(e) = (v_m, b_i) \in \mathcal{R}(e)$;

Procedure II. TRUSSNESS COUNTING

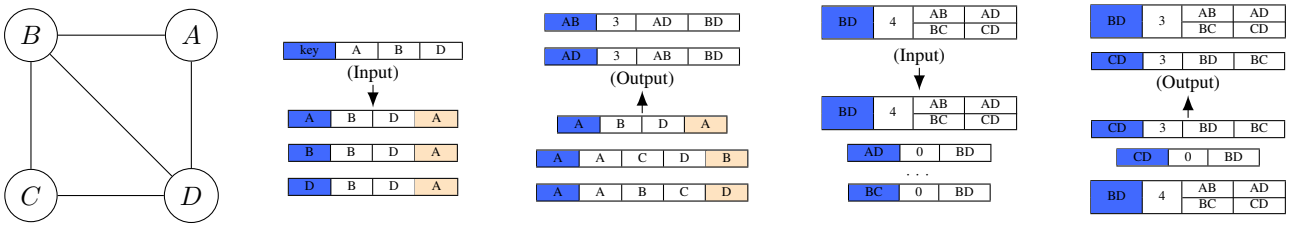
Input: The set of each edge $e \in E$, the potential trussness $truss(e)$, and each edge $r_i(e) \in \mathcal{R}(e)$: $I_2 = \{e, truss(e), r_1(e), r_2(e), ..., r_{|\mathcal{R}(e)|}(e)\}$. The value c passed by configuration, which is used to indicate the condition on trussness.
Output: The output format is the same as the input of type I_2.
Map(Records of Type I_2):
1: emit $k = e$ and $val = (truss(e), r_1(e), r_2(e), ..., r_{|\mathcal{R}(e)|}(e))$;
2: **if** $truss(e) \geq c$ **then**
3: **for** each $r_i(e) \in \mathcal{R}(e)$ **do**
4: emit $k = r_i(e)$ and $val = (0, e)$;
Reduce(Key $k = e_m$, Value $val[1, ..., r]$):
5: $L \leftarrow$ new list, $L_o \leftarrow$ new list;
6: **for** each $val \in val[1, ..., r]$ **do**
7: **if** the first cell in val is $truss(e_m)$ **then**
8: $L \leftarrow (r_1(e_m), ..., r_{|\mathcal{R}(e_m)|}(e_m))$ and $S \leftarrow truss(e_m)$;
9: **else if** the first cell in val is 0 **then**
10: $L_o \leftarrow (e')$ which is in the second cell of val;
11: **if** $S < c$ **then**
12: Output$(e_m, truss(e_m), r_1(e_m), ..., r_{|\mathcal{R}(e_m)|}(e_m))$ where $r_i(e_m) \in \mathcal{R}(e_m)$ is recorded in L and $truss(e_m) = S$;
13: **else**
14: $B \leftarrow \{b : b \in L \cap L_o$ and b forms triangles with e_m and another $b' \in L \cap L_o\}$;
15: **if** $|B|/2 + 2 < c$ **then**
16: Output$(e_m, truss(e_m), r_1(e_m), ..., r_{\mathcal{R}(e_m)}(e_m))$ where $truss(e_m) = \phi(e_m) = c - 1$;
17: **else**
18: Output$(e_m, truss(e_m), r_1(e_m), ..., r_{\mathcal{R}(e_m)}(e_m))$ where $truss(e_m) = |B|/2 + 2$;

propose an improved version called i-MRTruss as Algorithm I composed of two main procedures: Triangle Finding and Trussness Counting. The pseudo codes of these 2 procedures are given in Procedure I and Procedure II. The first procedure is called only in the first job to transform the whole graph into an *edge-centric* one. That is, each record emitted by Triangle Finding is an edge e with its potential trussness $truss(e)$, and the edge triangle relationship $\mathcal{R}(e)$. Note that $truss(e)$ may not equal $\phi(e)$. In each output record of Triangle Finding, $truss(e)$ is set to $sup(e) + 2$ for each e. The second procedure is called in each iteration and will maintain the same information as the one passed by Triangle Finding and the potential trussness $truss(e)$ of each e will be finally updated to its real trussness $\phi(e)$.

Consider the graph G in Fig. 4a as an example. The map phase in Triangle Finding generates a *neighbor list* for the vertex v specified in each input record $I_1 = \{v, nb_1(v), nb_2(v), \cdots, nb_{|nb(v)|}(v)\}$. A vertex u in the bisque cell (Fig. 4b) is added at the end of a neighbor list to indicate the owner of this list, named the *indicator* afterwards (Fig. 4b: A mapper generates a neighbor list of A and also passes it to A's neighbors, B and D). Therefore, in the reduce phase, a reducer will get not only the neighbor list of the main vertex v_m specified in its key, but also a collection of neighbor lists of $nb(v_m)$. Since each indicator u in the bisque cell indicates the owner of each list, a reducer will output the edge $e = (v_m, u)$, the potential trussness $truss(e) = sup(e) + 2$, and edge triangle relationship $\mathcal{R}(e)$, where the support $sup(e)$ is the number of the common neighbors of v_m and u, and $\mathcal{R}(e)$ can also be retrieved from the common neighbors of v_m and u. Finally, each output record of the reduce phase has the form $O_1 = \{e, truss(e), r_1(e), r_2(e), ..., r_{|\mathcal{R}(e)|}(e)\}$ where each edge $r_i(e) \in \mathcal{R}(e)$ (Fig. 4c: A reducer compares A's neighbor list to B's and D's to find common neighbors respectively, and finally generates edges AB and AD with potential trussnesses and edge triangle relationships).

The map phase in Trussness Counting takes the edge-centric result of Triangle Finding as the input and does the following actions. A trussness threshold c is specified by i-MRTruss in each iteration. Every record will be emitted as the key-value pair $KV_1 = \langle key = e, value = (truss(e), r_1(e), r_2(e), \cdots, r_{|\mathcal{R}(e)|}(e))\rangle$ in the map phase. In addition, for each

Authorized licensed use limited to: INDIAN INSTITUTE OF TECHNOLOGY DELHI. Downloaded on March 14,2023 at 17:31:35 UTC from IEEE Xplore. Restrictions apply.



(a) Graph $G = (V, E)$ (b) Map phase for the first record (A, B, D) (c) Reduce phase for the reducer having $key = A$ (d) Map phase for the record with the edge BD in the iteration of $c = 4$ (e) Reduce phase for reducers with $key = CD$ and $key = BD$

Fig. 4. An example for i-MRTruss. (a): the input graph G. (b)(c): two phases in Triangle Finding. (d)(e): two phases in Trussness Counting.

$r_i(e)$ in the record with potential trussness $truss(e) \geq c$, a new key-value pair is emitted as the $KV_2 = \langle key = r_i(e), value = (0, e)\rangle$, which means e wants to tell $r_i(e)$ that it still exists in the graph in this iteration, i.e., it's not removed in previous iterations (Fig. 4d: Since $truss(BD) = 4 \geq 4$, the mapper emits not only a pair of KV_1 but also pairs of KV_2 for BD's neighbors: AB, AD, BC, and CD). Therefore, in the reduce phase, a reducer will get a collection of values corresponding to the main edge e_m specified in its key. Note that if the first cell of a value in a key-value pair is 0, the type of this pair is detected as KV_2 because $truss(e)$ is always larger than 0. Thus, $\mathcal{R}(e_m)$ and $truss(e_m)$ can be retrieved from KV_1. If $truss(e_m) < c$, no comparison is needed since $truss(e_m) = \phi(e_m)$ already. However, if $truss(e_m) \geq c$, based on pairs of type KV_2, the presence of edges in $\mathcal{R}(e_m)$ can be verified, and the potential trussness $truss(e_m)$ is updated. Then, if this new potential trussness does not satisfy the trussness threshold c, it will be modified to be $c - 1$ as its final trussness $\phi(e_m)$ (Fig. 4e: Since $truss(CD) = 3 < 4$, therefore $truss(CD) = \phi(CD)$, and then no comparison is needed; $truss(BD) = 4 \geq 4$, but since no KV_2 pairs with $key = BD$ are received, $truss(BD)$ is then updated based on the above description). The format of the final output in the reduce phase is the same as O_1.

The time complexity of MRTruss or i-MRTruss is the same as that of the traditional batch k-truss decomposition. However, because of the preservation of edge triangle relationships, i-MRTruss is much more IO-efficient than MRTruss, and thus the running time of i-MRTruss is much better.

V. Distributed k-Truss Decomposition based on Bulk Synchronous Parallel Model

The performance of our improved algorithm, i-MRTruss, is still intrinsically bounded by the framework of MapReduce since there are too many intermediate results between jobs, incurring unwanted disk I/O operations. Therefore, we further propose an algorithm based on the graph-parallel abstraction in which a vertex-centric model is used to give an intuitive computational environment for graph processing. Furthermore, the graph-parallel abstraction will not have lots of IO overhead, which is the main performance issue in MapReduce for graph computing.

Moreover, we want to solve the k-truss decomposition in a brand new aspect: the trussness $\phi(e)$ of an edge $e \in E_G$ can be decided by the trussnesses of a subset of edges in the graph G. This idea provides a new computation logic different from the traditional batch algorithms. In this section, we first derive a theorem to prove the locality property in k-truss to decide the enough range of the subset of edges in a graph for computing the trussness $\phi(e)$ of an edge e. Finally, we propose an algorithm named GPTruss based on the theorem.

A. Locality of k-truss

The locality of k-truss shows that the trussness $\phi(e)$ of an edge e is only related to the trussnesses of edges in $enb(e)$. Therefore, to design an algorithm under graph-parallel abstractions for computing the trussness $\phi(e)$ of each edge e, we can use this property to ensure a small range of message transmit of an edge. With this property, each edge only needs to communicate to edges nearby instead of edges in the whole graph, and therefore the amount of redundant long-term communication among edges can be decreased. Moreover, the amount of communication over different partitions located on different machines can be reduced in the distributed system. Before introducing the theorem of locality, we first define the induced graph.

Definition 6 (Induced Graph)**.** $G(C) = (V|C, C)$ is a subgraph of G induced by a subset of edges C, where $V|C = \{u, v \in V : (u, v) \in C\}$.

The theorem of locality is derived as follows[3].

Theorem 1 (Locality)**.** $\forall e \in E_G$*:* $\phi(e) = k$ *if and only if*

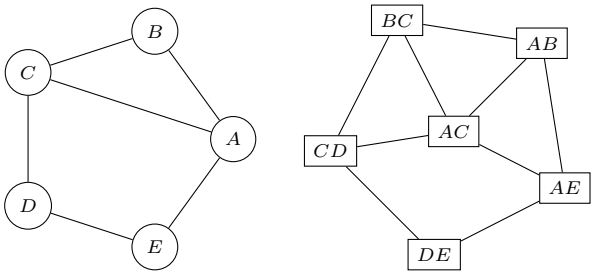
1) *there exists a subset* $E_k \subseteq enb(e)$ *such that* $|E_k| = 2(k-2)$*, edges in* E_k *forms total* $(k-2)$ *triangles with* e*, and for each edge* $e' \in E_k$*,* $\phi(e') \geq k$*;*
2) *there is no subset* $E_{k+1} \subseteq enb(e)$ *such that* $|E_{k+1}| = 2(k-1)$*, edges in* E_{k+1} *forms total* $k-1$ *triangles with* e*, and for each edge* $e' \in E_{k+1}$*,* $\phi(e') \geq k+1$*.*

Proof:

$\Rightarrow$ By the definition of trussness, $\phi(e) = k, W_k \subseteq E$ exists such that $e \in W_k$ and $G(W_k)$ is a k-truss, and there is no set $W_{k+1} \subseteq E$ such that $e \in W_{k+1}$ and $G(W_{k+1})$ is a $(k+1)$-truss.
The part 1 follows since $\phi(e) = k$ implies $sup(e) \geq k-2$, which means at least $k-2$ pairs

[3]We prove the theorem in a similar approach to proving the locality of k-core [19].

Authorized licensed use limited to: INDIAN INSTITUTE OF TECHNOLOGY DELHI. Downloaded on March 14,2023 at 17:31:35 UTC from IEEE Xplore. Restrictions apply.



(a) Graph $G = (V, E)$ (b) The line graph $L(G) = (V_{L(G)}, E_{L(G)})$ of G

Fig. 5. An example of line graph transformation.

of edges form a triangle with e. Therefore at least $2(k-2)$ neighbors of e form triangles with it belong to k-truss $G(W_k)$.

The part 2 follows by contradiction: assume that $l_1, \ldots, l_{2(k-1)}$ are $2(k-1)$ edge neighbors of e with trussness $k+1$ or more, and they form total $k-1$ triangles with e. For $i = 1, \ldots, 2(k-1)$, denote $W_i \subseteq E$ such that $l_i \in W_i$ and $G(W_i)$ is a $(k+1)$-truss. Consider the set $U = \{\{e\} \cup \bigcup_{i=1}^{2(k-1)} W_i\}$, if $l \in U$, then indeed $sup(l, G(U)) \geq k-1$: if $e = l$, obviously $sup(e, G(U)) \geq k-1$, whereas if $e \neq l \in W_i$, $sup(e, G(U)) \geq sup(l, G(W_i)) \geq k-1$. Hence, a $(k+1)$-truss exists and contains $G(U)$ ($G(U)$ may not be maximal), so it is the $(k+1)$-truss for e, which leads to a contradiction.

$\Leftarrow$ For each edge $l_i \in E_k$ ($1 \leq i \leq 2(k-2)$, $\phi(l_i) \geq k$), there exists $W_i \subseteq E$ such that $G(W_i)$ is a k-truss and $l_i \in W_i$. Consider the set $U = \{\{e\} \cup \bigcup_{i=1}^{2(k-2)} W_i\}$. With the same argument used above, we see that for each $l \in U$, $sup(l, G(U)) \geq k-2$: if $e = l$, obviously $sup(e, G(U)) \geq k-2$, whereas if $e \neq l \in W_i$, $sup(e, G(U)) \geq sup(l, G(W_i)) \geq k-2$. Again, this proves that $\phi(e) \geq k$.

Suppose now that $\phi(e) = k' \geq k+1$, by contradiction. This means that there is a subset $W \subseteq E$ such that $G(W)$ is a k'-truss containing e, i.e., e has at least $2(k'-2) \geq 2(k-1)$ neighbors in $G(W)$, and therefore each of them with trussness $k' \geq k+1$, which contradicts the hypothesis 2. Thus, $\phi(e) = k$.

■

Since the existing graph-parallel abstractions are vertex-centric, while the local property of k-truss is edge-centric, we need a line graph transformation for the practical implementation.

Definition 7 (Line Graph)**.** Given a simple graph $G = (V_G, E_G)$, its line graph $L(G) = (V_{L(G)}, E_{L(G)})$ is a graph where each vertex $v' \in V_{L(G)}$ represents an $e \in E_G$ ($1-1$ correspondence), and two vertices in $V_{L(G)}$ are adjacent if and only if their corresponding edges in E_G share a common endpoint.

Fig. 5 shows an example line graph. However, it is clear to see that, compared to the original graph, the line graph transformation introduces more vertices and edges, which leads to heavy memory usage and high computational time. We further propose a pruning method described in Definition 8 to help reduce useless edges of $E_{L(G)}$ in a line graph.

Algorithm II. GPTRUSS

Input: $G = (V_G, E_G)$
Output: $T = \{\phi(e) : e \in E_G\}$
1: S, T, and $PL(G) \leftarrow \emptyset$;
▷ S is expected to contain $truss(e) = sup(e) + 2, \forall e \in E_G$
2: $\{PL(G), S\} \leftarrow$**Triangle Finding(**G**)**;
3: $T \leftarrow$**Trussness-Parallel Computing(**$PL(G)$**,** S**)**;
4: **return** T;

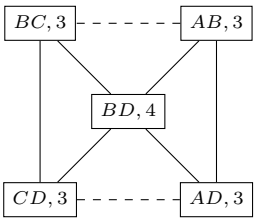


Fig. 6. The pruned line graph $PL(G)$ constructed by Trussness-Parallel Computing based on the output of Triangle Finding on graph G in Fig. 4a (The dashed line represents edges pruned from the original line graph)

Definition 8 (Pruned Line Graph)**.** Given a simple graph $G = (V_G, E_G)$ and its line graph $L(G) = (V_{L(G)}, E_{L(G)})$, the pruned line graph of G is $PL(G) = (V_{PL(G)}, E_{PL(G)})$ where $V_{PL(G)}$ is the same as the $V_{L(G)}$, but $E_{PL(G)}$ is reduced by the constraint: two vertices in $V_{PL(G)}$ are adjacent if and only if their corresponding edges $e_1, e_2 \in E_G$ forming a triangle with another edge $e \in E_G$.

Definition 8 directly follows Theorem 1 and it points out that not all connections between an edge $e \in E_G$ and its $enb(e)$ needs to be constructed. Therefore, it can reduce the useless edges in a line graph, which decreases the number of messages sent in a single round.

B. Algorithm in Graph Parallel Abstraction

With the theoretical basis, we present the GPTruss, which is shown in Algorithm II, with two main procedures: Triangle Finding and Trussness-Parallel Computing. The pseudo codes of these 2 procedures are given in Procedure III and Procedure IV. The first procedure is reused from i-MRTruss to find the pruned line graph based on Definition 8. The main procedure in GPTruss is Trussness-Parallel Computing, which is designed in the graph-parallel abstraction. Therefore, a pruned line graph will be constructed by Trussness-Parallel Computing based on the output records from Triangle Finding. Each vertex-program located on the vertex of a pruned line graph will communicate with each other and updates its local value according to Theorem 1. The program will terminate within a few supersteps when no message sent or all vertex-programs turn to halt. We design Trussness-Parallel Computing in the abstraction of Pregel. We next give a running example of GPTruss.

Fig. 6 shows a pruned line graph $PL(G)$ based on the output of Triangle Finding which input is the graph G in Fig. 4a.

Authorized licensed use limited to: INDIAN INSTITUTE OF TECHNOLOGY DELHI. Downloaded on March 14,2023 at 17:31:35 UTC from IEEE Xplore. Restrictions apply.

Procedure III. TRUSSNESS-PARALLEL COMPUTING

Input: $PL(G) = (V_{PL(G)}, E_{PL(G)})$, $S = \{truss(e) = sup(e) + 2 : e \in E_G = V_{PL(G)}\}$ where $G = (V_G, E_G)$
Output: $T = \{\phi(e) : e \in E_G\}$
▷ For easier reading, though this procedure takes the pruned line graph $PL(G)$ as its input in reality, we still use the edges in original graph G to present this method.
Class Edge:
1: Edge index : $\{u, v\}$ ▷ $e = (u, v)$
2: Edge value : $truss \leftarrow sup(e) + 2$, $M \leftarrow$ new tablet
3: **function** COMPUTE
4: **if** the current superstep is equal to $= 0$ **then**
5: sendMesageToAllNeighbors(u, v, $truss$);
6: **if** the current superstep is greater than 0 **then**
7: **for** each received message (x, y, tr) **do**
8: $un \leftarrow \text{unCommonNode}_{u,v}(x, y)$;
9: $co \leftarrow \text{CommonNode}_{u,v}(x, y)$;
10: **if** un is not in M or co does not exist in $M(un)$ **then**
11: insert (co, tr) to $M(un)$;
12: **else if** (co, tr') in $M(un)$ and $tr' > tr$ **then**
13: replace (co, tr) to (co, tr') in $M(un)$;
14: $newtruss \leftarrow$ **LocalityCount**(M, $truss$);
15: **if** $newtruss < truss$ **then**
16: $truss \leftarrow newtruss$;
17: sendMesageToAllNeighbors(u, v, $truss$);
18: **else**
19: VoteToHalt();

Procedure IV. LOCALITY COUNTING

Input: a table M and an integer $truss$
Output: an integer k
1: **for** i from 1 to $truss$ **do** $count[i] \leftarrow 0$;
2: **for** each element un in M **do**
3: **if** $M(un).size = 2$ **then**
4: $j \leftarrow min(truss, tr1, tr2)$
where elements $(co1, tr1)$ and $(co2, tr2)$ are in $M(un)$;
5: $count[j] \leftarrow count[j] + 1$;
6: **for** i from $truss$ down to 3 **do**
7: $count[i-1] \leftarrow count[i-1] + count[i]$;
8: $k \leftarrow truss$;
9: **while** $k > 2$ and $count[k] < k - 2$ **do**
10: $k \leftarrow k - 1$;
return k;

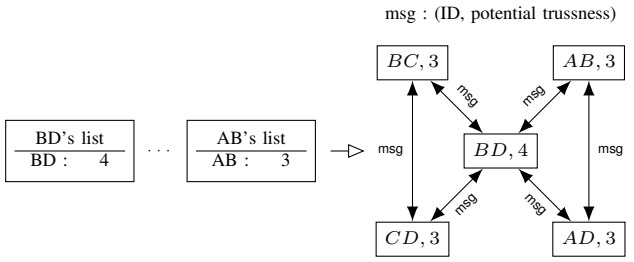


Fig. 7. The first superstep in Trussness-Parallel Computing with $PL(G)$ in Fig. 6.

Every vertex in $PL(G)$ represents an edge e in the original graph G and associates with a potential trussness $truss(e)$, initialized to $sup(e)+2$, and every edge in $PL(G)$ reflects the edge triangle relationship of each $e \in E_G$. However, for the ease of reading, we explain the actions of Trussness-Parallel Computing based on the edges in the original graph G, instead of the vertices in the pruned line graph $PL(G)$.

At the first superstep shown in Fig. 7, each vertex-program only knows its potential trussness in the beginning, which is maintained in a data structure called *list*, and then, each program sends message of its potential trussness to neighbors' programs.

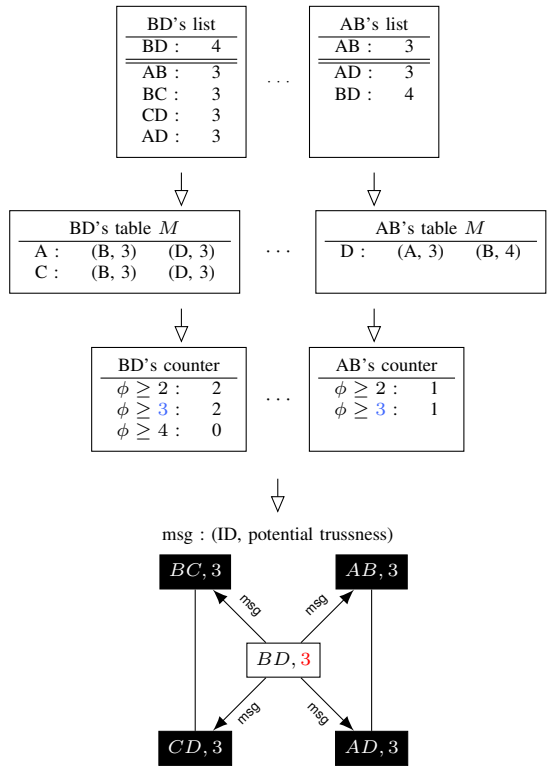


Fig. 8. The second superstep in Trussness-Parallel Computing with $PL(G)$ in Fig. 6.

Fig. 8 shows the second superstep in Trussness-Parallel Computing. The lists maintained by vertex-programs corresponding to edges AB, BC, CD and $AD \in E_G$ are updated based on the received messages sent from the previous superstep. Based on the lists, each program records the edge triangle relationship $\mathcal{R}(e)$ of its corresponding edge $e \in E_G$ in another data structure called *table*. For example, for the program corresponding to AB, in AB's table M, the entry "$M(D) : (A, 3), (B, 4)$" means that the edges AD and BD form one triangle with AB. Then, in AB's *counter*, which is implemented by an array to keep the result based on Theorem 1, "$\phi \geq 2 : 1$" means that there is one pair of edges, forming one triangle with AB, with trussness greater than 2. The blue value in each vertex-program's counter indicates the new estimated potential trussness (the value may be the same as the previous value), whereas the potential trussness value (colored in red) in the pruned line graph is different from the previous one. Since other vertex-programs, except the one of BD, do not have a different potential trussness after computation, they volt to halt (black vertices). Only the vertex-program of BD sends its new potential trussness to its neighbors.

Since not all programs halt, the third superstep is needed. In Fig. 9, each vertex-program again updates its list based on the received messages, and then updates its table. Because the vertex-program of BD receives nothing now, its table remains unchanged. Other programs receive only BD's message. Thus, for example, the program of AB updates its table by changing $(B, 4)$ to $(B, 3)$. However, the new estimated potential trussness (the blue value) still remains unchanged. Finally, no

Authorized licensed use limited to: INDIAN INSTITUTE OF TECHNOLOGY DELHI. Downloaded on March 14,2023 at 17:31:35 UTC from IEEE Xplore. Restrictions apply.

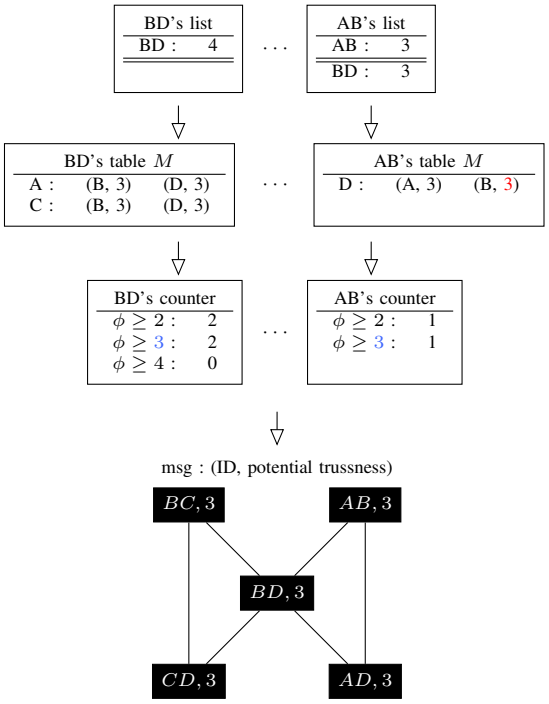


Fig. 9. The third superstep in Trussness-Parallel Computing with $PL(G)$ in Fig. 6.

programs update the potential trussnesses, and they all volt to halt (black vertices). Therefore, the procedure is complete with the local variable of each vertex-program equal to $\phi(e)$ of the corresponding edge $e \in E_G$.

VI. Experimental Analysis

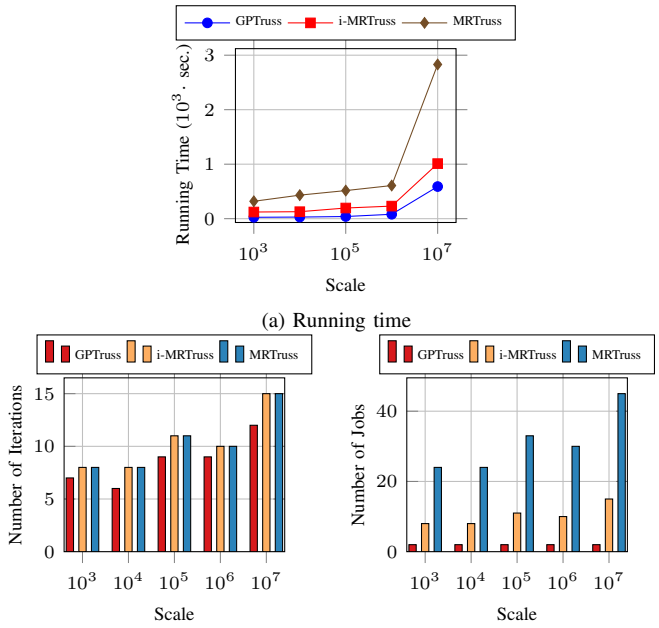
This section reports the experimental results with both synthetic datasets and real world datasets. All graphs from these datasets are preprocessed to be a simple graph without self-loops and multiple edges. The experimental analysis will cover the 5 aspects: running time, the number of required iterations, memory usage, disk usage, and the number of required jobs. We implement MRTruss, i-MRTruss and GPTruss by hadoop-1.2.1 and hama-0.6.3 in a cloud environment consisting of 4 machines. One of the machines is served as both master and slave, and the other 3 machines are solely slaves. All the experiments are performed on machines with 2.5GHz Intel Xeon CPU. The memory usage and the required core number in each machine does not exceed 105GB and 7 cores.

A. Experiments on Synthetic Data

We use the Kronecker graph generation model [22], which is designed to be naturally obeyed the several properties in real network graphs, to generate five synthetic graph datasets in different scales. These five datasets have the same Kronecker matrix setting: $\{0.999\ 0.327; 0.348\ 0.391\}$, which ensures the similar attributes among them. The statistics of five synthetic datasets are summarized in Table I. The meaning of each row of the table is described as follows: the scale corresponds to the number of vertices in each graph , and i is the iteration of

TABLE I. Statistics of Synthetic Graph Datasets

Scale	10^3	10^4	10^5	10^6	10^7
i	10	14	18	20	24
$\|V\|$	1024	16384	262144	1048576	16777216
$\|E\|$	1368	25514	465679	1986937	36146725
d_{avg}	2.6718	3.1145	3.5528	3.7898	4.3087
sup_{max}	7	6	11	13	29



(a) Running time

(b) Total number of iterations to convergence.

(c) Total number of required jobs.

Fig. 10. Efficiency of MRTruss, i-MRTruss and GPTruss over the 5 synthetic datasets.

Kronecker matrix multiplication; $|V|$ and $|E|$ is the number of vertices and the number of edges; d_{avg} and sup_{max} represent the average degree and the maximal support. In each dataset, the range of k is controlled to be from $k = 2$ to $k = 4$.

Through these synthetic datasets, Fig. 10 shows the performance of GPTruss, i-MRTruss and MRTruss with the common setting: 4 reducers and 4 BSP peers. Note that the number of mappers in MapReduce is decided by the block size and the input file size. The default block size 64MB is used in our experiments, and the number of mappers is corresponding to the size of each dataset, instead of the same value. This setting is reasonable since tasks in mappers are often simple routines with low complexity, and then the number of mappers scaling up with input file size does not affect the experiments. Since the complexity of our algorithms is dominated by reducers, we only fix the number of reducers to be the same value to show the scalability of our algorithms.

In Fig. 10a, the running time of our algorithms GPTruss and i-MRTruss is much faster than that of MRTruss, especially when the scale of the dataset grows up to 10^7. In Fig. 10b, since the trussness range of k for these five datasets is fixed to 3, which is small, there is only a small difference in the required number of iterations among GPTruss, i-MRTruss and MRTruss. Note that both of i-MRTruss and MRTruss follow

Authorized licensed use limited to: INDIAN INSTITUTE OF TECHNOLOGY DELHI. Downloaded on March 14,2023 at 17:31:35 UTC from IEEE Xplore. Restrictions apply.

TABLE II. STATISTICS OF REAL WORLD NETWORK DATASETS

Name	com-Youtube	loc-Gowalla	roadNet-TX	com-DBLP
$\|V\|$	1134890	196591	1379917	317080
$\|E\|$	2987624	950327	1921660	1049866
d_{avg}	5.265	9.668	2.785	6.622
sup_{max}	4034	1297	3	312
k_{max}	19	29	4	114

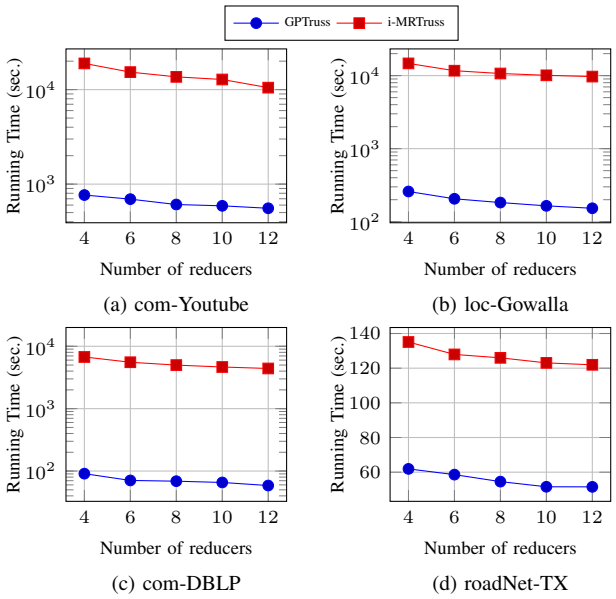


Fig. 11. The running time of i-MRTruss and GPTruss over the 4 real datasets.

the same concept of the traditional batch algorithm for k-truss decomposition, and they hence have the same iteration number. Fig. 10c further shows the required job number for each method. The number of required jobs for MRTruss and i-MRTruss are the number of MapReduce jobs. As for GPTruss, the number of required jobs is equal to the number of MapReduce jobs plus the number of Hama jobs. It is clear to see that although there is only a small difference between the required iterations (Fig. 10c), the required jobs, related to disk I/O overhead, of them have a crucial variation. The reason that GPTruss outperforms the other two is that both the required iterations and the required jobs for GPTruss are always 2, which is much less than the others.

B. Experiments on Real World Data

To evaluate our methods on the real data, we choose four datasets[4] from the real world , which attributes are summarized in Table II, where the additional row header k_{max} is the maximal k for k-truss. Among these four datasets, com-Youtube, loc-Gowalla and com-DBLP are three dense datasets with high average degree, and RoadNet-TX is the largest dataset with over one million vertices. However, since RoadNet-TX has $d_{avg} < 3$, it is viewed as a sparse dataset. In the following experiments, only the experiment results of GPTruss and i-MRTruss are included because the running time of MRTruss is crucially too long.

[4]All datasets are available in http://snap.stanford.edu/data/index.html

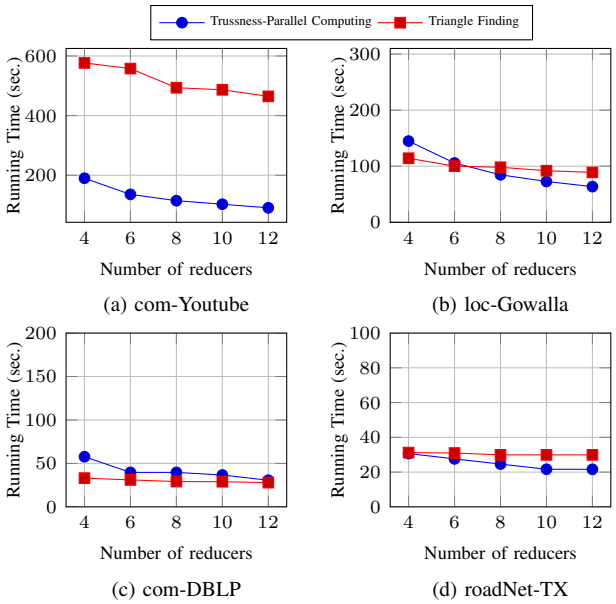


Fig. 12. The running time of two procedures in GPTruss over the 4 real datasets.

From Fig. 11a, 11b, and 11c, GPTruss is obviously more efficient than i-MRTruss, especially in the dense datasets. In the sparse dataset like roadNet-TX (Fig. 11d), the difference between GPTruss and i-MRTruss is not huge, but GPTruss still outperforms i-MRTruss. In addition, as the number of reducers increases, the running time of i-MRTruss does not decrease substantially.

Fig. 12 shows the running time of the two procedures: Triangle Finding and Trussness-Parallel Computing in GPTruss. The running time for both procedures is roughly similar in loc-Gowalla, com-DBLP, and roadNet-Tx. However, in com-Youtube, the running time of Triangle Finding is much longer than that of Trussness-Parallel Computing, which indicates that when a graph has a large number of edges, the running time of Triangle Finding will dominate the performance of GPTruss.

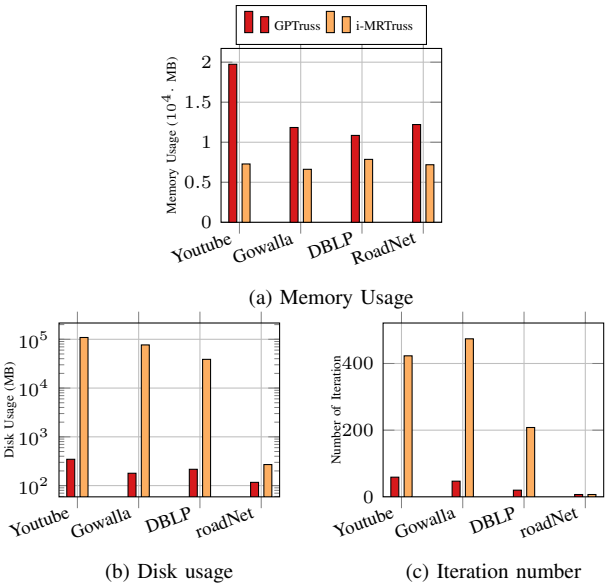
Furthermore, Fig. 13 shows the detailed analysis on memory usage, disk usage and the number of required iterations. Note that the memory usage of GPTruss is the average memory usage in each machine, while the memory usage of i-MRTruss is the average memory usage for a single job in each machine. Disk usage for both i-MRTruss and GPTruss is the total file size of immediate results and final results. In Fig. 13a, it shows that GPTruss consumes more memory than i-MRTruss, but it is cost-effective for both disk usage (Fig. 13b) and time consumption. In Fig. 13c, the number of iterations of i-MRTruss is quite larger than the one of GPTruss, which leads to more IO overhead (Fig. 13b) and a longer running time of i-MRTruss.

In summary, for a large and dense graph, GPTruss is much more efficient than MRTruss and i-MRTruss and for a sparse dataset, GPTruss still runs faster than MRTruss and i-MRTruss.

VII. CONCLUSION

In this paper, we first propose an improved method, called i-MRTruss, which is based on the existing distributed k-truss

Authorized licensed use limited to: INDIAN INSTITUTE OF TECHNOLOGY DELHI. Downloaded on March 14,2023 at 17:31:35 UTC from IEEE Xplore. Restrictions apply.



(a) Memory Usage

(b) Disk usage

(c) Iteration number

Fig. 13. The analysis of iterations of i-MRTruss and GPTruss over the 4 real datasets.

decomposition in the MapReduce framework. Nevertheless, MapReduce is not suitable for iterative graph computing, which usually incurs large disk usage, e.g., too many intermediate results and lots of unwanted disk I/O operations. Thus, we further propose a distributed k-truss decomposition, named GPTruss, in the graph-parallel abstractions. GPTruss is based on our derived locality theorem for k-truss, which can reduce the amount of communication and the iteration number in the graph-parallel abstractions, and therefore GPTruss has more efficient performance. In the future work, it is worth studying how to process the pruned line graph efficiently when a graph contains many edges, which causes longer running time as shown in the experimental.

REFERENCES

[1] D. Kempe, J. Kleinberg, and E. Tardos, "Maximizing the spread of influence through a social network," in *KDD*, 2003, pp. 137–146.

[2] X. He, G. Song, W. Chen, and Q. Jiang, "Influence blocking maximization in social networks under the competitive linear threshold model," in *SDM*, 2012, pp. 463–474.

[3] D.-N. Yang, C.-Y. Shen, W.-C. Lee, and M.-S. Chen, "On socio-spatial group query for location-based social networks," in *SIGKDD*, 2012.

[4] D.-N. Yang, Y.-L. Chen, W.-C. Lee, and M.-S. Chen, "On social-temporal group query with acquaintance constraint," in *VLDB*, 2011.

[5] H. Yu, A. Paccanaro, V. Trifonov, and M. Gerstein, "Predicting interactions in protein networks by completing defective cliques," *Bioinformatics*, vol. 22, pp. 823–829, 2006.

[6] J. D. Cohen, "Trusses: Cohesive subgraphs for social network analysis," *National Security Agency Technical Report*, 2008.

[7] J. Wang and J. Cheng, "Truss decomposition in massive networks," in *VLDB*, 2012.

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

[9] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," in *OSDI*, 2004.

[10] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: a system for large-scale graph processing," in *SIGMOD*, 2010.

[11] L. G. Valiant, "A bridging model for parallel computation," *Communications of the ACM*, vol. 33, no. 8, pp. 103–111, 1990.

[12] S. Seo, E. J. Yoon, J. Kim, S. Jin, J.-S. Kim, and S. Maeng, "Hama: An efficient matrix computation with the mapreduce framework," in *CloudCom*, 2010.

[13] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein, "Distributed graphlab: a framework for machine learning and data mining in the cloud," in *VLDB*, 2012.

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

[15] J. Cheng, L. Zhu, Y. Ke, and S. Chu, "Fast algorithms for maximal clique enumeration with limited memory," in *SIGKDD*, 2012.

[16] W. Fan, X. Wang, and Y. Wu, "Performance guarantees for distributed reachability queries," in *VLDB*, 2012.

[17] S. Chu and J. Cheng, "Triangle listing in massive networks and its applications," in *SIGKDD*, 2011.

[18] R.-H. Li, J. X. Yu, and R. Mao, "Efficient core maintenance in large dynamic graphs," *TKDE*, vol. 26, no. 10, pp. 2453–2465, 2014.

[19] A. Montresor, F. De Pellegrini, and D. Miorandi, "Distributed k-core decomposition," *TPDS*, vol. 24, no. 2, pp. 288–300, 2013.

[20] J. Cheng, Y. Ke, S. Chu, and M. T. Ozsu, "Efficient core decomposition in massive networks," in *ICDE*, 2011.

[21] A. E. Sarıyüce, B. Gedik, G. Jacques-Silva, K.-L. Wu, and U. V. Catalyürek, "Streaming algorithms for k-core decomposition," in *VLDB*, 2013.

[22] J. Leskovec and C. Faloutsos, "Scalable modeling of real graphs using kronecker multiplication," in *ICML*, 2007.

Authorized licensed use limited to: INDIAN INSTITUTE OF TECHNOLOGY DELHI. Downloaded on March 14,2023 at 17:31:35 UTC from IEEE Xplore. Restrictions apply.