Efficient Cohesive Subgraphs Detection in Parallel

Yingxia Shao[♯] simon0227@pku.edu.cn
Lei Chen[§] leichen@cse.ust.hk
Bin Cui[♯] bin.cui@pku.edu.cn
[♯]Key Lab of High Confidence Software Technologies (MOE), School of EECS, Peking University
[§]Department of Computer Science and Engineering, HKUST

ABSTRACT

A cohesive subgraph is a primary vehicle for massive graph analysis, and a newly introduced cohesive subgraph, k-truss, which is motivated by a natural observation of social cohesion, has attracted more and more attention. However, the existing parallel solutions to identify the k-truss are inefficient for very large graphs, as they still suffer from huge communication cost and large number of iterations during the computation. In this paper, we propose a novel **p**arallel and **e**fficient **t**russ detection **a**lgorithm, called PETA. The PETA produces a triangle complete subgraph (*TC-subgraph*) for every computing node. Based on the TC-subgraphs, PETA can detect the local k-truss in parallel within a few iterations. We theoretically prove, within this new paradigm, the communication cost of PETA is bounded by three times of the number of triangles, the total computation complexity of PETA is the same order as the best known serial algorithm and the number of iterations for a given partition scheme is minimized as well. Furthermore, we present a subgraph-oriented model to efficiently express PETA in parallel graph computing systems. The results of comprehensive experiments demonstrate, compared with the existing solutions, PETA saves 2X to 19X in communication cost, reduces 80% to 95% number of iterations and improves the overall performance by 80% across various real-world graphs.

Categories and Subject Descriptors

H.2.8 [**Database Applications**]: Data Mining; D.1.3 [**Programming Techniques**]: Concurrent Programming—*Parallel Programming*

Keywords

Graph Algorithm; Cohesive Subgraph; Truss; Subgraph-Oriented

1. INTRODUCTION

A cohesive subgraph [25] is an important vehicle for the analysis of massive graphs. The most intuitive definition for a cohesive subgraph is a clique [13] in which each vertex is adjacent to every other vertex. However, for graphs in the real-world, enormous cliques in small size exist while the number of larger cliques is limited, thus it is difficult to capture a meaningful subgraph for analysis, which reveals the clique definition is too strict to be helpful. Consequently, many other relaxed forms of cohesive subgraphs are proposed. n-clique [3] loosens the distance constraint between any two vertices from 1 to n. n-clan [15] is an n-clique in which the diameter should be no greater than n, while n-club [15] is just a subgraph whose diameter is no greater than n without n-clique restriction. Furthermore, several other definitions weakening the degree constraint of the clique are also given. k-plex [21] is a subgraph containing n vertices in which each vertex connects to no fewer than $n - k$ vertices. In contrast, k-core [20] only requires that each vertex has at least k neighbors.

All the aforementioned definitions of cohesive subgraph, except k-core, are faced with enormous enumeration problem (too many results) and computational intractability. Taking the clique as an example, it requires exponential number of enumerations and will discover at most $3^{n/3}$ maximal cliques [23, 16]. For k-core, it can be found in polynomial time and does not encounter the enumeration problem. Whereas the k-core conception is too general, the results are not that cohesive [4, 27].

Recently, a new type of the cohesive subgraph, called **k-truss** [4], was introduced. In k-truss, each edge is involved in at least $k - 2$ triangles, and the number of involved triangles is called the *support* of that edge. k-truss is more rigorous than k-core but still looser than clique, and the results can be highly cohesive [4]. Especially in a social network, k-truss ensures that each pair of friends has at least $k - 2$ common friends, and this is consistent with sociological researches [8, 26]. Meanwhile, k-truss avoids the enumeration problem and can be detected in polynomial time as well. The formal definition of k-truss and its important properties will be described in Section 2.

The majority of existing approaches for k-truss detection are all sequential and executed on a single-node [4, 24, 27]. In the presence of large scale graphs, which cannot be stored in a single node, the parallel solutions are required. Cohen [5] proposed a MapReduce-based algorithm for the k-truss detection. As the MapReduce [6] framework is disk-oriented and graph structure unaware, Cohen's solution is inefficient in practice [24]. L. Quick [18] introduced a parallel algorithm on Pregel-like graph computing systems [14, 19, 1], which distributively store the graph in memory and run graph algorithm with vertex-centric model in parallel, to improve the performance.

However, all the existing parallel solutions [5, 18] deploy the same strategy to detect the trusses. The algorithm iteratively prunes the graph based on the definition of k-truss via three sub-routines, i.e., enumerating all the triangles in the graph, for each edge counting the number of triangles containing it, removing the edges with insufficient support. The algorithm discovers a valid k-truss until

SIGMOD'14, June 22 - 27 2014, Snowbird, UT, USA.

ACM 978-1-4503-2376-5/14/06 ...$15.00.
http://dx.doi.org/10.1145/2588555.2593665.

no more edges can be dropped. Unfortunately, this strategy still suffers from high communication cost and large number of iterations to prune, furthermore, the entire computation complexity is hardly to be close to the best known order, $\mathcal{O}(m^{\frac{3}{2}})$, of serial algorithms.

The inefficiency of the state-of-the-art parallel algorithms is caused by the repeated triangle counting and improper programming models. The repeated triangle counting not only brings more iterations, but also increases the communication and computation cost. From the programming model aspect, the MapReduce is too loose to capture the graph structure, and the vertex-centric model is too restricted to operate the local graph flexibly.

In this paper, we propose a novel parallel and efficient truss detection algorithm, named PETA. The algorithm constructs a triangle complete subgraph (or *TC-subgraph* for short), for each computing node. On basis of TC-subgraphs, the algorithm can find the local k-trusses in parallel. Furthermore, we prove the simple union of these local k-trusses from TC-subgraphs is exactly the global k-truss. Moreover, the algorithm avoids the tediously recounting triangles by applying the *seamless detection* (detection without restart) between successive iterations. Thus, in our new framework, the worst communication cost is bounded by a tight upper limit, and the real communication cost is only related to the size of invalid triangles. The total computation complexity can achieve the same order as the best known sequential algorithm and the number of iterations is minimized.

In order to efficiently implement the PETA, we first identify the power-law distribution between the frequency and edges' initial supports, named as *Edge-Support Law*. The edge-support law ensures that PETA can achieve a low space cost for graphs in the real world. The space cost can be further reduced by applying an edge-balanced partition scheme. Moreover, we extend the popular vertex-centric model into a subgraph-oriented model. The subgraph-oriented model treats the local subgraph (partition) as the minimal operating unit and allows users to access and update the local graph directly. Thus, we can easily equip the existing in-memory algorithms to enhance the local performance.

In summary, we list our contributions in this work as follows.

- We design an efficient parallel k-truss detection algorithm, named PETA, which constructs triangle complete subgraphs and avoids redundant computations by the seamless detection technique.
- We prove PETA has the same order as the best serial solution, and also achieves the minimal number of iterations under certain partition schemes.
- We propose the edge-support law in real world graph, which ensures the algorithm achieves an appropriate space cost in practice.
- We present a subgraph-oriented model. The model relaxes the vertex-centric model, and can express graph algorithms more efficiently.

Organization: The formal problem definitions are described in Section 2. Section 3 introduces the exsiting solutions and their limitations. The novel PETA is elaborated in Section 4, followed by analyzing the efficiency of PETA (Section 5) and discussing its practical optimizations (Section 6). The experimental results are presented in Section 7. The last two sections go through the related work and conclude this paper.

2. PROBLEM DEFINITION

In this section, we re-formalize several core concepts related to k-truss cohesive subgraph, which are first introduced by Cohen [4], followed by the definition of k-truss detection problem. Then, we describe the parallel settings for the k-truss detection problem.

Symbols	Description
$G = (V, E)$	An undirected graph
n, m	The size of vertex/edge set in graph G
(u, v)	An edge in graph G
$N(v)$	All vertices adjacent to v in graph G
$d(v)$	The degree of v, i.e., $d(v) = \|N(v)\|$
$N(V)$	The union of $N(v), v \in V$
k	The threshold for truss detection problem
$\theta_G(e), \theta(e)$	The support of an edge e in graph G
$\Gamma(G, k), \Gamma_k$	The maximal k-truss in graph G
$\overline{\rho}$	The average number of replications of an edge
γ	The edge cut ratio of a graph partitioning
T_{vuw}	A triangle formed by vertices v, u, w
$\|\triangle_G\|, \|\triangle\|$	The number of triangles in graph G

Table 1: Notations Summary

2.1 Preliminaries

The purpose of our work is to efficiently detect the cohesive subgraph, k-truss, in an undirected graph G in parallel. The graph G consists of a vertex set V and an edge set E. An edge $e \in E$ is undirected and joins two vertices $v, u \in V$, denoted by (u, v) or (v, u). We also use $n(= |V|)$, $m(= |E|)$ to simplify the representations of the number of vertices and the number of edges in G, respectively. The symbol $d(v)$ stands for the degree of a vertex v, which equals to $|N(v)|$. $N(v)$ contains all the vertices that are adjacent to the vertex v. Moreover, the neighbor set of a vertex set V' is the union of each single vertex's neighbors, i.e., $N(V') = \bigcup_{v \in V'} N(v)$. All the notations frequently used in this paper are summarized in Table 1.

A *triangle* in a graph is a cycle of length three and denoted by T_{uvw} if its three vertices are u,v,w. The *support* of an edge based on the triangle can be stated as follows,

DEFINITION 1. (SUPPORT). *The support of an edge $e = (u, v)$ in graph G, denoted by $\theta_G(e)$, is defined as the number of triangles that the (u, v) is involved, i.e., $\theta_G(e) = |\{T_{uvw} | (u, w), (v, w) \in E\}|$.*

When the context is clear, we simplify the $\theta_G(e)$ into $\theta(e)$. Based on the support of an edge, we proceed to define the k-truss in a graph G.

DEFINITION 2. (k-TRUSS). *Given a graph $G = (V, E)$, if a subgraph $G_s = (V_s, E_s)$ in G, satisfies that $\forall\ e = (u, v) \in E_s$, $\theta_{G_s}(e) \geq k - 2$, then the subgraph G_s is a k-truss cohesive subgraph in G.*

Furthermore, the *maximal k-truss*, denoted by $\Gamma(G, k)$, is the one that cannot be contained by any other k-truss in G. We simply use Γ_k to represent $\Gamma(G, k)$ when the context of graph G is clear. In this paper, we assume that the empty subgraph is a k-truss of graph G for arbitrate k. Thus any graph G at least has a k-truss for the arbitrate k.

The following property guarantees that there is only one Γ_k in a graph G.

PROPERTY 1. (UNIQUENESS OF THE MAXIMAL k-TRUSS). *Only one maximal k-truss exists in a graph G for a fixed threshold k.*

PROOF. Suppose only q (q>1) different maximal k-trusses exist in a graph G, and they are Γ_{k1}, Γ_{k2}, ... , Γ_{kq}. According to

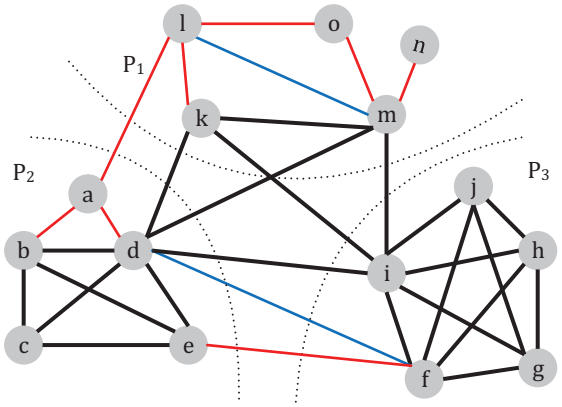


Figure 1: A toy graph G and it is partitioned into P_1, P_2, P_3. The subgraph with black thick edges is Γ_4

Definition 2 and the maximal property, $\bigcup_{1\leq i\leq q}\Gamma_{ki}$ is also a maximal k-truss in graph G. This leads to the contradiction that only q different maximal k-truss exist, since we now have $q+1$ different ones. So q can only be one. □

In this paper, we focus on the following problem, which is the key issue for the truss-related task [27, 24, 4].

Problem: (k-TRUSS DETECTION). *Given a graph $G=(V,E)$ and a threshold k, finding the maximal k-truss in G.*

Here is an example in Figure 1. Given a graph G and threshold k=4, the goal of above problem is to discover the subgraph with black thick edges, which is a Γ_4.

2.2 Fundamental Operation

The k-truss detection problem can be solved in polynomial time, but the various implementations of the fundamental operation result into different computation complexity. The *fundamental operation* is responsible for counting the triangles around a certain edge. According to Definition 1, $\theta(e)$ is the exact number of triangles that the edge e is involved. Therefore, $\theta(e)$ can be computed simply by counting the triangles around that edge e. Two popular solutions for the fundamental operation are listed below.

Classic solution. It first sorts the neighbors of each vertex in ascending order on their IDs. Then for an edge (u,v), the algorithm can calculate the number of triangles around (u,v) in $\mathcal{O}(d(v)+d(u))$. This method helps discover the k-truss in $\mathcal{O}(\sum_{(u,v)\in E}(d(v)+d(u)))$ = $\mathcal{O}(\sum_{v\in V} d^2(v))$ time complexity and has been applied in [4, 27].

Index-based solution. It is the most recent solution, proposed in [24]. When operating an edge (u,v), the algorithm only enumerates neighbors w of the vertex u with smaller degree, and tests the existence of the third edge (w,v) in graph G. Since the testing can be done in constant time with the help of a hashtable, the fundamental operation can be completed in $\mathcal{O}(\min\{d(v),d(u)\})$. Thus, the total computing complexity is decreased to $\mathcal{O}(m^{\frac{3}{2}})$, which is the best known time complexity.

2.3 Parallel Computing Context

Before proceeding to the parallel k-truss detection algorithms, we clarify the parallel computing context for the k-truss detection.

The original graph $G=(V,E)$ is divided into W partitions, which are P_1, P_2, ... , P_W. A single partition P_i=(V_i, E_i), $1\leq i\leq W$, satisfies $V_i\subseteq V$ and E_i=$\{(u,v)|u\in V_i \vee v\in V_i\}$. Additionally, the W partitions satisfy that $V=\bigcup_{1\leq i\leq W} V_i$, and $V_i \bigcap V_j=\phi, i\neq j$. Initially, the W partitions are distributively stored among W computing nodes' memory.

Since the graph G is partitioned, some edges will cross different partitions, called *cross edges*. The remained edges are *internal edges*. We define the *edge cut ratio*, denoted by γ, as the ratio between the number of cross edges and the number of edges in graph G. Meanwhile, the cross edges will be replicated in order to maintain the connectivity. We use *edge replicating factor*, $\overline{\rho}$, to represent the average number that an edge is replicated. $\overline{\rho}$ is the ratio between the number of edges maintained by the parallel algorithm and the original size of E in graph G.

In the parallel computing context, all triangles in the partitioned graph G are classified into three different types according to the effect of cross edges.

- **Type I** is the triangle only consisting of internal edges.
- **Type II** is the triangle including both internal edges and cross edges.
- **Type III** is the triangle only containing cross edges.

For example, in Figure 1, edges (k,m), (l,k), (l,m) are internal edges, so T_{kml} is Type I. Edges (d,k), (d,m), (d,i), (k,i) are cross edges, so T_{dkm} is Type II and T_{dki} belongs to Type III.

3. THE CUTTING-EDGE PARALLEL SOLUTIONS

As briefly mentioned in Section 1, two parallel algorithms for k-truss detection have been developed. One is implemented on the MapReduce framework, and the other is improved by adapting the MapReduce solution to the Pregel-like systems. However, both approaches follow the same logical framework, which is first introduced by Cohen [5] and listed below.

1. Enumerate triangles.
2. For each edge, record the number of triangles, $\theta(e)$, containing that edge.
3. Keep only edges with $\theta(e)\geq k-2$.
4. If step 3 dropped any edges, return to step 1.
5. The remained graph is the maximal k-truss.

This type of algorithms requires the triangle counting routine in every *logical iteration* (steps 1-4). Furthermore, due to the algorithm's restriction, it will incur many iterations. Since the MapReduce framework flushes all the intermediate results to the disk and is unaware of the graph structure, it is inefficient for the iterative graph workloads [14]. The inefficiency of the MapReduce solution for k-truss detection is also pointed out in [24]. Although, the L. Quick's solution tries to eliminate the shortages of the MapReduce solution by applying the Pregel-like systems, which is in-memory and graph aware, the performance is still hindered by the internal limitations of the logical framework.

In the following subsections, we introduce our improvement on the L. Quick's solution first, and then show the inherent limitations of the existing parallel solutions.

3.1 Improved L. Quick's Algorithm

L. Quick's [18] parallel solution uses the popular graph computing system, Pregel, and runs on an ordered directed graph, where each edge joins two vertices from the one with small degree to the one with large degree and breaks ties based on vertices' id. The

Algorithm 1 Improved vertex algorithm for k-truss detection

```
1.  /* v represents the vertex where this algorithm runs. */
2.  inerStep ← getSuperstep() % 3
3.  switch (inerStep)
4.  case 0:
5.     foreach pair (u, w), u, w ∈ N(v) do
6.        send message (v, u, w) to the vertex u for completing
          edge (u, w). {d(v) ≤ d(u) ≤ d(w).}
7.     end foreach
8.  case 1:
9.     foreach incoming message (u, v, w) do
10.       if w ∈ N(v) then
11.          send message (u, v, w) to u, v. {notify the sources of
             three edges in the triangle T_uvw.}
12.       end if{v ∈ N(u) ∧ w ∈ N(u) holds from case 0.}
13.    end foreach
14. case 2:
15.    foreach incoming message (v', u', w') do
16.       if v=v' then
17.          increase θ((v, u')), θ((v, w'))
18.       else if v=u' then
19.          increase θ(v, w')
20.       end if{v=w' is impossible!}
21.    end foreach
22.    foreach u ∈ N(v) do
23.       if θ((u, v)) < k − 2 then
24.          remove edge (v, u).
25.       end if
26.    end foreach
27. end switch
```

approach splits a logical iteration into four separated supersteps, between which a synchronization will happen. Thus, the synchronization explodes in four times. We improve it by simply getting rid of the fourth superstep, which removes the vertex with zero degree in every logical iteration. The improved approach drops the isolated vertices altogether in the end, since the isolated vertices do not bring additional computation in the middle of the algorithm. This modification not only reduces the number of synchronization (supersteps) in Pregel-like systems, but also decreases the communication cost (the original version requires messages for the vertex deletion).

The new vertex algorithm for the improved solution is listed in Algorithm 1. The algorithm still runs on an ordered directed graph, but repeats in every three supersteps until no edges are dropped. In the first superstep (Lines 4-7), each vertex v enumerates vertex pairs from its neighbors and sends corresponding triad messages for completing the third edge. The second superstep (Lines 8-13) is responsible for verifying the existence of the third edge in a triad message and notifying the sources of three edges if a triangle is found. In the third superstep, each vertex counts the number of triangles for its adjacent edges (Lines 15-21) and removes the edge whose $\theta(e)$ is below the threshold $k-2$ (Lines 22-26).

3.2 Limitations of the Algorithm

Although the parallel graph computing systems facilitate the solution and solve the problem in-memory, the approach is still hindered by several inherent limitations. First, the algorithm still enlarges the synchronized frequency by three times. Second, the repeated triangle counting brings in massive redundant computations and communications. For example, the triangles reserved in the final found k-truss will be repeatedly computed as much as the number of iterations. Third, the restriction of the vertex-centric model in Pregel only supports to implement the classic solution for the fundamental operation. This leads to that the computation complexity of the algorithm in one iteration is $\mathcal{O}(d_{max}\sum_1^n \hat{\mathrm{d}}^2(v))$, where $\hat{\mathrm{d}}(v)$=$|\{u|u \in N(v) \wedge d(v) \leq d(u)\}|$, and d_{max}=$\max\{\hat{\mathrm{d}}(v)\}$.

Example. Detecting Γ_4 in graph G as shown in Figure 1. The improved algorithm takes three logical iterations, which totally consumes nine supersteps. The red edges and blue edges are eliminated in the first and second logical iterations, respectively. The third iteration guarantees that the result graph doesn't change any more. In a single iteration, a Type II or Type III triangle will cause four remote messages, in which one for querying and three for notifying. So the algorithm sends 60 messages altogether in the three iterations. However, the algorithm only occupies a small amount of memory benefit from the ordered directed graph, and the edge replicating factor $\overline{\rho}$ is one.

To summarize, these limitations inhibit existing algorithms from obtaining a satisfactory performance. We need to redesign the k-truss detection framework, and require a more flexible programming model with graph structure preserving to implement the detection framework.

4. THE IDEA OF PETA

We propose a novel **p**arallel and **e**fficient **t**russ detection **a**lgorithm, called PETA. The basic idea in PETA is that each computing node simultaneously finds the local maximal k-truss independently. Once all the local maximal k-truss subgraphs are discovered, the computing nodes exchange the graph mutations with each other and continue to refine the previous local maximal k-truss subgraphs. The process is repeated until all the local maximal k-truss subgraphs are stable (no more external edges are removed). Finally, the maximal global k-truss subgraph is the simple union of all local maximal k-truss subgraphs.

In following subsections, we first introduce the subgraph-oriented model, which enables parallel graph computing systems to express more rich graph algorithms, like PETA. Then a special subgraph, TC-subgraph, is presented. Finally, the local subgraph algorithm on top of TC-subgraph in PETA is described.

4.1 Subgraph-Oriented Model

The classic vertex-centric model is too constrained to (efficiently) express a rich set of graph algorithms, since a vertex algorithm is limited to access its neighbors only and the other parts of local subgraph cannot be accessed. We extend it into a subgraph-oriented model.

```
class Subgraph{
    public abstract void subgraphCompute();
    public Vertex getVertex(long vid);
    public Vertex removeVertex(long vid);
    public void addVertex(long vid, Vertex v);
    public Iterator<Vertex> getVertexIterator();
    public Edge getEdge(long src, long dest);
    public Edge removeEdge(long src, long dest);
    public void addEdge(Edge e);
    public Iterator<Edge> getEdgeIterator();
    public void sendMessage(long vid, Message msg);
}
```

Figure 2: API abstraction in subgraph-oriented model

The subgraph-oriented model removes the vertex constraints, treats the local subgraph as the minimal operable unit, and allows users to directly access and update the local graph. Figure 2 shows the typical APIs for a subgraph program. For instance, at any time, user can randomly access a vertex via getVertex() method, and then operate the vertex as what can be done in the vertex-centric model. The parallel computing framework executes many subgraph programs concurrently, and each of them is the minimal computing

unit. The subgraph programs exchange messages with each other between the successive iterations.

Comparing to the vertex-centric model, the subgraph-oriented model is a coarse-grained one and opens up new opportunities to efficiently express local graph algorithms. Because of the flexibility of operating local graph in subgraph-oriented model, the algorithms can access vertices and edges on demand. Like in PETA, the edges of local subgraph are visited in the removing order. It is impossible to implement such an edge access order in vertex-centric model. Furthermore, the vertex-centric model is just a special case of the subgraph-oriented model, (i.e., the subgraph-oriented model becomes the vertex-centric one when the subgraphCompute() method is implemented as sequentially visiting the vertices of local subgraph), so the subgraph-oriented model is able to express all the graph algorithms which can be realized on vertex-centric model. These graph algorithms can be optimized further by their corresponding sequential algorithms. For instance, implementing BFS on subgraph-oriented model, in each iteration, a computing node is initialized by external vertices, which have been visited by other computing nodes, and executes the sequential BFS algorithm on local subgraph. Thus the number of iterations will be reduced, since more than one-hop vertices are explored in each iteration.

To summarize the above discussion, subgraph-oriented model can express more complicated graph algorithms by eliminating the vertex constraints. It is a cornerstone for the PETA to be expressible in a parallel graph computing system.

4.2 Triangle Complete Subgraph

Naturally, a raw local partition on each computing node doesn't include the complete information of every edge's common neighborhood, so that the Type II and Type III triangles cannot be directly captured in the local partition. Therefore, it is impossible to compute the local maximal k-truss subgraph just relying on the local partition. For instance, in Figure 1, the local partition P_2 cannot be aware of the existence of triangles T_{dim} and T_{dif}, because P_2 is unable to access vertices m, i and f's neighborhoods at local.

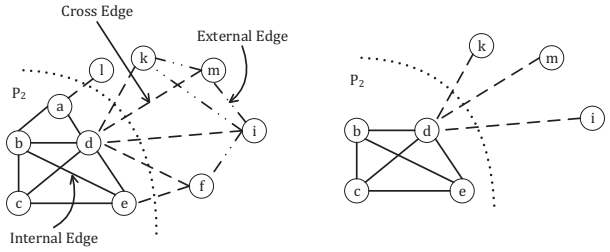
In the PETA, it works on the *triangle complete subgraph* to do the local computation. The definition is given as follows,

DEFINITION 3. (TRIANGLE COMPLETE SUBGRAPH). *Given a graph $G = (V, E)$, and its subgraph $G_s = (V_s, E_s), V_s \subseteq V, E_s \subseteq E$. The triangle complete subgraph, **TC-subgraph** for short, $G_s^+ = (V_s^+, E_s^+)$ satisfies $V_s^+ = \{V_s \cup N(V_s)\}$ and $E_s^+ = E \cap (\{(u,v)|\ u \in V_s \vee v \in V_s\} \cup \{(u,v)|u, v \in N(V_s) \wedge \exists w \in V_s, s.t.\ (w,u),(w,v) \in E_s^+\})$.*

In accordance with above definition, the edges in a TC-subgraph belong to three categories. Besides the internal edge and cross edge, the third type is ***external edge***. The external edge is the one whose both end vertices are belong to $N(V_s)$. It is a phantom of internal or cross edge in some other TC-subgraphs. Figure 3(a) shows a TC-subgraph based on the local partition P_2 in graph G. In the TC-subgraph, (b, e) is an internal edge, (d, m) is a cross edge and (m, i) is an external edge.

Note that, the notation of TC-subgraph is more restricted than the induced subgraph. In Figure 3(a), the edge (l, m) is not included in the TC-subgraph while it will be in the corresponding induced subgraph. Fortunately, Theorem 1 points out the TC-subgraph has the full triangle information for all internal and cross edges, so that the TC-subgraph consumes less memory while preserving sufficient knowledge.

THEOREM 1. *TC-subgraph $G_s^+ = (V_s^+, E_s^+)$ of the subgraph $G_s = (V_s, E_s)$ in graph $G = (V, E)$ contains the same number of triangles for every edge $e = (u, v)$, in which $(u \in V_s \vee v \in V_s)$ holds, as the graph G does.*



(a) TC-subgraph (b) Local Γ_4

Figure 3: TC-subgraph and local Γ_4 in P_2 of graph G.

PROOF. Using proof by contradiction. Let the edge $e_0 = (u_0, v_0)$, $u_0 \in V_s$, has different number of triangles between G_s^+ and G.

1): Assume e_0 has more triangles in G_s^+. Then there exists at least one vertex w_0, which forms a triangle $T_{u_0 w_0 v_0}$ in G_s^+ and does not form one in G. This fails to hold the condition $(u_0, w_0) \in E_s^+ \wedge (u_0, w_0) \in E$.

2): When e_0 has fewer triangles, G contains a vertex w_0 in contrast. If w_0 is not in V_s^+, then the condition $V_s^+ = \{V_s \cup N(V_s)\}$ fails, because u_0 is in V_s, and w_0 should be in $N(V_s)$. If w_0 exists in G_s^+, then the edge $(u_0, w_0), (v_0, w_0)$ should be included by the definition of E_s^+.

In conclusion, the theorem is correct. □

Based on Theorem 1, the node can compute on the internal edges and cross edges locally and correctly. For simplicity, we call both edges as the *core edges* in a TC-subgraph. We proceed to define the *local maximal k-truss* in a TC-subgraph.

DEFINITION 4. (LOCAL MAXIMAL k-TRUSS). *A subgraph $G_k = (V_k, E_k)$ in TC-subgraph $G_s^+ = (V_s^+, E_s^+)$, $V_k \subseteq V_s^+$, $E_k = E_s^+ \cap \{(u,v)|u \in (V_s \cap V_k) \vee v \in (V_s \cap V_k)\}$, is the local k-truss if and only if $\theta_{G_k'}(e) \geq k-2$, $e \in E_k$, where $G_k' = (V_k, E_s^+ \cap \{(u,v)|u \in V_k \wedge v \in V_k\})$. The local maximal k-truss is the one no other local k-truss contains it.*

Figure 3(b) illustrates the corresponding local maximal 4-truss in the TC-subgraph of P_2. Similar to the maximal k-truss, the local maximal k-truss also has Property 1. The following theorem ensures the correctness of PETA. The algorithm can discover the correct maximal k-truss with iteratively finding the local maximal ones.

THEOREM 2. *When all the local maximal k-trusses are stable, the union of these maximal local k-trusses is the global maximal k-truss subgraph.*

PROOF. According to Theorem 1 and Definition 4, the local Γ_k indicates that its core edges belong to the global Γ_k if the external edges still exist in global. Since all the local Γ_ks are stable, which implies no graph mutations, the external edges in one local Γ_k must be reserved in some other local Γ_k as the core edges. So the above theorem holds. □

4.3 The Local Subgraph Algorithm in PETA

Here we elaborate the local algorithm of PETA, which is also a subgraph program in the subgraph-oriented model. Thanks to the TC-subgraph, each computing node can detect the local maximal k-truss independently during an iteration. So the local subgraph algorithm is divided into two distinct phases. The first one is the

Algorithm 2 Local Algorithm in PETA

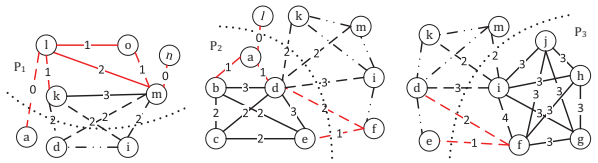
Input: local partition $P = (V_i, E_i)$, threshold k.
Output: stable local maximal k-truss.

1. /*initialization phase*/
2. $G_s^+ \leftarrow P$
3. send triad message (v, u, w) to the subgraph of containing vertex u. $\{v \in V_i, u, w \in N(v), d(v) \leq d(u) \leq d(w).\}$
4. *synchronized barrier* {one iteration}
5. **foreach** incoming message (v, u, w) **do**
6. notify v, u, w by message (v, u, w), if the edge (u, w) exists
7. **end foreach**
8. *synchronized barrier* {one iteration}
9. **foreach** incoming message (v, u, w) **do**
10. **foreach** edge e in $\{(u, v),(v, w),(u, w)\}$ **do**
11. **if** e is the core edge **then**
12. $\theta(e) \leftarrow \theta(e) + 1$ /*calculate initial $\theta(e)$.*/
13. **else**
14. $E_s^+ \leftarrow E_s^+ \cup \{e\}$ /*materialize external edge.*/
15. **end if**
16. **end foreach**
17. **end foreach**
18. $invalidQueue \leftarrow \phi$ /* stores removable edges */
19. **foreach** core edge e in E_s^+ **do**
20. **if** $\theta(e) \leq k - 2$ **then**
21. enqueue e into $invalidQueue$
22. **end if**
23. **end foreach**
24. /*detection phase.*/
25. **repeat**
26. enqueue removed external edge into $invalidQueue$
27. /*detect local maximal k-truss in G_s^+*/
28. **while** $invalidQueue \neq \phi$ **do**
29. $e \leftarrow$ dequeue $invalidQueue$
30. call Index-based solution of fundamental operation on e to decrease the support of other two edges
31. enqueue the invalid core edges into $invalidQueue$.
32. **end while**
33. notify graph mutations to remote subgraphs if necessary.
34. *synchronized barrier* {one iteration}
35. **until** All local maximal k-trusses are stable.

initialization phase, in which TC-subgraphs are constructed and the initial $\theta(e)$ are calculated for the core edges. The other one is responsible to find the local maximal k-truss and is called *detection phase*. The pseudo-code is illustrated in Algorithm 2.

In the *initialization phase*, computing nodes generate their TC-subgraph through the triangle counting routine. The only difference is that the computing nodes should materialize triangles in local if they are Type II or Type III which involve an external edge (Line 14). Along with the TC-subgraph construction, the initial $\theta(e)$ of core edges are calculated as well (Line 12). The initialization phase takes three iterations and the third iteration mingles with the first detection iteration (Lines 9-23).

In the *detection phase*, the local maximal property ensures that each computing node can do the detection continuously between iterations without any redundant computation. Thus the whole computation on a TC-subgraph, although separated by synchronization (Line 33), is same as the one in a single node, we name it as *seamless detection*. The seamless detection starts to refine the local maximal k-truss only based on the removed external edges (Line 26), except the first detection iteration. Since the local maximal k-truss has been discovered in previous iteration, the current iteration doesn't need to modify it unless some external edges are deleted in some other TC-subgraphs. Thus, $\theta(e)$ is decreased only due to the triangle missing that is related to the external edges' removal. The algorithm doesn't need to scan over all the local core edges in each detection iteration, only follows the edge removing order to update the local maximal k-truss. For the first detection iteration, the algorithm simply scan over all core edges once, and enqueue the edges with insufficient $\theta(e)$ to start the detection (Lines 19-23). After the local maximal k-truss is discovered, the algorithm notifies other TC-subgraphs that are influenced by the local core edges' removal (Line 32). The detection phase is accomplished when no external edges are removed.



(a) No message. (b) Send (e, f) removal message to P_3. (c) No message.

Figure 4: The 3^{th} iteration of 4-truss detection on graph G. The initial $\theta(e)$s are assigned on the core edges.

Example. PETA finishes in four iterations for solving the 4-truss detection in graph G of Figure 1. The first two iterations construct the TC-subgraph. The third one initializes $\theta(e)$ of the core edges and finds the local Γ_4 successively. Figure 4 illustrates the third iteration. All red edges will be removed by each computing node, and only one message that causes T_{def} in P_3 missing will be sent from P_2. The last iteration refines the local Γ_4 according to the message of external edges removal and discovers all the local Γ_4s are stable. Overall, the algorithm takes four iterations with sending 25 messages (24 messages are from the TC-subgraph construction).

5. EFFICIENCY ANALYSIS OF PETA

In this section, we analyze the efficiency of PETA. The analysis comes from four different aspects. They are space cost, computation cost, communication cost, and the number of iterations. Overall, within the new paradigm, i.e., TC-subgraph and seamless detection, the communication cost of PETA is bounded by $3|\triangle|$. The total computation complexity is $\mathcal{O}(\frac{\overline{\rho}^{\frac{3}{2}}}{\sqrt{W}} m^{\frac{3}{2}})$ and PETA also achieves the minimal number of iterations for a given partition.

5.1 Space Cost

Since the TC-subgraph requires to materialize the external edges for the triangle completeness, the memory overhead is incurred, which can be measured by $\overline{\rho}$. Theorem 3 shows the expectation of $\overline{\rho}$ for a random partitioned graph G.

THEOREM 3. *Given a graph G, which is random partitioned into W parts, then*

$$E(\overline{\rho}) = (2 - \frac{1}{W}) + \frac{3|\triangle|}{m}(1 - \frac{1}{W})^2.$$

PROOF. In a random partition, an edge is a cross one with probability $(1 - \frac{1}{W})$, so $(1 - \frac{1}{W})m$ edges are replicated for maintaining the graph connectivity.

The other edge replication is from external edges for the completeness of Type II and Type III triangles. The number of external edges can be estimated as follows.

$$\sum_{v \in V} \{(1 - \frac{1}{W})^2 \frac{1}{2} \times \sum_{u \in N(v)} \theta((v, u))\} = 3|\triangle|(1 - \frac{1}{W})^2,$$

where $(1 - \frac{1}{W})^2$ means a triangle with two cross edges brings in an external edge.

In summary, $E(\overline{\rho})$ is

$$E(\overline{\rho}) = \frac{1}{m}(m + (1 - \frac{1}{W})m + 3|\triangle|(1 - \frac{1}{W})^2)$$
$$= (2 - \frac{1}{W}) + \frac{3|\triangle|}{m}(1 - \frac{1}{W})^2.$$

□

Following corollary is directly from above theorem,

COROLLARY 1. *Given a graph G, the upper bound of $\overline{\rho}$ is $2 + \frac{3|\triangle|}{m}$.*

The upper bound in Corollary 1 can be achieved, when each single vertex in graph G forms a partition. When the upper bound is reached if the graph G is a clique, then every TC-subgraph becomes the entire clique. At that time, although the core edges are small, the space cost might be too large to be unacceptable. Fortunately, for the most real-world graphs, the upper bound is relatively small, and the actual $\overline{\rho}$ hardly reaches the upper bound even if the random partition scheme is used. This will be discussed in Section 6.1.

5.2 Computation Complexity

The subgraph-oriented model supports us to implement the fundamental operation in the index-based way. Thus, the parallel algorithm is able to detect the local maximal k-truss as efficient as the best serial approach [24]. Theorem 4 shows that the parallel algorithm also can achieve the same complexity order as the best serial one in total. This reveals that PETA is really efficient in computation cost.

THEOREM 4. *The total computation complexity of the parallel algorithm is $\mathcal{O}(\frac{\overline{\rho}^{\frac{3}{2}}}{\sqrt{W}} m^{\frac{3}{2}})$.*

PROOF. Here we assume the random partition scheme is used, so that the edges are distributed in a perfect balance.

For the TC-subgraph construction, the main logic is the same as triangle counting, whose computation complexity is $\mathcal{O}(m^{\frac{3}{2}})$ [11, 2].

The seamless detection technique ensures that each edge from the three types in a TC-subgraph executes the fundamental operation at most once. Combining with the index-based implementation of fundamental operation, the computation complexity in local can be $\mathcal{O}(m_i^{\frac{3}{2}})$. Then the total complexity of the parallel algorithm is

$$\sum_1^W m_i^{\frac{3}{2}} \approx \sum_1^W (\frac{\overline{\rho} m}{W})^{\frac{3}{2}} \quad \because \text{ Balanced Partition}$$
$$= \mathcal{O}(\frac{\overline{\rho}^{\frac{3}{2}}}{\sqrt{W}} m^{\frac{3}{2}}). \tag{1}$$

Overall, the theorem holds when $\overline{\rho} \ll m$, which is true for the real-world graphs. □

Additionally, in accordance with the above theorem, an improvement of space cost (small $\overline{\rho}$) also enhances the total computation cost.

5.3 Communication Cost

THEOREM 5. *The communication cost of detection phase in Algorithm 2 is bounded by $3|\triangle|$.*

PROOF. In detection phase, communication only occurs when the Type II and Type III triangles are eliminated. In the worst case, a triangle is removed in three independent TC-subgraphs at the same time. Moreover, Algorithm 2 guarantees that each triangle is deleted exactly once, so the communication upper bound is $3|\triangle|$. □

Theorem 5 gives a bound of communication cost for the detection phase in PETA. Since the real communication is only from the removals of Type II and Type III triangles, we can further decrease the communication cost with a partition which contains small number of cut-triangles. This implies that a small $\overline{\rho}$ (few external edges) will reduce the communication cost as well. By the way, the communication of the TC-subgraph construction is the same with triangle counting process.

5.4 Number of Iterations

On account of the more iterations are executed, the more synchronization is required and the heavier influence is caused by the imbalance. It is necessary to obtain a low iteration number. Fortunately, Theorem 6 guarantees that PETA achieves the minimized number of iterations for a given partition.

First, we introduce the following lemma,

LEMMA 1. *In Algorithm 2, the number of removed edges during an iteration of the detection phase is maximized.*

PROOF. This can be directly derived from the algorithm. As it finds the local maximal k-truss in each iteration, there must be no core edges, whose $\theta(e) < k-2$, in the remained TC-subgraphs. □

On basis of Lemma 1, we yield

THEOREM 6. *Given a partitioned graph $G = (V, E)$ and the parameter k, Algorithm 2 achieves the minimal iterations (synchronization).*

PROOF. Suppose Algorithm 2 $\mathcal{A}$ finishes in S_1 iterations, and each iteration removes e_i edges, $1 \le i \le S_1$. The optimal one $\mathcal{A}^*$ runs $S_2 (\le S_1)$ iterations with eliminating e_i^* in different iterations.

Property 1 implies that the total size of removed edges is fixed by the problem, i.e., $\sum_{i=1}^{S_1} e_i = \sum_{i=1}^{S_2} e_i^*$.

According to Lemma 1, the constraint $e_i \ge e_i^*$ holds. So $\sum_{i=1}^{S_2} e_i \ge \sum_{i=1}^{S_2} e_i^*$ holds as well.

Finally, we get $\sum_{i=S_2+1}^{S_1} e_i = 0$, which indicates there are no edges removed from iteration $S_2 + 1$ to S_1. Recall the stopping condition of Algorithm 2 is that no graph mutation exists, so the $\mathcal{A}$ must finished at S_2 and $\mathcal{A} = \mathcal{A}^*$. □

6. PRACTICAL OPTIMIZATIONS

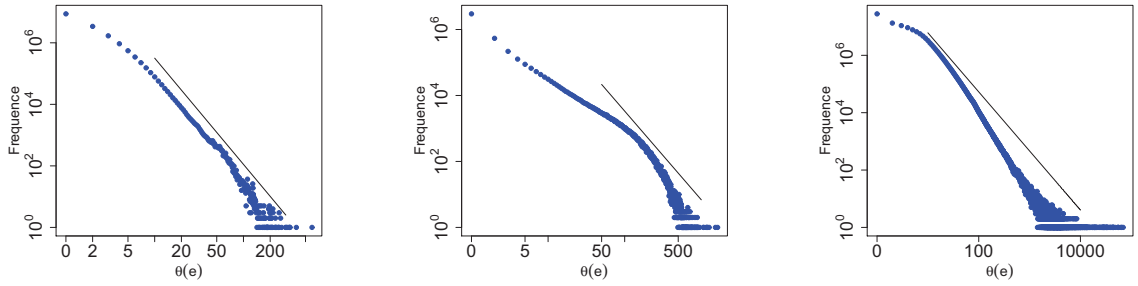
In this section, we first address the power-law distribution of initial $\theta(e)$ in real-world graphs. With the help of the skewed distribution, $\overline{\rho}$ can be small enough even a random partition is used. Then, we discuss the influences of different partition schemes on PETA. Finally, we introduce several implementation details.

6.1 Edge-Support Law

Through a bunch of studies on various types of real-world graphs, we find that the relationship between $\theta(e)$'s frequence and $\theta(e)$ satisfies the power-law distribution. We summarize it as the *edge-support law* as follows,

PROPERTY 2. *(EDGE-SUPPORT LAW) The frequency distribution of initial $\theta(e)$ in the real-world graphs satisfies the power-law, i.e.,*

$$P(\theta(e)) \propto (\theta(e) + 1)^{-\alpha}, \alpha > 2.$$



(a) us-patent (b) wikitalk (c) dbpedia

Figure 5: Initial $\theta(e)$ ~Frequence distribution on various graph

Figure 5 shows the frequence distribution of $\theta(e)$ in us-patent, wikitalk, dbpedia as the representatives. They clearly illustrate that most of edges' support values are around zero, and only a few number of edges have a large $\theta(e)$ in the real-world graphs. Assume the edge-support law is a perfect power-law distribution, then the expectation of edge support, $E[\theta(e)]$, is

$$\begin{aligned} E[\theta(e)] &= \frac{\sum_{\theta=0}^{\theta max} \theta(\theta+1)^{-\alpha}}{\sum_{\theta=0}^{\theta max}(\theta+1)^{-\alpha}} \\ &\approx (\alpha-1)\int_0^{\infty}(\theta+1)^{-\alpha+1}d\theta \\ &= \frac{\alpha-1}{\alpha-2}, \alpha > 2, \end{aligned}$$

where $\sum_{\theta=0}^{\theta max}(\theta+1)$ and $\alpha-1$ are both the normalized factor.

This implies that $E[\theta(e)]$ is a relatively small value when the perfect power law holds. Although in reality, the edge-support law is not the perfect power-law, $E[\theta(e)]$ is indeed small comparing with the scale of the graph. In Table 2, the $E[\theta(e)]$ column shows average $\theta(e)$ across several popular graph datasets. The largest one is just 20.0 from the livejournal graph which contains 4.8M vertices and 43M edges.

Graph	W	$E[\theta(e)]$	$\overline{\rho}_{est}$	$\overline{\rho}_{rand}$	$\overline{\rho}_{METIS}$
livejournal	32	20.00	20.74	8.99	1.77
us-patent		2.36	3.25	3.13	1.19
wikitalk		5.93	7.53	4.52	3.31
dbpedia		7.61	9.11	6.77	2.10
livejournal	16	20.00	19.52	6.72	1.62
us-patent		2.36	3.14	2.95	1.15
wikitalk		5.93	7.15	3.65	2.67
dbpedia		7.61	8.63	5.60	1.85

Table 2: Estimated and actual $\overline{\rho}$ in real-world graphs

Since $\frac{3|\triangle|}{m} = E[\theta(e)]$, on basis of Corollary 1, the Edge-Support Law ensures a small upper bound of $\overline{\rho}$. Table 2 also lists the estimated $\overline{\rho}_{est}$ and accurate $\overline{\rho}_{rand}$ when the graph is randomly partitioned into 16 and 32 subgraphs. We find that even the random partition scheme is applied, $\overline{\rho}$ is already small enough to be practical. For instance, the $\overline{\rho}_{rand}$ of livejournal can be decreased to 8.99 with 32 partitions.

6.2 Partition Influence on PETA

As analyzed in Section 5, a small $\overline{\rho}$ reduces the space and communication cost, also improves computing efficiency. In general, we can give the formulation of computing $\overline{\rho}$ with edge-cut ratio, γ, as

$$\overline{\rho} = 1 + \gamma + \gamma E[\theta(e)],$$

where the term $1+\gamma$ represents the original edges and cut ones, while the term $\gamma E[\theta(e)]$ means whenever an edge is cut, the triangles around that edge are also cut, thus bringing external edges.

Hence, a good partition, which has small γ, can help decrease $\overline{\rho}$ further. This implies a good partition will improve the efficiency of PETA from space cost, communication cost and the computation complexity. Actually, a good partition with small edge cut ratio reduces the number of iterations in heuristics as well. First, let's define θ^c_{min} (θ^c_{max}) is the minimal (maximal) $\theta(e)$ among all the cut edges. Then the number of iterations for any k belongs to the following three cases,

- $k-2 \leq \theta^c_{min}$: No cut edges are deleted, so the algorithm finishes after the first iteration in detection phase.
- $\theta^c_{min} < k-2 \leq \theta^c_{max}$: A portion of cut edges are deleted, and the algorithm may need other several iterations after the first iteration in detection phase.
- $\theta^c_{max} < k-2$: All cut edges are removed during the first iteration in the detection phase. One additional iteration is sufficient to finish the whole computation.

Except the case $\theta^c_{min} < k-2 \leq \theta^c_{max}$, the number of iterations of the other cases are deterministic, which is three and four, respectively. For the second case, a smaller $\theta^c_{max} - \theta^c_{min}$ tends to get fewer iterations. A good partition happens to meet this heuristic rule in most circumstances. Moreover, due to the local community and clustering properties [7, 17] of the real world graphs, those graphs usually have natural groups, which results in a small γ.

However, in parallel computing, another important factor, that affecting the performance, is the balance. It is really difficult to achieve both balance and the minimal edge cuts at the same time [10]. But for PETA, it is sufficient to obtain a partition in balance with a proper γ. For instance, simply decreasing γ from 100% to 50%, $\overline{\rho}$ can be improved at least 25%. In this paper, we create the core edge balanced partition with an applicable edge cut ratio, whose implementation is described in the next subsection, to improve the performance of PETA. Since all the computations focus on the local core edges, we balance the core edges across the partitions, not the vertices any more, for the k-truss detection. After applying the good partition scheme, we find that the six $\overline{\rho}$ of popular datasets are all below 3.5, which is shown in the last column of Table 2.

To sum up, the Edge-Support law guarantees our parallel algorithm have a small upper bound of the space cost in real-world

graphs. The core edge balanced partition scheme with a small edge cut ratio improves it further in practice.

6.3 Implementation Details

We implement PETA on giraph-1.0.0 [1] by extending the vertex-centric model into the subgraph-oriented model. In the subgraph-oriented model, the logic of the subgraph algorithm follows Algorithm 2. Besides, the algorithm separately manages the core edges and external edges of a TC-subgraph. This is because the external edges only trigger the execution of subgraph program for each iteration, and during the local maximal k-truss detection, the core edges are sufficient for the computation. The fundamental operation is implemented as the index-based solution, in which the local k-truss algorithm enumerates triangles for an edge with hashtable supported. These flexible implementations are benefit from the subgraph-oriented model. The communications between computing nodes are asynchronously executed by message passing mechanism, so the computation and communication happen concurrently.

The default partition scheme is the random partition, $\mathcal{R}$ partition for short. Furthermore, we use a good partition, which balances the core edges with a small edge cut ratio, to improve the performance of PETA. Currently, we simply use METIS [9] to generate the reasonable partition instead of developing a new partition algorithm, and we call it as $\mathcal{M}$ partition. Since the METIS is unable to balance the core edges directly, we assign each vertex's degree as its weights, and balance the degree as an indicator for core edge balance. The degree balance factor is limited in 1%. Table 3 shows the γ and actual core edge balance factor, when the graphs are partitioned into 32 subgraphs by the Random and METIS methods. For a heuristic partition strategy, like $\mathcal{M}$ partition, the system requires an index to record the vertex-partition mapping schemes. Since such a map costs a small footprint of memory, (i.e., $\sim$130M for dbpedia dataset), in current version of PETA, each computing node stores a copy of the map to avoid communication.

Graph	γ		Actual balance factor	
	$\mathcal{M}$	$\mathcal{R}$	$\mathcal{M}$	$\mathcal{R}$
wikitalk	53.57%	96.87%	40%	1%
us-patent	17.93%	96.88%	1%	0%
livejournal	25.55%	96.89%	4%	1%
dbpedia	32.37%	96.88%	6%	0%

Table 3: The statistics of graphs with 32 partitions.

7. PERFORMANCE EVALUATION

In this section, we evaluate the performance of PETA for the k-truss detection. The experimental environment is described in the next subsection. We show the performance of PETA can be improved by the edge balanced partition scheme, and verify the efficiency of PETA by comparing with three other parallel solutions. At last, the scalability of PETA is presented as well.

7.1 Environment Setup

Graph	$\|V\|$	$\|E\|$	$\|E\|/\|V\|$	k_{max}
wikitalk	2.4M	4.7M	1.95	53
us-patent	3.8M	16.5M	4.38	36
livejournal	4.8M	42.9M	8.84	362
dbpedia	17.2M	117.4M	6.84	52

Table 4: Graph statistics

All experiments are conducted on a cluster with 23 physical nodes. Each node has two 2.60GH AMD Opteron 4180 CPUs with 48GB memory and a 10T disk RAID. Four graph datasets are used in this paper. All the graphs have been processed into undirected ones and the meta-data of graphs are listed in Table 4. The dbpedia is available on KONECT[1], and the remained graphs are downloaded from SNAP[2] project.

Algorithms and implementations. The implementation of PETA and $\mathcal{R}$ ($\mathcal{M}$) partition schemes used in experiments are described in Section 6.3. In addition, three other parallel solutions are compared. They are orig-LQ, impr-LQ and Cohen-MR. **Orig-LQ** is the original L. Quick's solution [18] and **impr-LQ** is the improved one as presented in Section 3.1. Since the classic solution of counting triangle around an edge is slower than the index-based one (Section 2.2), we implement both algorithms in subgraph-oriented model, thus the index-based solution can be implemented as well, but the logical k-truss detection framework (Section 3) is reserved. **Cohen-MR** is short for Cohen's MapReduce solution and it is implemented on Hadoop-0.20.2 using the efficient triangle counting approach [2], which counts the triangle in a single map-reduce round.

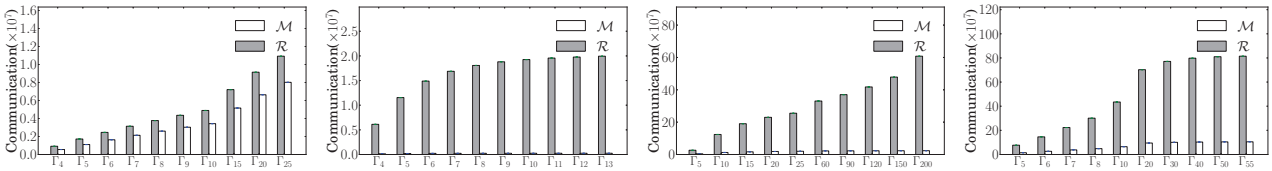
Measures. In this paper, we evaluate the algorithms in four different aspects, which are *space cost*, *communication cost*, *number of iterations* and *the overall performance*. Since the space cost is only related to partition schemes and is measured by edge replicating factor, $\overline{\rho}$, which has been presented in Section 6.1, we do not illustrate it again in the following sections and readers can refer the metric $\overline{\rho}$ in Table 2. The smaller $\overline{\rho}$ implies a better space cost. Additionally, the value of $\overline{\rho}$ is one for all baselines. Thus, we can see that PETA has small space overhead when $\mathcal{M}$ partition method is used.

Threshold selection strategy. For the k-truss detection problem, a regular input parameter besides the graph is the threshold k. We denoted k_{max} as the maximal k that the k-truss of the graph is non-empty. Table 4 also shows the k_{max} of the four graphs. With the increase of k, the size of k-truss in a graph is shrinking, and the more edges will be deleted. Consequently, the measured metrics vary as well. In the following experiments, we only visualize the corresponding costs with several selected k. The strategy of selecting k satisfies that the sizes of results are almost uniformly distributed across all possible sizes. For instance, in Figure 6(c), we select ten different k (x-axis) for livejournal, thus the percentages of vertices of the selected k-trusses change from 36.79% to 0.06%, while the percentages of edges vary from 54.72% to 1.11%. Note that we use k and Γ_k interchangeably to refer to the x-axis in following figures.

7.2 Performance of PETA on Different Partition Schemes

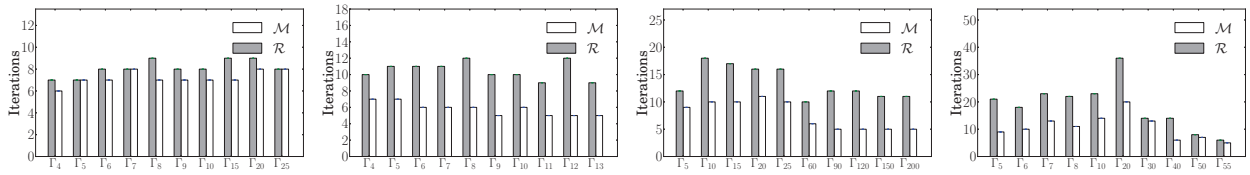
To verify the benefit of an edge balanced partition with a reasonable edge cut ratio (Section 6.2), we evaluate the performance of PETA on random ($\mathcal{R}$) and METIS-based edge balanced ($\mathcal{M}$) partition schemes. Since the different W has the similar enhancement of $\overline{\rho}$ (Table 2) which leads to similar outcomes, we only present the results of 32 partitions, whose additionally statistics are listed in Table 3. The experiment results demonstrate that a partition with small edge cut ratio reduces the communication cost sharply and decreases the number of iterations as well. Therefore, the partition with small γ improves the overall performance. However, the partition with small γ cannot always guarantee to achieve a better overall performance, because the imbalance of partitions may ruin the benefit brought by the small γ in the parallel computing. The detailed explanations are presented from three different metrics.

[1]http://konect.uni-koblenz.de
[2]http://snap.stanford.edu



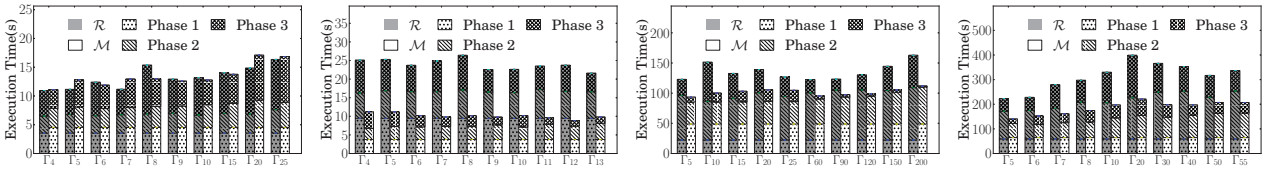
(a) wikitalk (b) us-patent (c) livejournal (d) dbpedia

Figure 6: Communication cost in detection phase. The y axis illustrates the number of sending messages.



(a) wikitalk (b) us-patent (c) livejournal (d) dbpedia

Figure 7: Number of iterations



(a) wikitalk (b) us-patent (c) livejournal (d) dbpedia

Figure 8: Overall performance. From bottom to top, each bar is split into phase one, phase two and phase three.

Communication performance. Here we show that the partition with small edge cut ratio can enhance the communication efficiency. In Figure 6, across four different graphs, the number of sending messages in detection phase is sharply reduced when the good partition method $\mathcal{M}$ is used. For example, $\mathcal{M}$ method can reduce 85.11% communications compared with $\mathcal{R}$ partition scheme to detect Γ_{10} in dbpedia. This is because more invalid edges can be removed in one iteration without extra communications on a partition with smaller γ. The improvement in Figure 6(a) is not that large as others, because the close $\overline{\rho}$ between partition $\mathcal{R}$ and $\mathcal{M}$ (Table 2) indicates the similar cut triangles, which incurs the real communication cost. Besides, the communication tends to increase for a larger threshold k (x-axis), due to more external edges are likely to be removed as k goes up.

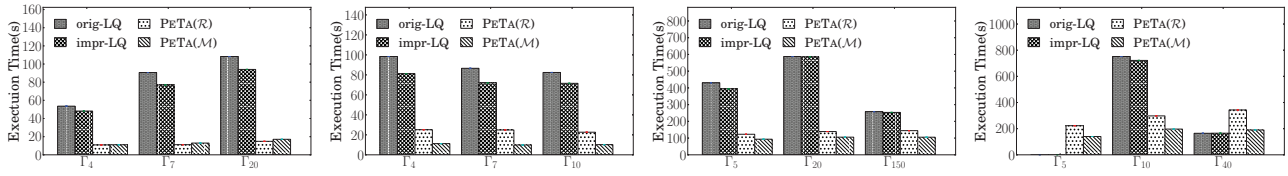
Number of iterations. Usually, the number of iterations can be decreased by improving the partition quality as well. In Figure 7(d), to detect Γ_{20} in dbpedia, the number of iterations decreases from 36 to 20 when the partition method changes from $\mathcal{R}$ partition to $\mathcal{M}$ partition. Since the good partition only heuristically (does not guarantee) reduces the number of iterations, the improvement can be small sometimes. Like detecting Γ_5 in wikitalk in Figure 7(a), the improvement is zero. Another interesting observation is that the number of iterations may not increase with the increase of the threshold. For instance, it takes 10 iterations when detecting Γ_{10} in livejournal with good partition $\mathcal{M}$, while only 6 iterations are needed for the Γ_{60} detection, in Figure 7(c). This is because for a small k, some edges are deleted in several successive iterations, while those can be eliminated in one iteration when the threshold k is increased.

Overall performance. Since the overall performance is affected by several factors, such as communication, balance factor, iterations, we analyze it in three different phases. *Phase one* is the first two iterations which is responsible for exchanging the triangle information across the parallel framework. *Phase two* is the third iteration, which materializes the TC-subgraphs and executes the first detection iteration. The *last phase* completes the remaining detection.

Figure 8 illustrates the overall performance of k-truss detection with different partition schemes. Since phase one (bottom part) is the same with classic triangle counting on graph computing systems, it will enumerate the pairs of every vertex's neighbors. Thus, the influence of imbalance is quadratic. In Figure 8, except the us-patent, all other graphs with $\mathcal{M}$ partition perform worse than the ones with $\mathcal{R}$ partition. This is caused by the imbalance of $\mathcal{M}$ partition. Refer to Table 3, livejournal, dbpedia, wikitalk have various imbalance of core edges from 4% to 40%. Although livejournal only has a 4% imbalance, the performance of phase one is still decreased by around two times. However, the us-patent achieves the similar balance on both partition $\mathcal{M}$ and $\mathcal{R}$. The imbalance of partition heavily affects the performance of phase one.

The cost of phase two (middle part) is mainly determined by the number of incoming messages for materializing the TC-subgraph and the size of eliminated edges in the first detection iteration. For the fixed threshold k, except the wikitalk, other graphs perform better with a small edge-cut ratio. According to $\overline{\rho}$ in Table 2, these three graphs achieve a smaller space cost when the $\mathcal{M}$ partition is used, so fewer incoming messages are processed for creating external edges. However, in wikitalk, the slight improvement of the space cost between two partition schemes reveals that the cut triangles are similar, and this leads to the similar performance of the phase two. When the threshold k increases, the cost of phase two goes up as well. This is caused by more edges are deleted in the first detection iteration. A good partition which indeed decreases the space cost helps enhance the performance of phase two.

In the phase three (top part), besides a smaller $\overline{\rho}$ leads to better performance, fewer number of iterations improve the performance as well. For example, though the wikitalk has similar $\overline{\rho}$ between two partition schemes, for Γ_8 and Γ_{15} detection, the $\mathcal{M}$ partition can still achieve better performance (Figure 8(a)) in phase three by reducing the number of iterations (Figure 7(a)).



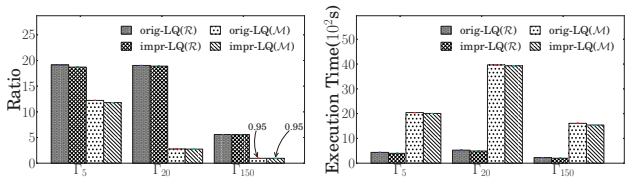
(a) wikitalk (b) us-patent (c) livejournal (d) dbpedia

Figure 9: Overall performance comparison.

To sum up, if a partition with small γ can reduce the $\overline{\rho}$ with affordable imbalance overhead, the overall performance can be improved as well. Otherwise, the random partition is a reasonable choice, such as the wikitalk case, though it has a higher γ.

7.3 Performance Comparison

Here we compare PETA with orig-LQ, impr-LQ and Cohen-MR in communication cost, number of iterations and overall performance respectively. All the experiment results reveal that PETA surpasses the state-of-the-art solutions in almost all aspects of the measurement even if the random partition scheme is used. Unless detecting the k-truss with large k, the existing solutions can achieve a comparable performance against PETA. Since the results are similar on various graphs and the space is limited, the communication comparison is only presented on livejournal dataset, while the number of iterations comparison is shown on dbpedia dataset.



(a) Communication ratio (b) Computation comparison

Figure 10: Results on livejournal

Communication comparison. PETA detects k-truss by constructing TC-subgraphs once and applying the seamless detection, thus it avoids the repeated triangle counting, which usually incurs huge communication. Figure 10(a) shows the communication cost of orig-LQ and impr-LQ on livejournal with different partitions, and the cost has been normalized to the one of PETA. It is clear to see that PETA save the communication cost by 2X to 19X on different partition schemes. However, when k is large enough that the majority of triangles are eliminated at the first detection iteration and a few triangles are remained, the communication overhead of repeated triangle counting can be small, so that the total communication cost can be as small as or smaller than the one of PETA. For example when detecting Γ_{150} on livejournal with $\mathcal{M}$ partition, impr-LQ saves about 5% communication compared with PETA.

Number of iterations comparison. On all the datasets, PETA can reduce the number of iterations around 3X to 20X on average by avoiding the repeated triangle counting and finding local maximal k-truss. Table 5 lists the number of iterations used for detecting Γ_5, Γ_{10}, Γ_{40} on dbpedia. For instance, when finding Γ_5, PETA($\mathcal{M}$) only takes nine iterations, while orig-LQ needs 2212 iterations, which is 246X higher than the former. The number in parenthesis means the count of logical iterations in the original k-truss detection framework (steps 1-4 in Section 3) of three baselines. Compared with this number, PETA still achieves about 4X to 60X improvement, which is benefit from finding the local maximal k-truss. Moreover, the results reveal the existing framework is partition independent. A good partition will not decrease the number of iterations, while the one in PETA can be reduced by a good partition as analyzed in Section 7.2.

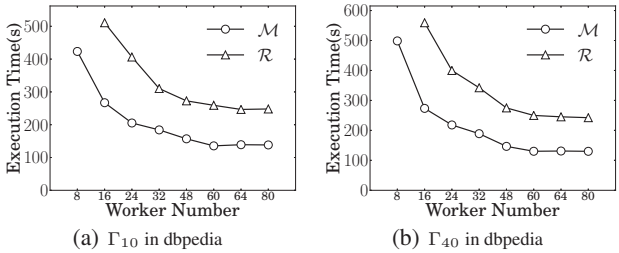
k-truss	Orig-LQ $\mathcal{R}\&\mathcal{M}$	Impr-LQ $\mathcal{R}\&\mathcal{M}$	Cohen-MR	PETA $\mathcal{R}$	PETA $\mathcal{M}$
Γ_5	2212(503)	1509(503)	1006(503)	21	9
Γ_{10}	272(68)	204(68)	136(68)	23	14
Γ_{40}	112(28)	84(28)	56(28)	14	6

Table 5: Number of iterations on dbpedia.

Overall performance comparison. With the help of TC-subgraph and seamless detection, PETA achieves a highly efficient performance on various graphs, no matter random partition or edge-balanced partition is used. Figure 9 illustrates the overall performance of PETA($\mathcal{R}$), PETA($\mathcal{M}$), orig-LQ and impr-LQ. Since Cohen-MR and detecting Γ_5 on dbpedia via orig-LQ and impr-LQ are at least 10X slower than the corresponding performance of PETA($\mathcal{M}$), their results are not visualized for figures' clarity. It is easy to figure out that PETA performs better than all the baselines. For instance, when detecting Γ_{20} on livejournal, orig-LQ is 5.6X slower than PETA($\mathcal{M}$). The results on other three graphs are similar. Although impr-LQ performs better than orig-LQ, it still cannot win over the PETA because of the repeated triangle counting. However, when detecting k-truss for a large k, orig-LQ and impr-LQ may achieve a comparable performance to PETA. For instance, when detecting Γ_{40} in dbpedia, orig-LQ and impr-LQ has similar performance to PETA($\mathcal{M}$), and perform better than PETA($\mathcal{R}$). This is because, with a large k, most of triangles are eliminated early, then the overhead of repeated triangle counting is small as mentioned before.

In addition, the performance of orig-LQ and impr-LQ in Figure 9 is measured on random partition, due to both baselines suffer from the good partition $\mathcal{M}$. Figure 10(b) shows the performance of orig-LQ and impr-LQ on partition $\mathcal{M}$ is 5X to 8X slower than the one on partition $\mathcal{R}$. The reason is that the good partition tends to cluster the local communities in the same local partition which often belongs to the same k-truss, thus the good partition leads to a heavy imbalance of the distribution of a k-truss, while this type of imbalance seriously hinders the performance of the repeated triangle counting.

7.4 Scalability Testing



(a) Γ_{10} in dbpedia (b) Γ_{40} in dbpedia

Figure 11: Scalability of PETA on different partition schemes.

Finally, to evaluate the scalability of PETA, we run experiments of detecting Γ_{10} and Γ_{40} on dbpedia dataset with partition schemes $\mathcal{R}$ and $\mathcal{M}$. Figures 11(a) and 11(b) illustrate that, with the increase of the number of workers, the performance of PETA improves gracefully on both partition scheme $\mathcal{R}$ and $\mathcal{M}$. For example,

when detecting Γ_{10} on dbpedia with $\mathcal{M}$ partition strategy, it spends 424s with 8 workers while the cost is 184s when 32 workers are provided. However, when the number of workers increases, each partition tends to be in small size and the communication cost begins to dominate the cost for each partition, thus the improvement is not ideal linear. Furthermore, the limited number (resources) of physical computers restricts the performance improvement, so the more workers run on a single node, the slower each worker operates. When the number of workers exceeds 60 for dbpedia, the performance improvement becomes slight.

8. RELATED WORK

Cohesive subgraph is one of key components that capture the latent properties in a graph. Various definitions on cohesive subgraph have been introduced, such as clique, n-clique, n-clan, n-club, k-plex, k-core, which were briefly reviewed in Section 1. In this paper, we focus on the newly proposed k-truss based cohesive subgraph.

Cohen [4] first introduced the notation of k-truss based cohesive subgraph and designed a polynomial in-memory algorithm to find the maximal k-truss of a graph in $\mathcal{O}(\sum_{v \in V} d^2(v))$. Recently, Wang et al. [24] improved the Cohen's algorithm through efficient in-memory triangle counting algorithm [11], and the improved algorithm achieves $\mathcal{O}(m^{\frac{3}{2}})$ time complexity with the same space $\mathcal{O}(n+m)$ to that of Cohen. Whereas, it is infeasible to process a big graph in-memory on a single node. Wang et al. [24] also developed two disk I/O-efficient algorithms, i.e., bottom-up approach and top-down approach. The bottom-up external algorithm finishes the computation in $\mathcal{O}((\frac{m}{M} + k_{max})scan(|G|) + \sum_{H \in H} |\triangle_H|)$ I/Os. Feng et al. [27] designed a similar I/O-efficient external algorithm, which is facilitated by graph databases.

However, all aforementioned algorithms are limited on a single node, and cannot scale to the parallel computing framework. Cohen introduced the first parallel k-truss detection algorithm via MapReduce framework in [5]. Due to the limitation of MapReduce framework for the iterative jobs, Cohen's parallel algorithm is inefficient in practice. L. Quick [18] developed the parallel k-truss detection algorithm in Pregel-like graph computing systems. Since both parallel solutions follow the same framework which requires repeated triangle counting and generates huge communication with large number of iterations, they yield a poor performance in practice. Unlike the previous methods, PETA proposed in this paper is built on TC-subgraph, and avoids the reductant communication and computation.

Another related research field is the parallel graph computing systems, where the vertex-centric model was used. A lot of similar systems [14, 12, 19] popped up recently. Although, the vertex-centric model has already expressed a bunch of graph algorithms, it is still not flexible to some graph applications [22]. This is because the vertex-centric model restricts the algorithm to operate the local graph in view of a single vertex. In this paper, we extended the vertex-centric model into the subgraph-oriented model to efficiently implement our PETA.

9. CONCLUSION

In this paper, we developed a parallel and efficient truss detection algorithm, named PETA. The novel algorithm has several advantages over the state-of-the-art parallel solutions, such as bounded communication cost, computation complexity with the same order as the best known serial solutions, and minimized number of iterations. Moreover, we extended the vertex-centric model into subgraph-oriented model for efficiently implementing PETA in parallel. At last, we conducted comprehensive experiments to verify the efficiency of PETA against the existing approaches.

ACKNOWLEDGMENTS

This research is sponsored by the National Natural Science Foundation of China under Grant No. 61272155. Furthermore, Lei Chen's work is supported in part by the Hong Kong RGC/NSFC Project N_HKUST637/13, National Grand Fundamental Research 973 Program of China under Grant 2012-CB316200, Microsoft Research Asia Gift Grant and Google Faculty Award 2013.

10. REFERENCES

[1] Giraph. https://github.com/apache/giraph.
[2] F. N. Afrati, D. Fotakis, and J. D. Ullman. Enumerating subgraph instances using map-reduce. ICDE, 2013.
[3] R. D. Alba. A graph-theoretic definition of a sociometric clique. J. Math. Sociol., 1973.
[4] J. Cohen. Trusses: Cohesive subgraphs for social network analysis. NSA., 2008.
[5] J. Cohen. Graph twiddling in a mapreduce world. Comput. Sci. Eng., 2009.
[6] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. OSDI, 2004.
[7] M. Girvan and M. E. J. Newman. Community structure in social and biological networks. PNAS, 2002.
[8] M. S. Granovetter. The Strength of Weak Ties. Am. J. Sociol., 1973.
[9] G. Karypis and V. Kumar. Parallel multilevel graph partitioning. IPPS, 1996.
[10] K. Lang. Finding good nearly balanced cuts in power law graphs. Technical report, 2004.
[11] M. Latapy. Main-memory triangle computations for very large (sparse (power-law)) graphs. Theor. Comput. Sci., 2008.
[12] 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. PVLDB, 2012.
[13] R. Luce and A. Perry. A method of matrix analysis of group structure. Psychometrika, 1949.
[14] 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. SIGMOD, 2010.
[15] R. J. Mokken. Cliques, clubs and clans. Qual. Quant., 1979.
[16] J. Moon and L. Moser. On cliques in graphs. Israel J. Math., 1965.
[17] M. Newman. Detecting community structure in networks. Eur. Phys. J B, 2004.
[18] L. Quick, P. Wilkinson, and D. Hardcastle. Using pregel-like large scale graph processing frameworks for social network analysis. ASONAM, 2012.
[19] S. Salihoglu and J. Widom. Gps: a graph processing system. SSDBM, 2013.
[20] S. B. Seidman. Network structure and minimum degree. Social Networks, 1983.
[21] S. B. Seidman and B. L. Foster. A graph-theoretic generalization of the clique concept. J. Math. Sociol., 1978.
[22] B. Shao, H. Wang, and Y. Xiao. Managing and mining large graphs: Systems and implementations. SIGMOD, 2012.
[23] E. Tomita, A. Tanaka, and H. Takahashi. The worst-case time complexity for generating all maximal cliques and computational experiments. Theor. Comput. Sci., 2006.
[24] J. Wang and J. Cheng. Truss decomposition in massive networks. PVLDB, 2012.
[25] S. Wasserman and K. Faust. Social network analysis: Methods and applications. Cambridge university press, 1994.
[26] D. R. White and F. Harary. The cohesiveness of blocks in social networks: Node connectivity and conditional density. Sociol. Methodol., 2001.
[27] F. Zhao and A. K. H. Tung. Large scale cohesive subgraphs discovery for social network visual analysis. PVLDB, 2013.