International Journal of Software and Informatics, ISSN 1673-7288
http://www.ijsi.org, ijsi@iscas.ac.cn, +86-10-62661048
IJSI, 2022, 12(1): 89–106, doi: 10.21655/ijsi.1673-7288.00277

International Journal of Software and Informatics

Research Article

Distributed Tip Decomposition on Large Bipartite Graphs

Xu Zhou (周旭)[1], Tongfeng Weng (翁同峰)[1], Zhibang Yang (杨志邦)[1], Boren Li (李博仁)[1], Ji Zhang (张吉)[3], Kenli Li (李肯立)[1,2]

[1] (College of Computer Science and Electronic Engineering, Hunan University, Changsha 410082, China)
[2] (National Supercomputing Center in Changsha, Changsha 410082, China)
[3] (Zhejiang Lab, Hangzhou 311100, China)
Corresponding author: Tongfeng Weng, wengtongfeng@hnu.edu.cn

Abstract Tip decomposition has a pivotal role in mining cohesive subgraphs in bipartite graphs. It is a popular research topic with wide applications in document clustering, spam group detection, and analysis of affiliation networks. With the explosive growth of the bipartite graph data scale in these scenarios, it is necessary to use distributed methods to realize its effective storage. For this reason, this paper studies the problem of the tip decomposition on a bipartite graph in the distributed environment for the first time. Firstly, a new relay-based communication mode is proposed to realize effective message transmission when the given bipartite graph is decomposed in a distributed environment. Secondly, the Distributed Butterfly Counting (DBC) algorithm and the Distributed Tip Decomposition (DTD) algorithm are designed. In particular, a controllable parallel vertex activation strategy is proposed to solve the problem of memory overflow when DBC decomposes large-scale bipartite graphs. Finally, the message pruning strategy based on vertex priority and message validity pruning strategy are introduced to further improve the efficiency of the algorithm by reducing redundant communication and computing overhead. The experiment is deployed on the high-performance distributed computing platform of the National Supercomputing Center. The effectiveness and efficiency of the proposed algorithms are verified by experiments on several real datasets.

Keywords bipartite graph; butterfly counting; distributed systems; tip decomposition

Citation Zhou X, Weng TF, Yang ZB, Li BR, Zhang J, Li KL. Distributed tip decomposition on large bipartite graphs. *International Journal of Software and Informatics*, 2022, 12(1): 89–106. http://www.ijsi.org/1673-7288/277.htm

1 Introduction

Bipartite graphs are a special type of graph in which the vertex set can be divided into two disjoint subsets U and V, and edges only exist between the vertices in different subsets, i.e., $G = (U, V, E)$, satisfying $\{e = (u, v) \in E \mid u \in U \wedge v \in V\}$. In recent years, mining

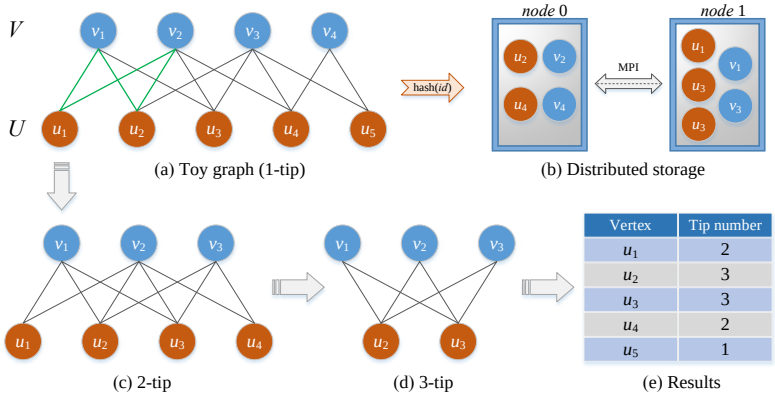
This is the English version of the Chinese article ‘‘面向大规模二部图的分布式 Tip 分解算法研究. 软件学报, 2022, 33(3): 1043–1056. doi: 10.13328/j.cnki.jos.006457”.
Funding items: National Natural Science Foundation of China (61772182, 61802032, 69189338, 62172146, 62172157); Open Fund of Zhejiang Lab (2021KD0AB02); Fund of Key Laboratory of Information System Engineering of National University of Defense Technology
Received 2021-07-01; Revised 2021-07-31; Accepted 2021-09-13; IJSI published online 2022-03-28

techniques for cohesive subgraphs in bipartite graphs have received much attention and have been applied to practical applications such as document clustering[1], author–paper relationship analysis, user–product relationship analysis[2], and spam group detection[3].

There exists a considerable amount of research on decomposition techniques for dense subgraph structures such as k-core[4, 5] and k-truss[6] on unipartite graphs. These techniques can be used to analyze simple graphs obtained from bipartite graph transformation[7], which however results in the loss of the original structural information. In addition, the scale of the graphs increases by 6 orders of magnitude when bipartite graphs are transformed into co-occurrence (projection) graphs[8], which leads to significant additional space storage overhead. Therefore, the existing decomposition techniques for unipartite graphs cannot be applied to large bipartite graph. As a result, the algorithm for decomposing bipartite graphs needs to be investigated.

Butterfly subgraphs are the smallest dense subgraphs in bipartite graphs, which contain a bipartite cluster of four vertices from two vertex sets (U/V), and the number of butterflies corresponding to each vertex is the butterfly degree of that vertex. Thus, when more butterflies are shared by any two vertices on the same side, the connection between them is closer. This structure can be used to measure the dense relationship between vertices on the same side in a bipartite graph. k-tip is defined as a cohesive community $H = (U', V', E')$ satisfying that any $u \in U'$ exists in at least k butterflies. For a particular vertex u, the tip number of u is k if it can only exist in k-tip but not in $k+1$-tip. The tip decomposition algorithm studied in this paper is used to compute the tip numbers of all vertices in a given graph[9].

Sariyüce *et al.*[8] propose a hierarchical stripping-based algorithm for the decomposition of a given bipartite graph. This algorithm iteratively removes the vertex with the smallest butterfly degree, and the butterfly degree of the vertex when it is deleted is its tip number. As shown in Figure 1(a), there are four vertices v_1, v_2, v_3, and v_4 in V and five vertices u_1, u_2, u_3, u_4, and u_5 in U. The vertices u_1, u_2, v_1, and v_2 form a butterfly, i.e., a $(2, 2)$-bipartite cluster. If each vertex $u \in U$ in G exists in at least one butterfly, G is a 1-tip community. Similarly, the subgraphs in Figure 1(c) and (d) are 2-tip and 3-tip communities, respectively. The tip number of each vertex in set U is shown in Figure 1(e).



Vertex	Tip number
u_1	2
u_2	3
u_3	3
u_4	2
u_5	1

Figure 1 Distributed tip decomposition of a bipartite graph

The existing butterfly subgraph counting (hereinafter referred to as butterfly counting) algorithm and tip decomposition algorithm[10–12] are built on the basis that the memory of a single machine can satisfy the storage requirements of the original graph data and intermediate computation results. As the scale of bipartite graphs grows, it is difficult to satisfy the storage requirements for a large bipartite graph with the memory of a single machine. For this purpose,

the distributed storage of the large graph is required, and the corresponding distributed bipartite graph processing technique should be studied. Secondly, there are a large number of vertices with the same id in two vertex sets due to the special characteristics of the bipartite graph structure, and the existing distributed graph computation system mainly targets simple graphs, where its communication mode depends on the vertex id, so there is a message send/receive mismatch when bipartite graphs are processed, which results in incorrect computation results. Finally, there is no direct edge connection between vertices on the same side of bipartite graphs, while tip decomposition aims to explore the dense relationship between vertices on the same side, which requires a 2-hop message transmission to achieve the butterfly counting algorithm and the tip decomposition algorithm, resulting in a large amount of communication overhead and thus affecting the efficiency of the decomposition.

To address the above challenges, this paper explores the butterfly subgraph counting and tip decomposition algorithms for large bipartite graphs in a distributed environment, in which the distributed storage environment can effectively solve the original data storage of large bipartite graphs and the storage of intermediate computation results. For distributed computing environments, this paper designs a relay-based communication mode, which forwards messages through relay vertices to ensure the effectiveness of message delivery, so as to break the limitation that existing distributed graph computing systems rely on the vertex id for message delivery. Specifically, for a given bipartite graph $G = (U, V, E)$, the tip numbers of all vertices in U are computed. The vertices in V are used as relay vertices, which do not need to perform computation and are only responsible for forwarding the received messages, while the vertices in U perform different computation functions as needed. Since the relay vertex (V-side vertex) can directly obtain the information of the vertex to be decomposed (U-side vertex), such as vertex degree and tip number, the message pruning strategy can be designed depending on these attributes to reduce the additional communication overhead and thus the computation in the next round of iterations. Accordingly, this paper proposes a Distributed Butterfly Counting (DBC) algorithm relying on the relay-based communication mode and introduces a message pruning strategy based on vertex priority to reduce the redundant communication overhead. Then, with the results of butterfly counting, the paper designs a Distributed Tip Decomposition (DTD) algorithm to calculate the tip values of all vertices. In the tip decomposition process, the vertex is stored hierarchically by the butterfly degree through maintaining a butterfly tree *BFTree*. Finally, according to the minimum key of the *BFTree*, the vertex in the corresponding leaf is peeled, and when the *BFTree* is empty, the tip numbers of all vertices can be obtained.

The main contributions of this paper are as follows:

- a relay-based communication mode is designed and a distributed graph computation system for the analysis of cohesive subgraphs in bipartite graphs is constructed;
- an efficient DBC is proposed. In particular, to address the memory overflow when DBC decomposes large bipartite graph data, this paper introduces a controllable parallel vertex activation strategy for efficient computation of butterfly counting while improving the algorithm parallelism;
- DTD based on the *BFTree* is proposed, and a vertex priority-based message pruning strategy is introduced to reduce redundant communication and computation overhead so that the efficiency of the algorithm can be further improved; and
- experiments are deployed on the high-performance distributed computing platform of the National Supercomputing Center to verify the algorithm effectiveness and efficiency on several real datasets.

The paper is organized as follows: Section 2 summarizes the related work and introduces the algorithms for distributed graph computing systems, butterfly counting, and tip decomposition

problems; Section 3 gives the related definitions and problem definitions used in this paper; Section 4 presents the proposed relay-based communication mode and DBC; Section 5 describes the DTD; Section 6 introduces the experimental results and analysis; finally, Section 7 concludes this paper with an outlook.

2 Related Work

This section introduces the related work on distributed graph computing systems and butterfly counting algorithm and tip decomposition algorithm for bipartite graphs.

2.1 Distributed graph computing systems

Distributed graph computing systems are mostly designed on the basis of the BSP model, which is a computational model suitable for graph computing. As shown in Figure 2, the superstep is the basic iterative unit, which consists of three steps: receive messages, computing, and send messages. At the end of each superstep, there is a global barrier for synchronizing messages.

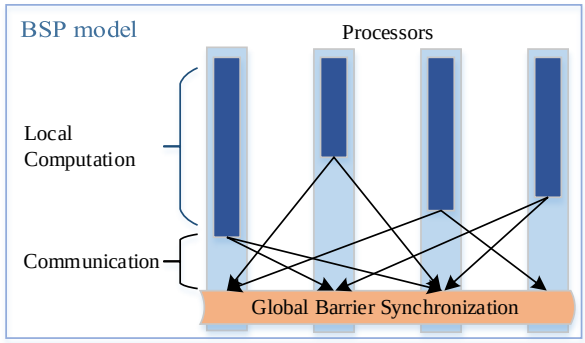


Figure 2 BSP model

On the basis of the BSP model, vertex-centric graph computing systems such as Pregel[13], Giraph[14], GraphX[15], and Pregel+[16] have been proposed one after the other. For large graph data, Yu *et al*.[17] reviewed the key issues in the computational model, communication mechanism, segmentation strategy, index structure, and fault-tolerance management in large graph processing from the two aspects of graph data management and processing mechanism. Lu *et al*.[18] proposed two feature parameters, variation coefficient of degree and slice accessibility, for optimizing the performance of graph computing system research platform and online segmentation algorithms loading feature sensing in a CPU+GPU heterogeneous environment. Zhang *et al*.[19] proposed a framework for evaluating fault-tolerance techniques based on three dimensions of cost, efficiency, and quality according to the uncertainty factors and robustness problems in distributed graph computing frameworks.

Compared with graph computing systems[20–22] and graph algorithms[23–25] for simple graphs, systems and graph algorithms designed to handle bipartite graphs are few. Chen *et al*.[26] proposed a vertex-centered system BiGraph implemented on PowerGraph to efficiently partition bipartite graphs for machine learning algorithms. Liu *et al*.[27] designed an efficient index construction and maintenance algorithm for computing (α-β)-core on bipartite graphs that can be used for the analysis on dynamic bipartite graphs. However, these systems and algorithms do not introduce a special optimization strategy for the butterfly counting problem.

In summary, existing graph computing systems and graph algorithms cannot be used effectively to deal with the butterfly counting and tip decomposition problems.

2.2 Butterfly counting and tip decomposition of bipartite graphs

Butterfly is an important graph structure in bipartite graphs, which has been widely used in bipartite graph structure mining such as k-tip community detection. In this paper, we focus on butterfly counting and tip decomposition problems on bipartite graphs.

To optimize the butterfly counting algorithm, Chiba and Nishizeki[28] designed a vertex priority-based counting algorithm to optimize the edge search process in graphs. Wang *et al*.[9] further introduced a cache-aware strategy into a vertex priority-based algorithm to reduce the time complexity. Sanei-Mehri *et al*.[11] proposed an approximation algorithm for butterfly counting and gave a proof for its accuracy.

To speed up tip decomposition, Shi and Sun[9] proposed the ParButterfly framework to design parallel algorithms for butterfly counting and tip decomposition based on OpenMP. Lakhotia *et al*.[12] proposed a parallel tip decomposition algorithm with shared memory which consists of coarse-grained decomposition and fine-grained decomposition to obtain higher parallelism and improve the resource utilization of multi-core systems.

The above-mentioned algorithms are based on the assumption that the memory of a single machine is large enough to store a given bipartite graph and the intermediate computation results. However, with the explosive growth of graph scale, the memory bottleneck of a single machine can no longer be ignored. Multiple machines not only provide scalable distributed storage space but also have high-performance computational resources for improving the parallelism of algorithms.

In addition, although the existing algorithms can efficiently handle butterfly counting and tip decomposition problems, vertex relabeling and reordering operations introduce significant communication overhead when the algorithms are applied to distributed systems. For example, in Figure 1(a), vertices are stored in clusters in a distributed manner according to the hash value of their id (cf. Figure 1(b)). Existing algorithms need to relabel the vertices in U in descending order of degree. In node 0, the label of u_4 is smaller than that of u_2 because they have the same degree and the id of u_4 is larger than that of u_2. Similarly, label $\mathrm{label}(u_3) < \mathrm{label}(u_5) < \mathrm{label}(u_1)$ in node 1. These are the local labels of vertices in each machine. If we want to get the global labels of all vertices, the local labels need to be compared and reordered from the largest to the smallest for different machines. This operation has a high time complexity and will incur plenty of communication overhead.

Table 1 Summary of notations

Name	Description
$G = (U, V, E)$	A bipartite graph consisting of vertex sets U, V, and an edge set E
N_u	Neighbors of vertex u
$d_G(u)$	Number of neighbors of vertex u
$H = (U', V', E')$	A subgraph of graph G, abbreviated as H
$\bowtie_u$	A butterfly (subgraph) containing vertex u
$d_{\bowtie}$	Number of butterflies in which a vertex is involved
$d^u_{\bowtie}$	Number of butterflies involving vertex u, i.e., the butterfly degree
$\bowtie^{u_2}_{u_1}$	A butterfly containing u_1 and u_2 vertices
T_u	Tip number of vertex u
H^u_k	k-tip that contains vertex u and has the largest k value

3 Problems and Definitions

This section introduces the theory related to bipartite graph and tip decomposition. Table 1 lists the names of the used symbols and their descriptions.

An undirected bipartite graph $G = (U, V, E)$ is given, in which U and V refer to two disjoint vertex sets and E refers to the edge set in the graph, which satisfies $\{e = (u, v) \in E \mid$

$u \in U \wedge v \in V\}$. The adjacent points of any vertex $u \in U$ exist only in V, which are denoted as N_u, where $d_g(u) = |N_u|$ refers to the degree of u. Since there is no direct edge connection between vertices on the same side, the close relationship between vertices in the same set can be measured by the butterfly subgraph. $d_{\bowtie}$ corresponds to the concept of degree in simple graphs; the number of butterflies containing vertex u is expressed as $d^u_{\bowtie}$; for two vertices u_1 and u_2 in the same butterfly, $\bowtie^{u_2}_{u_1}$ is used to indicate that they can be connected by a butterfly; $|\bowtie^{u_2}_{u_1}|$ can be used to measure the closeness between two vertices u_1 and u_2.

Definition 1 (butterfly subgraph, or butterfly)**.** For a given bipartite graph $G = (U, V, E)$, there exists a subgraph $H = (U', V', E')$, where $U' = \{u_1, u_2\} \subseteq U, V' = \{v_1, v_2\} \subseteq V$. $H = (U', V', E')$ is a butterfly if and only if u_1 and u_2 are common neighbors of v_1 and v_2, i.e., H is a $(2, 2)$-bipartite cluster.

In general, a community consists of a set of closely related homogeneous entities. k-core is a cohesive subgraph structure in simple graphs, where any vertex has at least k neighbors in the same community. In a bipartite graph, vertices on the same side belong to the same class of entities, while each vertex only maintains its neighbors located on the opposite side. Thus, the connectivity between the vertices on the same side cannot be obtained according to the degrees of the vertices. If two vertices are in at least one same butterfly, u_1 can be mapped as a neighbor of u_2. In this paper, we focus on the cohesive structure of vertices on the U-side. On this basis, the k-tip community is defined as follows:

Definition 2 (k-tip)**.** For a given bipartite graph $G = (U, V, E)$, the subgraph $H = (U', V', E')$ is a k-tip when and only when the following three conditions are satisfied:

(1) Connectivity: Each pair of vertices belonging to U' in the k-tip can be connected directly or indirectly by one or more butterflies.
(2) Tightness: Every vertex $u \in U'$ exists in at least k butterflies.
(3) Maximality: There exists no other k-tip involving H.

Definition 3 (tip number)**.** For a given bipartite graph $G = (U, V, E)$, if vertex $u \in U$ can only exist in a k-tip but not in any $(k + 1)$-tip, the tip number of u is k, which is denoted as T_u, and the corresponding k-tip is denoted as H^u_k.

According to the above analysis on bipartite graphs and related definitions, this paper focuses on the following two problems:

Problem 1 (butterfly counting)**.** In a given bipartite graph $G = (U, V, E)$, for each vertex $u \in U$, count the number of butterflies involving u, i.e., H^u_k.

Problem 2 (tip decomposition)**.** For each vertex $u \in U$ in a given bipartite graph $G = (U, V, E)$, the tip number of u is computed.

As shown in Figure 1, vertices u_1, u_2, v_1, and v_2 form a butterfly. For vertex u_2, there are 5 butterflies involving it, including 1 butterfly $\bowtie^{u_2}_{u_1}$, 3 butterflies $\bowtie^{u_3}_{u_2}$, and 1 butterfly $\bowtie^{u_4}_{u_2}$. Depending on the results of butterfly counting of all the vertices, the vertex with the smallest butterfly degree is peeled iteratively until the graph becomes null. In this way, the tip value of each vertex is obtained, and the 1-tip, 2-tip, and 3-tip communities can be constructed accordingly.

4 DBC

Although tip decomposition is similar to core decomposition in simple graphs, the core decomposition algorithm cannot be directly applied as the former due to the special characteristics of bipartite graph structures. For the construction of the relationship between each pair of vertices on the same side, it is necessary to enumerate all butterflies in a bipartite graph (i.e., to find $\bowtie^{u_2}_{u_1}$ for all $u_1 \in U$ and $u_2 \in U$). In this section, DBC based on the vertex-

centric computing model is proposed. In addition, since the original bipartite graph is stored in a cluster in a distributed manner, the existing single-machine butterfly counting algorithm cannot be directly applied to distributed systems. To cope with the limitation that existing distributed graph computing systems cannot be applied to bipartite graphs, this paper introduces a flexible relay-based communication mode.

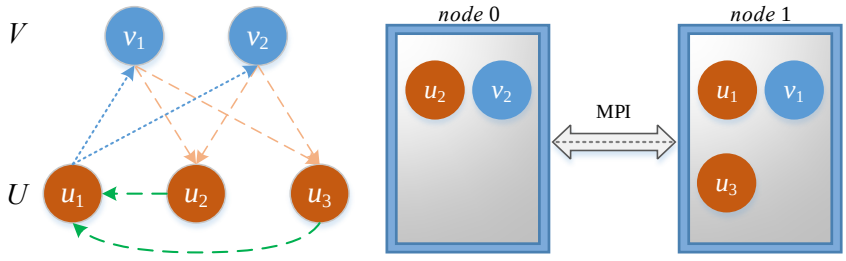
4.1 Relay-based communication mode

Due to the special characteristics of the bipartite graph structure, it is necessary to design a message transmission mode suitable for communication between 2-hop neighbors. For vertex $u \in U$ in a given bipartite graph $G = (U, V, E)$, it only obtains its neighbors belonging to V. For counting the butterflies containing u (i.e., $|\{ \bowtie_u^{u'} \mid u' \in U \}|$), it is necessary to obtain the number of shared neighbors of different 2-hop neighbors.

In a vertex-centered system, vertices can be activated by receiving messages from adjacent points. In light of this idea, a more flexible communication mode for analyzing bipartite graphs on distributed systems is proposed in the following:

Strategy 1 (Relay-based communication mode)**.** Firstly, vertices in U are manually activated to send messages to their neighbors in V. These neighbors will be activated in the next superstep. Then, the activated vertices in V will send messages to activate the 2-hop neighbors of the manually activated vertices in U. Finally, the activated vertices in U are 2-hop neighbors with each other and can send messages to each other directly.

As shown in Figure 3, for the given bipartite graph $G = (U, V, E)$, where U contains 3 vertices u_1, u_2, and u_3, and V contains 2 vertices v_1 and v_2, these vertices are stored in clusters in a distributed manner according to the hash values of their id. For vertex $u_1 \in U$, it is necessary to obtain the shared neighbors between its 2-hop neighbors in U and those in V. u_1 is first activated and sends messages to its neighbors v_1 and v_2. Then, v_1 and v_2 send the *id*1 of u_1 to u_2 and u_3 (except for vertex u_1). u_2 and u_3 will receive message 1 (the id of u_1) from v_1 and v_2 in the next superstep. This means that they have two shared neighbors with u_1 respectively. After that, u_2 and u_3 identify u_1 as their 2-hop neighbors and send their id (2 and 3) and the information of shared neighbors to u_1. The above analysis indicates that four supersteps are required to access and return the information of 2-hop neighbors.



(a) Bipartite graph (b) Distributed storage

Figure 3 Example of relay-based communication mode

4.2 Butterfly counting algorithm

A butterfly containing vertices u_1 and u_2 can be considered as an abstract edge between vertices, and the number of butterflies is used as the weight of the edge. The weight is directly related to the number of shared neighbors of u_1 and u_2.

Theorem 1. For a given bipartite graph $G = (U, V, E)$, if two vertices $u_1 \in U$ and $u_2 \in U$ have n shared neighbors in V, the number of butterflies containing both u_1 and u_2 is $n \times (n-1)/2$.

Proof: According to Definition 1, a butterfly is a $(2,2)$-bipartite cluster containing two vertices on each side and all four possible edges in it. For two vertices u_1 and u_2, if there exist two vertices $v_1 \in V$ and $v_2 \in V$ both adjacent to u_1 and u_2, these four vertices can form a butterfly $\bowtie_{u_1,u_2,v_1,v_2}$. Therefore, any two shared neighbors can form a butterfly with u_1 and u_2. The total number of butterflies containing each pair of vertices is calculated as follows:

$$\text{ButterflyCnt} = C_n^2 = \frac{n!}{(n-2)! \times 2!} = \frac{n \times (n-1)}{2}$$

where n refers to the number of shared neighbors.

As shown in Figure 1, four supersteps are required to obtain the number of shared adjacent points of each vertex with its 2-hop neighbors according to the relay-based communication mode. Vertices u_1 and u_2 have shared adjacent points v_1 and v_2, and there exists a butterfly $\bowtie_{u_1,u_2,v_1,v_2}$ containing them. For vertices u_2 and u_3, vertices v_1, v_2, and v_3 are their shared neighbors. Therefore, u_1 and u_2 participate in three butterflies simultaneously.

It is too expensive to apply existing butterfly counting algorithms on a single machine to distributed systems. This is because they mostly need to reorder and renumber all the vertices, which introduces not only a rather large extra communication overhead but also a lot of vertex migration overhead. To improve the efficiency of DBC, we design a vertex priority-based message pruning strategy to reduce redundant communication.

Strategy 2 (vertex priority-based message pruning strategy)**.** For a given $G = (U, V, E)$, each vertex $u \in U$ only needs to count the number of butterflies with 2-hop neighbors whose id values are larger than its own id. Therefore, the activated vertices in V only need to send the id of the manually activated vertex u to its neighbors whose id value is larger.

As shown in Figure 3, when activating u_2, v_1 and v_2 will receive messages and be activated in the next superstep. According to Strategy 2, v_1 and v_2 only need to send the id (2) of u_2 to u_3, which thus avoids the double computation of butterfly between u_1 and u_2.

Depending on Strategy 1, Strategy 2, and Theorem 1, the main idea of DBC is to find the shared neighbors of each pair of vertices in U. To improve the resource utilization, this paper introduces the idea of parallelism to improve the algorithm efficiency.

As shown in Algorithm 1, the bipartite graph to be decomposed has been stored in the distributed machines according to the hash value of each vertex id, and $U' \subset U$ refers to those vertices stored in the current machine. In the initialization phase, *ActiveVector* (Line 2) is used to store the activated vertices. In the butterfly counting phase, each iteration takes four supersteps. Firstly, m vertices are selected from U' and placed in *ActiveVector*, where m limits the number of initialized vertices to be processed in parallel on each machine. It should be noted that m is used here to implement a controllable parallel vertex activation strategy. Obviously, the larger m indicates the higher parallelism of the algorithm but increases the communication traffic, and thus may lead to memory overflow. Those manually activated vertices send messages to their neighbors (Superstep 1, Lines 5–10). Secondly, according to Strategy 2 (Superstep 2, Lines 12–16), the vertices in V that receive messages from U are activated and the ids of the manually activated vertices are forwarded to their neighbors. Next, the number of butterflies is calculated according to Theorem 1, and the result is sent to the corresponding manually activated vertices (Superstep 3, Lines 18–22). Finally, those vertices activated in the first superstep receive messages from their 2-hop neighbors and obtain the butterfly counting result (Superstep 4, Lines

24–27). After these four supersteps, some of the vertices in U can determine the number of butterflies in which they are located. The butterfly counting algorithm ends when all the vertices in U have gone through the same operation.

Algorithm 1. DBC

Input: Bipartite graph $G = (U, V, E)$.
Output: Butterfly counting results for each vertex $u \in U$.
/* Initialization phase on all machines */
ActiveVector $\leftarrow 0$
/* Butterfly counting phase on all machines */
repeat
  Superstep 1: Manually activate vertex
  Select m vertices from U' for parallel processing and put them into *ActiveVector*
  $U' = U' \backslash$ *ActiveVector*
  for $u \in$ *ActiveVector* **do**
    for $v \in N_u$ **do**
      Send a message to v and activate it in the next superstep
    end for
    Communication synchronization barrier
  end for
  Superstep 2: Forward relay messages
  Vertices in V receive messages from U which are then put into *ActiveVector*
  for $v \in$ *ActiveVector* **do**
    for $u' \in N_v$ **do**
      If u'.*id* is greater than the id of the manually activated vertex, send their *id*s to u'
    end for
  end for
  Synchronization by global barrier
  Superstep 3: Do butterfly counting on two-hop neighbors
  Vertices in U receive messages from V which are then put into *ActiveVector*
  for $u' \in$ *ActiveVector* **do**
    According to Theorem 1, count the number of butterflies with the received id of the manually activated vertex and add the result to $d_{\bowtie}^{u'}$
    Send the result to the corresponding manually activated vertex
  end for
  Synchronization by global barrier
  Superstep 4: Calculate butterfly counting result
  Vertices in U receive messages from U which are then put into *ActiveVector*
  for $u \in$ *ActiveVector* **do**
    Add the butterfly counting result to $d_{\bowtie}^{u}$
  end for
until U' on each machine is not null
return the value of $d_{\bowtie}^{u}$ for each vertex $u \in U$

For each iteration, four supersteps are needed to obtain the number of butterflies containing manually activated vertices. The m vertices in this batch are activated in the first superstep, and the butterfly counting result can be obtained in four supersteps. Therefore, the DBC can converge in the superstep $4 \times U'/m$. The communication cost can be analyzed by the number of messages passed in the whole process. In the first superstep, the manually activated vertices need to send messages to all their neighbors. Then, the vertices in V send messages according to Strategy 2. Therefore, the total number of messages in the first two supersteps is $\sum_{u \in U} \left(N_u + \sum_{v \in N_u} \left(\sum_{u' \in N_v} \mathrm{sgn}(u'.id > u.id) \right) \right)$, in which u is the manually activated vertex in the first superstep and u' is the 2-hop neighbor of u. In addition, the 2-hop neighbors send the result of butterfly counting to the corresponding manually activated vertices in the third

superstep, which is the number of u' whose id is larger than $u.id$.

5 DTD

After the execution of the DBC algorithm proposed in Section 4, each vertex $u \in U$ has its butterfly degree known. Similar to the degree of vertices in simple graphs, $d^u_{\bowtie}$ can only be used to roughly analyze the structure of bipartite graphs. Thus the tip number of each vertex needs to be obtained to explore the finer-grained structural features of bipartite graphs. In this section, DTD based on hierarchical stripping is proposed to obtain the tip numbers of the vertices in U.

Strategy 1 can solve the problem that messages cannot be sent directly between vertices on the same side, on the basis of which the DTD can be studied. For purpose of introducing parallelism to the DTD, the vertex with the lowest number of butterflies is peeled in each iteration, and the tip numbers of the remaining vertices are updated.

Theorem 2. Given a bipartite graph $G = (U, V, E)$ and the minimum $d_{\bowtie}$ value $d_{\bowtie\min}$ in U, the tip number of each vertex in U is not smaller than $d_{\bowtie\min}$.

Proof: There is a bipartite graph $G = (U, V, E)$, and the minimum value of $d_{\bowtie}$ in U is $d_{\bowtie\min}$. According to Definition 2, G is a $d_{\bowtie\min}$-tip, and every vertex $u \in U$ exists in at least $d_{\bowtie\min}$-tip. In combination with Definition 3, it can be deduced that the tip value of each vertex is at least $d_{\bowtie\min}$.

The vertex is peeled to determine its tip value, but in the original graph data, the vertex still exists in the adjacent point lists of its neighbors. In addition, in the vertex-centric mode, the vertices in V cannot know the stripped vertices in U. If the vertices of V continue to forward messages to all their neighbors in subsequent iterations, a large number of redundant messages will be generated.

Strategy 3 (message validity pruning strategy)**.** For a given bipartite graph $G = (U, V, E)$, vertices in V have neighbors only in U. For a vertex $v \in V$, after it receives a message from a censored neighbor $u \in U$, u is set to the inactive state, i.e., 0. When v forwards a message, it is not necessary to consider the adjacent point with state 0.

The main idea of DTD is to peel vertices according to the value of $d_{\bowtie\min}$. To solve the problem that vertices cannot be sorted under distributed storage, this paper constructs a butterfly tree *BFTree* to store the id of the vertices hierarchically. *BFTree* is composed of multiple key-value pairs, where "key" represents the $d_{\bowtie}$ of a vertex, and "value" is a container to store the id of a vertex. The pseudo-code is shown in Algorithm 2.

As shown in Algorithm 2, the *BFTree* is first generated depending on the butterfly degree of each vertex (Lines 5–11). In the first superstep, the global minimum $d_{\bowtie}$ value, *GlobalMinKey*, is obtained from the keys of the *BFTree* (Line 15). The vertex id with the minimum $d_{\bowtie}$ value is stored in *BFTree*[*GlobalMinKey*], which will be censored and send messages to its neighbors (Lines 16–21). In addition, the state of these manually activated vertices is set to be peeled (Line 22). In the second superstep, the vertices in V receive messages and forward the source ids of these messages to the neighbors that have not yet been stripped (Lines 26–34). In the third superstep, the 2-hop neighbors of these manually activated vertices from the first superstep update their $d_{\bowtie}$ according to Theorem 1 (Lines 37–40). According to Theorem 2, the tip values of the still unstripped vertices are at least *GlobalMinKey*. Therefore, if the tip number of a vertex after line 40 of Algorithm 2 is smaller than *GlobalMinKey*, it is set to *GlobalMinKey* (Lines 41–43). It should be noted that updating the $d_{\bowtie}$ values of vertices will affect the structure of the *BFTree*.

The *BFTreeUpdate* function is designed to update the *BFTree* when the $d_{\bowtie}$ of a vertex changes (Line 40). The specific update method is similar to the *BFTree* construction process.

After the first three supersteps, the manually activated vertices in the first superstep have confirmed their tip numbers. The DTD starts another iteration until the *BFTree* is null (Lines 13–46).

Algorithm 2. DTD

Input: Bipartite graph $G = (U, V, E)$, $d^u_{\bowtie}$ value of each vertex $u \in U$.
Output: Tip decomposition result of each vertex $u \in U$.

1. /* Initialization phase on all machines */
2. *ActiveVector* $\leftarrow 0$
3. *GlobalMinKey* $\leftarrow 0$
4. Construct a butterfly tree *BFTree* according to $d_{\bowtie}$ for each $u \in U$.
5. **for** each u in U **do**
6. **if** *BFTree.find*($d^u_{\bowtie}$) = *BFTree.end*() **then**
7. *BFTree*[$d^u_{\bowtie}$] $\leftarrow$ {*u.id*}
8. **else**
9. *BFTree*[$d^u_{\bowtie}$].*push_back*(*u.id*)
10. **end if**
11. **end for**
12. /* Tip decomposition phase on all machines */
13. **repeat**
14. **Superstep 1:** Manually activate vertices
15. *GlobalMinKey* $\leftarrow$ Get the minimal key in *BFTree*
16. *ActiveVector* $\leftarrow$ *BFTree*[*GlobalMinKey*]
17. *BFTree**BFTree*[*GlobalMinKey*]
18. **for** $u \in$ *ActiveVector* **do**
19. **for** $v \in N_u$ **do**
20. Send a message to v, and activate it in the next superstep
21. **end for**
22. u.*peeled* $\leftarrow 1$
23. **end for**
24. Synchronization by global barrier
25. **Superstep 2:** Forward relay messages
26. The vertices in V receive messages from U which are then put into the *ActiveVector*
27. **for** $v \in$ *ActiveVector* **do**
28. **for** $u' \in N_v$ **do**
29. **if** u'.*peeled* $= 0$ **then**
30. Send the ids of manually activated vertices tou'
31. **end if**
32. **end for**
33. **end for**
34. Synchronization by global barrier
35. **Superstep 3:** Update the number of butterflies on 2-hop neighbors
36. The vertices in U receive messages from V which are then put into the *ActiveVector*
37. **for** $u' \in$ *ActiveVector* **do**
38. *preBFcnt*$\leftarrow d^{u'}_{\bowtie}$
39. Calculate the butterfly according to the result of the received id of manually activated vertex minus $d^{u'}_{\bowtie}$ according to Theorem 1.
40. **if** $d^{u'}_{\bowtie} <$*GlobalMinKey* **then**
41. $d^{u'}_{\bowtie} \leftarrow$*GlobalMinKey*
42. **end if**
43. *BFTreeUpdate*(*preBFcnt*, $d^{u'}_{\bowtie}$)
44. **end for**
45. **until** *BFTree* on each machine is not null
46. **return** $d^u_{\bowtie}$ value of each vertex $u \in U$

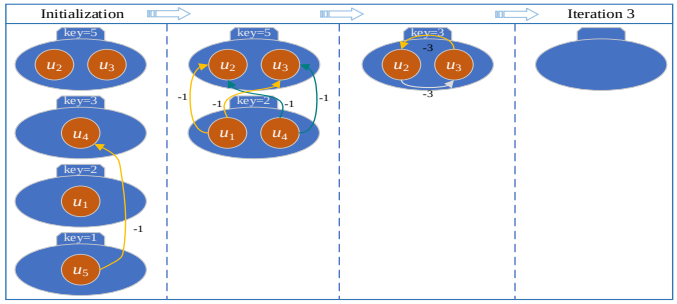
Algorithm 3. *BFTreeUpdate*(*preBFcnt*, *curBFcnt*)

Input: vertex u, *BFTree*, *preBFcnt*, *curBFcnt*
Output: updated *BFTree*
1. Strip $u.id$ from *BFTree*[*preBFcnt*]
2. **if** *BFTree*[*preBFcnt*] is null, **then**
3. Strip *BFTree*[*preBFcnt*] from *BFTree*
4. **end if**
5. **if** *BFTree*[*curBFcnt*]!=*BFTree.end*() **then**
6. *BFTree*[*curBFcnt*].*push_back*($u.id$)
7. **else**
8. *BFTree*[*curBFcnt*] $\leftarrow \{u.id\}$
9. **end if**
10. **return** *BFTree*

Figure 4 illustrates the tip decomposition process of the bipartite graph in Figure 1(a). As shown in Figure 4(a), the $d_{\bowtie}$ of vertices u_{1-5} are 2, 5, 5, 3, and 1 respectively. The corresponding *BFTree* is generated from these vertices and their $d_{\bowtie}$; its structure is shown in Figure 4(b). In the first iteration, u_5 has the minimum $d_{\bowtie}$ and is peeled from the original graph. Since u_4 and u_5 jointly participate in a butterfly, this reduces $d_{\bowtie}^{u_4}$ from 3 to 2, and the structure of the *BFTree* is also updated accordingly. In the second iteration, u_1 and u_4 are placed in the *ActiveVector*, because they both have the current minimum $d_{\bowtie}$ value of 2. When u_1 and u_4 are peeled from U, the remaining vertices in U are u_2 and u_3, which will be peeled in the third iteration. In these iterations, the $d_{\bowtie}$ of the vertex at the time of being peeled is the tip number. The decomposition result is shown in Figure 1(e).

Vertex	Initialization	Iteration-1	Iteration-2	Iteration-3
u_1	2	2	-	-
u_2	5	5	3	-
u_3	5	5	3	-
u_4	3	2	-	-
u_5	1	-	-	-

(a) $d_{\bowtie}$ value of vertex in iteration process



(b) *BFTree* in iteration process

Figure 4 Tip decomposition on a toy bipartite graph

For a given bipartite graph $G = (U, V, E)$, Algorithm 2 requires at most $3 \times |U|$ supersteps to obtain the tip numbers of all vertices in U. Assuming that only one vertex has the minimum $d_{\bowtie}$ in each iteration, three supersteps are consumed to peel the vertex u and update the $d_{\bowtie}$ of the 2-hop neighbors of u according to Algorithm 2. In the first superstep, the activated vertex u

sends a message to its neighbors N_u. The $v \in N_u$ that receives the message forwards the source id (i.e., *u.id*) to $u' \in U$ (i.e., the 2-hop neighbors of u) in the next superstep. After that, those 2-hop neighbors update their respective tip numbers according to Theorem 1. In summary, each vertex needs three supersteps to obtain its tip number and update the $d_{\bowtie}$ of its 2-hop neighbors. Thus, the total superstep cost is $3 \times |U|$.

6 Experimental Results and Analysis

6.1 Experimental setup

(1) Experimental environment: The experimental code in this paper is written in C++; the message communication between different computing nodes is based on Message Passing Interface (MPI), which is then compiled by mpicxx into an executable file. The relay-based communication mode designed in this paper is embedded in a distributed graph computing system which is deployed on the National Supercomputing Center (TH-I) in Changsha. TH-I is equipped with a 160 Gb/s high-speed interconnection system with two Intel® Xeon® CPUs and 48 GB main memory in each single computing node. The underlying MPI of TH-I is implemented in-house and compiled by the Intel compiler, and the experimental code of the algorithm in this paper fits into this environment. This paper uses 10 computing nodes to build a distributed environment.

(2) Experimental setup: This paper is the first study on butterfly counting and tip decomposition in a distributed environment. For the DBC algorithm described in Section 4, the baseline is set to parallelism of 1, namely that m in Line 6 of Algorithm 1 is set to 1. The comparison experiment is $m = \{10, 30, 50, 100, 200, 500\}$, and the evaluation indexes are the total execution time (T) and the total number of supersteps (S). For the DTD algorithm described in Section 5, the baseline is the case without the message validity pruning strategy (Strategy 3), and the comparison algorithm adopts Strategy 3, with total tip decomposition time (T) and the communication traffic (C) being the evaluation indexes.

(3) Dataset: The basic information of the real bipartite graph datasets used in the experiments is shown in Table 2, which can be downloaded from the Konect website[1]. All the datasets are stored in a distributed way using hash, i.e., the machine number where the vertex is located is determined by the hash value of the vertex number. The hash-based graph partitioning technique is a commonly used method in distributed graph computing systems[29], which has the advantages of high partitioning efficiency, guaranteeing a balanced number of vertices stored on each computing node, and quickly determining the computing node where the destination vertex is located during the message transmission process.

Table 2 Real datasets

Dataset	$\|U\|$	$\|V\|$	$\|E\|$
Baidu	901,758	916,634	8,609,972
DBLP	172,072	53,400	293,673
DBLPau	1,953,085	5,624,219	12,282,059
Frwiki	757,621	8,829,774	52,950,008
Twitter	175,214	530,418	1,890,644

6.2 Analysis of experimental results

According to the experimental setup described in Section 6.1, this section analyzes the experimental results of two algorithms, DBC and DTD, in terms of execution time, the number of supersteps, and communication traffic.

[1]The dataset is downloaded from http://konect.cc/.

6.2.1 *Performance evaluation on DBC*

DBC (Algorithm 1) is designed to count the number of butterflies where all vertices (only U-side vertices are considered in this experiment) are located, and the communication between vertices of a bipartite graph can be achieved in a distributed environment according to Strategy 1 (relay-based communication mode). However, if all vertices are activated simultaneously in the first superstep, the communication traffic is too large and will lead to memory overflow. Thus, we need to adjust the algorithm parallelism manually during the experiment (Line 6 of Algorithm 1). The main purpose of this set of experiments is to verify the performance trend of DBC at different parallelism levels.

(1) Execution time: The execution time of an algorithm is the most direct and effective index for evaluating the performance of the algorithm itself. According to the execution flow of the two algorithms given in Section 4 and Section 5, the execution time of DBC on different datasets is calculated for each of 10 computing nodes with different numbers of vertices activated manually (1, 10, 30, 50, 100, 200, and 500, respectively). At this time, DBC has used Strategy 1 and Strategy 2 by default, and the statistical results are shown in Table 3. The results show that the time consumed by DBC decreases significantly when the number of vertices increases from 1 to 50, and then the performance of DBC stabilizes as the number of vertices activated by each computing node increases. It is found that although the parallelism of the algorithm is increased, the single-superstep communication traffic and computation load are also increased so that the overall efficiency improvement is not increased.

Table 3 Time cost of DBC on different datasets

Dataset	Parallelism m						
	1	10	30	50	100	200	500
Baidu	1,130	684	535	412	411	365	384
DBLP	67	28	21	17	16	16	16
DBLPau	206	28	14	10	8	8	7
Frwiki	1,128	994	911	724	787	791	719
Twitter	638	299	226	179	166	154	160

(2) Number of supersteps: In distributed systems, the communication overhead is an important part of the total overhead. The system deployed in this paper uses the BSP computational model, so the communication overhead can be measured by the number of supersteps. The parallelism level of Algorithm 1 is controlled by the parameter m in Line 6, and the number of execution supersteps for each dataset at different parallelism levels is shown in Table 4. The superstep cost is effectively reduced for all datasets as the parallelism level increases. Since the experiments in this paper are deployed on a high-performance computing platform in the National Supercomputing Center with a high-speed communication network on the bottom, the time reduction brought by the optimization of the number of supersteps is not obvious. It is foreseeable that the significant reduction in the number of supersteps can lead to higher performance improvement on a normal distributed cluster.

Table 4 Superstep cost with different parallelism

Dataset	Parallelism m						
	1	10	30	50	100	200	500
Baidu	361,116	40,148	13,384	7,224	3,612	1,808	724
DBLP	68,836	7,652	2,552	1,380	692	348	140
DBLPau	781,236	86,804	28,936	15,628	7,816	3,908	1,564
Frwiki	303,052	33,676	11,228	6,064	3,032	1,516	608
Twitter	70,420	7,900	2,636	1,412	708	356	144

6.2.2 *Performance evaluation on DTD*

DTD is used to calculate the tip values of all vertices, which indicate the k-tip community with the largest k-value that the vertex can exist in, and this index can be applied to mine the tight communities in bipartite graphs. This group of experiments mainly calculates the tip numbers of vertices on the U-side. However, since the vertices on the same side cannot directly access their 2-hop neighbors and cannot obtain the state (whether it is peeled or not) of their 2-hop neighbors, it is necessary to forward messages through the vertices (V-side vertices) on the opposite side. Meanwhile, the message validity pruning strategy is also executed by the relay vertex. This group of experiments permits to verify the effectiveness of the designed message pruning strategy and to quantify the reasons for the performance improvement from the perspective of communication traffic.

(1) Execution time: After DBC enables each vertex $u \in U$ in the dataset to obtain the butterfly degree, the experiments focus on comparing the execution time of DTD without Strategy 3 and that with Strategy 3 (expressed as DTD+) on different datasets. As indicated by the experimental results in Table 5, the message pruning strategy (Strategy 3) effectively reduces the total time cost, with the highest improvement being 51.9%. The reason for this is that the reduction in message volume reduces the communication traffic on the one hand and diminishes the number of vertices activated in the next superstep on the other hand, thus significantly decreasing the amount of redundant computation. In this algorithm, the vertices that have already determined the tip values do not need to receive messages to update their tip values. If Strategy 3 is not used, the vertices that have already determined the tip numbers may be activated several times, which introduces plenty of redundant computation.

Table 5 Time cost of tip decomposition (s)

Dataset	Algorithm		
	DTD	DTD+	Rate (%)
Baidu	960	462	51.9
DBLPau	89	77	13.5
Frwiki	3,299	2,980	9.7
Twitter	2,439	2,190	10.2

(2) Communication traffic: The above experiments reveal that the effectiveness of Strategy 3 comes from reducing redundant communication and thus redundant computation. This group of experiments quantifies the effectiveness of Strategy 3 by specific communication traffic. As shown in Table 6, the communication traffic of DTD+ is significantly less than that of DTD, and the maximum reduction rate of communication traffic is about 57.8%. According to the analysis of the BSP computation model, the communication traffic determines the number of vertices activated in the next round, which then further determines the computation load of each computing node. The decrease in communication traffic can effectively improve the performance of distributed algorithms.

Table 6 Communication cost of tip decomposition

Dataset	Algorithm		
	DTD	DTD+	Rate (%)
Baidu	285,815,206	126,092,382	55.9
DBLPau	4,903,833	2,069,477	57.8
Frwiki	653,528,468	534,658,219	18.2
Twitter	1,297,868,458	1,151,330,554	11.3

7 Conclusion and Outlook

To address the problems that existing butterfly counting algorithms and tip decomposition algorithms have storage bottlenecks on a single machine and a large communication overhead

in a distributed environment, this paper proposes a new relay-based communication mode and a more efficient DBC on this basis. In addition, redundant communication and extra overhead are significantly reduced by the introduction of a message pruning strategy based on vertex priority. Subsequently, DTD is proposed to improve the algorithm efficiency through the message validity pruning strategy. The experimental results show that the proposed algorithms can effectively solve the tip decomposition problem on large bipartite graphs. Dynamic changes have become an important characteristic of large bipartite graph data. For this reason, the distributed incremental maintenance algorithm of tip numbers on large dynamic bipartite graphs will be further studied in future work.

References

[1] Dhillon IS. Co-clustering documents and words using bipartite spectral graph partitioning. Proc. of the 7th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. 2001. 269–274.

[2] Wang K, Lin X, Qin L, *et al.* Efficient bitruss decomposition for large-scale bipartite graphs. 2020 IEEE 36th International Conference on Data Engineering (ICDE). IEEE, 2020. 661–672.

[3] Gibson D, Kumar R, Tomkins A. Discovering large dense subgraphs in massive graphs. Proc. of the 31st International Conference on Very Large Data Bases. 2005. 721–732.

[4] Seidman SB. Network structure and minimum degree. Social Networks, 1983, 5(3): 269–287.

[5] Malliaros FD, Giatsidis C, Papadopoulos AN, *et al.* The core decomposition of networks: Theory, algorithms and applications. The VLDB Journal, 2020, 29(1): 61–92.

[6] Cohen J. Trusses: Cohesive subgraphs for social network analysis. NSA, 2008.

[7] Newman MEJ. Scientific collaboration networks. I. Network construction and fundamental results. Physical Review E, 2001, 64(1): 016131.

[8] Sarıyüce AE, Pinar A. Peeling bipartite networks for dense subgraph discovery. Proc. of the 11th ACM International Conference on Web Search and Data Mining. 2018. 504–512.

[9] Shi J, Shun J. Parallel algorithms for butterfly computations. Proc. of the Symp. on Algorithmic Principles of Computer Systems. Society for Industrial and Applied Mathematics, 2020. 16–30.

[10] Wang K, Lin X, Qin L, *et al.* Vertex priority-based butterfly counting for large-scale bipartite networks. Proc. of the VLDB Endowment, 2019, 12(10): 1139–1152.

[11] Sanei-Mehri SV, Sariyuce AE, Tirthapura S. Butterfly counting in bipartite networks. Proc. of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining. 2018. 2150–2159.

[12] Lakhotia K, Kannan R, Prasanna V, *et al.* RECEIPT: REfineCoarsE-grained Independent Tasks for Parallel Tip decomposition of Bipartite Graphs. arXiv: 2010.08695, 2020.

[13] Malewicz G, Austern MH, Bik AJC, *et al.* Pregel: A system for large-scale graph processing. Proc. of the 2010 ACM SIGMOD International Conference on Management of data. Indianapolis, 2010. 135–146.

[14] Sakr S, Orakzai F M, Abdelaziz I, *et al.* Large-scale Graph Processing using Apache Giraph. Springer, 2016.

[15] Gonzalez JE, Xin RS, Dave A, *et al.* Graphx: Graph processing in a distributed dataflow framework. Proc. of the 11th USENIX Symposium on Operating Systems Design and Implementation. Broomfield, 2014. 599–613.

[16] Yan D, Cheng J, Lu Y, *et al.* Effective techniques for message reduction and load balancing in distributed graph computation. Proc. of the 24th International Conference on World Wide Web. Florence, 2015. 1307–1317.

[17] Yu G, Gu Y, Bao YB, *et al.* Large scale graph data processing on cloud computing environments. Chinese Journal of Computers/Ji Suan Ji Xue Bao, 2011, 34(10): 1753–1767. [doi: 10.3724/SP.J.1016.2011.01753]

[18] Lu L, Hua B. A platform-and-workload aware online graph partitioning algorithm. Chinese Journal of Computers/Ji Suan Ji Xue Bao, 2020, 43(7): 1230–1245. [doi: 10.11897/SP.J.1016.2020.01230]

[19] Zhang CB, Li Y, Jia T. Survey of state-of-the-art fault tolerance for distributed graph processing jobs. Journal of Software/Ruan Jian Xue Bao, 2021, 32(7):2078-2102. [doi: 10.13328/j.cnki.jos.006269]

[20] Engelhardt N, So HKH. Gravf: A vertex-centric distributed graph processing framework on fpgas. Proc. of the 2016 26th Int'l Conf. on Field Programmable Logic and Applications (FPL). IEEE, 2016. 1–4.

[21] Weng T, Zhou X, Li K, *et al.* Efficient distributed approaches to core maintenance on large dynamic graphs. IEEE Trans. on Parallel and Distributed Systems, 2021, 33(1): 129–143.

[22] Luo W, Zhou X, Yang J, *et al.* Efficient approaches to top-r influential community search. IEEE Internet of Things Journal, 2020, 8(16): 12650–12657.

[23] Ma C, Cheng R, Lakshmanan LVS, *et al.* Linc: A motif counting algorithm for uncertain graphs. Proc. of the VLDB Endowment, 2019, 13(2): 155–168.

[24] Fang Y, Cheng R, Luo S, *et al.* Effective community search for large attributed graphs. Proc. of the VLDB Endowment, 2016, 9(12): 1233–1244.

[25] Ma C, Fang Y, Cheng R, *et al.* Efficient algorithms for densest subgraph discovery on large directed graphs. Proc. of the 2020 ACM SIGMOD Int'l Conf. on Management of Data. 2020. 1051–1066.

[26] Chen R, Shi J X, Chen HB, *et al.* Bipartite-oriented distributed graph partitioning for big learning. Journal of Computer Science and Technology, 2015, 30(1): 20–29.

[27] Liu B, Yuan L, Lin X, *et al.* Efficient (α, β)-core computation in bipartite graphs. The VLDB Journal, 2020, 29(5): 1075–1099.

[28] Chiba N, Nishizeki T. Arboricity and subgraph listing algorithms. SIAM Journal on computing, 1985, 14(1): 210–223.

[29] Yan D, Cheng J, Özsu MT, *et al.* A general-purpose query-centric framework for querying big graphs. Proc. of the VLDB Endowment, 2016, 9(7): 564–575.

Xu Zhou, Ph.D., associate professor. Her research interests include parallel computing, graph data management, and graph computing.

Boren Li, master. His research interests include graph computing and community search.

Tongfeng Weng, Ph.D. His research interest is distributed graph computing.

Ji Zhang, professor. His research interests include data mining and graph data management.

Zhibang Yang, Ph.D., associate professor. His research interests include parallel computing and graph data management.

Kenli Li, Ph.D., professor, distinguished member of CCF, chairman of CCF Changsha. His research interests include parallel distributed processing, supercomputing, cloud computing, high-performance computing for big data and artificial intelligence.