Distributed Approaches to Butterfly Analysis on Large Dynamic Bipartite Graphs

Tongfeng Weng, Xu Zhou, Kenli Li, *Senior Member, IEEE*, Kian-Lee Tan, *Senior Member, IEEE*, and Keqin Li, *Fellow, IEEE*

Abstract—Tip decomposition has a pivotal role in mining cohesive subgraphs in bipartite graphs by computing the tip number of each vertex in accordance with the non-trivial motif butterfly ((2,2)-biclique). It has been a popular research topic with applications in document clustering, spam group detection, and analysis of affiliation networks. In such applications, the graphs are not only large, but they evolve quickly with new edges being continuously added/deleted. While existing centralized techniques could solve the tip decomposition problem for static bipartite graphs, they are not efficient for maintaining the tip numbers of vertices on large-scale graphs. In this paper, we study butterfly analysis problems on bipartite graphs in a distributed environment with the vertex-centric model. We first extend a centralized butterfly counting algorithm to a distributed version, called DBCA. An ingenious message aggregation strategy is designed to reduce massive redundant messaging and avoid the memory overflow problem while processing large-scale graphs. Based on the results of DBCA, we develop a distributed tip decomposition algorithm (DTDA) to get the tip number of each vertex in parallel. Finally, to maintain the tip numbers of vertices efficiently while graphs evolve over time, we explore a distributed tip maintenance algorithm (DTMA) along with a novel task-split strategy. Specifically, for an updated edge (insertion/deletion), several sub-tasks will be generated in line with the topology structure of the original bipartite graph. To our best knowledge, this is the first study to process the butterfly analysis problems in a distributed environment. In addition, comprehensive experiments have been conducted on real-world bipartite graphs. The experiment results demonstrate that our proposed algorithms are efficient and scalable.

Index Terms—Bipartite graph, butterfly counting, distributed algorithm, tip decomposition, tip maintenance

1 Introduction

In recent years, there has been an increasing interest in mining cohesive subgraphs in bipartite graphs [23]. In a bipartite graph, vertices are decomposed into two disjoint sets, U and V, and edges can only connect vertices from different sets. The cohesive subgraphs in a bipartite graph consist of vertices in the same set (U or V), i.e. vertices belong to the same entity. Bipartite graph modeling is important for a wide range of applications, including document clustering [5], author-paper and user-product relationships analyses [22], and spam group detection [7]. In such applications, the graphs have two common properties: they are large scale and evolve dynamically [1]. In this context, it is desirable to develop scalable distributed solutions to mine and maintain analyses results of large-scale dynamic bipartite graphs.

A considerable amount of literature has been published on uncovering dense structures on unipartite graphs, including core decomposition and truss decomposition [6], [8]. Although those off-the-shelf technologies can be utilized to analyze the projections of bipartite graphs [14], the original information is distorted and there are 6 orders of magnitude increase in graph size when converting bipartite graphs to co-occurrence (projection) graphs [17]. Therefore, it needs to find effective methods that can be applied to the original bipartite graphs.

In a unipartite graph, the triangle is the underlying structure that yields cohesive subgraphs. A butterfly is the smallest cohesive bipartite unit in bipartite graphs. As shown in Fig. 1, vertices v_1, v_2, u_1, and u_2 form a butterfly, which is a (2,2)-biclique, denoted as $\bowtie_{v_1,v_2,u_1,u_2}$. The number of butterflies that a vertex u can participate in is defined as the butterfly degree of u, denoted as $d^u_{\bowtie}$. The state-of-the-art vertex-priority-based algorithm [21] is proposed to calculate $d_{\bowtie}$ for each vertex. It is not easy to count butterflies in a distributed system. This is because a vertex cannot access its neighbors directly, and it is even harder for it to access its 2-hop neighbors. It needs two rounds of 2-hop message delivery to get common neighbors of a pair of vertices in the same set. For example, to enumerate the butterfly $\bowtie_{v_1,v_2,u_1,u_2}$, two messaging links $u_1 \rightarrow v_1 \rightarrow u_2$ and $u_1 \rightarrow v_2 \rightarrow u_2$ are needed. If we activate all vertices in parallel, it is inefficient to deal with such an amount of messages and may lead to the memory overflow problem while processing large-scale bipartite graphs.

- *Tongfeng Weng, Xu Zhou, and Kenli Li are with the College of Computer Science and Electronic Engineering, Hunan University, Changsha, Hunan 410082, China. E-mail: {wengtongfeng, zhxu, lkl}@hnu.edu.cn.*
- *Kian-Lee Tan is with the School of Computing, National University of Singapore, Singapore 119077. E-mail: tankl@comp.nus.edu.sg.*
- *Keqin Li is with the College of Computer Science and Electronic Engineering, Hunan University, Changsha, Hunan 410082, China, and also with the Department of Computer Science, State University of New York, New Paltz, NY 12561 USA. E-mail: lik@newpaltz.edu.*

Manuscript received 14 March 2022; revised 13 September 2022; accepted 8 November 2022. Date of publication 14 November 2022; date of current version 19 December 2022.
This work was supported in part by the National Key R&D Program of China under Grant 2020YFB2104000, in part by NSFC under Grants 62172146, 62172157, and 62102143, in part by the Natural Science Foundation of Hunan Province under Grant 2022JJ30009, in part by the Open Research Projects of Zhejiang Lab under Grant 2021KD0AB02, and in part by the Postgraduate Scientific Resarch Innovation Project of Hunan Province under Grant QL20210096. (Corresponding author: Xu Zhou.)
Recommended for acceptance by J. Zhai.
Digital Object Identifier no. 10.1109/TPDS.2022.3221821

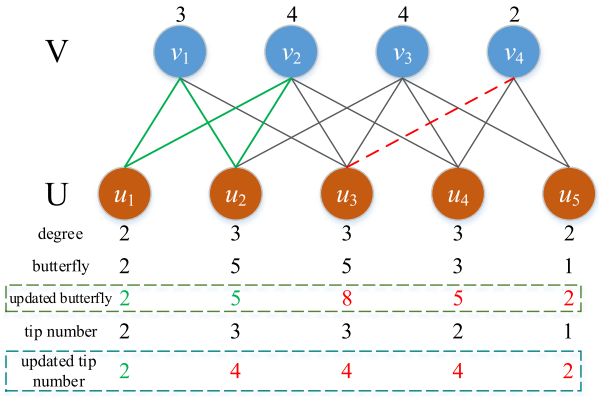

Fig. 1. A toy graph G. The edge (u_3, v_4) is inserted into G.

In [17], k-tip is investigated to measure the intensity of vertex participations in the butterfly structures and identifies the induced subgraphs with many butterflies. It consists of vertices that can participate in at least k induced butterflies. The tip number of a vertex u is the maximum k such that there exists a k-tip containing u. On the basis of the butterfly counting results, the tip number of each vertex $u \in U$ can be easily obtained by tip decomposition algorithms [10], [17], [19]. These works are hard to be extended to distributed systems due to the different storage and computing models. Although these algorithms can effectively process the core decomposition over static graphs, the time complexity $O(\sum_{v \in V} d(v)^2)$ [17] is too high to be applied to recalculate the tip numbers for each edge updated.

So far, however, there has been no discussion about the tip maintenance problem on time-evolving bipartite graphs. For an edge update in a unipartite graph, the endpoint degrees of the new edge is changed by constant 1 and the core numbers of all vertices can be changed by at most 1 [16]. Different from the core maintenance problem on unipartite graphs, the new edge may change the butterfly degrees $d_{\bowtie}$ of several vertices by more than 1. As a result, the existing theory of core maintenance cannot be applied to maintain tip numbers in bipartite graphs.

Challenges. In this paper, we mainly focus on the problems of distributed butterfly counting and tip number maintenance on large-scale bipartite graphs. According to the above analysis, the challenges are listed as follows. First, the massive messages generated during the process of butterfly counting will lead to inefficiency and the memory overflow problem. Second, it is difficult to identify the candidate vertices that will change the tip numbers after a graph change and the exact change value of the tip number of a vertex is also not easy to be determined.

To address the above challenges, we first design a message aggregation strategy to reduce massive redundant messaging and avoid the memory overflow problem while processing large-scale graphs. In specific, each vertex maintains a message vector according to the priority of neighbors and sends the message vector to each machine instead of its neighbors. After that, each machine activates local vertices and parses butterflies from the received messages. The proposed strategy reduces redundant messaging by avoiding the message passing between pairs of vertices. Then, a task-split strategy is conducted to maintain tip numbers when given bipartite graphs are updated. Specifically, for each updated edge (u, v), we refer to the task of maintaining the tip number as the original task (denoted as ori-task in the rest of the paper). If there exists a pair of vertices $u, u' \in U$ that has sharing butterflies changed on account of the edge update, some sub-tasks (u, u') will be generated, where the number of sub-tasks equals the change value of the sharing butterflies. For each sub-task, we reconstruct a subgraph consisting of candidate vertices and peel them according to their 2-hop neighbors' tip numbers. The remaining vertices in the reconstructed subgraph increase their tip numbers by 1. When all sub-tasks are completed, the ori-task is also done (i.e., the tip numbers are maintained.). Last but not least, a candidate-sharing theory is proved to reduce redundant computation of sub-tasks with two same endpoints.

Contributions. The major contributions of this study are listed as follows. We

- extend the centralized butterfly counting and tip decomposition algorithms to distributed versions. Specially, we conduct a message aggregation strategy to reduce massive redundant messaging and avoid the memory overflow problem.
- explore a distributed tip maintenance algorithm (DTMA) along with a novel task-split strategy. To improve efficiency, a candidate-sharing theory is developed to reduce redundant computation.
- verify the efficiency and effectiveness of our algorithms through various experiments on real-world graphs.

Outline. The remaining part of the paper proceeds as follows. The related work is reported in Section 2. Section 3 gives the preliminaries that formally define the problem. The distributed butterfly counting algorithm and the message aggregation strategy is described in Section 4. Section 5 gives the distributed tip number decomposition and maintenance algorithms. The experimental results are presented in Section 6. Finally, Section 7 concludes this paper.

2 Related Work

In this section, we review the related work about butterfly analysis on bipartite graphs and distributed graph computing systems.

2.1 Butterfly Analysis on Bipartite Graphs

In recent years, there has been an increasing interest in mining cohesive subgraphs in bipartite graphs [23]. As a basic motif in bipartite graphs, a butterfly contains two vertices on each side and all four possible edges among them (i.e., (2,2)-biclique) [20]. We focus on butterfly analysis on bipartite graphs.

A. Butterfly Counting

Chiba *et al.* [4] proposed an efficient vertex-priority quadrangle counting algorithm that traverses wedges with $O(1)$ work per wedge. Wang *et al.* [21] further explored a cache efficient version of the vertex-priority algorithm that reorders the vertices in decreasing order of degree. We extend the priority strategy to solve the distributed butterfly counting problem. In addition, Sanei *et al.* [15] designed randomized algorithms that can quickly approximate the number of

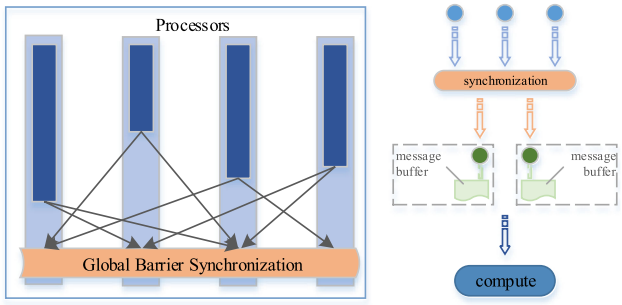

Fig. 2. The model of distributed graph computing systems.

TABLE 1
Summary of Notations

Notations	Description
$G{=}(U, V, E)$	A bipartite graph
$N(u)$	The neighbors of vertex u
$d_G(u)$	The number of neighbors of u
$H(U', V', E')$	The subgraph of graph G, abbreviated as H
$\bowtie_u$	A butterfly contains vertex u
$d^u_{\bowtie}$	The number of butterflies containing vertex u
$\bowtie^{u_2}_{u_1}$	A sharing butterfly of vertices u_1 and u_2
$\theta(u)$	The tip number of vertex u
H^u_k	The k-tip with maximal k contains vertex u

butterflies in a graph with a provable guarantee of accuracy. Sheshbolouki *et al.* [18] investigated the butterfly approximation framework on streaming bipartite graphs. A novel core algorithm is expressed for exact butterfly counting in streaming graph snapshots. It focuses on counting the total number of butterflies in a bipartite graph rather than considering a specific vertex, while our proposed algorithm can maintain the butterfly analysis results for each vertex. Existing work is constructed for the centralized environment, which is not scalable for large scale graphs. Therefore, we study the distributed butterfly counting algorithm in this work first.

B. Tip decomposition

k-tip [17] is a fundamental metric in analyzing bipartite graphs. It has been widely applied in many fields, including social network analysis [5], [7], [22]. There are several works that determined the tip number through a peeling-based method [17], [19]. Lakhotia *et al.* [10] introduced a parallel tip-decomposition algorithm by partitioning the vertices into multiple independent subsets that can be concurrently peeled. Also, these works are explored in a centralized environment without considering large scale graphs and dynamic graphs.

However, with the development of Internet information technology, graphs have been evolving on account of edge/vertex updates [1], [12]. Although many investigators explored incremental algorithms for maintaining the results of graph analysis [1], [2], [8], [25], there is no discussion about the tip maintenance problem on time-evolving bipartite graphs. To address this issue, we explore methods to maintain the tip number of vertices while graphs evolve.

2.2 Distributed Graph Computing Systems

Numerous research interests have been shown in designing distributed graph systems to process big graphs due to the bottleneck of single machine memory [24]. As shown in Fig. 2a, in a distributed graph computing system [13], [26], a superstep is a basic unit that contains three main steps: receiving messages, computing, and sending messages. In specific, vertices invoke the computing function to analyze messages that they have received. Then, after the synchronization of communication, messages are stored in the message buffer of corresponding vertices (Fig. 2b). Due to the large number of messages generated during the process of butterfly counting, the memory may overflow while processing large scale graphs. To analyze large scale bipartite graphs effectively and efficiently, we research the problems of butterfly analysis in a distributed environment.

3 Preliminaries

In this section, we introduce some definitions related to tip decomposition and maintenance on bipartite graphs. Table 1 gives the notations used in this work.

We focus on an unweighted and undirected bipartite graph $G{=}(U, V, E)$ in this paper, where U and V are two disjoint vertex sets and E contains all edges with two endpoints belonging to different vertex sets, i.e., U and V. Specifically, a vertex $u{\in}U$ only has neighbors in V, denoted as $N(u)$, and $d_G(u)$ is the degree of u. In general, vertices on the same side belong to the same category of entities. Because there's no direct connection between vertices on the same side, the concept of butterfly $\bowtie$ is introduced to represent their relationship. $\bowtie_u$ represents that a butterfly contains u and the number of butterflies that u can belong to is denoted as $d^u_{\bowtie}$. For two vertices u_1 and u_2 in the same butterfly, we use $\bowtie^{u_2}_{u_1}$ to represent a sharing butterfly and the number of sharing butterflies can be a metric to measure the structure tightness between the two vertices.

Definition 1. ***(Wedge)*** *Given a bipartite graph $G{=}(U, V, E)$, a wedge consists of three vertices and two edges (i.e., (u, v, u') or (v, u, v')).*

Definition 2. ***(Butterfly)*** *Given a bipartite graph $G{=}(U, V, E)$, there is a subgraph $H(U', V', E')$ containing two vertices u_1, u_2 in U and two vertices v_1, v_2 in V. $H(U', V', E')$ is a butterfly if and only if u_1 and u_2 are the neighbors of both v_1 and v_2 (i.e., H is a (2-2)-biclique containing two vertices on each side and all four possible edges among them). Obviously, a butterfly contains two wedges.*

In a bipartite graph, vertices on the same side belong to the same category of entities but their neighbors are always on the opposite side. We map u_1 to be a neighbor of u_2 if the two vertices are in at least one same butterfly. In this work, we focus on mining the cohesive structure of vertices inside U. Based on this, the definition of a k-tip community is given as follows.

Definition 3. ***(k-tip)*** *Given a bipartite graph $G{=}(U, V, E)$, a sub graph $H(U', V, E')$ is a k-tip if and only if it satisfies*

- *Connectivity: each pair of vertices belonging to U' are connected by a series of butterflies.*
- *Tightness: each vertex $u \in U'$ participates in at least k butterflies.*
- *Uniqueness: no other k-tip subsumes H.*

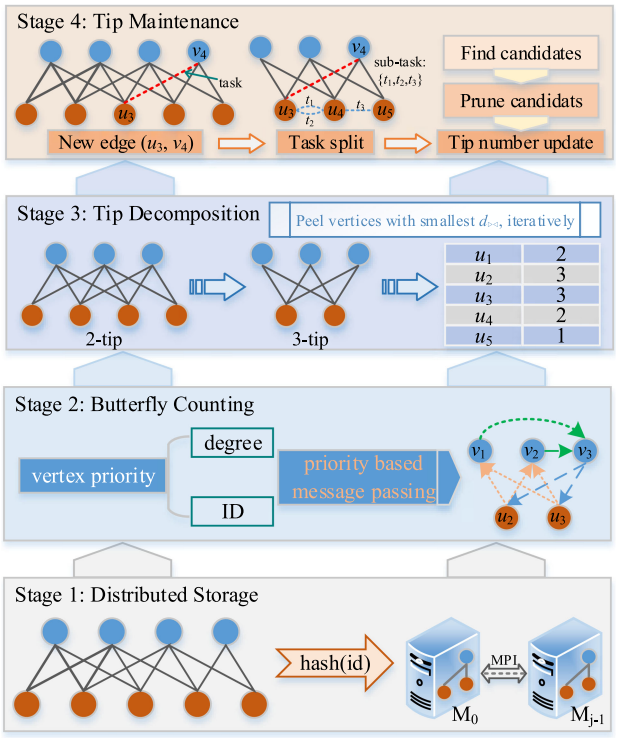

Fig. 3. System overview.

Definition 4. ***(Tip number)*** *Given a bipartite graph $G = (U, V, E)$, the tip number of a vertex $u \in U$ denoted as $\theta(u)$ is the maximum k such that there exists a k-tip (denoted as H_k^u) containing u.*

Based on these definitions, the problems that will be studied in this work are shown as follows.

Problem 1. ***(Butterfly Counting)*** *Given a bipartite graph $G = (U, V, E)$, for each $u \in U$, it aims to count the number of butterflies containing u (i.e., $d_{\bowtie}^u$).*

Problem 2. ***(Tip Decomposition)*** *Given a bipartite graph $G = (U, V, E)$, for each $u \in U$, tip decomposition is used to obtain the tip number of u.*

Problem 3. ***(Tip Maintenance)*** *Given a bipartite graph $G = (U, V, E)$ with an edge update, tip maintenance is proposed to maintain the tip numbers of vertices in U.*

Fig. 3 shows the system overview of this work. Given a bipartite graph, vertices are divided into different parts and stored in different machines according to the hash values of their IDs. For the given bipartite graph, butterfly counting is explored to calculate the butterfly degree of each vertex. And tip decomposition aims to obtain the tip number of each vertex on the basis of butterfly counting results. After that, when an edge update happens, tip maintenance is explored to maintain the tip numbers of corresponding vertices.

4 Distributed Butterfly Counting

To construct the relationship between each pair of vertices on the same side, we need to enumerate all butterflies in a bipartite graph (i.e., find $\bowtie_{u_1}^{u_2}$ for all $u_1 \in U$ and $u_2 \in U$). In this section, we study the distributed butterfly counting algorithm (DBCA) for large-scale bipartite graphs. The second stage in in Fig. 3 illustrates the basic idea of distributed butterfly counting. First, we extend the paradigm proposed in [21] to define the priority of a vertex according to its degree and ID (Lemma 1). Then, a priority-based message passing strategy is introduced to detect butterflies in the given graph. In specific, the vertex with higher priority will be selected as $start\_v$ and sent its ID to vertices $middle\_v$, which are neighbors of $start\_v$ and have smaller priority. After that, vertices $middle\_v$ forward messages to their neighbors that have priority lower than $start\_v$. In this way, we can get the number of sharing butterflies between each pair of vertices. And the butterfly degree $d_{\bowtie}$ of each vertex is calculated by Equations (1) and (2).

4.1 A Basic Algorithm

In a distributed vertex-centric graph computing system (see Section 6), each vertex just maintains its properties, including ID, degree, and neighbor list. If a vertex $u \in U$ wants to access its 2-hop neighbors, it needs to send a message to its neighbor $v \in N(u) \in V$ and v forwards the message to $u' \in N(v) \in U$ (2-hop neighbors of u). As stated in Section 1, to traverse a wedge in a bipartite graph, we need three supersteps to access the 2-hop neighbors of a vertex. According to Definition 2, a butterfly consists of two wedges. Therefore, the number of wedges that needs to be traversed determines the performance of DBCA.

The vertex-priority-based paradigm [21] is proposed to reduce the traversal of wedges when counting butterflies for each vertex. Specifically, a wedge with three vertices, denoted as $start\_v$, $middle\_v$ and end_v, is traversed if $p(start\_v) > p(middle\_v)$ and $p(start\_v) > p(end\_v)$, where p is a priority notion. We extend this notion of priority into our algorithm as below.

Lemma 1. ***(Priority)*** *Given a bipartite graph $G = (U, V, E)$, the priority $p(u)$ of a vertex u is determined by its degree, ID, and type.*

- *(vertices in the same set). For $p(u) > p(u')$, if $d(u) > d(u')$ or $d(u) = d(u') \wedge ID(u) > ID(u')$.*
- *(vertices in opposite sets). For $p(u) > p(v)$, if $d(u) > d(v)$ or $d(u) = d(v) \wedge u \in U$.*

DBCA is developed based on Lemma 1. The main idea is to find common neighbors of each pair of vertices u and u' in U. For two vertices u and u', they can form a butterfly with two of their common neighbors. Let n be the number of the common neighbors. The number of the butterflies containing both u and u' can be calculated by

$$B_1 = C_n^2 = \frac{n \times (n-1)}{2}. \tag{1}$$

Let vertices u and u' be the start and end vertices of these butterflies, respectively. The common neighbors of vertices u and u' are middle vertices. For each butterfly, it is composed of four vertices, three of which are vertices u, u' and w, and the fourth vertex of the butterfly is from the left $n-1$ common neighbors of u and u'. Hence, there are

$$B_2 = n - 1 \tag{2}$$

butterflies containing each common neighbor w, where n is the number of common neighbors.

As shown in Fig. 1, u_2 and u_3 have 3 common neighbors, v_1, v_2, and v_3. According to Equation (1), there are $C_3^2 = 3$ butterflies containing u_2 and u_3 and $3-1 = 2$ butterflies that v_1, v_2, and v_3 can participate in.

The pseudo-code of DBCA is shown in Algorithm 1. Given a bipartite graph $G=(U, V, E)$, we partition vertices in the graph to different working machines in a hashing way, which is the same as Pregel [13]. There are four supersteps to obtain $d_{\bowtie}$ for each vertex. In the first superstep (Lines 1-3), all vertices are activated as a $start\_v$ to send message ($start\_v$ id, t), where $start\_v$ is the start vertex of a wedge with the highest priority and t is the type (U or V) of the target vertex. In the second superstep (Lines 4-6), vertices receive message. The vertices, which we refer to as $middle\_v$, would then forward message ($start\_v$ id, t, $middle\_v$ id) to their neighbors that have priority lower than the $start\_v$. In the third superstep (Lines 7-12), the 2-hop neighbors (end_v) of these $start\_v$ are accessed to compute the two kinds of result $\mathbb{B}_1$ and $\mathbb{B}_2$ according to Equations (1) and (2). In specific, each end_v parses the received messages to construct a $hash\_map$ with key: $start\_v$, $value$: $middle\_v$ $list$. The length of the $middle\_v$ $list$ is the number of common neighbors of $start\_v$ and end_v. $\mathbb{B}_1$ and $\mathbb{B}_2$ can be calculated by the $hash\_map$ and the result will be sent to $start\_v$ and $middle\_v$, respectively. In the last superstep (Lines 13-17), vertices update their $d_{\bowtie}$ from the received messages.

Algorithm 1. Distributed Butterfly Counting Algorithm

Input: bipartite graph $G=(U, V, E)$
Output: butterfly counting result for each vertex $u \in U$
1: **Superstep 1:** (Activate all vertices as $start\_v$)
2: **for** $start\_v \in U \cup V$ **do**
3: Send message ($start\_v$ id, t) to $N(start\_v)$ with lower priority (Lemma 1).
4: **Superstep 2:** ($middle\_v$ forwards messages to end_v)
5: **for** each $middle\_v$ **do**
6: Send message ($start\_v$ id, t, $middle\_v$ id) to its neighbors that have priority lower than the $start\_v$.
7: **Superstep 3:** (end_v calculates $\mathbb{B}_1$ and $\mathbb{B}_2$)
8: **for** each end_v **do**
9: Construct a $hash\_map$: $key \rightarrow start\_v$, $value \rightarrow middle\_v$ $list$ from the received messages
10: Calculate $\mathbb{B}_1$ and $\mathbb{B}_2$ by fomular (1) and (2)
11: end_v: $d_{\bowtie} \leftarrow d_{\bowtie} + \mathbb{B}_1$
12: Send $\mathbb{B}_1$ to $start\_v$ and $\mathbb{B}_2$ to $middle\_v$
13: **Superstep 4:** ($start\_v$ and end_v update $d_{\bowtie}$)
14: **for** each $start\_v$ **do**
15: $d_{\bowtie} \leftarrow d_{\bowtie} + \mathbb{B}_1$
16: **for** each $middle\_v$ **do**
17: $d_{\bowtie} \leftarrow d_{\bowtie} + \mathbb{B}_2$
18: **return** $d_{\bowtie}^u$ for each vertex $u \in U$

Example. We illustrate the process of DBCA in Fig. 1. We take vertex u_2 as an example. According to Lemma 1, u_2 has higher priority than its neighbor v_1 and sends its id to v_1. But there is only one neighbor of v_1 having priority less than $start\_v$ u_2 (i.e., $p(u_1) < p(u_2)$). Because a butterfly consists of at least two wedges, no butterfly contains u_2, in which u_2 has the highest priority. Again considering Lemma 1, v_2 and v_3 send their id to neighbors u_1 to u_4 and u_2 to u_5, respectively. Then these neighbors forward the received message to their neighbors that have smaller priority than v_2 and v_3. The wedges traversed above can impact $d_{\bowtie}^{u_2}$ are (v_2, u_1, v_1), (v_2, u_2, v_1), (v_2, u_3, v_1), (v_3, u_2, v_1), (v_3, u_3, v_1), (v_3, u_2, v_2), (v_3, u_3, v_2), and (v_3, u_4, v_2). In these wedges, u_2 plays a $middle\_v$. By Equation (2), v_1 and v_2 sharing 3 and 2 butterflies with u_2, respectively. As a result, the total number of butterflies that u_2 can participate in is 5.

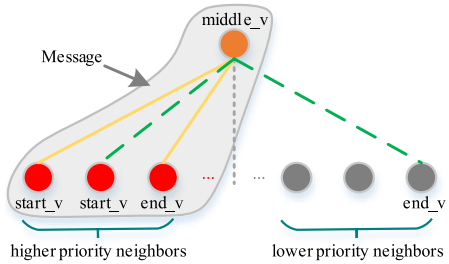

Fig. 4. Butterfly counting based on the aggregated messages.

Analysis. There are $|E|$ messages generated in the first superstep, and the memory cost is about $16 \times |E|$ bytes. The maximal number of messages passed in the second superstep equals $m \times (m-1)/2$, where $m = min(|U|, |V|)$, when the whole bipartite graph is a biclique. The maximal memory cost is about $24 \times m \times (m-1)/2$ bytes. Here, we use long int to define the message vector.

4.2 An Advanced Algorithm

It can be observed from the analysis of the basic algorithm that the number of messages will be too large to get high efficiency and even lead to the memory overflow problem while processing large-scale graphs. There is massive redundant messaging due to the repeated storage of messages. For example, in the second superstep, vertices u_{1-3} all need to send (ID(v_2), 1, *) to v_1. ID(v_2) and 1 are obviously repetitive. In addition, after the synchronization of each superstep, vertices maintain a message buffer to store the messages that are sent to themselves (see Fig. 2). As a result, the size of memory used by messages has been doubled.

To address these issues, we design a message aggregation strategy to reduce massive redundant messaging and count the butterflies directly by parsing messages without storing into message buffers.

Strategy 1. ***(Message aggregation)*** *Each vertex sorts its neighbors in descending order of their priority. The message sent by a vertex u consists of three parts.*

- *ID(u),*
- *the type of u, to indicate which set (U or V) u belongs to,*
- *higher priority neighbor list.*

The structure of the aggregated message is shown in Fig. 4. According to the vertex-priority-based paradigm [21], there are two kinds of wedges needed to be traversed. First, the wedge connected by yellow lines. The $start\_v$ and end_v both come from the higher priority neighbor list, but $p(start\_v)$ must be higher than $p(end\_v)$. Second, the wedge connected by green dotted lines. The $start\_v$ and end_v are

selected from the higher priority neighbor list and the lower priority neighbor list, respectively. These lower priority neighbors are not contained in the aggregated message. Here, $start\_v$ is included in the message and all vertices on each machine can be activated as end_v if they have neighbors that have been a $middle\_v$ of an aggregated message.

Algorithm 2. Distributed Butterfly Counting Algorithm Opt

Input: bipartite graph $G=(U, V, E)$
Output: butterfly counting result for each vertex $u \in U$
1: **Superstep 1:** (Construct the aggregated message)
2: **for** $u \in U \cup V$ **do**
3: step 1: Sort $N(u)$ according to Lemma 1
4: step 2: Construct message according to Strategy 1
5: step 3: Send the aggregated message to all **machines**
6: Synchronize communication message
7: Invoke function $Parse$(**Message**) to calculate $\mathbb{B}_1$ and $\mathbb{B}_2$
8: Send $\mathbb{B}_1$ and $\mathbb{B}_2$ to corresponding $start\_v$ and $middle\_v$ (see line 12 of Algorithm 1)
9: **Superstep 2:** (Update $d_{\bowtie}$)
10: **for** each $start\_v$ **do**
11: $d_{\bowtie} \leftarrow d_{\bowtie} + \mathbb{B}_1$
12: **for** each $middle\_v$ **do**
13: $d_{\bowtie} \leftarrow d_{\bowtie} + \mathbb{B}_2$
14: **return** $d^u_{\bowtie}$ for each vertex $u \in U$

Function Parse(**Message**)

Input: Message
Output: $\mathbb{B}_1$ and $\mathbb{B}_2$ for related vertices
1: **for** msg $\in$ **Message do**
2: Construct mid_hash_map: $key \rightarrow middle\_v$, $value \rightarrow start\_v$ $list$
3: **for** $u \in U \cup V$ **do**
4: **for** $v \in N(u)$ **do**
5: **if** v is a key in mid_hash_map **then**
6: Update $hash\_map$: $key \rightarrow start\_v$, $value \rightarrow middle\_v$ $list$ according to the two kinds of wedges shown in Fig. 4
7: Calculate $\mathbb{B}_1$ and $\mathbb{B}_2$ by Equations (1) and (2)
8: **return** $\mathbb{B}_1$, $\mathbb{B}_2$ for vertices

In the light of Strategy 1, we can parse the butterfly counting results more efficiently. The pseudo-code of the advanced algorithm is stated in Algorithm 2, called DBCA +. The improvement compared to the basic algorithm is mainly reflected in two aspects. First, each vertex generates an aggregated message according to Strategy 1 and sends the aggregated message to each machine (Lines 1-5). These $start\_v$ vertices are packed and sent to machines, which avoid messaging among vertices. Second, after the synchronization of the first superstep (Line 6), the function $Parse$(**Message**) is invoked to parse $\mathbb{B}_1$ and $\mathbb{B}_2$ from the messages received by each machine. As shown in Function Parse, a mid_hash_map is constructed according to **Message** (Lines 1-2). Referring to Fig. 4, we take the $middle\_v$ as key and the $start\_v$ list (higher priority neighbors) as $value$ in the map. Since, each vertex u is activated to check if it has a neighbor v that can be a key of mid_hash_map. If so, u updates its own $hash\_map$ (line 9 of Algorithm 1) according to $mid\_hash\_map[v]$ (Lines 3-6). It needs to append v into $hash\_map[u']$, where u' comes from $mid\_hash\_map[v]$ and has higher priority than u. Finally, $\mathbb{B}_1$ and $\mathbb{B}_2$ of related vertices can be obtained and used to update $d_{\bowtie}$ of vertices (Lines 8-13).

Example. Again, we take the example in Fig. 1 to illustrate how vertex u_2 obtains its $d_{\bowtie}$ by the operations of message aggregation and parsing. Assume that all vertices are distributed on two machines using the hash values of their IDs. According to Strategy 1, u_1, u_2, u_3, and u_4 send message (v_1, v_2, 1), (v_2, v_3, 1), (v_2, v_3, 1), and (v_2, v_3, 1) to the two machines, respectively. We parse these messages as follows by Function Parse. For the message (v_1, v_2, 1) sent by u_1, because v_1 and v_2 are neighbors of u_1 and v_2 has higher priority than v_1, v_1 puts v_2 into the $hash\_map$ (i.e., $hash\_map[v_2] \rightarrow \{u_1\}$). The other messages are processed in the same way. Finally, v_1 keeps the $hash\_map$ as $hash\_map[v_2] \rightarrow \{u_1, u_2, u_3\}$, $hash\_map[v_3] \rightarrow \{u_2, u_3\} v_2$ keeps the $hash\_map$ as $hash\_map[v_3] \rightarrow \{u_2, u_3, u_4\}$. It can be parsed from the $hash\_map$ that v_1 and v_2 have 3 common neighbors u_1, u_2, and u_3. Then $\mathbb{B}_2$ can be set as 2 according to Equation (2). v_2 and v_3 also have 3 common neighbors including u_2, so $\mathbb{B}_2$ has the same value 2. v_1 and v_3 have 2 common neighbors u_2 and u_3, then $\mathbb{B}_2$ equals to 1. As a result, the total number of butterflies containing u_2 is 5. And there are 192 bytes sent for counting $d^{u_2}_{\bowtie}$. It is worth noting that these messages can also be used to calculate the $d_{\bowtie}$ of other vertices.

Theorem 1. ***(Correctness.)*** *The butterfly counting results can be parsed from the aggregated messages correctly.*

Proof. Given a bipartite graph $G=(U, V, E)$, each butterfly in G consists of a vertex that has the highest priority and connects to two vertices with lower priority [21]. For each vertex u belonging to $U \cup V$, it collects its 2-hop neighbors with higher priority from the aggregated messages (Function Parse). And in each aggregated message, the $middle\_v$ also has priority lower than these $start\_v$ (Strategy 1). As a result, the butterfly found by u always has a vertex with the highest priority (i.e., the $start\_v$). Obviously, all butterflies can be found by this paradigm. □

Analysis. Given a bipartite graph $G=(U, V, E)$ and a cluster with j machines, maximal memory allocation may happen at line 5 of Algorithm 2 (i.e., step 3: send the aggregated message to all machines). For each edge $e=(u, v) \in E$, there must be an endpoint vertex with a larger priority according to Lemma 1. Based on Strategy 1, the vertex with larger priority will be in the higher priority neighbor list. Hence, there are $|E|$ vertices in higher priority neighbor lists and $j \times |E|$ integers will be sent. Because all vertices need to send the aggregated message to all machines, they will put their ID and the type flag into the aggregated message. For this purpose, $j \times 2 \times |V|$ integers will be sent. In summary, the space complexity for message passing is $O(j \times (2 \times |V| + |E|)$, which is linear to the size of the graph and much smaller than that of Algorithm 1.

5 Distributed Tip Decomposition and Maintenance

In this section, we focus on the tip decomposition and maintenance problems, especially for those vertices in U (i.e.,

calculate and maintain the tip number of each vertex $u \in U$). Tip decomposition is a necessary pretreatment to obtain the initial tip number of each vertex in a static graph, while tip maintenance is to maintain tip numbers in a dynamic graph.

5.1 Distributed Tip Decomposition

The tip number of a vertex u is the maximum k such that there exists a k-tip containing u, which can be used to query k-tip communities. Tip decomposition is to calculate the tip number of each vertex. According to Definition 4, we can obtain tip numbers of vertices by peeling vertices in a non-increasing order of their butterfly degrees $d_{\bowtie}$. The third stage in Fig. 3 illustrates the process of distributed tip decomposition. First, we peel vertices that have the smallest $d_{\bowtie}$ from the graph. The tip number of a vertex is identified when it is peeled. Then, for the neighbors of these vertices that have been peeled, their tip numbers are updated accordingly. Finally, after peeling all vertices, the tip decomposition process is completed.

The pseudo-code of DTDA has stated in Algorithm 3. Each vertex gets its tip number when it is peeled (Line 7). Especially, considering the parallel computing capability of distributed systems, we peel the vertices with the minimum $d_{\bowtie}$ in batch (Lines 5-9). To get these batch vertices efficiently, we construct a hash table called $CoreTree$ to index vertices with the same tip number. For each leaf in $CoreTree$, the key represents the tip number and the $value$ is a list consisting of the id of vertices with tip number key. Then, in the second superstep, a vertex $v \in V$ may receive messages from neighbors which have been peeled. The message sent in this scenario is formed as $(u\ list, 0)$, where $u\ list$ consists of these peeled vertices that are also neighbors of v, and 0 indicates this kind of message will be received by a vertex belonging to U (Lines 10-12). The remaining vertices update $d_{\bowtie}$ by subtracting sharing butterflies with peeled vertices (Lines 14-16).

Algorithm 3. Distributed Tip Decomposition Algorithm

Input: bipartite graph $G{=}(U, V, E)$
Output: tip number of each vertex $u \in U$
1: Invoke DBCA to get $d^u_{\bowtie}$ of each $u \in U$
2: **repeat**
3: **Superstep 1:** (Activate vertices with the minimum $d_{\bowtie}$)
4: $globalMin(d_{\bowtie}) \leftarrow MPI\_Allreduce(localMin(d_{\bowtie}))$
5: **for** $u \in U$ **do**
6: **if** $d^u_{\bowtie}{=}globalMin(d_{\bowtie})$ **then**
7: $\theta(u) \leftarrow d^u_{\bowtie}$
8: $G \leftarrow G \setminus u$
9: Send message $(u, 1)$ to neighbors
10: **Superstep 2:** (Forward messages)
11: **for** $v \in V$ **do**
12: Forward message $(u\ list, 0)$ to neighbors who have not been peeled (i.e., 2-hop neighbors of peeled vertices)
13: **Superstep 3:** (Update $d_{\bowtie}$)
14: **for** $u \in U$ **do**
15: Calculate $\mathbb{B}_1$ according to the received messages
16: $d^u_{\bowtie} \leftarrow d^u_{\bowtie} - \mathbb{B}_1$
17: **until** G on each machine is empty;
18: **return** $\theta(u)$ for each vertex $u \in U$

Optimization. It is easy to observe that Algorithm 3 also encounters the problem of excessive messaging when processing large-scale graphs. In particular, line 12 will forward so many messages that lead to low efficiency and the memory overflow problem. This is because the same message $(u\ list, 0)$ needs to be sent to neighbors repeatedly. The degree of vertices in a dense bipartite graph is always extremely large. To address this issue, we apply the message aggregation strategy proposed in Section 3.2 Strategy 1 to change the mode of message passing. Specifically, we modify the implementation of Algorithm 3 line 12. Vertex v sends message $(u\ list, 0, u'\ list)$ to all machines instead of its neighbors, where $u'list$ consists of v's neighbors that have tip number larger than $globalMin(d_{\bowtie})$. The other operations remain the same as that of Algorithm 3.

Analysis. Given a bipartite graph $G{=}(U, V, E)$, Algorithm 3 will take at most $3 \times |U|$ supersteps to get the tip numbers of all vertices in U. The time complexity of DTDA is $O(\sum_{v \in V} d(v)^2)$ [17].

5.2 Distributed Tip Maintenance

In this paper, the main challenge is to solve the problem of tip maintenance on bipartite graphs in a distributed environment. It is time-consuming to recalculate the tip numbers of all vertices when a large-scale graph is updated in real-time. To gain better maintenance performance, we explore a distributed incremental algorithm for tip maintenance. The fourth stage in Fig. 3 illustrates the process of distributed tip maintenance. First, a task split strategy (Strategy 2) is utilized to convert a new edge update task into several sub-tasks. In the figure, the red dotted line in the bipartite graph is the new edge. According to Strategy 2, there are three sub-tasks $\{t_1, t_2, t_3\}$ generated and marked as blue dotted lines. Then, for each sub-task, there are two main operations (i.e., Functions CandidateFind and CandidatePeel) invoked to update the tip numbers of vertices in an incremental way. After all sub-tasks are completed, the tip numbers of vertices can be updated correctly.

5.2.1 Theoretical Basis

Given a bipartite graph $G{=}(U, V, E)$, there is an edge $e{=}(u, v)$ inserted into or removed from G. Due to the update of $e{=}(u, v)$, some vertices in U will have their $d_{\bowtie}$ changed, which in turn may change the tip numbers of some vertices.

Observation 1. *Given a bipartite graph $G{=}(U, V, E)$ and an update edge $e{=}(u, v)$, vertices may have their $d_{\bowtie}$ changed are the vertex u and the neighbors of v.*

Proof. The number of sharing butterflies of any two vertices is determined by their common neighbors. v will be a neighbor of u after inserting the new edge $e{=}(u, v)$. Then v is a new common neighbor of u and $u' \in \{u \cup N(v)\}$. As a result, each vertex $u \in \{u \cup N(v)\}$ may has $d_{\bowtie}$ changed. The observation is then proved. □

It is difficult to find the candidate vertices in U whose tip numbers may be changed due to the following two reasons. First, for an edge update, there are several pairs of vertices have sharing butterflies changed. These vertices always have different tip numbers in the original bipartite graph.

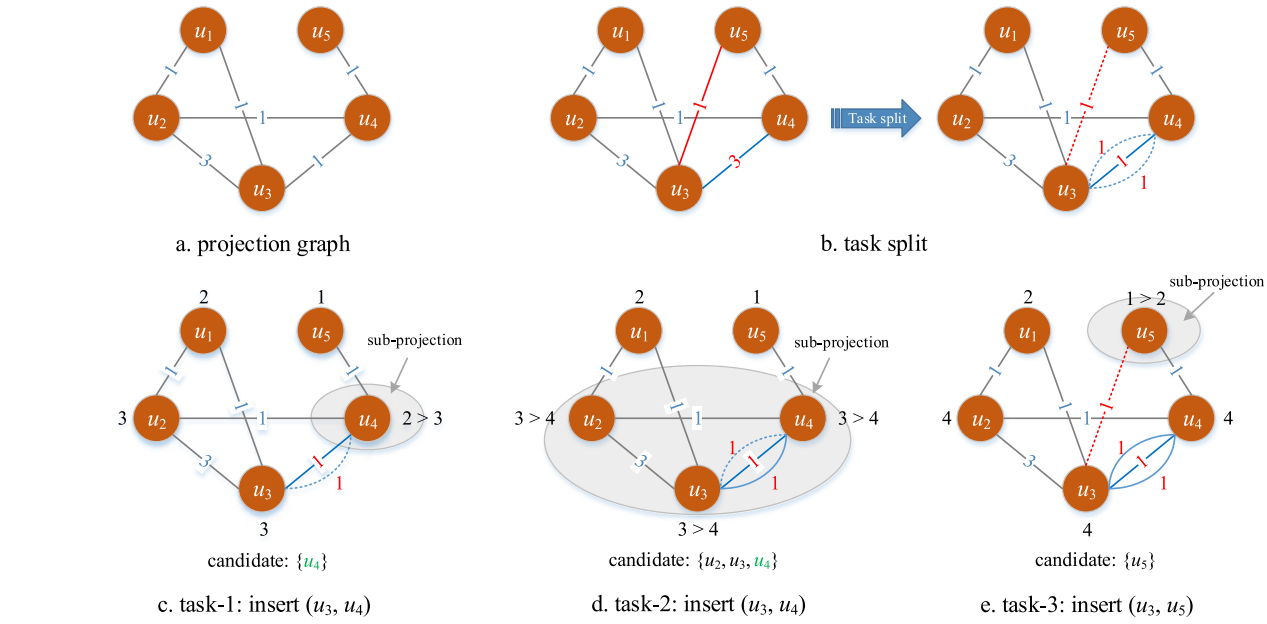

Fig. 5. Task split based tip maintenance.

Second, for each pair of vertices with sharing butterflies changing, the change value may be more than 1. Many researchers have utilized the incremental algorithm to maintain core numbers in unipartite graphs [16], [25], the algorithm can only process edge update without weight (i.e., the edge weight equals 1.) and vertices whose degrees are changed are the two endpoints of the new edge. Therefore, we cannot apply it to solve the tip maintenance problem in bipartite graphs.

In order to facilitate the understanding, we use the projection graph (Fig. 5a) of Fig. 1 to illustrate our methods in the following. The weight of each edge represents the number of sharing butterflies between the two endpoints. But it should be noted that we cannot analyze bipartite graphs by the corresponding projection graphs due to the too large memory overhead (see Section 1). As shown in Fig. 5b, after inserting edge $e=(u_3, v_4)$ into G (Fig. 1), the weight of edge $e=(u_3, u_4)$ in the projection graph changes from 1 to 3 and there is an new edge $e=(u_3, u_5)$ generated with weight 1.

In line with the above analysis and Observation 1, we design a task split strategy to break down this complex problem.

Strategy 2. ***(Task split)*** *Given a bipartite graph $G=(U, V, E)$ and an update edge $e=(u, v)$, for each pair of vertices (u, u') having sharing butterflies changed, there are l sub-tasks $t=(u, u')$ generated to replace the ori-task (i.e., the edge insertion $e=(u, v)$). l is the change value of the weight of (u, u').*

Observation 2. *For each sub-task $t=(u, u')$ generated by Strategy 2, it can be considered to add a sharing butterfly to both u and u'.*

According to Observation 2, we can find the candidate vertices that may have tip numbers changed locally.

Theorem 2. *Given a bipartite graph $G=(U, V, E)$ and a sub-task $t=(u, u')$ with $\theta(u) < \theta(u')$, we have the following theorems.*

- *All vertices in U will have tip numbers changed by at most 1.*
- *Any vertex u'' that may have its tip number updated must have the same tip number with u (i.e., $\theta(u'')=\theta(u)$) and must be connected to u via a butterfly path consisting of vertices that have the same tip numbers.*

Proof. In a bipartite graph $G=(U, V, E)$, the tip number of a vertex $u \in U$ is determined by that of its 2-hop neighbors. According to Observation 2, only u and u' have their $d_{\bowtie}$ changed and the change value is 1. Then the tip numbers of all vertices in U will have tip numbers changed by at most 1.

Because $\theta(u) < \theta(u')$, u cannot support u' to increase $\theta(u')$. Therefore, the first vertex that may have tip number changed is u and the change value is up to 1. Obviously, the sharing butterfly $\bowtie_u^{u'}$ must be in the new $\theta(u')'$-tip $H^{u'}_{\theta(u')'}$, where $\theta(u')'$ is the updated tip number of u and equals $\theta(u')+1$.

For edge insertion, we consider two cases:(i) where $\theta(u_1) > \theta(u)$ and (ii) where $\theta(u_2) < \theta(u)$.

For a vertex u_1 with $\theta(u_1) > \theta(u)$, it will have tip number increased by 1. The new sharing butterfly $\bowtie_u^{u'}$ will be in a $\theta(u_1)'$-tip $H^{u_1}_{\theta(u_1)'}$, where $\theta(u_1)'$ equals $\theta(u_1)+1$. Because $\theta(u)' < \theta(u_1)'$, u cannot participate in $H^{u_1}_{\theta(u_1)'}$, which is a contradiction.

For a vertex u_2 with $\theta(u_2) < \theta(u)$, it will have tip number increased by 1. $\theta(u)$ is the first candidate vertex that has its tip number changed. Therefore, $\theta(u)' > \theta(u_2)'$ and u can participate in $H^{u_2}_{\theta(u_2)'}$. Removing the sharing butterfly $\bowtie_u^{u'}$ from $H^{u_2}_{\theta(u_2)'}$, $\theta(u)''=\theta(u)'-1 \geq \theta(u_2)'$, then $H^{u_2}_{\theta(u_2)'} \setminus \bowtie_u^{u'}$ can still be a $\theta(u)''$-tip. As a result, u_2 can participate in $H^{u_2}_{\theta(u_2)'}$ without $\bowtie_u^{u'}$, i.e., a contradiction.

We use similar arguments for the removal case. Again, we consider two cases.

For a vertex u_1 with $\theta(u_1) > \theta(u)$, it will have tip number decreased by 1. The removed butterfly $\bowtie_u^{u'}$ was not in

$H^{u_1}_{\theta(u_1)}$ due to $\theta(u_1) > \theta(u)$. Thus $H^{u_1}_{\theta(u_1)}$ is still a $\theta(u_1)$-tip after the removal, creating a contradiction.

For a vertex u_2 with $\theta(u_2) < \theta(u)$, it will have tip number decreased by 1. After the removal, we have $\theta(u_2)' < \theta(u)'$. If we put the sharing butterfly back, $\theta(u_2)'$ cannot be changed since $\theta(u_2)' \neq \theta(u)'$. A contradiction is generated.

For satisfying the constraint of connectivity in Definition 3, if a vertex u_3 has $\theta(u_3)=\theta(u)$ but cannot be connected to u via a butterfly path consisting of vertices with tip numbers equaling $\theta(u)$, there is a subgraph H consisting of vertices with tip numbers equaling $\theta(u)$ and $u_3 \in H$. Because the none-induced neighbors of vertices in H will not have tip numbers changed and there is no new butterfly inserted into H, the tip numbers of vertices in H will not be changed. The theorem is then proved. □

Definition 5. ***(Support)*** *Given a bipartite graph $G=(U, V, E)$, the support of a vertex $u \in U$ is the number of butterflies that connect 2-hop neighbors u' where $\theta(u') \geq \theta(u)$.*

Theorem 3. *Given a bipartite graph $G=(U, V, E)$ and a sub-task $t=(u, u')$ with $\theta(u) \leq \theta(u')$, for edge insertion, a candidate vertex increases its tip number by 1 if and only if its support value is larger than $\theta(u)$. For edge deletion, a candidate vertex who has support value smaller than $\theta(u)$ can decrease its tip number by 1.*

Proof. The proof is similar to the theory on unipartite graphs [16], [25]. □

Theorem 4. *Given a bipartite graph $G=(U, V, E)$ and an update edge $e=(u, v)$, the generated sub-task set is $T=\{t_1, t_2, t_3, ...\}$. After all sub-tasks are completed in serial, the tip numbers of vertices in U are updated correctly. (Note: these sub-tasks are performed on the original graph without $e=(u, v)$.)*

Proof. For each sub-task $t=(u, u')$, it can be considered to add a sharing butterfly to u and u'. According to Theorem 2, we can find the candidate vertices that may have tip numbers changed. By Theorem 3, we peel the vertex that cannot get tip number updated by its 2-hop neighbors' tip numbers. After all sub-tasks are completed, these new butterflies generated by the ori-task are all inserted to G. As a result, the final tip numbers can be maintained. The theorem is then proved. □

Example. We use the projection graph in Fig. 5 to illustrate these theorems. As shown in Fig. 5b, after inserting edge $e=(u_3, v_4)$ into G (Fig. 1), there are three sub-tasks $\{t_1=(u_3, u_4), t_2=(u_3, u_4), t_3 = (u_3, u_5)\}$. For sub-task $t_1=(u_3, u_4)$ with $(\theta(u_3)=3) > (\theta(u_4)=2)$, we get the candidate set $\{u_4\}$ according to Theorem 2. It can be calculated that $support(u_4)=3$ after the new sharing butterfly $\bowtie^{u_4}_{u_3}$ inserted is larger $\theta(u_4)$. Thus, $\theta(u_4)$ is increased from 2 to 3. Serially, sub-task $t_2=(u_3, u_4)$ is processed in the following. Since t_2 is performed on the basis of t_1, the sharing butterfly added in t_1 needs to be considered in the calculation process of t_2. Other processes are the same as t_1. The final tip numbers are shown in Figs. 5d and 5e.

5.2.2 Algorithm for Edge Insertion

Based on strategies and theories (see Section 4.2.1), we investigate distributed tip maintenance algorithm (DTMA) to update tip numbers incrementally. Given a bipartite graph $G=(U, V, E)$, there is an edge e that will be inserted into G. For the new edge $e=(u, v)$ (the ori-task), we construct a set of sub-tasks $\{t_1, t_2, t_3, ...\}$. These sub-tasks are processed in serial.

According to Strategy 2, the sub-tasks generated by the same ori-task may have the same endpoints. This character helps speed up candidate vertices selection.

Observation 3. *Given a bipartite graph $G=(U, V, E)$ and an inserted edge $e=(u, v)$, the generated sub-task set is $\{t_1, t_2, t_3, ...\}$. t_1 and t_2 have the same endpoints (u, u'). Assume there is a vertex set $\mathbb{R}$, vertices in $\mathbb{R}$ have tip numbers changed after the completion of t_1, then each vertex in $\mathbb{R}$ must be a candidate vertex of t_2.*

Proof. Give a sub-task $t_1=(u, u')$, there are two cases need to be considered: (i) $\theta(u) < \theta(u')$ and (ii) $\theta(u)=\theta(u')$.

For case (i), according to Theorems 2 and 3, $\mathbb{R}$ consists of vertices with tip number $\theta(u)'$, where $\theta(u)'$ is the updated tip number and equals $\theta(u)+1$. Absolutely, $u \in \mathbb{R}$. Because $\theta(u)' \leq \theta(u')$, the candidate vertices $\mathbb{C}$ of t_2 is connected to u and have tip numbers $\theta(u)'$. Thus, $\mathbb{R} \subseteq \mathbb{C}$.

For case (ii), according to Theorem 2, we have $u \in \mathbb{R}$ and $u' \in \mathbb{R}$. Because the new sharing butterfly must be in $H^{u}_{\theta(u)'}$, u' will also have tip number changed and $\theta(u')'=\theta(u)'$. Therefore, vertices in $\mathbb{R}$ have the same tip number $\theta(u)'$ and are connected to u and u'. The observation is then proved. □

Algorithm 4. Distributed Tip Maintenance Algorithm

Input: bipartite graph $G=(U, V, E)$, an new edge $e = (u, v)$
Output: updated tip number of each vertex $u \in U$

1: Generate sub-task set $T=\{t_1, t_2, t_3, ...\}$
2: **for** $t=(u, u') \in T$ **do**
3: **Phrase 1:** Initialize sub-task
4: $\theta(t) \leftarrow min(\theta(u), \theta(u'))$
5: Put u/u' with smaller tip number into candidate set $\mathbb{C}$
6: Expand $\mathbb{C}$ by Observation 3
7: **for** $u \in \mathbb{C}$ **do**
8: Construct sub_map: $u' \rightarrow (s, \theta(u'))$, where s is the number of sub-tasks with endpoints (u, u')
9: **Phrase 2:** Find candidate
10: CandidateFind(sub-task $t=(u, u')$)
11: **Phrase 3:** Peel candidate
12: CandidatePeel (sub-projection, $\mathbb{C}$)
13: **for** $u \in \mathbb{C}$ **do**
14: $\theta(u) \leftarrow \theta(u) + 1$

DTMA for Edge Insertion. Given a bipartite graph $G=(U, V, E)$, there is an edge $e = (u, v)$ inserted into G. The sub-task set $T = \{t_1, t_2, t_3, ...\}$ can be generated by Strategy 2 (Line 1). There are three main phrases for each sub-task, including sub-task initial, find candidate, and peel candidate. In the sub-task initial phrase, we first initialize the sub-task $t = (u, u')$. The endpoint with a smaller tip number (i.e., $\theta(u')$) is selected as a candidate (Lines 4-5). In addition, we expand $\mathbb{C}$ by Observation 3 (Line 6). In the find candidate phrase, we apply a BFS based manner to find candidate vertices via a butterfly connected path consisting of vertices with tip number $\theta(u')$ (see Function CandidateFind). Here, we only consider the vertices in U. Thus, vertices in V just need to forward messages among different $u \in U$. In particular,

we construct a sub-projection graph consisting of candidates to avoid 2-hop communication in the third phase (i.e., peel candidate). The sub-projection graph for a sub-task contains the following information. (1) 2-hop adjacency list. An edge needs to be created between two candidates if they have sharing butterflies. (2) weight of these new edges. The weight is equal to the number of sharing butterflies of two vertices. What's more, when a sub-task is being processed, the corresponding edge insertion e=(u, v) has not been put into G. To address this issue, we construct a sub_map for vertices, which are endpoints of sub-tasks, to store the sharing butterflies with its 2-hop neighbors. sub_map can be used to find new candidates that cannot be found in the original graph (line 14 of Function CandidateFind) and update $support$ value of the vertex (line 18 of Function CandidateFind). In the third phrase, we peel candidates from $\mathbb{C}$ if their $support$ values are not larger than $\theta(u')$ and update their 2-hop neighbors' $support$ value by the sub-projection graph directly (lines 1-6 of Function CandidatePeel). Finally, those vertices still in $\mathbb{C}$ increase their tip number by 1 (Lines 13-14).

Function CandidateFind (Sub-Task $t = (u, u')$)

Input: sub-task $t = (u, u')$
Output: candidate set $\mathbb{C}$
1: **Superstep 1:** (Activate ancestral candidates)
2: Candidate vertices send message $(u, 1)$ to neighbors
3: **Superstep 2:** (Forward messages)
4: **for** v that has received messages **do**
5: Construct a candidate list $\mathbb{L}$ by message[0]
6: Send message $(v, 0, \mathbb{L})$ to $u' \in N(v)$ with $\theta(u') \geq \theta(t)$
7: **Superstep 3:** (Find new candidates and sharing butterflies)
8: **for** u' that has received messages **do**
9: **if** $\theta(u')$=$\theta(t)$ **then**
10: $\mathbb{C} \leftarrow u'$
11: Calculate $\mathbb{B}_1$ of each u in $\mathbb{L}$
12: Construct sub-projection according to $\mathbb{L}$ and $\mathbb{B}_1$
13: Send message $(u', 0, \mathbb{B}_1)$ to corresponding u
14: Find new candidates from sub_map and put into $\mathbb{C}$
15: **Superstep 4:** (Calculate $support$ of ancestral candidates)
16: **for** u that has received messages **do**
17: Calculate $support(u)$ by $\mathbb{B}_1$ in received messages
18: Update $support(u)$ according to sub_map
19: Update sub-projection according to messages and $\mathbb{B}_1$
20: Repeat Superstep 1-4 until no new candidate is found

Function CandidatePeel (Sub-Projection, $\mathbb{C}$)

Input: sub-projection, $\mathbb{C}$
1: **Superstep 1:** (Peel candidate vertex)
2: **for** $u \in \mathbb{C}$ and $support(u) \leq \theta(t)$ **do**
3: $\mathbb{C} \leftarrow \mathbb{C} \setminus u$
4: Send message $(u, 0, 2{-}hop\_neighbor)$ // $2{-}hop\_neighbor$ can be find in the sub-projection
5: **Superstep 2:** (Update $support$)
6: Update $support$ for u that have received messages
7: Repeat Superstep 1-2 until no candidate needs to be peeled

Example. In Fig. 5, the sub-task set generated from the new edge e=(u_3, v_4) is T=$\{t_1$=$(u_3, u_4), t_2$=$(u_3, u_4), t_3$= $(u_3, u_5)\}$. We first process t_1=(u_3, u_4). The two endpoints u_3 and u_4 construct a sub_map as $sub\_map[u_4]$ $\rightarrow\theta(u_4)$ and $sub\_map[u_3]$ $\rightarrow\theta(u_3)$, respectively. u_4 is the first candidate. u_2, u_3, and u_5 are the 2-hop neighbors of u_4 and can be accessed by lines 1-13 of Function CandidateFind. In this process, the sub-projection graph is constructed by candidates. u_2 and u_3 send message $(u_2/u_3, 0, 1)$ to u_4, where 0 indicates the vertex in U will receive this message and 1 is the number of sharing butterflies between u_2/u_3 and u_4. According to these messages and sub_map, we can obtain the value of $support(u_4)$. For this sub-task, $\mathbb{C}$=$\{u_4\}$, $support(u_4)$=3, then $\theta(u_4)$ is increased from 2 to 3. Next, sub-task t_2=(u_3, u_4) is processed serially. By Observation 3, u_4 can be put into $\mathbb{C}$ directly. Because t_1 and t_2 have same endpoints and the tip number of u_4 is changed after the completion of t_1. For t_2, $\mathbb{C}$=$\{u_2, u_3, u_4\}$, $support(u_2)$=4, $support(u_3)$=6, and $support(u_4)$=4, then $\theta(u_2), \theta(u_3)$, and $\theta(u_4)$ are increased from 3 to 4. The sub-task t_3 is handled in a similar manner.

5.2.3 Discussion for Edge Deletion

Given a bipartite graph G=(U, V, E), there is an edge e=(u, v) deleted from G. Similarly, we generate a sub-task set T=$\{t_1, t_2, t_3, ...\}$ according to Strategy 2. These sub-tasks are processed in serial. For each sub-task t, the endpoint(s) with smaller tip number is(are) set as candidate(s). Different from the process of edge insertion, we don't need to find all candidates and apply the $CandidatePeel$ function to peel candidates that will not have tip numbers changed. Based on Theorem 3, if the $support$ value of a candidate vertex is smaller than its tip number, we can decrease its tip number by 1 immediately. In addition, only the vertex who have tip number changed can expand more candidates. Specifically, we can determine if the tip number of a vertex u changes at Superstep 4 of Function CandidateFind (Line 18). The sub-task completes when no vertex tip number has changed.

As shown in Fig. 5e, the tip number of vertices in U is updated by DTMA. Assuming that we delete edge (u_3, u_4) from Fig. 1, the sub-task set is T=$\{t_1$=$(u_3, u_4), t_2$=$(u_3, u_4), t_3$ =$(u_3, u_5)\}$, which is the same as the insertion scenario. We remove those edges in Fig. 5e one by one. For sub-task t_1, the initial candidates are u_3 and u_4. $support(u_4)$ reduces from 4 to 3 and its tip number is reduced by 1. Then, we need to update $support(u_3)$ to 3 and $\theta(u_3)$ to 3. Now, u_3 cannot support u_2 to maintain its tip number 4. According to Definition 4, $\theta(u_3)$ should be updated to 3. For sub-task t_2, $support(u_4)$ reduces from 3 to 2 and $\theta(u_4)$ should be 2. $support(u_2)$ and $support(u_3)$ are equal to 3, which indicates that u_2 and u_3 can keep their tip numbers. For sub-task t_3, the only influenced vertex is u_5. $\theta(u_5)$ reduces from 2 to 1. Finally, all vertices get their new tip numbers.

6 Experimental Evaluation

In this section, we conduct comprehensive experiments to evaluate the performance of our algorithms.

6.1 Experiment Setting

Schemes. To our best knowledge, this is the first study to process the butterfly counting, tip decomposition, and tip maintenance problems in a distributed environment. We apply the vertex priority paradigm [21] to count butterflies for each vertex (Algorithm 1) and set it as a baseline, denoted as DBCA, which can be seen as a distributed version of the

TABLE 2
Real-World Bipartite Graph Datasets

Graph	$\|E\|$	$\|U\|$	$\|V\|$	RES
discogs	5,302,276	1,754,823	270,771	4.5 GB
amazon	5,743,258	2,146,057	1,230,915	7.1 GB
dblp-au	12,282,059	1,953,085	5,624,219	15.9 GB
Google+	20,592,961	5,998,790	4,443,631	22.2 GB
orkut	327,037,487	8,730,857	2,783,196	56.1 GB
RMAT-23	29,821,707	2,497,203	2,402,566	12.4 GB
RMAT-24	73,942,504	4,971,665	4,986,525	27.0 GB
RMAT-25	149,842,312	9,976,161	9,759,895	53.5 GB
RMAT-26	261,740,855	18,713,683	18,608,791	100.0 GB

algorithm proposed in [21]. For Algorithm 2, a novel message aggregation strategy is introduced into DBCA, denoted as DBCA+. Based on the result of butterfly counting, Algorithm 3 (DTDA) is designed to compute the tip number of each vertex. Also, the optimized version with a message aggregation strategy is denoted as DTDA+. Then we evaluate the performance of Algorithm 4 (DTMA) by comparing the time cost and superstep cost of maintaining tip numbers for per edge update with Algorithm 3 (DTDA). In addition, we verify the scalability of all algorithms while varying the number of machines.

Datasets. The performance of the proposed algorithms is evaluated on five real-world and four synthetic bipartite graphs. Table 2 shows the properties of the five real-world graphs, including dblp-au, discogs, amazon, Google +(IMC12), and orkut. All of these datasets can be downloaded from http://konect.cc/. For synthetic datasets, SNAP [11] python package is used to generate graphs with varying vertex numbers from 2^{23} to 2^{26} following R-MAT model [3]. We refer to the parameter configuration for dataset generation in [25]. After that, we convert these synthetic datasets into bipartite graphs by the method proposed in [21], called RMAT-23 to 26, in which RMAT-25 and RMAT-26 have billion edges. In a vertex-centric system, each vertex is an independent object. The memory cost of this storage mode is much larger than that of CSR (Compressed Sparse Row) or CSC (Compressed Sparse Column). The resident memory usage (RES) after loading each dataset is shown in the last column of Table 2.

Environment. The proposed algorithms and distributed graph computing system are implemented in C++ and compiled by mpicxx, which compiles and links MPI programs written in C++. The experiments are deployed on the National Supercomputing Center in Changsha, named TH-I. TH-I has Intel Quad Core Xeon E5540 2.53 GHz/E5540 3.0 GHz CPU and 32 GB of RAM for each machine. We conduct the experiments on 10 nodes of TH-I, and the total memory is about 320 GB. As for the dataset orkut and synthetic graphs, we experiment by starting 10 MPI processes on a single server with 256 GB memory due to the limited running time of the distributed cluster.

6.2 Evaluation of Algorithms

In the following, we evaluate the performance of different algorithms in line with the experiment setting.

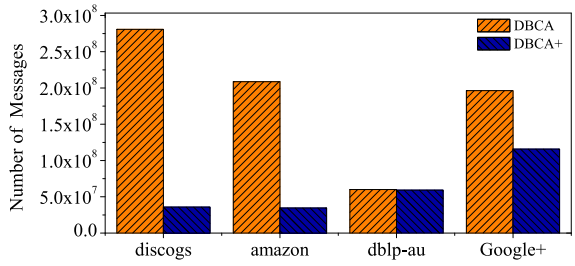

Fig. 6. Communication of butterfly counting.

6.2.1 Performance of Butterfly Counting

Memory Usage. Memory usage is linear with the number of messages and Strategy 1 focuses on reducing redundant messages. We compare the number of messages generated by DBCA and DBCA+. Because each superstep in a vertex-centric model is independent, the memory overflow will happen in the superstep that generates the most messages. As analyzed in Section 4, the superstep 1 in DBCA (i.e., lines 1-4) and the step 3 of superstep 1 in DBCA+ (i.e., line 5) will generate the most messages. Fig. 6 shows the number of messages of the superstep that generate the most messages while counting butterflies for each vertex. The traffic has been reduced by up to 87.2%. But for dblp-au, it is the sparsest one among all datasets, in which the number of edges and vertices are on the same order of magnitude. As a result, the potential of Strategy 1 is not well realized. For dataset orkut, the memory overflow problem happened and the results cannot be obtained if we apply DBCA rather than DBCA. In summary, our proposed distributed butterfly counting algorithm can be applied to handle large scale graphs and avoid the memory overflow problem effectively.

Time Cost. We evaluate the time cost of butterfly counting algorithms with 10 machines on TH-1 and the results are shown in Fig. 7. The performance of DBCA+ on these datasets is better than DBCA. The degree of improvement varies for different datasets. It should be noted that the main purpose of the message aggregation strategy (Strategy 1) is to solve the memory overflow problem due to the large number of messages generated while processing large-scale graphs. There is an out of memory exception thrown while processing the dataset orkut. But we can use DBCA+ to get the butterfly counting results of orkut. This further supports the effectiveness of Strategy 1.

Varying the Number of Machines. The number of machines in a distributed system reflects its parallel computing capabilities. In this set of experiments, we evaluate the scalability of DBCA+ while varying the number of machines. Fig. 8 presents the trend of time cost with a dwindling number of machines, where "Node-*" indicates the number of machines used in the experiment. Due the memory limitation of a single machine in our distributed environment, we set "Node-2" as the initial reference. Therefore, the improvement rate of "Node-2" is 1 for every datasets. As for other configurations, we exhibit the improvement rate comparing to its former one. For example, given a specific dataset, the improvement rate of Node-8 is calculated using:

$$rate = \frac{T_{Node-5} - T_{Node-8}}{T_{Node-5}}. \quad (3)$$

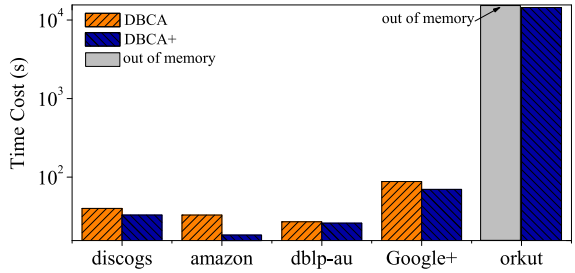

Fig. 7. The time cost of butterfly counting.

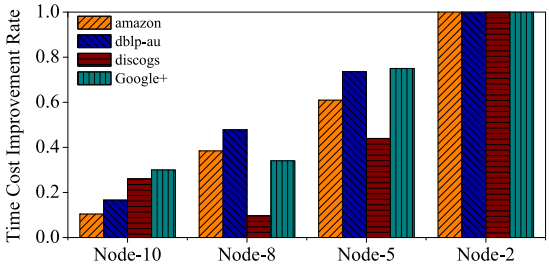

Fig. 8. The improvement rate of time cost of butterfly counting with varying machines.

Obviously, the descending slope of time cost is inversely proportional to the number of machines. The single most striking observation to emerge from the data comparison was that as the number of machines increases, the rate of the time cost decline tends to converge. Although more machines can provide more computing power, they also introduce more cross edges, which in turn increases the communication cost.

6.2.2 Performance of Tip Decomposition and Maintenance

A. Tip decomposition

Time Cost. The distributed tip decomposition algorithm (DTDA) is designed to compute the tip numbers of vertices in a given bipartite graph. The time cost of DTDA and DTDA+ for each dataset on 10 nodes is shown in Table 3. *CoreTree* is constructed to split vertices according to tip numbers. It will be used in both DTDA and DTDA+. The symbol "\" indicates the memory overflow that happened when we apply DTDA to decompose graphs. Also, we count the number of messages sent in the second superstep (line 12 of Algorithm 3), which produces the most messages. It can be observed that DTDA+ shows better performance on both efficiency and effectiveness for most datasets. But for the dataset dblp-au, DTDA+ appears to be less efficient than the baseline. Referring to the message column of Table 3, the number of messages sent by DTDA+ is one order of magnitude more than that of DTDA. This is because there are too many vertices that have degrees less than the number of machines used in the experiment. Here, we use 10 machines to evaluate our experiments. As shown in Fig. 9, 99.6% vertices in set V of dataset dblp-au have degree less than 10. DTDA+ sends a message to 10 machines in the second superstep. As a result, much more messages are introduced, which leads to lower efficiency. But for large-scale graphs, for the dataset orkut, there are only 25.6% vertices that have degrees less than 10. In summary, DTDA+ is more applicable for processing large-scale dense graphs.

TABLE 3
Performance of DTDA and DTDA+

Dataset	Time Cost (s)			Message	
	CoreTree	DTDA	DTDA+	DTDA	DTDA+
discogs	20	\	1370	\	3.6E7
amazon	55	114	43	2.2E8	3.7E7
dblp-au	22	15	28	1.5E7	1.1E8
Google+	142	\	3964	\	1.2E8
orkut	17	-	155148	-	-

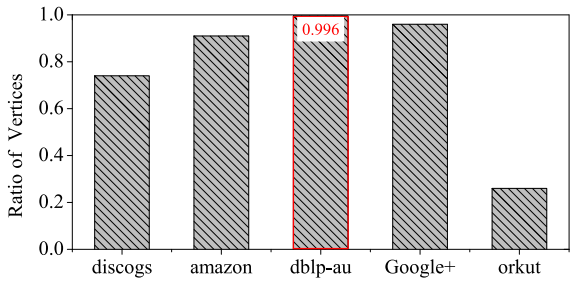

Fig. 9. Ratio of vertices have degree less than 10.

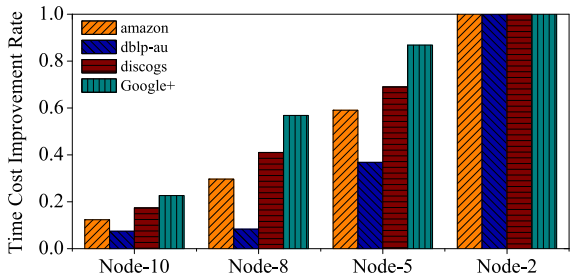

Fig. 10. The improvement rate of time cost of DTDA+ with varying machines.

Varying the Number of Machines. We have also verified the scalability of DTDA+ with varying the number of machines. The results are shown in Fig. 10 and are similar to the trend of DTDA+. Here we show the results and will not repeat the analysis.

B. Tip maintenance

Task Generation. According to Strategy 2, for an update edge $e{=}(u, v)$, the number of sub-tasks is closely related to the number of common neighbors between u and $u' \in N(v)$. As shown in Fig. 1, if we insert an edge $e{=}(u_1, v_4)$ into G, there is still no sharing butterflies containing both u_1 and u_5. Thus, no sub-task has endpoints (u_1, u_5). Assuming that edge $e{=}(u_4, v_2)$ does not exist, u_1 and u_4 would also have no sharing butterfly. Then there would be no sub-task generated. In this case, we do not need to apply tip decomposition or tip maintenance algorithm to maintain the tip numbers of vertices. Therefore, it does not make sense to evaluate the performance with this kind of edge update. To avoid this issue, we start with a random search for two vertices $u_1, u_2 \in U$ such that $N(u_1) \cap N(u_2) \neq \emptyset$. Then we randomly select two vertices $v_1, v_2 \in N(u_2) \wedge v_1, v_2 \notin N(u_1)$ and generate the new ori-tasks as $e_1{=}(u_1, v_1), e_2{=}(u_1, v_2)$. For

TABLE 4
Tasks Generation

Task	discogs	amazon	dblp	google	orkut
edge insertion	500	500	500	500	500
sub-task	146,778	1,396	6,047	18,816	3,420

TABLE 5
Time Cost of Tip Maintenance

Dataset	Time Cost		Improvement (%)
	500 edges	per edge	
discogs	4054.4 s	8.1 s	99.41
amazon	14.5 s	29 ms	99.93
dblp-au	120.0 s	240 ms	99.14
Google+	4411.9 s	8.8 s	99.78
orkut	28607.0 s	57.2 s	99.96

TABLE 6
Superstep Cost of Tip Maintenance

Dataset	Superstep Cost			Improvement(%)
	DTDA	DTMA		
		500 edges	per edge	
discogs	154,619	1,128,121	2,256	98.54
amazon	18,001	9,882	20	99.89
dblp-au	17843	46,429	93	99.48
Google+	64,910	260,387	521	99.20
orkut	997,137	20,900	42	99.99

each dataset, we generate 500 new edges and the number of sub-tasks is shown in Table 4.

To demonstrate the correctness of the proposed tip number maintenance algorithms, we use Algorithms 2 and 3 to get the updated tip numbers of all vertices after inserting the 500 new edges. Then we compare the results with the results generated by DTMA. After comparing one by one, the two methods obtain the same results. This indicates that DTMA can maintain tip numbers effectively.

Time Cost. In this set of experiments, we evaluate the efficiency of DTMA (Algorithm 4). Based on the task generation scheme, we generate 500 new edges and accumulate the total time that DTMA takes to process the 500 edge insertions. Then, we compare the time cost per edge update of DTMA (Algorithm 4) with DTDA+ (Algorithm 3). It can be seen from the data in Table 5 that the time cost per edge is significantly less than that of DTDA+ (see Table 3). The improvement rate is calculated by

$$rate = \frac{T_{DTDA+} - T_{per}}{T_{DTDA+}}, \quad (4)$$

where T_{DTDA+} and T_{per} represents the time cost of DTDA+ and DTMA (per edge), respectively. Specifically, the time cost is reduced by 2 to 4 orders of magnitude. This is because the number of impacted vertices that may have tip numbers changed is much less than the size of the original graph while inserting a new edge. DTMA incrementally maintains tip numbers, which significantly reduces redundant calculations.

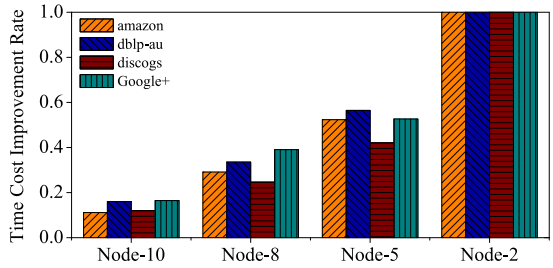

Fig. 11. The improvement rate of time cost of tip maintenance with varying machines.

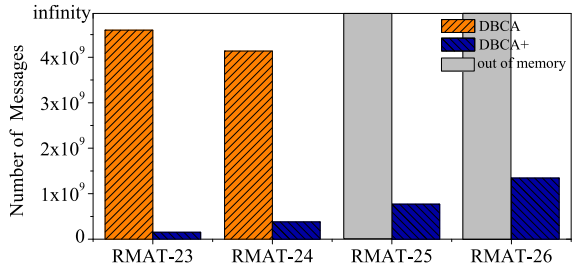

Fig. 12. Communication of butterfly counting on RMAT-23 and RMAT-24.

Superstep Cost. Massively parallel algorithms in vertex-centric model admit the number of supersteps. For butterfly counting and tip decomposition, the message aggregation strategy is designed to reduce the number of messages. So we don't evaluate this factor of the two algorithms. We compare the superstep cost between DTDA and DTMA and the results are shown in Table 6. For each edge update, the total superstep cost by DTMA is reduced by 98.54% to 99.99% (the last column of Table 6). Here, the improvement rate is calculated similarly with Equation (4). It can be seen that the superstep of 500 edges may be larger than the cost of DTDA (the blue marked results in the third column). Especially for discogs, the superstep cost is nearly an order of magnitude larger. This is because the dataset is denser than others, in which the average degree or the tip number is much larger. Then there are more subtasks generated after the process of task generation (see Table 4). As a result, it needs more superstep to maintain tip numbers while processing dense graphs.

Varying the Number of Machines. We illustrate the scalability of DTMA+ (Algorithm 4) while varying the number of machines. As shown in Fig. 11, a clear benefit of the scale of the distributed environment in the reduction of time cost could be identified in this analysis. Also, the rate of the time cost decline tends to converge with the increase of the number of machines.

6.2.3 Evaluation on Synthetic Bipartite Graphs

To verify the scalability of our proposed algorithms, we conduct experiments on four different scale synthetic graphs. Because the maximum memory usage happens at line 5 of Algorithm 2 , which sends the aggregated message to all machines for each vertex, we first compare the number of messages sent by DBCA and DBCA+. Fig. 12 shows the

TABLE 7
Tasks Generation for Synthetic Graphs

Task	RMAT-23	RMAT-24	RMAT-25	RMAT-26
edge insertion	500	500	500	500
sub-task	17324	37351	33175	17223
time(s)/task	3.2	14.3	22.6	20.4

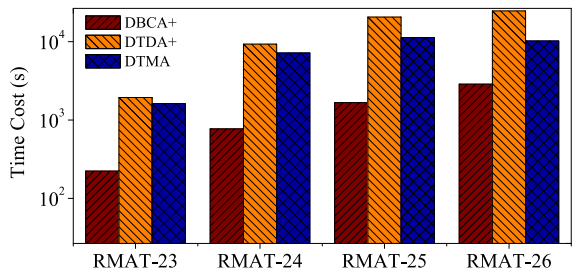

Fig. 13. Time cost of DBCA+, DTDA+, and DTMA on RMAT-23 and RMAT-24. Here, the time cost of DTMA is the total time consumption of processing 500 new edges.

results of the supersteps of DBCA and DBCA+ that generate the maximum messages. For RMAT-23 and RMAT-24, DBCA+ saves 96.7% and 90.8% messages compared to DBCA, respectively. For RMAT-25 and RMAT-26, the memory overflow problem will occur if we apply DBCA while DBCA+ can get the results accurately. We use the gray bar to indicate the memory overflow case in the figure. In addition, it can be observed that the number of messages grows linearly as the graph size grows. Summarily, our proposed algorithms can be applied to process billion scale graphs.

Then, we evaluate the time cost of DBCA+, DTDA+, and DTMA on given synthetic graphs. Here, we also insert 500 new edges into each synthetic graph. The number of sub-tasks and the average time cost of a task is shown in Table 7. As shown in Fig. 13, for each edge update, DTMA can save more than 99.8% time cost on average compared to DTDA+. Therefore, our proposed tip maintenance algorithm can be used to maintain tip numbers on dynamic bipartite graphs efficiently.

6.2.4 Discussion

We discuss the superstep cost of the tip maintenance algorithm for edge deletion in this section. We generate a sub-task set from the deleted edge according to Strategy 2. For each sub-task, we find its candidates based on Theorem 2. Here, the most important difference compared to the edge insertion scenario is that we do not need to find all candidates. Similar to the core maintenance problem on unipartite graphs [25], the tip number of a candidate can be determined when it is accessed. Although some 2-hop neighbors of a candidate have the same tip number with it, they will not be accessed if the current candidate's tip number does not change. But for the edge insertion scenario, we need to access all candidates according to Function CandidateFind and prune candidates by Function CandidatePeel. Given a sub-task $t{=}(u, u')$, u has a smaller tip number and the distance between u and the farthest candidate from u is Φ. For edge insertion, it needs at least $2 \times \Phi$ supersteps to find all candidates and at most Φ supersteps to prune candidates. As a result, the minimum and maximal superstep cost of a sub-task for edge insertion are $2 \times \Phi$ and $3 \times \Phi$, respectively. But for edge deletion, if u do not need to update its tip number, the process is completed. Because there is no candidate pruning operation in processing edge deletion, the minimum and maximal superstep cost of a sub-task for edge deletion are 1 and $2 \times \Phi$, respectively.

7 Conclusion

In this paper, we have investigated butterfly counting, tip decomposition, and tip maintenance problems in larger-scale dynamic bipartite graphs on a distributed environment for the first time. Specifically, the novel message aggregation is proposed to improve the efficiency and effectiveness of butterfly counting. Our algorithms have been tested on the vertex-centric graph computing system deployed on TH-I. The experimental results on real-world bipartite graphs have validated the efficiency and effectiveness of our algorithms.

In the future, we will investigate the tip maintenance problems when edge insertion and deletion happen at the same time. In addition, most existing dynamic graph compression algorithms cannot be directly applied to distributed systems [9], [12]. In [27], a distributed data compression framework was proposed which focuses on orthogonal processing on compression. It is also interesting to introduce this technique to further improve memory efficiency.

Acknowledgments

The authors would like to thank the three anonymous reviewers for their kind suggestions.

References

[1] A. Abidi, L. Chen, R. Zhou, and C. Liu, "Searching personalized k-wing in large and dynamic bipartite graphs," 2021, *arXiv:2101.00810*.
[2] E. Akbas and P. Zhao, "Truss-based community search: A truss-equivalence based indexing approach," *Proc. VLDB Endowment*, vol. 10, no. 11, pp. 1298–1309, 2017.
[3] D. Chakrabarti, Y. Zhan, and C. Faloutsos, "R-mat: A recursive model for graph mining," in *Proc. SIAM Int. Conf. Data Mining*, 2004, pp. 442–446.
[4] N. Chiba and T. Nishizeki, "Arboricity and subgraph listing algorithms," *SIAM J. Comput.*, vol. 14, no. 1, pp. 210–223, 1985.
[5] I. S. Dhillon, "Co-clustering documents and words using bipartite spectral graph partitioning," in *Proc. 7th ACM SIGKDD Int. Conf. Knowl. Discov. Data Mining*, 2001, pp. 269–274.
[6] S. N. Dorogovtsev, A. V. Goltsev, and J. F. F. Mendes, "K-core organization of complex networks," *Phys. Rev. Lett.*, vol. 96, no. 4, 2006, Art. no. 040601.
[7] D. Gibson, R. Kumar, and A. Tomkins, "Discovering large dense subgraphs in massive graphs," in *Proc. 31st Int. Conf. Very large Data Bases*, 2005, pp. 721–732.
[8] X. Huang, H. Cheng, L. Qin, W. Tian, and J. X. Yu, "Querying k-truss community in large and dynamic graphs," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2014, pp. 1311–1322.
[9] J. Ko, Y. Kook, and K. Shin, "Incremental lossless graph summarization," in *Proc. 26th ACM SIGKDD Int. Conf. Knowl. Discov. Data Mining*, 2020, pp. 317–327.
[10] K. Lakhotia, R. Kannan, V. Prasanna, and C. A. F. De Rose, "Receipt: Refine coarse-grained independent tasks for parallel tip decomposition of bipartite graphs," 2020, *arXiv:2010.08695*.
[11] J. Leskovec and R. Sosič, "SNAP: A general-purpose network analysis and graph-mining library," *ACM Trans. Intell. Syst. Technol.*, vol. 8, no. 1, pp. 1–20, 2016.

[12] Z. Ma, J. Yang, K. Li, Y. Liu, X. Zhou, and Y. Hu, "A parameter-free approach for lossless streaming graph summarization," in *Proc. Int. Conf. Database Syst. Adv. Appl.*, 2021, pp. 385–393.
[13] G. Malewicz et al., "Pregel: A system for large-scale graph processing," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2010, pp. 135–146.
[14] M. EJ Newman, "Scientific collaboration networks. i. network construction and fundamental results," *Phys. Rev. E*, vol. 64, no. 1, 2001, Art. no. 016131.
[15] S.-V. Sanei-Mehri, A. E. Sariyuce, and S. Tirthapura, "Butterfly counting in bipartite networks," in *Proc. 24th ACM SIGKDD Int. Conf. Knowl. Discov. Data Mining*, 2018, pp. 2150–2159.
[16] A. E. Sariyüce, B. Gedik, G. Jacques-Silva, K.-L. Wu, and Ü. V. Çatalyürek, "Streaming algorithms for k-core decomposition," *Proc. VLDB Endowment*, vol. 6, no. 6, pp. 433–444, 2013.
[17] A. E. Sarıyüce and A. Pinar, "Peeling bipartite networks for dense subgraph discovery," in *Proc. 11th ACM Int. Conf. Web Search Data Mining*, 2018, pp. 504–512.
[18] A. Sheshbolouki and M. T. Özsu, "sGrapp: Butterfly approximation in streaming graphs," *ACM Trans. Knowl. Discov. Data*, vol. 16, no. 4, pp. 1–43, 2022.
[19] J. Shi and J. Shun, "Parallel algorithms for butterfly computations," 2019, *arXiv:1907.08607*.
[20] J. Wang, A. W.-C. Fu, and J. Cheng, "Rectangle counting in large bipartite graphs," in *Proc. IEEE Int. Congr. Big Data*, 2014, pp. 17–24.
[21] K. Wang, X. Lin, L. Qin, W. Zhang, and Y. Zhang, "Vertex priority based butterfly counting for large-scale bipartite networks," *Proc. VLDB Endowment*, vol. 12, no. 10, pp. 1139–1152, 2019.
[22] K. Wang, X. Lin, L. Qin, W. Zhang, and Y. Zhang, "Efficient bitruss decomposition for large-scale bipartite graphs," in *Proc. IEEE 36th Int. Conf. Data Eng.*, 2020, pp. 661–672.
[23] K. Wang, W. Zhang, X. Lin, Y. Zhang, L. Qin, and Y. Zhang, "Efficient and effective community search on large-scale bipartite graphs," in *Proc. IEEE 37th Int. Conf. Data Eng.*, 2021, pp. 85–96.
[24] X. Wang, D. Wen, L. Qin, L. Chang, Y. Zhang, and W. Zhang, "Scaleg: A distributed disk-based system for vertex-centric graph processing," *IEEE Trans. Knowl. Data Eng.*, to be published, doi: 10.1109/TKDE.2021.3101057.
[25] T. Weng, X. Zhou, K. Li, P. Peng, and K. Li, "Efficient distributed approaches to core maintenance on large dynamic graphs," *IEEE Trans. Parallel Distrib. Syst.*, vol. 33, no. 1, pp. 129–143, Jan. 2022.
[26] D. Yan et al., "A general-purpose query-centric framework for querying big graphs," *Proc. VLDB Endowment*, vol. 9, no. 7, pp. 564–575, 2016.
[27] F. Zhang, J. Zhai, X. Shen, O. Mutlu, and X. Du, "POCLib: A high-performance framework for enabling near orthogonal processing on compression," *IEEE Trans. Parallel Distrib. Syst.*, vol. 33, no. 2, pp. 459–475, Feb. 2022.

Tongfeng Weng received the master's degree from the School of Information Engineering, Wuhan University of Technology, in 2018. He is currently working toward the doctoral degree with the College of Computer Science and Electronic Engineering, Hunan University, Changsha, China. His research interests include distributed graph computing and parallel computing.

Xu Zhou received the master's degree from the College of Computer Science and Electronic Engineering, Hunan University, in 2009. She is currently an associate professor with the Department of Information Science and Engineering, Hunan University, Changsha, China. Her research interests include parallel computing and data management.

Kenli Li (Senior Member, IEEE) received the PhD degree in computer science from the Huazhong University of Science and Technology, China, in 2003. He is currently a Cheung Kong professor of computer science and technology with Hunan University, the dean with the College of Computer Science and Electronic Engineering, Hunan University. His major research interests include parallel and distributed processing. He serves on the editorial board for *IEEE Transactions on Computers*.

Kian-Lee Tan (Senior Member, IEEE) received the PhD degree in computer science from the National University of Singapore, Singapore, in 1994. He is a professor of computer science with the School of Computing, National University of Singapore (NUS), Singapore. His current research interests include query processing and optimization, database performance, data science, and distributed graph computing. He is a member of ACM.

Keqin Li (Fellow, IEEE) is a SUNY distinguished professor of Computer Science with the State University of New York. He is also a national distinguished professor with Hunan University, China. His current research interests include cloud computing, fog computing and mobile edge computing, energy-efficient computing and communication, embedded systems and computer architectures and systems. He has authored or coauthored more than 860 journal articles, book chapters, and refereed conference papers, and has received several best paper awards. He is an AAIA fellow. He is also a member of Academia Europaea.