Towards Distributed Bitruss Decomposition on Bipartite Graphs

Yue Wang*
Shenzhen Institute of Computing Sciences
yuewang@sics.ac.cn

Ruiqi Xu*
National University of Singapore
ruiqi.xu@nus.edu.sg

Xun Jian
Alexander Zhou
Lei Chen
Hong Kong University of Science and Technology
{xjian,atzhou@connect,leichen@cse}.ust.hk

ABSTRACT

Mining cohesive subgraphs on bipartite graphs is an important task. The k-bitruss is one of many popular cohesive subgraph models, which is the maximal subgraph where each edge is contained in at least k butterflies. The bitruss decomposition problem is to find all k-bitrusses for $k \geq 0$. Dealing with large graphs is often beyond the capability of a single machine due to its limited memory and computational power, leading to a need for efficiently processing large graphs in a distributed environment. However, all current solutions are for a single machine and a centralized environment, where processors can access the graph or auxiliary indexes randomly and globally. It is difficult to directly deploy such algorithms on a shared-nothing model. In this paper, we propose distributed algorithms for bitruss decomposition. We first propose SC-HBD as the baseline, which uses $\mathcal{H}$-function to define bitruss numbers and computes them iteratively to a fix point in parallel. We then introduce a subgraph-centric peeling method SC-PBD, which peels edges in batches over different *butterfly complete subgraphs*. We then introduce local indexes on each fragment, study the *butterfly-aware edge partition* problem including its hardness, and propose an effective partitioner. Finally we present the *bitruss butterfly-complete* subgraph concept, and divide and conquer DC-BD method with optimization strategies. Extensive experiments show the proposed methods solve graphs with 30 trillion butterflies in 2.5 hours, while existing parallel methods under shared-memory model fail to scale to such large graphs.

PVLDB Reference Format:
Yue Wang, Ruiqi Xu, Xun Jian, Alexander Zhou, and Lei Chen. Towards Distributed Bitruss Decomposition on Bipartite Graphs. PVLDB, 15(9): XXX-XXX, 2022.
doi:XX.XX/XXX.XX

1 INTRODUCTION

A bipartite graph G, which contains two disjoint node sets $U(G)$ and $L(G)$, and edges from one set to another, is usually used to model relationships between two types of entities in real-world applications. For instance, consumer-product purchase records, author-paper academic networks, actor-movie information, and so on. Dense subgraph mining is an importance task in graph analysis. It has real-world applications including spam detection, social recommendation, anomaly detection, and the like. While many works have discovered hierarchical dense structures on unipartite networks, such as k-core [23] and k-truss [13, 27], dense subgraph discovery on bipartite graphs is now attracting research attention.

In this paper, we focus on the k-bitruss model, which is a butterfly-based dense structure introduced in [25, 42]. A butterfly, or a (2, 2)-biclique, is the smallest fundamental unit of a cohesive structure in a bipartite graph. A k-bitruss Γ_k is such a maximal subgraph of G that each edge is contained in at least k butterflies, and the bitruss number ϕ_e of an edge e is the largest k such that a k-bitruss contains e. We study the bitruss decomposition problem, which computes ϕ_e for $\forall e \in G$. The k-bitruss model provides a compact way to reveal the hierarchically dense structure on bipartite graphs, and it can be viewed as the analogy of the popular k-truss model on unipartite graphs. The k-bitruss model can be used in fraud detection in social networks and recommendation over user-item structures [37].

<u>Recommendation System.</u> Given a user-product bipartite graph, the k-bitruss model can help identify cohesive subgraphs where users and products are densely connected, showing similarities among users and products. In a recommendation system, such similarities can be used to recommend potential products to users, and associate products with potential buyers [29].

<u>Anomaly Detection.</u> In social media such as Facebook, Twitter, Weibo and TikTok, users often form a "following/followed by" relationship. To increase popularity and influence, fake accounts may be created to follow a particular group of users [5]. These vicious users tend to form a closely connected group, and the k-bitruss model can help locate those anomaly communities at different levels of granularity for further investigation. Similarly, k-bitruss can be used to detect a group of web attackers who access a set of webpages frequently to make their rankings higher.

Owing to the large amount of data generated from various information systems in daily life, it has become more and more necessary to process and analyse large-scale graphs. For example, during Singles' Day 2017, Alibaba coped with 256 000 payment transactions per second, and a total of 1.48 billion transactions were processed by Alipay in the entire 24 hours [40]. This means the number of user-product relations grew extremely fast. According to [39], Facebook has 2.8 billion monthly active users (as of 31 December 2020), indicating billions of following relations and user-post relations. On such bipartite networks, finding k-bitruss structures is also useful in hierarchical group finding, anomaly transaction detection, and personalized recommendations. Processing large-scale graph problems efficiently is going beyond the capability of a single machine (or a shared-memory model) due to its limited memory and restricted concurrency. Developing efficient distributed algorithms for large-graphs, which are too big to load and process in the memory, or too costly to process on a single machine, is an urgent need.

*Co-first authors.

Proceedings of the VLDB Endowment, Vol. 15, No. 9 ISSN 2150-8097.
doi:XX.XX/XXX.XX

Most current methods can solve the bitruss decomposition problem on a single machine. However, none of them aim to design effective algorithms on a shared-nothing model, where G is partitioned among multiple workers to lower the memory requirement of a single machine and enhance parallelism. Existing works [25, 42] follow a similar sequential bottom-up peeling framework (details in Appendix B), *i.e.,* during each round the edge e with the minimum support is labeled and peeled from G, and the support of affected edges is then updated. [37] further boosts the peeling process by proposing the BE-Index, which compacts the edge-butterfly information into bloom-edge structures, so that the redundant workload of enumerating butterflies can be avoided. It is difficult to deploy the above algorithms on a shared-nothing model, due to the following challenges: (1) peeling edges one by one is sequential by nature, which restricts the granularity of parallelism; (2) the shared-memory model allows globally random access to G, related auxiliary data structures and indexes, that are not directly supported on a shared-nothing model; and (3) distributed algorithms incur additional costs of synchronization and communication, which is neither considered nor optimized in current methods.

Facing the above challenges, we aim to answer the following basic questions in this paper: Q1: How to make the process of bitruss decomposition parallelizable using multiple workers on a shared-nothing model? Q2: Given parallel methodologies, how to partition the graph to achieve better communication and overall performance? Q3: How to accelerate computation for each worker locally while guaranteeing correctness? Q4: How to further reduce communication cost and synchronization overhead?

For Q1, firstly, as a baseline, we adopt an alternative interpretation of bitruss number using $\mathcal{H}$-function (which is currently used as a second definition for core numbers). Based on this, we first define the concept of *butterfly-complete subgraph*, and propose SC-HBD, which computes bitruss numbers iteratively to a fix point by *thinking like a subgraph* (Sect. 3). The convergence is shown. SC-HBD incurs a large total workload, thus we present a second batch peeling framework, which peels edges in batches (Sect. 4). Based on that we introduce SC-PBD, a subgraph-centric peeling algorithm, which performs peeling concurrently on different subgraphs and guarantees the accuracy by communication and synchronization.

For Q2, to balance the local computation and reduce the communication cost for SC-HBD and SC-PBD, we formulate a *butterfly-aware edge partition* problem, which is different from existing ones that balance nodes or edges directly. We show the problem is NP-hard and hard to approximate, and present a heuristic partitioner with a quality guarantee, which estimates the workload of each partition efficiently and accurately during partitioning (Sect. 6).

For Q3, to speed up local computation over multiple butterfly-complete subgraphs for SC-HBD and SC-PBD, we build a local index on each individual subgraph and use it to accelerate butterfly retrieval locally (Sect. 5). The key observation is that we do not need to store all wedges in the subgraph, and only those have at least one of its inner edge.

For Q4, to further reduce the communication and synchronization during batch peeling, we introduce the concept of *bitruss butterfly-complete subgraph*, and present a divide-and-conquer method DC-BD (Sect. 7). It first divides G into such subgraphs that each subgraph can perform peeling locally and independently, without any further communication and synchronization among workers in the conquer phase. We further show how to divide G under this framework and propose optimization strategies to boost local computation on local bitruss butterfly-complete subgraphs.

Using both real-life and synthetic datasets, we conduct extensive experiments, and the results show the effectiveness and efficiency of our proposed methods (Sect. 8): (1) Our optimizations speed up SC-HBD, SC-PBD and DC-BD by 18, 26 and 6.4 times, respectively. (2) Our methods are parallel scalable: the respond time of SC-HBD, SC-PBD and DC-BD reduce by 4.6, 3.1 and 3.6 times, when p grows from 8 to 96. (3) DC-BD can process graphs with 30T of $|\bowtie_G|$, in 2.5 hours, whereas current methods like BiT-BU and BiT-PC fail to handle such large graphs. (4) DC-BD consistently beats SC-HBD, SC-PBD, BiT-BU and ParButterfly by at least 72, 3.9, 1.3 and 8.3 times, respectively, and is on average 1.9 times faster than BiT-PC.

2 PRELIMINARIES

In this section, we first present related definitions of bitruss decomposition, then we introduce the environment of distributed graph computation. Notation table and proofs of the paper are in Appendix.

Related Definitions. We consider an undirected bipartite graph $G(V(U, L), E)$, where $U(G)$ is the set of nodes in the upper layer, $L(G)$ is the set of nodes in the lower layer, $U(G) \cap L(G) = \emptyset$, $V(G) = U(G) \cup L(G)$ is the node set, and $E(G) \subseteq U(G) \times L(G)$ is the edge set. Denote by (u, v) or (v, u) the edge between u and v. Denote by $N_G(u) = \{v : v \in V(G) \wedge (u, v) \in E(G)\}$ the set of neighbors of u in G. Given $E' \subseteq E$, the subgraph of G induced by E' is the graph formed by edges in E'.

Definition 2.1 (Butterfly). *Given a bipartite graph G and four nodes $u, v, w, x \in V(G)$, where $u, w \in U(G)$ and $v, x \in L(G)$, a butterfly induced by the nodes u, v, w, x is a (2,2)-biclique of G,* i.e., *both u and w are linked to w and x, respectively, by edges (u, v), (u, x), (w, v) and (w, x).*

Denote by ${}^{u}_{v}{\bowtie}^{w}_{x}$ the butterfly induced by nodes $\{u, v, w, x\}$, $\bowtie_G$ the set of butterflies in G, and $\bowtie_{e,G}$ the intersection of the set of butterflies of G and those edge e takes part in. Here $|\bowtie_{e,G}|$ is called the butterfly support of e in G. Denote by $\bowtie_{G_1,G_2}$ the set of butterflies in G_2 with edges in G_1, *i.e.,* $\bowtie_{G_1,G_2} = \bigcup_{e \in G_1} \bowtie_{e,G_2}$.

Definition 2.2 (k-bitruss). *Given a bipartite graph G, and a positive integer k, a k-bitruss Γ_k is a maximum subgraph of G such that $|\bowtie_{e,\Gamma_k}| \geq k$ for each $e \in \Gamma_k$.*

Definition 2.3 (bitruss number). *The bitruss number ϕ_e of an edge $e \in G$ is the largest k such that a k-bitruss in G contains e.*

Definition 2.4 (bitruss decomposition). *Given a bipartite graph G, the bitruss decomposition problem is to compute ϕ_e for each $e \in E(G)$.*

Denote by $\delta(G)$ the minimum support of all edges in G, *i.e.,* $\delta(G) = \min_{e \in E(G)} |\bowtie_{e,G}|$, ϕ_e also satisfies the following equation.

Lemma 2.1. $\phi_e = \max_{G' \subseteq G, e \in G'} \delta(G')$.

Distributed Graph Computation. We use a coordinator-based shared-nothing model, consisting of a master (coordinator) W_0 and a set of p workers $\mathcal{W} = \{W_1, \cdots, W_p\}$. The workers (including W_0) are pairwise connected by bi-directional communication channels. Meanwhile, G is fragmented into $\mathcal{F} = (F_1, \cdots, F_p)$ and distributed among these workers, where each $F_i = (V_i, E_i)$ is a subgraph of G, such that $V = \cup_{i\in[1,p]} V_i$ and $E = \cup_{i\in[1,p]} E_i$. Each worker W_i hosts and processes a fragment F_i of G. We follow the Bulk Synchronous Parallel (BSP) model for computing. Under BSP, computation and communication are performed in supersteps: at each superstep, each worker W_i first reads messages (sent in the last superstep) from other workers, and then performs the local computation, and finally sends messages (to be received in the next superstep) to other workers. The barrier synchronization of each superstep is coordinated by W_0. One desired property of a distributed graph algorithm ρ is parallel scalability, *i.e.,* T_ρ (time taken by ρ) decreases with an increasing p, indicating the more resources added, the more efficient ρ is. To ensure parallel scalability, it is crucial to make the workload of ρ balanced across different workers. In contrast, existing methods for bitruss decomposition are based on shared-memory model and are abstracted as SeqPeel, detailed in Appendix B.

3 SUBGRAPH-CENTRIC H-FUNCTION DECOMPOSITION

Peeling edges one by one is hard to parallelize in nature. Is there any other paradigm that could make bitruss decomposition parallelizable? Recently, [21] introduces an alternative interpretation for k-core decomposition based on $\mathcal{H}$-function (Def. 3.1). This results in a parallel paradigm where each node v updates its coreness by evaluating $\mathcal{H}(\cdot)$ on coreness of $N_G(v)$ at each round. [26] further extends this idea for general parallel nucleus decomposition over unipartite graphs on shared-memory model. Since k-bitruss is a cohesive model following similar intuition with k-core/truss, it is possible to extend above idea to parallel bitruss decomposition. In the following, first we show this parallel diagram paradigm and its correctness in Sect. 3.1, then present our subgraph-centric algorithm SC-HBD in Sect. 3.2, which is a baseline.

Definition 3.1 ($\mathcal{H}$-function). *Given a multiset N of natural numbers, the $\mathcal{H}$-function $\mathcal{H}(N)$ returns the largest integer y such that there are at least y elements in N whose values are at least y.*

3.1 Overview

In this section, we present H-BD (Algo. 1). During H-BD, each edge computes and maintains a value $\gamma^{(i)}(e)$ which is initialized as $|\bowtie_{e,G}|$ (line 1). At round i, $\gamma^{(i)}(e)$ is updated as follows (lines 5-10). For each butterfly $\bowtie$ of $\bowtie_{e,G}$, the value $\rho(e,\bowtie)$ is computed, which is the minimum $\gamma^{(e)}(i-1)$ among edges in $\bowtie$ other than e. Then $\gamma^{(i)}(e)$ is updated as $\mathcal{H}(N_e)$, where N_e is the set of $\rho(e,\bowtie)$s that evaluated on all butterflies in $\bowtie_{e,G}$. H-BD terminates when $\gamma^{(i)}(e)$ converges to ϕ_e for all edges. At each round of i, all edges update $\gamma^{(i)}(e)$ locally and simultaneously (line 5). Next, we show $\gamma^{(i)}(e)$ is non-increasing, and converges to ϕ_e. To show the correctness of H-BD and make the paper self-contained, we follow similar arguments to [26], with the specialization to bitruss model.

Lemma 3.1 ([26]). *For all $i \geq 1$, $\forall e \in E, \gamma^{(i)}(e) \leq \gamma^{(i-1)}(e)$.*

Lemma 3.2 ([26]). *For all $i \geq 0$, $\gamma^{(i)}(e) \geq \phi_e$.*

Since $\gamma^{(0)}(e), \cdots, \gamma^{(i)}(e), \cdots$ is a non-increasing sequence and has a non-negative lower bounds, it converges in finite steps.

Theorem 3.3 ([26]). *$\forall e \in E$, the sequence $\gamma^{(i)}(e)$ for $i \geq 0$ converges to ϕ_e.*

Algorithm 1 H-BD

```
Input: $G(V, E)$
Output: $\phi_e$ for each $e \in E$
1: $\gamma^{(0)}(e) \leftarrow |\bowtie_{e,G}|, \forall e \in E$
2: $converge \leftarrow False, i \leftarrow 0$
3: while not $converge$ do
4:     $i \leftarrow i + 1, converge \leftarrow True$
5:     for all $e \in E$ in parallel do
6:         $N_e \leftarrow \emptyset$
7:         for all $\bowtie \in \bowtie_{e,G}$ do
8:             $\rho_{e,\bowtie} = \min_{e' \in \bowtie, e' \neq e} \gamma^{(i-1)}(e'),\ N_e \leftarrow N_e \cup \{\rho_{e,\bowtie}\}$
9:         $\gamma^{(i)}(e) \leftarrow \mathcal{H}(N_e)$
10:        if $\gamma^{(i)}(e) \neq \gamma^{(i-1)}(e)$ then $converge \leftarrow False$
11: $\phi_e \leftarrow \gamma^{(i)}(e), \forall e \in E$
```

We show an upper bound for the number of iterations of H-BD in Sect. 4.1, which is tighter than [26]. Next, we introduce our distributed solution based on H-BD.

3.2 Subgraph-Centric Decomposition

We introduce subgraph-centric algorithm SC-HBD (Algo. 2). Suppose E is partitioned into disjoint set $E_1, \cdots, E_p$, and each F_i is the subgraph induced by E_i. In order to update $\gamma^{(\cdot)}(e)$ locally on F_i, it is necessary to "enlarge" F_i to contain additional edges to retrieve supporting butterflies for edges in F_i. This is because edges in a butterfly may cross different fragments. Next, we introduce the concept of butterfly complete subgraph.

Definition 3.2 (Butterfly complete subgraph). *Given a subgraph F_i induced by E_i, the butterfly complete graph of F_i is F_i^+ induced by E_i^+, where $E_i^+ = E_i \cup \{e' : e' \in E \setminus E_i$ and there exists such an edge $e \in E_i$ that e, e' are in the same butterfly of $G\}$.*

Briefly speaking, F_i^+ includes those $e' \notin F_i$ which forms butterflies with any $e \in F_i$ in G (see Appendix C for building F_i^+). For any $e \in E_i^+$, if $e \in E_i$, we call e an *inner* edge (also denoted by $e \in F_i$), otherwise, e is called an *external* edge.

Lemma 3.4. $\forall e \in F_i, |\bowtie_{e,G}| = |\bowtie_{e,F_i^+}|$.

SC-HBD follows similar logic to H-BD, but has the following differences: (1) SC-HBD only works on F_i^+; (2) synchronizations of $\gamma^{(i)}(e)$ among different fragments are performed to maintain the consistency. SC-HBD first initializes $\gamma^{(0)}(e)$ as $|\bowtie_{e,F_i^+}|$ for each inner edge e (lines 2-3). At each following superstep, SC-HBD first receives updated $\gamma^{(i)}(e)$ in last superstep for external edges (lines 7-7) and then computes $\gamma^{(i)}(e)$ (lines 9-12) for each inner edge. Moreover, if any e is updated to a different value and is contained as an external edge on another fragment F_j^+, message is sent to F_j^+ to notify the change (lines 13-15). SC-HBD terminates when it converges, *i.e.,* all workers volt to halt (line 15).

<u>Cost Analysis.</u> At each iteration, the number of butterflies accessed (line 10 of Algo. 2) is $\sum_{e\in E_i} |\bowtie_{e,F_i^+}|$. Computing $\rho_{e,\bowtie}$ (line 11) can be done in constant time, since each butterfly has exactly four edges. Computing $\gamma^{(i)}(e)$ can be done in linear time by incrementally building a hash table. Therefore, the computation time of each iteration on F_i^+ can be done in $O(\sum_{e\in E_i} |\bowtie_{e,F_i^+}|) = O(|\bowtie_{F_i,F_i^+}|)$, by exploiting the local index on F_i^+ (introduced in Sect. 5). The number of messages sent by F_i^+ is $\sum_{e\in E_i} \min\{|\bowtie_{e,F_i^+}|, p-1\}$, since messages

Algorithm 2 SC-HBD (Subgraph-Centric Decomposition)

```
Input: $F_i^+$, $MSG_r$
Output: $\phi_e$ for $e \in E_i$
 1: $s \leftarrow getSuperstep()$
 2: if s = 0 then /* Initialization */
 3:     for all $e \in E_i$ do $\gamma^{(0)}(e) \leftarrow |\bowtie_{e,F_i^+}|$
 4: else /* Iterative Update */
 5:     if $MSG_r = \emptyset$ then $W_i$ volts to halt
 6:     else
 7:         for all $(e, value) \in MSG_r$ do $\gamma^{(s-1)}(e) \leftarrow value$
 8:         $MSG_s \leftarrow \emptyset$
 9:         for all $e \in E_i$ do
10:             for all $\bowtie \in \bowtie_{e,F_i^+}$ do
11:                 $\rho_{e,\bowtie} = \min_{e' \in \bowtie, e' \neq e} \gamma^{(s-1)}(e')$, $N_e \leftarrow N_e \cup \{\rho_{e,\bowtie}\}$
12:             $\gamma^{(s)}(e) \leftarrow \mathcal{H}(N_e)$, $N_e \leftarrow \emptyset$
13:             if $\gamma^{(s)}(e) \neq \gamma^{(s-1)}(e)$ then
14:                 if $\exists F_j^+$ such that $e \in F_j^+ \wedge j \neq i$ then
15:                     Send message $(e, \gamma^{(s)}(e))$ to $W_j$
    return $\gamma^{(s-1)}(e)$ for all $e \in E_i$ when all workers volt to halt
```

sent by any inner edge e is bounded by $|\bowtie_{e,G}|$, and is also bounded by the number of remote workers $p-1$. Therefore, the amount is bounded by $O((p-1)|F_i|)$ and $O(|\bowtie_{F_i,F_i^+}|)$. The messages received by F_i^+ is $O(|F_i^+| - |F_i|)$. The total messages exchanged among all workers during one iteration is $O(\sum_{i=1}^{p} |F_i^+| - |F_i|)$.

Putting these together, we can see the cost of SC-HBD is in $O(T \cdot \max_{i\in[1,k]} |\bowtie_{F_i,F_i^+}|)$. Here T is the number of iterations required for convergence, which is decided solely by G (same as H-BD) and is irrelevant to p. When the fragment is balanced (partitioning G is discussed in Sect. 6) *w.r.t.* the number of butterflies, *i.e.,* $|\bowtie_{F_i,F_i^+}| = O(|\bowtie_G|/p)$, the algorithm is in $O(T|\bowtie_G|/p)$.

4 SUBGRAPH-CENTRIC BATCH PEELING

Though H-BD is parallelizable and leads to our distributed solution SC-HBD, the total workload of all workers of SC-HBD is $O(T|\bowtie_G|)$. This is larger than existing sequential counterparts. In this section, we first introduce a batch peeling framework (Sect. 4.1), which has the same workload as current sequential solutions, and then we present a subgraph-centric program based on it (Sect. 4.2).

4.1 Batch Peeling Framework

Here we introduce BatchPeel (Algo. 3), which peels a batch of edges at each round, to increase the granularity of parallelism following SeqPeel. Specifically, BatchPeel initializes $sup(e)$ as the support of each e (line 1). At each round i, the minimum value MS of $sup(e)$ is identified (line 3), then all edges with $sup(e) \leq MS$ are repeatedly removed in batch (line 5), and labeled with MS as ϕ_e (line 6). Meanwhile, $sup(e)$ for remaining edges are updated, this round continues until all edges have support number larger than MS (line 4-7). BatchPeel terminates when G becomes empty (line 2).

We show the correctness of BatchPeel. Denote by $MS^{(i)}$ the MS in i-th round, $S^{(i)}$ the set of edges removed in the i-th iteration, and $L^{(i)}$ the subgraph of G induced by edges $S^{(i)} \cup S^{(i+1)} \cup \cdots$.

Claim 4.1. *For any $0 \leq j < i$, $MS^{(i)} \geq MS^{(j)}$.*

Proof. This is guaranteed by the condition in line 4. □

Algorithm 3 BatchPeel

```
Input: $G(V, E)$
Output: $\phi_e$ for each $e \in E$
 1: $\forall e \in G$, $sup(e) \leftarrow |\bowtie_{e,G}|$ , $i \leftarrow 0$
 2: while $G \neq \emptyset$ do
 3:     $MS \leftarrow \min_{e\in E(G)} sup(e)$
 4:     while $\exists e \in E$ such that $sup(e) \leq MS$ do
 5:         $S \leftarrow \{e : sup(e) \leq MS\}$, $G \leftarrow G \setminus S$
 6:         for all $e \in S$ do $\phi_e \leftarrow MS$
 7:         update $sup(\cdot)$ for edges affected by removing $S$, $i \leftarrow i+1$
 8: return $\phi_e$ for each $e$
```

Algorithm 4 SC-Peel (Subgraph-centric Peeling on k)

```
Input: $F_i^+$, $k$, $MSG_r$
Output: $e \in E_i$ whose $\phi_e > k$
 1: $s \leftarrow getSuperstep()$
 2: if s = 0 then /* Initialization */
 3:     $Q \leftarrow \{e : e \in F_i \wedge |\bowtie_{e,F_i^+}| \leq k\}$, $R \leftarrow$ SubPeel($Q$)
 4: else/* Iterative Peeling */
 5:     $R \leftarrow$ SubPeel($MSG_r$)
 6: for all $e \in R$ and $W_j$, such that $e \in F_j^+ \wedge j \neq i$ do send $\{e\}$ to $W_j$
 7: if no messages are sent then $W_i$ volts to halt
 8: function SubPeel(Q)
 9:     $R \leftarrow \emptyset$;
10:     while $Q \neq \emptyset$ do
11:         $\phi_e \leftarrow k$;   $e \leftarrow Q.pop()$;
12:         for all $\bowtie \in \bowtie_{e,F_i^+} \cap \bowtie_{F_i,F_i^+}$ do
13:             for all $e' \in \bowtie$ and $e' \neq e$ do
14:                 if $e'$ is an inner edge on $F_i^+$ then
15:                     $supp(e') \leftarrow supp(e') - 1$
16:                     if $supp(e') \leq k$ then $Q.add(e')$
17:         $F_i^+ \leftarrow F_i^+ \setminus \{e\}$, $R \leftarrow R \cup \{e\}$
18:     return $R$;
```

Theorem 4.2. *At the end of i-th round of BatchPeel ($i \geq 0$), $\forall e_i \in S^{(i)}$, ϕ_{e_i} is correctly assigned,* i.e., *$\phi_{e_i} = MS^{(i)}$.*

Corollary 4.3. *For any $i, j \geq 0$ and $i \geq j$, for any $e_i \in S_i$, $e_j \in S_j$, $\phi_{e_i} \geq \phi_{e_j}$.*

Note that there is another version of peeling by batch MinBatchPeel [28], which peels edges whose support is *exactly* the minimum. On the contrary, BatchPeel peels edges whose support is $\leq MS^{(i)}$, which is at least the minimum of the current support of G. Therefore, BatchPeel can peel more edges than MinBatchPeel in an iteration and take less iterations in total. BatchPeel also provides an upper bound for T in H-BD.

Theorem 4.4. *If e is removed at i-th iteration in BatchPeel ($e \in S^{(i)}$), then $\gamma^{(t)}(e) = \phi_e$ for $t \geq i$,* i.e., *$\gamma^{(\cdot)}(e)$ converges to ϕ_e within i iterations.*

4.2 Subgraph-centric Peeling

In this section, we introduce our subgraph-centric peeling approach called SC-PBD. The basic idea is that, we treat each fragment as a subgraph, and peel edges in each subgraph independently. During the peeling, messages are only exchanged when it is necessary.

SC-PBD follows the skeleton of BatchPeel. However, SC-PBD calls the subgraph-centric SC-Peel (Algo. 4) for the major peeling phase, which corresponds to the main loop (lines 4-7) of Algo. 3. Other parts of logic are controlled by coordinator W_0.

Given F_i^+ and k (assigned as MS at each round invoked by W_0), SC-Peel peels such edges $e \in F_i$ that $sup(e) \leq k$, and updates the support of the remaining inner edges. It consists of: (1) the initial stage which peels edges affected by initial unqualified inner edges (lines 2-3); (2) and an iterative peeling stage which peels edges affected by the removal of external edges (lines 4-5). Initial stage has 1 superstep, it identifies those inner edges Q whose support is $\leq k$, and uses Q as the "seed" to perform the sequential peeling algorithm SubPeel(Q), which returns a set R of removed inner edges that can influence some other fragment F_j^+. The iterative peeling stage has multiple supersteps. It first receives messages MSG_r which contain external edges of F_i^+ that are removed as inner edges in some other fragments in the last superstep. Next it invokes SubPeel(MSG_r) for peeling, then after peeling with Q or MSG_r, it checks whether R is empty or not. If R is empty, W_i volts to halt since peeled inner edges in F_i do not affect other fragments (line 7). Otherwise, W_i sends edges in R to corresponding fragments for notifying the removal of the external edges (line 6). It terminates when all workers volt to halt.

SubPeel(Q) is a sequential procedure works on F_i^+. Given a set Q of starting edges to be removed, SubPeel(Q) views edges in Q as "seeds" and peels edges in F_i affected by Q as much as possible. For each $e \in Q$, it iterates over $\bowtie \in \bowtie_{e,F_i^+} \cap \bowtie_{F_i,F_i^+}$ (line 12), *i.e.*, butterflies associated with e with at least one edge in F_i. Then for each edge $e' \in \bowtie$ (line 13): if e' is an inner edge, its support decreases by 1 (line 15). Furthermore, if $|\bowtie_{e',F_i^+}| \leq k$, e' is added to Q for further peeling (line 16). After processing edges affected by e, e is removed from F_i^+ (line 17). In addition, if there exists some other fragment F_i^+ containing e as an external edge, then e is added to R for further notifying other fragments in the next superstep where e is removed (line 17). It terminates until Q becomes empty. Next, we show the correctness of Algo. 4.

Theorem 4.5. *When Algo. 4 terminates, for any remaining inner edge $e \in F_i$, $\phi_e > k$.*

Cost of SC-PBD. We analyse the computation and communication cost incurred by any fragment F_i^+ during SC-PBD. The computation cost incurred by peeling an edge $e \in F_i^+$ is bounded by the number of butterflies associated with e in $\bowtie_{F_i,F_i^+}$ (line 13, detailed implementation in Sect. 5). The total computation cost on F_i^+ is $O(\sum_{e \in F_i^+} |\bowtie_{e,F_i^+} \cap \bowtie_{F_i,F_i^+}|) = O(|\bowtie_{F_i,F_i^+}|)$, since a butterfly in $\bowtie_{F_i,F_i^+}$ is accessed at most once (when it is destroyed). Next, we analyse the communication cost of F_i^+. For each inner edge e, it has at most $\min\{|\bowtie_{e,G}|, p-1\}$ mirrors on other fragments, the same as the message size of notifying other fragments of its removal. The message amount sent by F_i^+ is bounded by $O((p-1)|F_i|)$ and $O(|\bowtie_{F_i,F_i^+}|)$. For an external edge e, it receives at most 1 message from the other fragment F_j where e is an inner edge, notifying the removal of e. Therefore, the total communication cost on F_i^+ is $O(\sum_{e \in F_i} |\bowtie_{e,F_i^+}| + \sum_{e \notin F_i} 1) = O(|\bowtie_{F_i,F_i^+}|)$, since each external edge is contained in at least one butterfly in $\bowtie_{F_i,F_i^+}$.

It can been seen that SubPeel(Q) is similar to SeqPeel. The differences are: (1) SubPeel(Q) is performed on a fragment F_i^+, while SeqPeel is performed on G; (2) SubPeel(Q) is a subroutine of SC-Peel for local peeling, and SC-Peel also performs communication among fragments. This reveals the advantage of thinking like a subgraph: the computation of a fragment is sequential, and multiple fragments run in parallel. This makes it flexible to adjust the granularity of parallelism to improve the efficiency by graph partition.

Algorithm 5 Enumerating Butterflies in $\bowtie_{F_i,F_i^+}$

```
Input: $F_i$, $F_i^+$, edge $e = (u, v)$, $e \in F_i^+$ and local index $H_i$ and $\bar{H}_i$
Output: The set of butterflies $\bowtie_{e,F_i^+} \cap \bowtie_{F_i,F_i^+}$
1:  Suppose $p(u) > p(v)$, otherwise swap$(u, v)$; $B_e \leftarrow \emptyset$;
2:  for all $w$ such that $(w, v) \in F_i^+ \wedge p(u) > p(w)$ do
3:      if $(w, v) \in F_i \vee (u, v) \in F_i$ then
4:          for all ${}_{x}\angle_{u}^{w} \in H_i(u, w)$ such that $x \neq v$ do
5:              $B_e \leftarrow B_e \cup \{{}^{u}_{v}\bowtie^{w}_{x}\}$
6:      else
7:          for all ${}_{x}\angle_{u}^{w} \in \bar{H}_i(u, w)$ do
8:              $B_e \leftarrow B_e \cup \{{}^{u}_{v}\bowtie^{w}_{x}\}$
9:  for all $w$ such that $(w, u) \in F_i^+ \wedge p(w) > p(u)$ do
10:     execute lines 2-8 with swapped $u$ and $v$
    return $B_e$
```

5 LOCAL INDEX ON F_i^+

The key operation of both the subgraph-centric algorithm SC-HBD and SC-Peel is to iterate over $\bowtie_{e,F_i^+}$ for a given e (line 10 of Algo. 2 and line 12 of Algo. 4). This operation dominates the cost over F_i^+ for the two algorithms. In this section, we discuss how to build a local index and use it to retrieve butterflies.

Enumerating Butterflies with Index. To efficiently enumerate butterflies in $\bowtie_{e,F_i^+}$, we explicitly store the wedges shared by each pair of vertices over F_i^+. Denoted as H_i, the index maps vertex pair $(u, w) \in V \times V$ to the set of wedges ${}_{v}\angle_{u}^{w}$ with u, w as endpoints. It is formally defined as follows:

$$H_i(u, w) = \{{}_{v}\angle_{u}^{w} \in \mathbb{W}_{F_i^+}\}.$$

Here $\mathbb{W}_{F_i^+}$ denotes a subset of wedges in F_i^+, such that each wedge ${}_{v}\angle_{u}^{w}$ in it satisfies $p(u) > \max(p(v), p(w))$. The priority $p()$ defines a total order over V. It serves to reduce the size of index and speeds up its construction (explained in Algo. 11 in Appendix D).

For each set $H_i(u, w)$, we also explicitly maintain a subset of it, denoted as $\bar{H}_i(u, w)$, such that:

$$\bar{H}_i(u, w) = \{{}_{v}\angle_{u}^{w} \in \mathbb{W}_{F_i^+} \mid (u, v) \in F_i \vee (w, v) \in F_i\}.$$

That is, $\bar{H}_i(u, w)$ only contains wedges that have at least one inner edge in F_i. We denote the collection of all $\bar{H}_i(u, w)$ in F_i^+ as $\bar{\mathbb{W}}_{F_i^+}$.

Each pair of wedges in $H_i(u, w) \times \bar{H}_i(u, w)$ forms a butterfly in $\bowtie_{F_i,F_i^+}$. We denote the corresponding set of butterflies as

$$\bowtie_i(u, w) = \{{}^{u}_{v}\bowtie^{w}_{x} \mid {}_{v}\angle_{u}^{w} \in H_i(u, w) \wedge {}_{x}\angle_{u}^{w} \in \bar{H}_i(u, w) \wedge v \neq x\}.$$

Note that the butterfly sets of $\bowtie_i$ are not explicitly stored. They are instead implied from the corresponding wedge sets of H_i. Since each butterfly in $\bowtie_i(u, w)$ contains one wedge in $\bar{H}_i(u, w)$ that consists of at least one inner edge in F_i, we can see:

Lemma 5.1. $\bowtie_i(u, w) \subset \bowtie_{F_i,F_i^+}$.

Although H_i only contains a subset of all wedges in F_i^+, the sets of butterflies in the induced $\bowtie_i$ covers all butterflies in $\bowtie_{F_i,F_i^+}$ without redundancy. More specifically:

Lemma 5.2. *The butterfly sets $\bowtie_i$ in F_i^+ form a partition of the butterfly set $\bowtie_{F_i,F_i^+}$.*

With the help of the index $H_i, \bar{H}_i$ and their induced butterfly sets $\bowtie_i(u,w)$, the algorithm for enumerating butterflies associated with e in $\bowtie_{F_i,F_i^+}$ is seen in Algo. 5. The algorithm takes as input the fragment F_i and its enlarged counterpart F_i^+, together with an edge $e \in F_i^+$ and indexes of H_i and $\bar{H}_i$. It generates the subset of butterflies in $\bowtie_{F_i,F_i^+}$ containing e. The algorithm first scans the neighbor w of v in F_i^+ such that ${}_v\angle_u^w \in \mathbb{W}_{F_i^+}$ (line 2). If the wedge scanned contains at least one edge in F_i, the algorithm scans the wedges of $H_i(u,w)$ and adds the corresponding butterflies into the set B_e (lines 3-5). Otherwise, it only scans wedges in $\bar{H}_i(u,w)$ and enumerates the butterflies (lines 6-8). It then scans wedges with v and w as wedge endpoints (lines 9 -10) in a similar way.

To see the algorithm is correct, note that 1) in the first loop (lines 3-8), we scanned all butterflies in $\bowtie_{e,F_i^+} \cap \bowtie_{F_i,F_i^+} \cap \bowtie_i(u,w)$ for an edge $e \in F_i$; 2) all wedges containing e in $\mathbb{W}_{F_i^+}$ are processed. Putting this together with Lemma 5.2, we can see the Algo. 5 successfully returns all butterflies in $\bowtie_{F_i,F_i^+}$ that contains e for a given edge. As for the cost of the algorithm, by Lemma 5.2, each of the butterflies returned are scanned only once, and no butterflies outside of the returned $\bowtie_i$ are scanned. Hence the cost is in $O(|\bowtie_{e,F_i^+} \cap \bowtie_{F_i,F_i^+}|)$. The building of the local index is simple and discussed in Appendix D. Note that current index such as the BE-Index [37] cannot be directly used here, since it retrieves $\bowtie_{e,G}$ instead of $\bowtie_{e,F_i^+} \cap \bowtie_{F_i,F_i^+}$, and it does not distinguish inner edges and external edges on F_i^+.

Pruned Indexes. For SC-PBD, we can further avoid scanning external wedges in $H_i(u,v) \setminus \bar{H}_i(u,v)$, *i.e.,* wedges with both edges in $F_i^+ \setminus F_i$. This is because the algorithm only processes edges of $\bowtie_{e,F_i^+}$ that fall in F_i (at line 14 of Algo 4). Consequently, the indexes can be pruned for SC-PBD by storing only $\bar{\mathbb{W}}_{F_i^+}$, instead of $\mathbb{W}_{F_i^+}$. In contrast, such pruned indexes are not available for SC-HBD. Because the algorithm needs to scan all $sup(e')$ for edges $e' \in \bowtie_{e,F_i^+}$, in order to compute N_e and $\mathcal{H}(N_e)$ on edge $e \in F_i$.

Lemma 5.3. *The total size of pruned indexes is bounded,* i.e., $\sum_{i\in[1,p]} |\bar{\mathbb{W}}_{F_i^+}| \leq 2|\mathbb{W}_G|$.

6 PARTITIONING

Given the subgraph-centric peeling algorithms of SC-HBD and SC-PBD, we discuss how to partition G in this section. We first formalize the graph partition problem and show its hardness (Sect. 6.1), and then propose a partitioner that guarantees both efficiency and effectiveness (Sect. 6.2).

6.1 Butterfly-Aware Edge Partition Problem

We first show why existing conventional partitions are not a fit for the parallel computation of Algo.s 2 and 4. Then we formalize this new partitioning problem as *Butterfly-aware Balanced Graph Partition Problem* and present the hardness result.

Since each F_i^+ expands F_i by including all associated butterflies, it is easy to see random hash edge partition does not work well. Such a partition may incur a high replication of edges and each expanded fragment F_i^+ will contain almost the entire graph G after replication. Worse still, there are no partitioners that can be readily applied to the problem. Conventional edge partitioners are developed based on objectives of 1) balancing the size of each partition $|F_i|$ and 2) minimizing the communication between edge copies. However, the cost of SC-PBD and SC-HBD over F_i^+ is determined by $|\bowtie_{F_i,F_i^+}|$. Hence we need to balance $|\bowtie_{F_i,F_i^+}|$, rather than $|F_i|$. Meanwhile, the communication also takes place among replicated external edges, rather than among copies of vertices. Towards this end, we formalize the problem of partitioning *w.r.t.* SC-PBD and SC-HBD and show its hardness. Denote by $C(G)$ a disjoint edge partition over G of size p.

Definition 6.1 (Butterfly-aware Balanced Graph Partition (BABGP)). *Given a bipartite graph $G(V,E)$, p, and ϵ, output an edge partition $C(G)$ such that for $\forall i \in [i,p]$, $|\bowtie_{F_i,F_i^+}| \leq \epsilon \cdot B$, and the total number of external edges on all fragments t is minimized, where $B = \frac{\sum_{i=1}^{p} |\bowtie_{F_i,F_i^+}|}{p}$, $\epsilon \geq 1$, and $t = \sum_{i=1}^{p}(|F_i^+| - |F_i|)$.*

Theorem 6.1. *BABGP problem is NP-hard.*

Corollary 6.2. *BABGP has no polynomial time approximation algorithm with finite approximation factor unless P=NP.*

6.2 Butterfly-Aware Balanced Partitioner

Given the NP-hardness of BABGP, we next develop a heuristic parallel edge partitioner, named Butterfly-Aware Balanced Partitioner (BABP). The partitioner efficiently divides the bipartite graph into balanced fragments and effectively boosts the performance of Algo.s 2 and 4. Next we present its sequential procedure for simplicity, and how it is parallelized is shown in Appendix E.

Recall the computation cost of Algo.s 2 and 4 are all bounded by $O(\max_{i\in[1,p]} |\bowtie_{F_i,F_i^+}|)$. Hence the goal of our BABP is to speed up the computation by reducing $\max_{i\in[1,p]} |\bowtie_{F_i,F_i^+}|$.

In a nutshell, the partitioner begins with p empty partitions and grows each partition simultaneously until all edges are partitioned, so that: 1) $\max_{i\in[1,p]} |\bowtie_{F_i,F_i^+}|$ is bounded, 2) the total workload of $\sum_{i\in[1,p]} |\bowtie_{F_i,F_i^+}|$ is reduced, and 3) the cost of partitioner is in $o(|\bowtie_G|)$, so that the speedup introduced by the partition is not canceled by the cost of partitioning.

As shown in Algo. 6, the partitioner takes as input G, p, and a user-defined threshold ϵ, and it generates a p-way partition as output. The algorithm first declares a set of edges S_i for each initially empty F_i. The set S_i includes all edges e whose maximum heuristic score falls into F_i, *i.e.,* $\arg_j \max_{j\in[1,k]} f_{\text{gain}}(e, F_j) = i$ (line 3, the heuristic score f_{gain} is described later in this section). It also initializes $sup(e)$ and indexes $H(\cdot)$ as auxiliary structures (line 4). It declares a maximum workload as $B_{\max} = 1/p \cdot \sum_{e\in E} sup(e)$ (line 5) and uses B_i as a *workload estimation* of $|\bowtie_{F_i,F_i^+}|$.

During the partitioning process (lines 6-13), the partitioner grows one partition each time in a round-robin fashion. The partitions F_i with a large B_i are prevented from growing (line 8). That is, when 1) B_i has already exceeds the workload limit of $B_{\max}$, or 2) B_i is relatively too large compared with other fragments, characterized by $\epsilon \min_{j\in[1,p]} B_j$. If the fragment F_i is allowed to grow, it tries to pick the edge e in S_i with the maximum score of $f_{\text{gain}}(e, F_i, G)$ (line 10). The edge e is picked randomly if S_i is empty (line 12). The estimated workload B_i and S_i are updated accordingly (line 13).

Next, we describe 1) how the workload over fragment F_i is estimated by B_i efficiently; and 2) how the heuristic score f_{gain} is defined for edge selection.

Workload Estimation. In order to reduce the workload and keep them bounded, we need to keep track of the workload of the partitions. However, it is too costly to maintain the exact cost of $|\bowtie_{F_i,F_i^+}|$ for F_i, as it will require scanning butterflies in $\bowtie_{e,F_i^+}$ upon

Algorithm 6 BABP (Butterfly-Aware Balanced Partitioner)

```
Input: $G(V(U, L), E)$, a positive $p$, a user-defined threshold $\epsilon$ ($\epsilon > 1$)
Output: A $p$-way edge partitions $C(G) = \{F_1, F_2, \dots, F_p\}$.
1:  for all $i \in [1, p]$ do
2:      $F_i \leftarrow \emptyset$; $B_i \leftarrow 0$;
3:      let $S_i$ be the subset of unassigned edges with $f_{\text{gain}}(e, F_i) = \max_{j\in[1,k]} f_{\text{gain}}(e, F_j)$
4:  Initialize $sup(e) = |\bowtie_{e,G}|$, and indexes $H(\cdot)$;
5:  $B_{\max} = \frac{1}{p}\sum_{e\in E} sup(e)$;
6:  while $\bigcup_{i\in[1,p]} F_i \neq G$ do
7:      for all $i \in [1, p]$ do
8:          if $B_i \geq B_{\max}$ or $B_i \geq \epsilon \min_{j\in[1,p]} B_j$ then continue;
9:          else if $S_i \neq \emptyset$ then
10:             $e \leftarrow \arg_e \max_{e\in S_i} f_{\text{gain}}(e, F_i)$;
11:         else
12:             pick an unassigned edge $e$ in $G$
13:         $F_i \leftarrow F_i \cup \{e\}$; update $B_i$ and $S_j$ for $j \in [1, p]$ accordingly;
```

adding a new edge e into F_i. On the contrary, it is much faster to use $\sum_{e\in F_i} |\bowtie_{e,G}|$ as an estimation of $|\bowtie_{F_i,F_i^+}|$, since computing $|\bowtie_{e,G}|$ for edges in G only requires scanning wedges in $\mathbb{W}_G$, where $|\mathbb{W}_G| << |\bowtie_G|$ for real-life graphs. However, this estimation is not accurate enough. It can not help us to reduce the total workload of $\sum_{i\in[1,p]} |\bowtie_{F_i,F_i^+}|$, since $\sum_{i\in[1,p]}\sum_{e\in F_i} |\bowtie_{e,G}| = \sum_{e\in G} |\bowtie_{e,G}| = 4|\bowtie_G|$. That is, the sum of this estimated workload is fixed and does not reflect the replications of butterflies for a given edge partition.

Toward this, we strike a balance between efficiency and accuracy, and estimate the cost of partition F_i (denoted as B_i) as follows,

$$B_i = \sum_{e\in F_i} |\bowtie_{e,G}| - \sum_{u,v\in V} |{}_{u}\bowtie_{v}|(\sigma_1(F_i, u, v) + 2\sigma_2(F_i, u, v)) \quad (1)$$

Here the notion of ${}_{u}\bowtie_{v}$, σ_1 and σ_2 are defined as follows:

- denote by ${}_{u}\bowtie_{v} = \{{}^{x}_{u}\bowtie^{w}_{v} \mid \max(p(u), p(v)) > \max(p(w), p(x))\}$;
- σ_1 denotes a boolean function with input of F_i and $u, v \in V$, it returns true if and only if: there exists $y \in V$, such that $p(y) < \max(p(u), p(v))$ and $(u, y) \in F_i$ and $(v, y) \in F_i$;
- σ_2 denotes a boolean function with input of F_i and $u, v \in V$, it returns true if and only if: for each $y \in V$, such that $p(y) < \max(p(u), p(v))$, then $(u, y) \in F_i$ and $(v, y) \in F_i$;

The set ${}_{u}\bowtie_{v}$ denotes a set of butterflies in G, associated of a pair of vertices (u, v) (on the same side of $V(L)$ or $V(U)$). Each of the butterflies in this set are all formed by two edges with (u, v) as endpoints in $\mathbb{W}_G$, and hence the cardinality of the set can be efficiently computed as $|{}_{u}\bowtie_{v}| = \binom{W}{2}$, where $W = |\{{}_{w}\angle^{v}_{u} \in \mathbb{W}_G\}|$. The two boolean functions σ_1 and σ_2 indicate sufficient conditions for situations where 1) all butterflies with in the set of ${}_{u}\bowtie_{v}$ contain at least two edges in F_i; and 2) all butterflies within the set of ${}_{u}\bowtie_{v}$ consist solely of the edges in F_i, respectively.

Theorem 6.3. *The estimated workload B_i is a tighter upper bound of actual workload $|\bowtie_{F_i,F_i^+}|$ than the estimation $\sum_{e\in F_i} |\bowtie_{e,G}|$, i.e.,*

$$|\bowtie_{F_i,F_i^+}| \leq B_i \leq \sum_{e\in F_i} |\bowtie_{e,G}| \quad (2)$$

Heuristic Score. The selection of edge at line 10 is based on the heuristic function, defined as follows:

$$f_{\text{gain}}((u, v), F_i) = \begin{cases} |\{w \mid (v, w) \in F_i \wedge {}_{v}\angle^{u}_{w} \in \mathbb{W}_G\}| + & p(u) > p(v) \\ |\{w \mid (u, w) \in F_i \wedge {}_{u}\angle^{w}_{v} \in \mathbb{W}_G\}| & \\ f_{\text{gain}}((v, u), F_i) & p(u) < p(v) \end{cases}$$

That is, the gain for including an edge e is defined as the number of new wedges in $\mathbb{W}_G$ with both edges in F_i. Intuitively, the gain measures the locality of e in F_i: with a higher gain, expanding e will introduce more wedges in $\mathbb{W}_G$ to F_i, leading to more local butterflies with all 4 edges in F_i. Consequently, a partition with better locality has less replicated butterflies and incurs a lower total workload of $\sum_{i\in[1,p]} |\bowtie_{F_i,F_i^+}|$.

Analysis of BABP. We next analyze the complexity and the quality of BABP.

Cost of BABP. Algo. 6 assigns one edge to one partition each time. For each edge e, the cost consists of 1) selecting the edge with the maximum score (line 10), 2) updating B_i (line 13) and 3) updating S_j for all affected edges (line 13).

Over an edge $e = (u, v)$, 1) can be performed in $O(\log(|E|))$ by popping the top element of a Fibonacci heap[12]. Each partition F_i maintains its own heap over the edges set of S_i, where $f_{\text{gain}}((u, v), F_i)$ serves as priority. For 2) and 3), we need to scan all wedges formed by the edge e and another adjacent e' in $\mathbb{W}_G$. For each affected wedge, the updates to B_i and $f_{\text{gain}}(e', F_i)$ can be performed in $O(1)$. With an updated $f_{\text{gain}}(e', F_i)$ on e', the associated sets S_j and corresponding heaps are updated accordingly. These can be done in $O(\log(|E|))$. Since the loop performs 1), 2) and 3) over all wedges in $\mathbb{W}_G$, the loop is in $O(\log(|E|)\sum_{(u,v)\in E}\min(d(u), d(v)))$. The indexing cost at line 4 is in $O(\sum_{(u,v)\in E}\min(d(u), d(v)))$. To sum up, we can see BABP is in $O(\log(|E|)\sum_{(u,v)\in E}\min(d(u), d(v)))$.

Quality of BABP. For a p-way edge partition output $C(G) = \{F_1, \cdots, F_p\}$, we show that 1) $\max_{i\in[1,p]} |\bowtie_{F_i,F_i^+}|$ is bounded, and 2) the total workload of $\sum_{i\in[1,p]} |\bowtie_{F_i,F_i^+}|$ is greatly reduced from the worst case of $4|\bowtie_{F_i,F_i^+}|$. For 1), note that any partition with $B_i \geq B_{\max}$ are prevented from growing (line 8). Hence $B_i < B_{\max} + \max_{e\in G} |\bowtie_{e,G}|$. Putting this together with Eq. 2, we can see Lemma 6.4 holds, *i.e.,* $|\bowtie_{F_i,F_i^+}|$ is bounded.

Lemma 6.4. $|\bowtie_{F_i,F_i^+}| < \frac{4}{p}|\bowtie_G| + \max_{e\in G} |\bowtie_{e,G}|$.

For 2), by summing up Eq. 2 for each fragment $F_i \in C(G)$, we can see the reduced total workload compared with the worst case of $4|\bowtie_G|$ is at least:

$$4|\bowtie_G| - \sum_{i\in[1,p]} |\bowtie_{F_i,F_i^+}| \geq \sum_{i\in[1,p]}\sum_{u,v\in V} |{}_{u}\bowtie_{v}|(\sigma_1(F_i, u, v) + 2\sigma_2(F_i, u, v)) \quad (3)$$

Parallelizing BABP and its cost analysis are shown in Appendix E.

7 DIVIDE & CONQUER

Previous subgraph-centric methods in Sect. 3.2 and 4.2 enable computing bitruss in parallel over partitioned graphs. However, over dense graphs, such solutions face the following challenges. 1) In dense graphs, there are often a few edges with high butterfly support, *i.e.,* "hub edges". As is observed in [38], more than 80% of updates are performed over the support of these "hub edges". In our distributed setting, it is even worse: the "hub edges" not only incur high computation costs of updating butterfly support locally, but also indicate large communication cost. This is because "hub edges" are more likely to have copies in other butterfly-completed partitions, where communications are required to synchronize the butterfly support . 2) Dense graphs also have very large $\phi_{\max}$. For each k in $[1, \phi_{\max}]$ such that the set of edges with $\phi_e = k$ is nonempty, a few supersteps are required to peel them. This leads to a high number of total supersteps to compute all k-bitruss, and results in a high communication and synchronization overheads.

To deal with these challenges, a natural question arises: can we divide G into such p fragments, such that the bitruss of edges in each fragments can be computed locally and independently *without* communication? This will not only prevent updating support of "hub edges" when peeling edges in remote fragments, but also reduce communication and synchronization overheads. Next we show the intuition.

Consider dividing a bipartite graph G as follows. For a given $t \in [1, \phi_{\max}]$, we partition edges in G into two parts, *i.e.,* Γ_t and $G \backslash \Gamma_t$. Then the bitruss of these two parts can be independently computed as follows: 1) for any edge e in Γ_t, the bitruss number of ϕ_e in G can be answered by computing the bitruss over the subgraph of Γ_t, *i.e.,* $\phi_e(G) = \phi_e(\Gamma_t)$; 2) for any edge e in $G \setminus \Gamma_t$, consider the butterfly-complete subgraph induced by $G \setminus \Gamma_t$, denoted as G', then the bitruss number of $e \in G \setminus \Gamma_t$ can be computed by running bitruss decomposition over G', *i.e.,* $\phi_e(G) = \phi_e(G')$. On the one hand, due to the hierarchical structure of the t-bitruss subgraphs, Γ_j can be computed from Γ_t for all $j \geq t$. On the other hand, for graph G', consider running a peeling algorithm over G, the introduced external edges in G' have bitruss number $\phi_e \geq t$, and are not peeled before all edges of $G \setminus \Gamma_t$ are purged. Hence the bitruss over $G \setminus \Gamma_t$ is also correctly computed.

Algorithm 7 DC-BD (Divide and Conquer Bitruss Decomposition)

```
Input: $G(V, E)$, $p$
Output: $\phi_e$ for each $e \in E$
1:  /* Phase I: Divide */
2:  Generate $p$-way partition $[1, t_1), \cdots, [t_{p-1}, +\infty)$ of bitruss numbers
3:  for all $i \in [1, p-1]$ do
4:      $\Gamma_{t_i} \leftarrow$ compute the $t_i$-bitruss of $G$, $F_i = \Gamma_{t_{i-1}} \setminus \Gamma_{t_i}$
5:  $F_p = \Gamma_{t_{p-1}}$;
6:  /* Phase II: Conquer */
7:  for all $i \in [0, p-1]$ in parallel do
8:      Construct subgraph $F_i$ on worker-$i$
9:      $F_i^B \leftarrow$ the bitruss butterfly-complete graph of $F_i$ in $G$
10:     Initialize index $\bar{H}_i$ and butterfly support $sup(\cdot)$ over $F_i^B$
11:     LocalPeel($sup(\cdot)$, $\bar{H}_i$, $F_i^B$)
12: function LocalPeel($sup(\cdot)$, $\bar{H}_i$, $F_i^B$)
13:     while there exists some edge in $F_i$ that is not peeled do
14:         $k \leftarrow \min_{e \in F_i} sup(e)$
15:         let $Q = \{e \mid e \in F_i \wedge sup(e) = k\}$
16:         while $Q \neq \emptyset$ do  Repeat lines 11-16 of Algo. 4
17:     return $\phi_e$ for each $e$ in $F_i$
```

We generalize the 2-way case to p-way as follows. The divide-and-conquer based framework (DC-BD) is illustrated in Algo. 7. The algorithm consists of two phases, namely the Divide phase (lines 2-5) and the Conquer phase (lines 7-11). In the former phase, the algorithm first divides the bitruss number interval of $[1, +\infty)$ into sub-intervals of $[t_{i-1}, t_i)$ for $i \in [1, p]$, with $t_0 = 1$ (line 2). The i-th fragment F_i consists of the edges with bitruss numbers falling into the interval $[t_{i-1}, t_i)$, and can be computed as $\Gamma_{t_{i-1}} \setminus \Gamma_{t_i}$ (lines 3-5). In phase Conquer, for each worker-i in parallel, we 1) reconstruct the subgraph F_i induced by edges in $\Gamma_{t_{i-1}} \setminus \Gamma_{t_i}$, 2) expand it by fetching edges in Γ_{t_i}, such that all butterflies with edges in $\Gamma_{t_{i-1}}$ are complete, 3) construct partial indexes $\bar{H}_i$ over F_i^B and initialize support $sup(\cdot)$, and 4) compute ϕ_e for edges e in F_i locally.

Next we first describe the Conquer phase and show the correctness of the framework in Sect. 7.1. Then in Sect. 7.2, for the Divide phase, we address the issue of bitruss number interval partition and propose an efficient partitioner that generates F_i with balance guarantees. Finally in Sect. 7.3, we equip the Divide phase with additional optimizations to boost its efficiency.

7.1 Conquer Phase

In this section we first define the *bitruss butterfly-complete subgraph*, which is different from a butterfly-complete one (Def. 3.2). Then we prove that bitruss can be correctly computed on such subgraphs independently without any communications. We describe how to revise SC-Peel (Algo. 4) to efficiently compute bitruss on these bitruss butterfly-complete subgraphs.

Definition 7.1 (bitruss butterfly-complete subgraph). *Consider the subgraph F of G, defined as $F = \Gamma_j \setminus \Gamma_l$ ($j < l$), the bitruss butterfly-complete subgraph of F, denoted as F^B, can be defined as the minimum subgraph of Γ_j that covers all butterflies in $\bowtie_{F,\Gamma_j}$.*

The *bitruss butterfly-complete subgraph* F^B induced by $F = \Gamma_j \setminus \Gamma_l$ is to include all edges in Γ_j that share butterflies with edges in F. In this way, we show that the bitruss number of edges within the interval $[j, l)$ can be correctly computed over F^B. That is, denoted by $\phi_e(G)$ the bitruss number of e over graph G, then:

Theorem 7.1. *Let $F = \Gamma_j \setminus \Gamma_l, (j < l)$, then for each $e \in F$, $\phi_e(F^B) = \phi_e(G)$.*

Lemma 7.2. *Let $F = \Gamma_j \setminus \Gamma_l, (j < l)$, then $\bowtie_{F,\Gamma_j} = \bowtie_{\Gamma_j} \setminus \bowtie_{\Gamma_l}$.*

Corollary 7.3. *Let $F = \Gamma_j \setminus \Gamma_l, (j < l)$, then $\bowtie_{F,\Gamma_j} = \bowtie_{F^B}$.*

Procedure LocalPeel. Next we show how to revise the sequential SubPeel and employ it as the Conquer phase running over bitruss butterfly-complete subgraphs. The revised procedure, denoted as LocalPeel, is shown in Algo. 7 (lines 12-17). The procedure takes in as input a bitruss butterfly-complete subgraph F_i^B, along with the butterfly supports $sup(\cdot)$ and indexes $\bar{H}_i$ over the subgraph. It outputs ϕ_e for each e in F_i.

Here we only focus on the revisions made in the new algorithm, listed as follows: 1) The procedure LocalPeel is a sequential peeling algorithm that is locally executed on each worker without any global communications. The outer loop peels edges that fall into $\Gamma_k \setminus \Gamma_{k+1}$ each time, and terminates when k reaches t_{i+1}. 2) The index $\bar{H}_i$ is similar with those employed in Algo. 4. However, it is based on the bitruss butterfly-complete subgraphs F_i^B, instead of F_i^+. Note that we only use the index $\bar{H}_i$ without the complete H_i one. This is because no edges in $F_i^B \setminus F_i$ are peeled in the procedure. Hence we do not need to access wedges that consist of both outer edges in $F_i^B \setminus F_i$, and $\bar{H}_i$ alone is sufficient.

Cost of phase Conquer. We can verify that the computation cost of function LocalPeel over fragment F_i^B is in $O(|\bowtie_{F_i,F_i^B}|) = O(|\bowtie_{F_i^B}|)$ (according to Corollary 7.3). With balanced partitions of $|\bowtie_{F_i^B}| = O(|\bowtie_G|/p)$ (shown in Sect. 7.2), the computation cost of phase Conquer is in $O(\max_{i \in [1,p]} |\bowtie_{F_i^B}|) = O(|\bowtie_G|/p)$. That is, phase Conquer is parallel scalable.

Note that, although both of the cost are in $O(|\bowtie_G|/p)$, the butterflies peeled in a bitruss butterfly-complete subgraph F_i^B by LocalPeel are much less than those by SubPeel over a corresponding butterfly-complete one F_i^+. This is because for LocalPeel we only peel inner edges in F_i. Each butterfly in $\bowtie_G$ is enumerated only once according to Lemma 7.2. Whereas SubPeel peels all edges in F_i^+, and each butterfly in $\bowtie_G$ might be peeled by up to 4 times by its 4 edges over 4 different fragments.

7.2 Divide Phase

It remains to answer how to divide G into p fragments. For the Algo. 7 to work, the fragments must be balanced, so that the performance of Conquer phase does not become degraded due to skewness. To achieve this, next we define the problem of *balanced hierarchical bitruss partition*, and propose an efficient heuristic partitioner with balance guarantee.

The phase Divide generates a *hierarchical bitruss partition* over graph G, formally defined as:

Definition 7.2 (Hierarchical Bitruss Partition). *Given a bipartite graph $G(V,E)$ and p, an increasing sequence of positive integers $\langle t_1,\cdots,t_{p-1}\rangle$, a corresponding hierarchical bitruss partition is defined as:*

$F_i = \Gamma_{t_{i-1}} \setminus \Gamma_{t_i}$, for each $i \in [1,p]$ ($t_0 = 1$ and $t_p = +\infty$).

The Partitioning Problem

The balance of the partition plays an important role in the following Conquer phase. Since the cost of the Conquer phase is in $O(\max_{i\in[1,p]} |\bowtie_{F_i^B}|)$, a balanced partition with $|\bowtie_{F_i^B}| \approx |\bowtie_G|/p$ will speed up the computation. We formally define the partitioning problem as follows.

Definition 7.3 (Balanced Hierarchical Bitruss Partition (BHBP)). *Given a bipartite graph $G(V,E)$ and p, find an increasing sequence $t_1,\cdots,t_{p-1}$ such that over the corresponding hierarchical bitruss partition, $\max_{i\in[1,p]} |\bowtie_{F_i^B}|$ is minimum.*

Theorem 7.4. *BHBP $\in P$.*

Hierarchical Partitioner. Though BHBP $\in P$, the DP algorithm used in the proof (in Appendix J) relies on computing ϕ_e of each edge in advance. However, the number of ϕ_e is unknown unless we run bitruss decomposition over G. To solve this chicken-and-egg paradox, we propose the Hierarchical Partitioner (HierarchPart), which efficiently partitions the graph with balance guarantees.

Algorithm 8 HierarchPart (Hierarchical Partitioner)

```
Input: $G(V,E)$, $p$
Output: fragment $F_i$ of $G$ for each $i \in [1,p]$, and $\phi_e$ for each $e \in G \setminus \bigcup_{i\in[1,p]} F_i$
1: Initialize index $H$ and edge support $sup(\cdot)$ over $G$
2: $B_{\max} = |\bowtie_G/p|$, $t_0 \leftarrow 1$
3: for all $i \in [1, p-1]$ do
4:     $t_i \leftarrow$ EstimateBitruNum($\Gamma_{t_{i-1}}$, $B_{\max}$)
5:     $\Gamma_{t_i} \leftarrow$ $k$-Bitruss($\Gamma_{t_{i-1}}$, $t_i$),   $F_i \leftarrow \Gamma_{t_i} \setminus \Gamma_{t_{i-1}}$
6:     while $|\bowtie_{F_i}| > B_{\max}$ do
7:         $t \leftarrow \min_{e\in F_i} sup(e)$
8:         Peel all edges $e$ with $\phi_e = t$ from $F_i$
```

Here we introduce the outline of HierarchPart in sequential for simplicity. Readers can refer to Appendix F for details, including estimation of the bitruss rank and the parallelization. Shown in Algo. 8, HierarchPart takes as input a graph G and the number of partition p. It outputs p disjoint subgraph F_i of G, such that $|\bowtie_{F_i^B}| \leq |\bowtie_G|/p$. Note that $\bigcup_{i\in[1,p]} F_i$ may not cover all edges in G. For those left out edges e, HierarchPart also outputs their bitruss numbers ϕ_e. This way, after the Conquer phase is performed on the fragments of F_i, ϕ_e for all edges $e \in G$ are answered.

HierarchPart first initializes index H and edge support and declares a maximum load of $B_{\max}$ (lines 1-2). It then generates $F_i = \Gamma_{t_{i-1}} \setminus \Gamma_{t_i}$ for $i \in [1,p]$ in an ascending order of t_i (lines 3-8). For each F_i, it first estimates a bitruss number t_i by calling EstimateBitruNum (line 4). The function EstimateBitruNum estimates the bitruss number t_i, such that for the new fragment F_i incurs a balanced load of approximately $|\bowtie_G|/p$. The algorithm computes the subgraph of Γ_{t_i} with the estimated t_i by running k-Bitruss over the previously computed $\Gamma_{t_{i-1}}$, and the new fragment of F_i is computed as $\Gamma_{t_{i-1}} \setminus \Gamma_{t_i}$ (line 5). To enforce the balance constraint, when $|\bowtie_{F_i^B}| > B_{\max}$, the algorithm further peels F_i, until $|\bowtie_{F_i^B}|$ is reduced below $B_{\max}$ (lines 6-8). We revise the algorithm SeqPeel and employ it for the computation of Γ_{t_i} at line 5 and the peeling at line 8. The revisions made are addressed later in Section 7.3. The algorithm terminates when all p fragments are computed.

7.3 Optimizations

Note that the Divide phase itself must be efficient, so that the speedup of phase Conquer is not canceled by the cost of the Divide phase. Although we can directly plug in the parallel SC-PBD in Section 4.2 for the function k-Bitruss at line 5, it incurs redundant computation and can be further sped up. Here we briefly describe two optimizations for speeding up k-Bitruss, and show the intuitions behind them. Refer to Appendix G for details.

Recounting Butterfly

An alternative way for peeling a set of edges Q from fragment F_i^+ is to directly delete Q and recompute *sup*() for each edges in $F_i^+ \setminus Q$ afterwards. We denote this method as Recount, in contrast with the peel-and-update-support based methods denoted as Peel. Note that the cost of Recount is in $O(|\bowtie_{Q,F_i^+}|)$, and the cost of Peel is in $O(|\mathbb{W}_{F_i^+\setminus Q}|)$. It is possible for Recount to run faster when 1) the set Q is large enough or 2) when F_i^+ is dense enough, since in both cases $|\mathbb{W}_{F_i^+\setminus Q}| < |\bowtie_{Q,F_i^+}|$. By switching to the faster methods among Recount and Peel, we are able to boost the computaion of k-Bitruss, which only employs Peel as a subroutine by default.

Delta-based Peeling

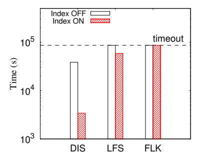
In function k-Bitruss (line 5 in Algo. 8), we are computing Γ_{t_i} from the current subgraph of $\Gamma_{t_{i-1}}$, hence it is overkill to directly plug in SC-PBD and compute ϕ_e for each peeled edge e in $\Gamma_{t_i} \setminus \Gamma_{t_{i-1}}$. To deal with this issue, we propose an optimization that boosts k-Bitruss() by avoiding explicitly computing ϕ_e for all edges.

The key idea is to first accumulate changes in a delta index $\Delta H_i(u,v)$, for each u, v. Then we propagate the accumulated changes to udpate edge support for wedges indexed by $H_i(u,v)$.

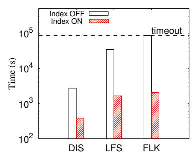
To see that this optimization works, observe that each time, *sup*() is decreased by an accumulated delta count indicated by $\Delta H_i(u,v)$, instead of only 1. For instance, when set of edges to peel Q is the whole fragment F_i, peeling via explicitly enumerating butterflies will have to scan all $|\bowtie_{F_i,F_i^+}|$ butterflies. In contrast, we can (1) accumulate changes into $\Delta H_i(u,v)$ for each u, v, and (2) propagate the changes to the associated edges. Both (1) and (2) can be done by one pass scanning over the local indexes with the size of $|\mathbb{W}_{F_i^+}|$, which is much faster.

Table 1: Datasets

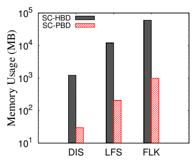
	Name	Abbr.	$\|G\|$	$\|\bowtie_G\|$	Network Type
Medium	Discog-lstyle	DIS	1.1×10^6	5.2×10^9	feature
	lasf.FM-song	LFS	4.4×10^6	3.2×10^{10}	interaction
	Flickr	FLK	8.5×10^6	3.5×10^{10}	affiliation
Large	Delicious	DEL	1.0×10^8	5.7×10^{10}	interaction
	Epinions	EPN	1.3×10^7	1.7×10^{11}	rating
	Jester150	JST	1.7×10^6	2.7×10^{11}	rating
	Movielens	MV	1.0×10^7	1.2×10^{12}	rating
	Livejournal	LJ	1.1×10^8	3.3×10^{12}	affiliation
	Reuters	RTS	6.1×10^7	7.5×10^{12}	text
	WebTracker	TRK	1.4×10^8	2.0×10^{13}	hyperlink



(a) Response Time of SC-HBD

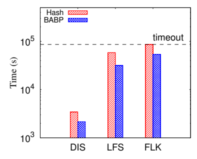


(b) Response Time of SC-PBD

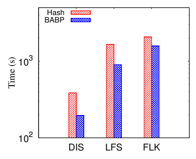


(c) Space Cost of Index

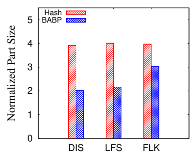
Figure 1: Effectiveness of Local Indexes



(a) Response Time of SC-HBD



(b) Response Time of SC-PBD



(c) Max Partition Size

Figure 2: Effectiveness of Partitioning

8 EXPERIMENTS

In this section, we conduct experiments with the aim of answering the following research questions: (1) Can local indexes (Sect. 5) speed up our distributed bitruss decomposition algorithms? How much extra memory space is required for storing the indexes? (2) Compared with naive hash based partitioning, can our partitioning methods (Sect. 6) reduce the parallel computation costs ? (3) Do our optimization strategies (Sect. 7.3) improve the efficiency of the Divide phase in DC-BD? (4) Are our distributed algorithms parallel scalable, *i.e.,* taking less time with more computation resources? (5) Can our methods scale to large datasets? (6) How do our methods compare to existing parallel solutions?

Datasets. We use 10 real-life bipartite graphs of various categories. The graphs summarized in Table 1, are ordered by their $|\bowtie_G|$. All the datasets are available on KONECT [1]. Graphs containing parallel edges including DIS, LFS, DEL and RTS are deduplicated before testing. We also generate synthetic bipartite graphs controlled by $|\bowtie_G|$ for testing the scalability of our algorithms.

Algorithms. We test our decomposition algorithms including SC-HBD (Sect. 3), SC-PBD (Sect. 4) and DC-BD (Sect. 7). Our algorithms are equipped with all optimizations described in this paper, unless otherwise stated. We also compare our algorithms with parallel state-of-art solutions, including the BiT-BU and BiT-PC methods in [38], and the ParButterfly algorithm in [28]. Note that there are two BiT-BU methods described in [38], *i.e.,* BiT-BU and BiT-BU^{++}. Since neither of the two consistently outperform one another, here we only report the optimal results of the two BiT-BU methods.

[1]http://konect.cc/

Implementation. The algorithms are all implemented in C++, using openMPI for inter-process communications. They are tested over a cluster of 8 machines, each equipped with an Intel Xeon CPU E5-2692 v2 @ 2.20GHz (12 CPU cores) and 64GB memory. Tests taking more than 24 hours are terminated and marked as "timeout". The parameter ϵ in BABP is set to 1.1 in our experiments. For the shared-memory parallel algorithms of BiT-PC, BiT-BU and ParButterfly, we use all 12 available cores on one machine, fixing the thread number to 24.

8.1 Effectiveness of Optimizations

Over datasets DIS, LFS and FLK, we first verify the effectiveness of our optimizations, including local index (Sect. 5), BABP partitioning (Sect. 6) and optimizations for Divide in DC-BD (Sect. 7.3).

Local Index. For SC-HBD and SC-PBD, we compare the methods equipped with index against their counterparts without index. All four methods work on F_i^+ derived from basic hash edge partitions.

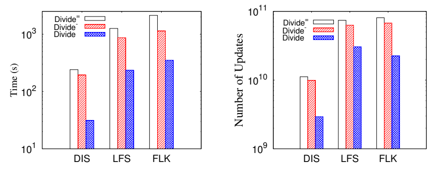
Impact on efficiency. Fixing $p = 96$, the total response time of SC-HBD (resp. SC-PBD) with and without local index is shown in Fig 1a (resp. Fig 1b). Note that the time for index construction is included in the total response time. We find that the local index significantly speeds up both SC-HBD and SC-PBD. More specifically: (1) SC-HBD with index is 11× faster than its counterpart over DIS, and (2) the local index on SC-PBD speeds up its overall response time by 7.1× and 21× on DIS and LFS, respectively. Taking less than 12% of the overall total response time, the indexes boost the overall efficiency by trading off space against time. By employing indexes, SC-HBD and SC-PBD can enumerate the butterflies associated with a given edge in time linear to the size of the butterfly subset. In contrast, without index, one has to compute the butterfly set from scratch each time. Thus, the redundant computation for enumerating edges that do not form a butterfly is pruned.

Memory usage. In the same setting, we also report the space cost of our indexes in Fig 1c. Note that over a fragment F_i, we have two different indexes, namely the full indexes of H_i and its pruned version $\bar{H}_i$. The former is used on SC-HBD and the latter is used for SC-PBD and DVC. The results show that the pruned index $\bar{H}_i$ is on average 53 times smaller than the full indexes H_i. This is because the total space cost of pruned indexes $\bar{H}_i$ over each fragment is bounded, *i.e.,* each wedge in $\mathbb{W}_G$ is indexed by at most two fragments. Whereas for the full index H_i there is no such guarantee. That is, our indexes successfully boost the computation while only consuming moderate space.

Partitioning. Next, we verify the effectiveness of BABP over SC-HBD and SC-PBD. We compare the algorithms running on hash partitions, against those running on BABP partitions.

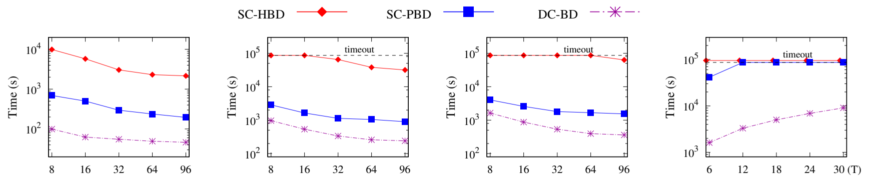
Impact on efficiency. Shown in Fig 2a-2b, we can see: (1) BABP boosts SC-HBD by 59% and 85% over DIS and LFS, respectively, compared with that running on hash partitions. (2) For SC-PBD, the gap of BABP and hash partitioning is ≥ 31% over all three datasets. (3) The partitioning cost are included in the total response time, which is less than 12%.

Maximum partition size. In the same setting, in Fig 2c, we also report the *normalized maximum partition size, i.e.,* $\max_{i\in[1,p]} |\bowtie_{F_i,F_i^+}|/B'$, where $B' = |\bowtie_G|/p$. The results show that: (1) the hash partition has a normalized maximum size of at least 3.9 over all 3 datasets. This means nearly all butterflies are replicated 4 times in the partitioned



(a) Response Time of Divide (b) Number of Updates

Figure 3: Effectiveness of Optimizations on Divide



(a) Varying p, DIS (b) Varying p, LFS (c) Varying p, FLK (d) Varying $|\bowtie_G|$

Figure 4: Efficiency of Parallel Bitruss Decompositions

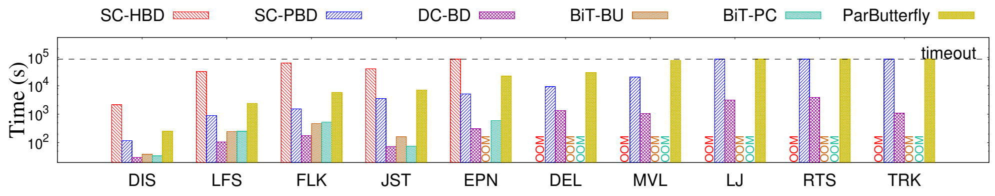


Figure 5: Comparison of Different Bitruss Decomposition Methods ($p = 96$, OOM denotes Out-Of-Memory Error)

butterfly-complete fragments. (2) In contrast, BABP reduces the maximum partition size by at least 32%. That is, BABP successfully boosts the computation by reducing the redundant computation incurred by replicated butterflies across the fragments.

Workload balance. In addition to the normalized maximum partition size, we also report the balance ratio of the partition, defined in BABGP (Sec 6.1), *i.e.,* $\max_{i\in[1,p]} |\bowtie_{F_i,F_i^+}|/B$, where $B = \sum_{i\in[1,p]} |\bowtie_{F_i,F_i^+}|/p$. When p varies from 8 to 96, the balance ratio of BABP is at most 1.02, 1.1 and 1.07 over DIS, LFS and FLK, respectively (not shown). These verify the balance of BABP.

Optimizations on DC-BD.

Next we verify the effectiveness of optimizations described in Sect. 7.3 including 1) recounting butterflies and 2) delta-index based peeling. We denote Divide without butterfly recounting as Divide⁻, and the baseline with neither optimizations as Divide⁼. Note that phase Divide derives from the distributed peeling of SC-PBD. As the effectiveness of optimizations tailored for SC-PBD has already been verified, they are equipped on Divide by default.

The overall efficiency of Divide is reported in Fig 3a. We find that: 1) The butterfly recounting technique on Divide⁻ on average speeds up Divide⁼ by 52%, and 2) Divide is at least 3.2× faster than Divide⁻. Observe that the baseline Divide⁼ is even slower than SC-PBD. This is because it not only needs to scan all butterflies in $\bowtie_G$, but also incurs the extra overhead of bitruss number estimations.

To see why these optimizations work, we also report the total number of updates on edges in Fig 3b. The results show that: (1) equipped with butterfly recounting, the total number of updates over edges is cut down by 14%, on average. (2) The delta-index based peeling further reduces the number of updates by at least 52%. That is, these optimizations reduce the number of edge updates performed by Divide, leading to a significant boost in its effciency.

8.2 Efficiency

Fixing the algorithms as their optimized versions, we test the impacts of varying 1) the number of CPU cores p and 2) the butterfly size $|\bowtie_G|$, over the total response time of our algorithms.

Parallel Scalability. Varying p from 8 to 96, we tested the parallel scalability of our parallel bitruss decompositions over DIS, FLS and FLK. The results are shown in Fig 4a-4c.

(a) All three algorithms take less time to decompose with more computation resources, as expected. SC-PBD and DC-BD on average speeds up by 3.1× and 3.6×, respectively, when p increases from 8 to 96. SC-HBD fails to solve the decomposition with in 1 day when p is low on LFS and LFS, while its speedup is 4.6× over DIS.

(b) For all p, SC-PBD consistently outperforms SC-HBD, by at least 9.7 times, whereas DC-BD is on average 4.3× faster than SC-PBD.

Scalability. Fixing $p = 96$, we test the impact of the number of butterflies on our algorithms. We generate synthetic bipartite graphs, varying $|\bowtie_G|$ from 6 Trillion to 30 Trillion. The results are shown in Fig 4d. We can see that 1) DC-BD successfully computes the bitruss decomposition of graphs with 30T butterflies in 2.5 hours. 2) When the $|\bowtie_G|$ grows from 6T to 30T, DC-BD is only 5.6× slower, *i.e.,* DC-BD scales well *w.r.t.* $|\bowtie_G|$. 3) SC-PBD fails to compute graphs with $|\bowtie_G| \geq 12$T within limited time, while SC-HBD times out on all synthetic graphs.

8.3 Comparison with Existing Methods

Fixing $p = 96$, we compare our methods against parallel algorithms of BiT-BU, BiT-PC and ParButterfly, over all datasets listed in Table 1. The results are shown in Fig 5. We find the following:

(a) DC-BD consistently beats other methods including BiT-BU and ParButterfly, by at least 1.3× and 8.3×, respectively. DC-BD is comparable with BiT-PC on JST with a gap of less than 2%, while on other graphs like FLK the gap can be upto 3.0×.

(b) Methods including DC-BD, SC-PBD and ParButterfly are able to handle large graphs, such as LJ, RTS and TRK. Among these three methods, only DC-BD successfully completed bitruss decomposition over all datasets within 24 hours. These verify the efficiency and scalability of DC-BD.

(c) BiT-BU and BiT-PC are not as scalable due to their exhaustive memory usage on large graphs. Both fail to handle graphs including DEL, LJ and RTS and reports OOM errors. This verifies the need for distributed bitruss decomposition.

(d) Our method may take more time than sequential methods with low number of workers on small graphs (not shown). For instance, over DIS, DC-BD outperforms the sequential version of BiT-PC only when $p > 16$. This is because on small graphs, very few edges are peeled between two rounds of global synchronization, causing a significant overhead of communication.

Summary. Our distributed algorithms solve bitruss decomposition in hours over graphs with trillions of butterflies. (1) Our optimizations effectively speed up the algorithms. Equipped with all optimizations, SC-HBD, SC-PBD and DC-BD become on average 18, 26 and 6.4 times faster, respectively. (2) Our methods are parallel scalable. When p grows from 8 to 96, SC-HBD, SC-PBD and DC-BD speeds up by 4.6, 3.1 and 3.6 times, respectively. (3) DC-BD can scale up to graphs with 30T of $|\bowtie_G|$, and terminate in 2.5 hours. (4) DC-BD is the most efficient bitruss decomposition method. Fixing $p = 96$, for all 10 real-life datasets, it consistently beats SC-HBD, SC-PBD, BiT-BU and ParButterfly by at least 72, 3.9, 1.3 and 8.3 times, respectively, and is on average 1.9 times faster than BiT-PC.(5) Existing methods of BiT-BU and BiT-PC run out of memory space on dense graphs such as LJ, RTS and TRK. This highlights the need for distributed bitruss decomposition.

9 RELATED WORKS

Parallel Bitruss Decomposition. There are also works about parallel bitruss decomposition [17, 28, 38] on shared-memory machines. [28] maintains a global bucketing structure which maps each edge to a bucket by butterfly count. Throughout the peeling, for each edge peeled, set interaction is performed to detect the affected edges, then information is grouped to update the bucketing structure simultaneously. Recently, [38] extends [37] to the multi-core environment, and presents methods for parallel BE-Index construction and peeling, by reducing writing conflicts when operating index using multiple threads. [17] partitions the BE-Index firstly, and then performs peeling on different parts of the BE-Index by dynamic task allocation. To design distributed methods, one may partition the graph, directly use above parallel solutions, and take the communication into account. However, it is challenging to maintain and partition related global auxiliary structures (bucketing in [28], BE-Index in [17, 38]) in addition to the graph itself, due to the randomness of reading/writing operations.

Cohesive Structures on Bipartite Graphs. Several cohesive structures have been proposed on bipartite graphs. The (α, β)-core [20] is a maximal subgraph where each node in the upper/lower layer has at least α/β neighbors. Maximal biclique [2] and quasi-biclique [30] are studied, while the latter is a maximal subgraph where each node in the upper layer (resp. lower layer) has at most ϵ (a positive integer) non-neighbors in lower layer (resp. upper layer). [22] finds such a biclique in G whose (p, q)-biclique density is largest. A k-tip [25] is a maximal subgraph in which each node takes part in at least k butterflies. We compare the above cohesive structures with k-bitruss. While enumerating maximal (quasi-)bicliques is NP-hard, decomposing bitruss can be done in polynomial time. Compared with (α, β)-core and density based (p, q)-biclique which need extra user inputs/parameters, k-bitruss is a parameter-free model and thus practical when users have not dived into the properties of the underlying graph. Besides, k-bitruss can provide hierarchical communities which can be used in different levels of granularity. The k-tip is similar to k-bitruss but a measure on nodes. Since k-tip finds vertex-induced subgraphs and k-bitruss finds edge-induced communities, k-bitruss can identify overlapping communities where a vertex can belong to multiple groups. This is practical in real-world applications, *e.g.,* a user can belong to different social groups based on his different hobbies.

Graph Partitioning. Graph partitioning is crucial for distributed graph computation. A number of methods (see surveys [6, 8]) are proposed for vertex-cut [7, 10, 16, 18, 24, 33] and edge-cut [3, 14, 15, 31, 32, 41]. Vertex-cut (resp. edge-cut) partitions edges (resp. vertices) into disjoint balanced subsets and reduce vertex (resp. edge) replication. As pointed out by [10, 35], vertex-cut is more suitable for power-law graphs with several nodes of high degree (hubs), since it allows better load balance by distributing those nodes among different machines; while edge-cut is better for graphs with many low-degree nodes since their adjacent edges are also transferred to the same machine. Hybrid partitioners are also proposed. [11] and [10] bond vertex-cut with edge-cut by cutting high-degree nodes controlled by parameters. [19] further merges close low-degree nodes into super nodes to avoid splitting them. Different from the above partitioners, BABP aims to balance the number of butterflies associated with inner edges and minimize the size of external edges incurred by completing butterflies. There also exists work about motif-aware graph clustering [4, 34]. [4] firstly builds a motif adjacency matrix, and then computes the spectral ordering from the normalized motif Laplacian matrix, and finally finds the best high-order cluster with the smallest conductance. [34] weights each edge according to the number of triangles it is contained in, and then simply removes edges whose weights are smaller than a threshold θ and outputs connected components as clusters. The above methods cannot be extended to solve BABGP since they cannot control the number of partitions in advance, and do not balance the size of the resulting components.

10 CONCLUSION

In this paper, we study the problem of distributed bitruss decomposition. We first propose SC-HBD, which uses $\mathcal{H}$-function to define bitruss numbers and computes them to a fixed point. We then introduce SC-PBD, a subgraph-centric batch peeling method executed over different butterfly-complete subgraphs. We also discuss how to build the local index on butterfly-complete subgraphs, and study the partition problem. We finally propose the concept of a bitruss butterfly-complete subgraph, and a divide and conquer method DC-BD. We also introduce various optimizations that improve our methods of SC-HBD, SC-PBD and DC-BD on average by 18, 26 and 6.4 times in practice. Extensive experiments show that the proposed methods solves graphs with 30 trillion of butterflies in 2.5 hours, while existing parallel methods under shared-memory model fail to scale to such large graphs. One topic of possible future work is to extend the framework to handle multipartite graphs.

REFERENCES

[1] [n.d.]. IMDB 5000 Movie Dataset. https://www.kaggle.com/carolzhangdc/imdb-5000-movie-dataset.

[2] Aman Abidi, Rui Zhou, Lu Chen, and Chengfei Liu. 2021. Pivot-based maximal biclique enumeration. In *Proceedings of the Twenty-Ninth International Conference on International Joint Conferences on Artificial Intelligence*. 3558–3564.

[3] Amine Abou-Rjeili and George Karypis. 2006. Multilevel algorithms for partitioning power-law graphs. In *Proceedings 20th IEEE International Parallel & Distributed Processing Symposium*. IEEE, 10–pp.

[4] Austin R Benson, David F Gleich, and Jure Leskovec. 2016. Higher-order organization of complex networks. *Science* 353, 6295 (2016), 163–166.

[5] Alex Beutel, Wanhong Xu, Venkatesan Guruswami, Christopher Palow, and Christos Faloutsos. 2013. Copycatch: stopping group attacks by spotting lockstep behavior in social networks. In *Proceedings of the 22nd international conference on World Wide Web*. 119–130.

[6] Charles-Edmond Bichot and Patrick Siarry. 2013. *Graph partitioning*. John Wiley & Sons.

[7] Florian Bourse, Marc Lelarge, and Milan Vojnovic. 2014. Balanced graph edge partition. In *SIGKDD*. 1456–1465.

[8] Aydın Buluç, Henning Meyerhenke, Ilya Safro, Peter Sanders, and Christian Schulz. 2016. Recent advances in graph partitioning. *Algorithm engineering* (2016), 117–158.

[9] Rong Chen, Jiaxin Shi, Yanzhe Chen, and Haibo Chen. 2015. PowerLyra: Differentiated Graph Computation and Partitioning on Skewed Graphs. In *EuroSys*. 1:1–1:15.

[10] Rong Chen, Jiaxin Shi, Yanzhe Chen, Binyu Zang, Haibing Guan, and Haibo Chen. 2019. Powerlyra: Differentiated graph computation and partitioning on skewed graphs. *ACM Transactions on Parallel Computing (TOPC)* 5, 3 (2019), 1–39.

[11] Dong Dai, Wei Zhang, and Yong Chen. 2017. IOGP: An incremental online graph partitioning algorithm for distributed graph databases. In *Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing*. 219–230.

[12] Michael L Fredman and Robert Endre Tarjan. 1987. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM (JACM)* 34, 3 (1987), 596–615.

[13] Xin Huang, Hong Cheng, Lu Qin, Wentao Tian, and Jeffrey Xu Yu. 2014. Querying k-truss community in large and dynamic graphs. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. 1311–1322.

[14] George Karypis and Vipin Kumar. 1998. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on scientific Computing* 20, 1 (1998), 359–392.

[15] George Karypis and Vipin Kumar. 1999. Parallel multilevel series k-way partitioning scheme for irregular graphs. *Siam Review* 41, 2 (1999), 278–300.

[16] Mijung Kim and K Selçuk Candan. 2012. SBV-Cut: Vertex-cut based graph partitioning using structural balance vertices. *Data & Knowledge Engineering* 72 (2012), 285–303.

[17] Kartik Lakhotia, Rajgopal Kannan, and Viktor Prasanna. 2021. Parallel Peeling of Bipartite Networks for Hierarchical Dense Subgraph Discovery. *arXiv preprint arXiv:2110.12511* (2021).

[18] Michael LeBeane, Shuang Song, Reena Panda, Jee Ho Ryoo, and Lizy K John. 2015. Data partitioning strategies for graph workloads on heterogeneous clusters. In *SC'15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–12.

[19] Dongsheng Li, Yiming Zhang, Jinyan Wang, and Kian-Lee Tan. 2019. TopoX: Topology refactorization for efficient graph partitioning and processing. *Proceedings of the VLDB Endowment* 12, 8 (2019), 891–905.

[20] Boge Liu, Long Yuan, Xuemin Lin, Lu Qin, Wenjie Zhang, and Jingren Zhou. 2019. Efficient (α, β)-core computation: An index-based approach. In *The World Wide Web Conference*. 1130–1141.

[21] Linyuan Lü, Tao Zhou, Qian-Ming Zhang, and H Eugene Stanley. 2016. The H-index of a network node and its relation to degree and coreness. *Nature communications* 7, 1 (2016), 1–7.

[22] Michael Mitzenmacher, Jakub Pachocki, Richard Peng, Charalampos Tsourakakis, and Shen Chen Xu. 2015. Scalable large near-clique detection in large-scale networks via sampling. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 815–824.

[23] Alberto Montresor, Francesco De Pellegrini, and Daniele Miorandi. 2012. Distributed k-core decomposition. *IEEE Transactions on parallel and distributed systems* 24, 2 (2012), 288–300.

[24] Fabio Petroni, Leonardo Querzoni, Khuzaima Daudjee, Shahin Kamali, and Giorgio Iacoboni. 2015. Hdrf: Stream-based partitioning for power-law graphs. In *Proceedings of the 24th ACM international on conference on information and knowledge management*. 243–252.

[25] Ahmet Erdem Sariyüce and Ali Pinar. 2018. Peeling Bipartite Networks for Dense Subgraph Discovery. In *Proceedings of the Eleventh ACM International Conference on Web Search and Data Mining, WSDM 2018, Marina Del Rey, CA, USA, February 5-9, 2018*, Yi Chang, Chengxiang Zhai, Yan Liu, and Yoelle Maarek (Eds.). ACM, 504–512. https://doi.org/10.1145/3159652.3159678

[26] Ahmet Erdem Sariyüce, C. Seshadhri, and Ali Pinar. 2018. Local Algorithms for Hierarchical Dense Subgraph Discovery. *Proc. VLDB Endow.* 12, 1 (2018), 43–56. https://doi.org/10.14778/3275536.3275540

[27] Yingxia Shao, Lei Chen, and Bin Cui. 2014. Efficient cohesive subgraphs detection in parallel. In *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*, Curtis E. Dyreson, Feifei Li, and M. Tamer Özsu (Eds.). ACM, 613–624. https://doi.org/10.1145/2588555.2593665

[28] Jessica Shi and Julian Shun. 2020. Parallel Algorithms for Butterfly Computations. In *1st Symposium on Algorithmic Principles of Computer Systems, APOCS@SODA 2020, Salt Lake City, UT, USA, January 8, 2020*. SIAM, 16–30. https://doi.org/10.1137/1.9781611976021.2

[29] Yue Shi, Martha Larson, and Alan Hanjalic. 2014. Collaborative filtering beyond the user-item matrix: A survey of the state of the art and future challenges. *ACM Computing Surveys (CSUR)* 47, 1 (2014), 1–45.

[30] Kelvin Sim, Jinyan Li, Vivekanand Gopalkrishnan, and Guimei Liu. 2009. Mining maximal quasi-bicliques: Novel algorithm and applications in the stock market and protein networks. *Statistical Analysis and Data Mining: The ASA Data Science Journal* 2, 4 (2009), 255–273.

[31] Isabelle Stanton and Gabriel Kliot. 2012. Streaming graph partitioning for large distributed graphs. In *Proceedings of the 18th ACM SIGKDD international conference on Knowledge discovery and data mining*. 1222–1230.

[32] Dan Stanzione, Bill Barth, Niall Gaffney, Kelly Gaither, Chris Hempel, Tommy Minyard, Susan Mehringer, Eric Wernert, H Tufo, D Panda, et al. 2017. Stampede 2: The evolution of an xsede supercomputer. In *Proceedings of the Practice and Experience in Advanced Research Computing 2017 on Sustainability, Success and Impact*. 1–8.

[33] Charalampos Tsourakakis, Christos Gkantsidis, Bozidar Radunovic, and Milan Vojnovic. 2014. Fennel: Streaming graph partitioning for massive scale graphs. In *Proceedings of the 7th ACM international conference on Web search and data mining*. 333–342.

[34] Charalampos E Tsourakakis, Jakub Pachocki, and Michael Mitzenmacher. 2017. Scalable motif-aware graph clustering. In *Proceedings of the 26th International Conference on World Wide Web*. 1451–1460.

[35] Shiv Verma, Luke M Leslie, Yosub Shin, and Indranil Gupta. 2017. An Experimental Comparison of Partitioning Strategies in Distributed Graph Processing. *Proceedings of the VLDB Endowment* 10, 5 (2017).

[36] Kai Wang, Xuemin Lin, Lu Qin, Wenjie Zhang, and Ying Zhang. 2019. Vertex Priority Based Butterfly Counting for Large-scale Bipartite Networks. *Proc. VLDB Endow.* 12, 10 (2019), 1139–1152. https://doi.org/10.14778/3339490.3339497

[37] Kai Wang, Xuemin Lin, Lu Qin, Wenjie Zhang, and Ying Zhang. 2020. Efficient Bitruss Decomposition for Large-scale Bipartite Graphs. In *36th IEEE International Conference on Data Engineering, ICDE 2020, Dallas, TX, USA, April 20-24, 2020*. IEEE, 661–672. https://doi.org/10.1109/ICDE48307.2020.00063

[38] Kai Wang, Xuemin Lin, Lu Qin, Wenjie Zhang, and Ying Zhang. 2021. Towards efficient solutions of bitruss decomposition for large-scale bipartite graphs. *The VLDB Journal* (2021), 1–24.

[39] Wikipedia contributors. 2021. Facebook — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=Facebook&oldid=1010764584 [Online; accessed 5-May-2022].

[40] Wikipedia contributors. 2021. Singles' Day — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=Singles%27_Day&oldid=1007941275 [Online; accessed 5-May-2022].

[41] Xiaowei Zhu, Wenguang Chen, Weimin Zheng, and Xiaosong Ma. 2016. Gemini: A computation-centric distributed graph processing system. In *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)*. 301–316.

[42] Zhaonian Zou. 2016. Bitruss Decomposition of Bipartite Graphs. In *Database Systems for Advanced Applications - 21st International Conference, DASFAA 2016, Dallas, TX, USA, April 16-19, 2016, Proceedings, Part II (Lecture Notes in Computer Science)*, Shamkant B. Navathe, Weili Wu, Shashi Shekhar, Xiaoyong Du, X. Sean Wang, and Hui Xiong (Eds.), Vol. 9643. Springer, 218–233. https://doi.org/10.1007/978-3-319-32049-6_14

APPENDICES

A Notations

Notations and symbols used in the paper are summarized in Table 2.

B Sequential Peeling

Existing solutions [25, 42] can be abstracted as SeqPeel (Algorithm 9). It has two phases: (1) counting phase: count $|\bowtie_{e,G}|$ for each e, labeled as $sup(e)$ (line 1); (2) peeling phase: at each iteration, the edge e with the minimum support is removed from G (line 8), labeled with ϕ_e as $sup(e)$ (line 3), and the $sup(\cdot)$ of affected edge by peeling e is updated (lines 4-7). The correctness of SeqPeel is proved in [42]. Different methods vary in how to retrieve butterflies and update support information, leading to different cost. It is shown the time complexity of SeqPeel in [42] is $O(m^2)$, while it is $O(\sum_{u\in U}\sum_{(v_1,u),(v_2,u)\in E}\max\{d(v_1),d(v_2)\}+\sum_{(u,v)\in E}\sum_{(w,v)\in E}\max\{d(u),d(w)\})$ in [25]. Recently, [37] improves the time complexity to $O(\sum_{(u,v)\in E}\min\{d(u),d(v)\}+|\bowtie_G|)$, by using vertex priority to boost butterfly counting and BE-Index to update support. As discussed in Sect. 1, it is difficult to directly deploy such algorithms on a shared-nothing model.

Algorithm 9 SeqPeel

```
Input: $G(V, E)$
Output: $\phi_e$ for each $e \in E$
1: $\forall e \in G$, $sup(e) \leftarrow |\bowtie_{e,G}|$
2: while $\exists e \in E$ such that $\phi_e$ is unset do
3:     $e \leftarrow \arg\min_{e' \in E} sup(e')$, $\phi_e \leftarrow sup(e)$
4:     for all $\bowtie \in \bowtie_{e,G}$ do
5:         for all $e \in \bowtie \wedge e \neq e'$ do
6:             if $sup(e') > sup(e)$ then
7:                 $sup(e') \leftarrow sup(e') - 1$
8:         $G \leftarrow G \setminus \{e\}$
9: return $\phi_e$ for each $e$
```

C Building Butterfly-Complete Subgraph

The butterfly-complete subgraph F_i^+ is constructed by scanning F_i in two passes, shown in Algo. 10. The algorithm takes in the initial subgraph F_i and the global degrees $d(\cdot)$. In the first pass scan, remote edges of each vertex are fetched via communications among different fragments (lines 1-8). Such edges may introduce new vertices (line 8). It can be verified that by including all adjacent edges for both endpoints of an edge $e \in F_i$, all vertices covered by the butterfly set $\bowtie_{e,G}$ are included in F_i'. Then in the second pass, remote edges of the enlarged fragment are synchronized again (lines 9-17). However this time we only add edges without introducing new vertices to current fragments.

Table 2: Notations and Symbols

Notation	Description
p	The number of workers
F_i, F_i^+	The i-th fragment and its butterfly-complete counterpart
$d(u)$, $d_i(u)$	The degree of u in G and in F_i^+, respectively
ϕ_e, $\phi_{\max}$	The bitruss number of e, the maximum bitruss number
${}^{u}_{v}\bowtie^{w}_{x}$	A butterfly that is composed of nodes u, v, w, x
$\bowtie_G$	The set of butterflies in G
$\bowtie_{e,G}$	The set of butterflies in G containing edge e
$\bowtie_{E,G}$	The set of butterflies in G containing edges in E
$\delta(G)$	The minimum support of edges in G
Γ_k	The k-bitruss of G
z	The number of distinct bitruss numbers of G
${}_v\angle_u^w$	A wedge with v as pivot and u, w as endpoints
$p(u)$	The priority of vertex u
$\mathbb{W}_G$	$\{{}_v\angle_u^w \mid (u, v), (v, w) \in G \wedge p(u) > \max(p(v), p(w))\}$

Cost Analysis. Denote by $F_i^{(2)}$ the 2-hop complete subgraph of F_i by including all edges that are within 2 hops from vertices in F_i. Then the parallel Algo. 10 constructs F_i^+ by scanning edges in $F_i^{(2)}$, and hence is in $O(\max_{i\in[1,p]}|F_i^{(2)}|)$.

Algorithm 10 Construct Butterfly Complete Graph

```
Input: $F_i = (V_i, E_i)$, $i \in [1, p]$, degrees $d(\cdot)$
Output: $F_i^+ = (V_i', E_i')$, $i \in [1, p]$
1: for all fragment $F_i$ in parallel do
2:     $F_i' \leftarrow F_i$; $MSG_s \leftarrow \emptyset$;
3:     for all $e = (u, v) \in E_i$ do
4:         if $d(u) \neq d_i(u)$ then $MSG_s \leftarrow MSG_s \cup \{(u, e)\}$
5:         if $d(v) \neq d_i(v)$ then $MSG_s \leftarrow MSG_s \cup \{(v, e)\}$
6: exchange messages in $MSG_s$ among fragments
7: for all fragment $F_i'$ in parallel do
8:     for all $(u, (u, v)) \in MSG_r$ do $E_i' \leftarrow E_i' \cup \{(u, v)\}$; $V_i' \leftarrow V_i' \cup \{u\}$
9: for all fragment $F_i'$ in parallel do
10:     $MSG_s \leftarrow \emptyset$;
11:     for all $e = (u, v) \in E_i'$ do
12:         if $d(u) \neq d_i(u)$ then $MSG_s \leftarrow MSG_s \cup \{(u, e)\}$
13:         if $d(v) \neq d_i(v)$ then $MSG_s \leftarrow MSG_s \cup \{(v, e)\}$
14: exchange messages in $MSG_s$ among fragments
15: for all fragment $F_i^+$ in parallel do
16:     for all $(u, (u, v)) \in MSG_r$ do
17:         if $v \in V_i'$ then $E_i' \leftarrow E_i' \cup \{(u, v)\}$;
18: Return $F_i'$ as $F_i^+$;
```

D Build Local Index

Building Local Index. The index H_i and $\bar{H}_i$ can be built by scanning all local wedges ${}_v\angle_u^w$ in $\mathbb{W}_{F_i^+}$ and adding them to the corresponding sets of $H_i(u,w)$ and $\bar{H}_i(u,w)$.

Illustrated in Algo. 11, the algorithm takes as input the fragment F_i together with its butterfly-complete counterparts F_i^+, and outputs the index of H_i and $\bar{H}_i$. The indexes are first initialized as empty sets (line 1). The algorithm iterates through edges (u,v) in F_i^+ and scans all wedges of the endpoints with the lower p as pivot (lines 2-4). Here p defines total order over all vertices in F_i^+. Each wedge ${}_v\angle_u^w$ scanned is added to the set of $H_i(u,w)$ (line 5), and inner wedges with edges in F_i are also recorded in corresponding set of $\bar{H}_i$ (lines 6-7).

Cost of Index Construction. For each edge (u,v) in F_i^+, we take the endpoint with higher priority $p()$, and scan its adjacent edges (line 4). This cost is in $O(d(x))$, where $x \in \{u,v\}$ and $p(x) = \max(p(u),p(v))$. Hence the total cost of index construction is in $O(\Sigma_{(u,v)\in F_i^+}d(x))$. In order to reduce the cost of index construction, we use $d()$ as $p()$ (similar trick in [36, 37]). This way, we can minimize the cost of index construction over F_i^+, which is in $O(\Sigma_{(u,v)\in F_i^+}\min_{x\in\{u,v\}}d(x))$. Since the index is built on each

Algorithm 11 Building Index H_i and $\bar{H}_i$

```
Input: $F_i$, $F_i^+$ and partial order $p()$ over vertices in $F_i^+$
Output: Index $H_i$ and $\bar{H}_i$
1: Initialize all $H_i(\cdot,\cdot)$ and $\bar{H}_i(\cdot,\cdot)$ as $\emptyset$;
2: for all $(u, v)$ in $F_i^+$ do
3:     if $p(u) < p(v)$ then swap$(u, v)$
4:     for all $(w, v)$ in $F_i^+$ such that $p(u) > p(v)$ do
5:         $H_i(u, v) \leftarrow H_i(u, v) \cup \{{}_v\angle_u^w\}$
6:         if $(u, v) \in F_i$ or $(w, v) \in F_i$ then
7:             $\bar{H}_i(u, v) \leftarrow \bar{H}_i(u, v) \cup \{{}_v\angle_u^w\}$
   return $H_i$ and $\bar{H}_i$
```

fragment F_i^+ without communications, the total parallel computation cost is in $O(\max_i \Sigma_{(u,v)\in F_i^+} \min_{x\in\{u,v\}} d(x))$. Through this paper, we use $d(u)$ as the total order $p(u)$ in wedge set $\mathbb{W}_{F_i^+}$, unless otherwise stated.

Remarks. The initial support for each edge e in F_i, *i.e.*, $sup(e) = |\bowtie_{e,F_i^+}|$, can be also initialized with the help of the index. Suppose $e = (u, v)$ and $p(u) > p(v)$, $sup(e)$ can be computed as follows:

$$|\bowtie_{e,F_i^+}| = \sum_{(u,w)\in F_i^+, p(u)>p(v)} (|H_i(u,w)| - 1) + \sum_{(w,v)\in F_i^+, p(w)>p(u)} (|H_i(w,v)| - 1) \quad (4)$$

This can be performed by a pass of scan over index H_i as a by-product of the indexing process.

E Parallelization of BABP

We describe how to extend the sequential BABP to a parallel algorithm. Here we only highlight the differences, listed as follows.

1) The parallelized BABP starts with a p-way random edge partition over G, denoted by $C'(G) = \{F'_1, \cdots, F'_p\}$. Here we use F'_i to distinguish from output partitions of F_i. The parallel algorithm takes $C'(G)$ as input, with each fragment F'_i assigned to a worker. The algorithm first expands each F'_i into a butterfly-complete subgraph $F_i'^+$ using Algo. 10.

2) The auxiliary structures of sup$(\cdot)$ and $H(\cdot)$ at line 4 of Algo. 6 are computed on each $F_i'^+$ locally and independently.

3) The set S_i at line 3 is divided into p different parts $S_{i,j}$, for j in $[1, p]$. Here $S_{i,j}$ is maintained on worker-j, and covers the subset of S_i that falls in F'_j.

4) The loop at lines 6-13 is executed on each worker in parallel. The information of B_i and $B_{\max}$ is synchronized globally before each iteration, hence the precondition at line 8 can be decided by each worker uniformly. To select the edge for F_i (line 10), an edge $e_{i,j}$ with the maximum gain in $S_{i,j}$ is first picked at worker-j, for each j in $[1, k]$. Then they are aggregated into e_i at worker-i using min as the aggregation operator. Each selected e_i is then broadcast to all workers, and line 13 is performed on each worker locally.

Cost of parallel BABP. The cost of parallel BABP on worker-i consists of the following two parts. 1) The cost of selecting local maximum edge $e_{i,j}$ and updating corresponding auxiliary structures upon edge assignment. These cost are all incurred by local edges in the initial fragment F'_i at worker i. Similar with its sequential counterpart, the cost is in $O(\log(|F'_i|)\sum_{(u,v)\in F'_i}\min(d(u), d(v)))$. 2) The cost of aggregating $e_{i,j}$ into e_i at worker-i. The aggregation can be performed in $O(\log p)$ for each edges in the output fragment F_i. Hence, this cost is in $O(\log(p)|F_i|)$.

Putting this together, we can see the parallel BABP is in $O(\max_{i\in[1,k]} \log(p)|F_i| + \log(|F'_i|)\sum_{(u,v)\in F'_i}\min(d(u), d(v)))$. In practice, when the number of partition p increases, both $\max_{i\in[1,k]} |F'_i|$ and $\max_{i\in[1,k]} |F_i|$ decreases. Whereas $\log(p)$ grows very slowly and can be treated as a constant. That is, the parallel BABP runs faster with more processors.

F Hierarchical Partitioning

Estimating Bitruss Number

The accuracy of the estimation on t_i by EstimateBitruNum plays a key role in the efficiency of the HierarchPart. An inaccurate estimation will lead to large workload of $\max_{i\in[1,p]} |\bowtie_{F_i^B}|$ for the initial fragments F_i at line 5. This in turn, will cause most of the edges peeled in the level-wise fashion when enforcing the balance constraints at lines 6-8. Towards this, we first give a lower bound of ϕ_e and then show how to estimate ϕ_e on a Γ_{t_i} of G.

We denote by η_e a lower bound of ϕ_e for an edge $e = (u, v)$, defined as follows

$$\eta_e = \max(\max_{{}_v\angle_u^w\in\mathbb{W}_G}\{|H(u,w)|\}, \max_{{}_u\angle_w^v\in\mathbb{W}_G}\{|H(w,v)|\}) - 1 \quad (5)$$

Here H is the index over G.

Lemma 10.1. *$\eta_e \le \phi_e$ for each e in G.*

Corollary 10.2. *For each $e \in \Gamma_k$, $\max(\eta_e, k) \le \phi_e \le |\bowtie_{e,\Gamma_k}|$.*

According to Lemma 10.1, we can narrow the estimation of ϕ_e into a sub-interval of bitruss numbers, shown in Corollary 10.2. Based on this range, we estimate ϕ_e over subgraph Γ_k as follows

$$\varphi_e = \alpha\max(\eta_e, k) + (1-\alpha)|\bowtie_{e,\Gamma_k}| \quad (6)$$

The estimation of ϕ_e, denoted as φ_e, is a bitruss number within the estimated range of Corollary 10.2. Here α is a parameter within range $(0, 1)$ (set to 0.5 in this paper).

With the estimation of φ_e, the function EstimateBitruNum estimates t_i over $\Gamma_{t_{i-1}}$ as:

$$t_i = \max_t\{t \mid \sum_{t_{i-1}\le j<t}\sum_{\varphi_e=j} sup(e) \le B_{\max}\} \quad (7)$$

That is, t_i is selected as the maximum bitruss number, such that the number of butterflies incurred by edges with $\varphi_e < t_i$ is less or equal to $B_{\max}$.

Through the computation of each t_i, the parameter α is also dynamically adjusted. More specifically, when the estimation of the last t_{i-1} leads to an overloaded F_{i-1} such that $|\bowtie_{F_{i-1}}| > B_{\max}$, we take in too many edges for F_{i-1}. This means the estimation of φ_e is on average lower than the actual bitruss number ϕ_e, and we increase φ_e by shifting it to the upper bound with decreased α. The parameter α is decreased to $\gamma\times\alpha$, where $\gamma\in(0,1)$ is a constant (set to 0.5 in the paper). On the contrary, when $|\bowtie_{F_{i-1}}| < B_{\max}$, α is increased to $\max(\alpha/\gamma, 1)$, so as to lower the estimation of φ_e.

Parallelization

To make HierarchPart parallel, we follow similar parallelization of SC-PBD. That is, we start with a p-way edge partition $C(G) = \{F'_1, \cdots, F'_p\}$, and expand them into butterfly-complete subgraphs first. Then the index H_i and edge support $sup(\cdot)$ at line 1 can be initialized locally without communication. Since the peeling at lines 5 and 8 are essentially revisions of SubPeel,

Algorithm 12 Delta-based Peeling on F_i^+

```
Input: Fragment $F_i^+$, edge set $Q$, $sup(\cdot)$ and index $H_i$
Output: Updated $sup(\cdot)$ and index $H_i$ after peeling $Q$ from $F_i^+$
 1: function DeltaSubPeel($Q$, $sup(\cdot)$, $H_i$)
 2:     Let $\Delta H_i(u,w) = \{{}_v\angle_u^w \in \mathbb{W}_{F_i^+} \mid (u,v) \in Q \vee (w,v) \in Q\}$
 3:     for all pairs of $(u, w)$ such that $\Delta H_i(u,w) \neq \emptyset$ do
 4:         for all ${}_v\angle_u^w \in \Delta H_i(u,w)$ do
 5:             if $(u,v) \notin Q$ then
 6:                 $sup(w,v) \leftarrow sup(w,v) - (|H_i(u,w)| - 1)$
 7:             else if $(w,v) \notin Q$ then
 8:                 $sup(u,v) \leftarrow sup(u,v) - (|H_i(u,w)| - 1)$
 9:         $H_i(u,w) \leftarrow H_i(u,w) \setminus \Delta H_i(u,w)$
10:         for all ${}_v\angle_u^w \in H_i(u,w)$ do
11:             $sup(u,v) \leftarrow sup(u,v) - |\Delta H_i(u,w)|$
12:             $sup(w,v) \leftarrow sup(w,v) - |\Delta H_i(u,w)|$
13:     Remove $Q$ from $F_i^+$
14:     return updated $sup(\cdot)$ and $H_i$
```

they can be parallelized in the same way as SC-PBD. As for function EstimateBitruNum, the information of φ_e for each edge e is updated locally at each fragment F_i first. Then the sum of $\sum_{\varphi_e=t} sup(e)$ is collected locally at each fragment and aggregated at the coordinator. With the collected information, the coordinator computes t_i according to Equation 6.

Cost of HierarchPart. The cost of parallel HierarchPart consists of the following: 1) the cost of initializing indexes and butterfly support $sup(\cdot)$, 2) the cost of butterfly peeling at lines 5 and 8, and 3) the cost of function EstimateBitruNum. Since the peeling is parallelized in the same way as SC-PBD, we can see 1) is in $O(\max_{i\in[1,p]} |\mathbb{W}_{F_i^+}|)$ and 2) is in $O(\max_{i\in[1,p]} |\bowtie_{F_i', F_i'^+}|)$. As for 3), it takes $O(\max_{i\in[1,p]} |F_i'|)$ and $O(\phi_{\max})$ for collecting the information of $\sum_{\varphi_e=t} sup(e)$ on each worker and computing t_i on coordinator, respectively. Put 1)-3) together, and note that $(\max_{i\in[1,p]} |F_i'|+\phi_{\max}) = o(\max_{i\in[1,p]} |\bowtie_{F_i', F_i'^+}|)$, we can see the parallel HierarchPart is in $O(\max_{i\in[1,p]} |\mathbb{W}_{F_i^+}| + \max_{i\in[1,p]} |\bowtie_{F_i', F_i'^+}|)$.

G Optimizations for Divide

Recounting Butterfly

To identify situations where Recount is the faster approach, we employ the following boolean function, defined as:

$$f(F_i^+, Q) = \sum_{e\in Q} sup(e) > \beta \times \sum_{(u,v)\in F_i^+ \setminus Q} \min(d_i'(u), d_i'(v))$$

here $d_i'()$ denotes the degree in the updated graph of $F_i^+ \setminus Q$, and β is a positive real number serving as the scaling factor depending on the machine hardware. The function $f(F_i^+, Q)$ is true when $\sum_{e\in Q} sup(e)$, *i.e.,* the cost of enumerating butterflies associated with edges in Q, is larger. This indicates that Recount is the faster method, and hence we switch to it. Otherwise, we use Peel. The function f can be computed by a linear scan of edges in F_i^+. With the efficiently estimated cost, the effectiveness of this optimization is guaranteed.

Delta-based Batch Peeling

To speed up the computation, we avoid explicitly enumerating the butterflies by accumulating the changes to $sup(\cdot)$ in a delta index first. We denote the collection of delta wedges with endpoints u, v as $\Delta H_i(u,v)$. It can be defined as wedges in $\mathbb{W}_{F_i^+}$, where at least one of the edges is included in edge set Q. After collecting the set of $\Delta H_i(u,v)$ by scanning wedges associated with Q, we can update the associated $sup()$ for remaining edges in the following steps:

(1) For edge $(u,w) \in F_i^+ \setminus Q$, such that ${}_w\angle_u^v \in \Delta H_i(u,v)$, $sup(u,w)$ is decreased by $|H_i(u,v)| - 1$, as the wedge is destroyed, and all the corresponding butterflies formed with ${}_w\angle_u^v$ and other wedges in $H_i(u,v)$ are to be removed.

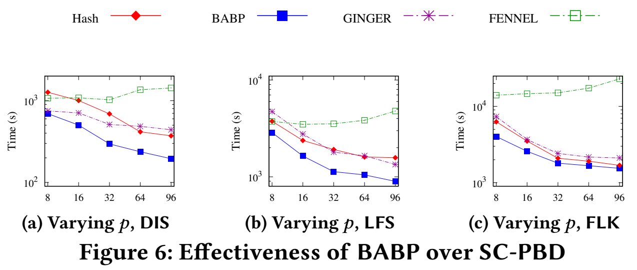
(2) For each wedge ${}_w\angle_u^v \in H_i(u,v) \setminus \Delta H_i(u,v)$, *i.e.,* the wedges that are not destroyed, both $sup(u,w)$ and $sup(v,w)$ are decreased by $|\Delta H_i(u,v)|$, as it no longer forms butterflies with wedges in $\Delta H_i(u,v)$.

The delta-based peeling algorithm, denoted as DeltaSubPeel, is shown in Algo. 12. Over F_i^+, the algorithm takes as input a set of edges Q to peel, together with edge support $sup(\cdot)$ and the local index H_i. The algorithm first computes the set of wedges $\mathbb{W}_{F_i^+}$ that are destroyed due to the deletion of edges in Q, and stores them in a delta index of ΔH_i (line 2). Then, the algorithm updates $sup(\cdot)$ associated with ΔH_i (lines 3-12). More specifically, 1) for wedges ${}_v\angle_u^w$ with only one edge in Q, the support of the other remaining edge is updated by removing all the butterflies it shared with other wedges in $H(u,w)$ (lines 4-8), and 2) for the remaining wedges in $H_i(u,w) \setminus \Delta H_i(u,w)$, the support for both edges of the wedge is updated by removing the destroyed butterflies formed with the wedge and others in $\Delta H_i(u,w)$ (lines 10-12).

The speed up of DeltaSubPeel is evident: the support of each affected edge is either decreased in batch by $|H_i(u,w)| - 1$ or by $|\Delta H_i(u,w)|$, whereas explicitly enumerating butterflies incurs only a unit decrement of butterfly count each time.

H More Experiment on Partitions

In addition to the experiments conducted in Sec 8, here we carry out more experiments to verify the effectiveness of our BABP partitioner. We test BABP against (1) the random hash edge-partition, denoted as Hash; (2) FENNEL[33], a stream partitioner under the vertex-partition model; (3) GINGER[9], a hybrid stream partitioner employed by PowerLyra. FENNEL is a sequential stream partitioner that partitions a vertex stream based on a greedy heuristic method. GINGER takes the heuristics of FENNEL for partitioning low-degree vertices, while separating edges of high-degree vertices into different partitions for a more balanced workload.



(a) Varying p, DIS (b) Varying p, LFS (c) Varying p, FLK

Figure 6: Effectiveness of BABP over SC-PBD

Response Time

Varying the number of partition p from 8 to 96, the impact of different partitions over the response time of SC-PBD is shown in Figure 6. Here SC-PBD is equipped with local index, and the time for partitioning and indexing have been accounted for the response time. We find the following. (1) BABP consistently beats other partitioners. Over all 3 tested datasets, it is on average 1.6, 1.6

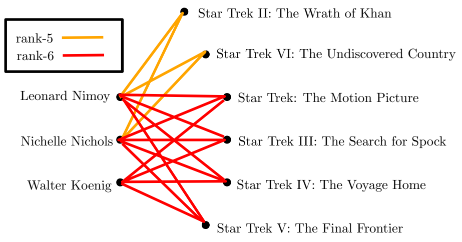

rank-5
rank-6
Leonard Nimoy
Nichelle Nichols
Walter Koenig
Star Trek II: The Wrath of Khan
Star Trek VI: The Undiscovered Country
Star Trek: The Motion Picture
Star Trek III: The Search for Spock
Star Trek IV: The Voyage Home
Star Trek V: The Final Frontier

Figure 7: A case study on IMDb movie-actor network

Table 3: Performance Evaluation of Partitioners (DIS, $p = 96$)

Algo		Hash	BABP	GINGER	FENNEL
	Normalized Max Size	4.01	2.02	5.36	16.1
	Partition Time (s)	0	1.02	1.67	1.59
SC-HBD	Eval. Time (s)	2917.96	2157.39	5631.98	15687.28
	Comm. Time (s)	0.96	0.69	0.62	0.59
	Respond Time (s)	2918.91	2159.1	5634.27	15689.46
SC-PBD	Eval. Time (s)	348.81	170.88	417.89	1412.16
	Comm. Time (s)	22.87	23.19	20.67	17.04
	Respond Time (s)	371.69	195.09	440.24	1430.79
Divide	Eval. Time (s)	39.59	27.77	36.54	42.29
	Comm. Time (s)	1.56	1.33	0.75	0.65
	Respond Time (s)	41.15	30.13	38.97	44.53

and 5.2 times faster than Hash, GINGER and FENNEL, respectively. Indeed, only BABP balances the actual workload incurred by SC-PBD over the partitions, while other partitioners balance either the number of edges or the number of vertices. (2) Over the partitions of Hash, BABP and GINGER, SC-PBD takes less time with larger numbers of partitions, as expected. (3) Over FENNEL partitions, SC-PBD takes a significantly longer time to finish. The response time rises with a larger p. This is because under vertex cut mode, FENNEL only balances the number of vertices of different partitions. In real-life bipartite graphs $G(V(U,L),E)$, there are often "hub vertices", *i.e.,* vertices in U (or L), with a large degree that connects a large portion of vertices in L (or U). For instance, in the dataset DIS, a single "hub vertices" in U connects to more than 13% of vertices in L. These "hub vertices" incur a large number of butterflies to enumerate for SC-PBD. Worse still, these "hub vertices" might be assigned to the same partition by FENNEL, as they all incur a unit cost under the vertex partitioning model. This leads to a more severe skewness. The impact of partitioners over the response time of SC-HBD and Divide are similar (not shown). These verify the effectiveness of our BABP.

Skewness and Communication Cost

In Table 3 we report a detailed analysis of partitioners, over DIS, with p fixed as 96. The results are as follows. (1) The response time of all 3 decomposition algorithms directly correlates with the skewness of the partition. The normalized maximum size of BABP is as small as 2.02, meaning that the largest fragment contains $\frac{2.02}{p}|\bowtie_G|$ butterflies to enumerate for the decomposition algorithms. In contrast, FENNEL has a normalized maximum size of 16.1, as it is designed to balance the number of vertices rather than number of butterflies across partitions. (2) The partition time takes only a small portion of the response time, *i.e.,* the time consumed by BABP is less than 0.05%, 0.5% and 3.4% for SC-HBD, SC-PBD and Divide, respectively. (3) The parallel computation cost for all 3 algorithms are dominated by the evaluation time for local computations. The communication cost only takes a small fraction of the total response time: it is less than 0.04%, 12% and 4.4% of the total time for SC-HBD, SC-PBD and Divide, respectively. Consequently, although partitioners like GINGER and FENNEL do lower the communication cost, they do not help with the overall efficiency due to the skewness of their workload distribution.

I Case Study

In addition to the case studies shown in previous work [25, 38], here we present another scenario to demonstrate the use of bitruss decompositions. We use the bipartite network of movie-actors extracted from IMDb [1].The bipartite graph consists of 4917 movies and 6255 actors. An edge linking one actor to a movie indicates the actor starred in the movie. In Fig 7, we show a bitruss subgraph found in the dataset. The edges colored in red and orange have bitruss rankings of 6 and 5, respectively. Note that actor Walter Koenig actually did star in Star Trek II and VI. However, since the datasets of IMDb only records 3 actors for each film, the corresponding edges linking Walter Koenig to these two movies are missing in the dataset. Despite the incomplete information, we can see the bitruss decomposition can still identify the movie franchise of the Star Trek series, together with the associated main actors including Walter Koenig, Nichelle Nichols and Leonard Nimoy. Note that we only use the topological information to find this cohesive subgraph formed by movie series and the corresponding main actors.

Comparing with other cohesive subgraph structures such as bicliques, we can see: (1) The Star Trek franchise can not be identified by $(3,6)$-biclque, due to the missing links on Walter Koenig. That is, bitruss subgraphs are more robust than bicliques, as they can tolerate a certain degree of incompleteness in graph topology. (2) Bitruss decomposition can identify nested cohesive subgraph structures with different levels of granularity.

J Proofs

Proof of Lemma 2.1

Proof. Let $A = \max_{G' \subseteq G, e \in G'} \delta(G')$. By definition of bitruss number, $\phi(e) = \delta(\Gamma_{\phi_e})$, we prove by showing $\delta(\Gamma_{\phi_e}) \geq A$ and $\delta(\Gamma_{\phi_e}) \leq A$:

- $\delta(\Gamma_{\phi_e}) \leq A$ since $e \in \Gamma_{\phi_e}$ and Γ_k is a subgraph of G;
- $\delta(\Gamma_{\phi_e}) \geq A$: Consider an arbitrary graph $G' \subseteq G$ and $e \in G'$, if $\delta(G') > \delta(\Gamma_{\phi_e})$, then $\phi_e > \delta(\Gamma_{\phi_e})$, contradicting that $\phi_e = \delta(\Gamma_{\phi_e})$. Therefore, $\delta(G') \leq \delta(\Gamma_{\phi_e})$. Since G' is chosen arbitrarily, then $\delta(\Gamma_{\phi_e}) \geq A$.

□

Proof of Lemma 3.1

Proof. We prove by induction. For the base case, $\gamma^{(0)}(e) = |\bowtie_{e,G}|$, then $\gamma^{(1)}(e) = \mathcal{H}(N_e) \leq |N_e| = |\bowtie_{e,G}| = \gamma^{(0)}(e)$ by the . Now assume $\forall e \in E, \gamma^{(i)}(e) \leq \gamma^{(i-1)}(e)$. Then in $i+1$-th iteration, for any e and $\bowtie$ which e takes part in, $\rho^{(i+1)}_{e,\bowtie} = \min_{e' \in \bowtie, e' \neq e} \gamma^{(i)}(e') \leq \min_{e' \in \bowtie, e' \neq e} \gamma^{(i-1)}(e') = \rho^{(i)}(e, \bowtie)$. Consequently, each element in $N_e^{(i+1)}$ has a value no greater than its previous iteration, then $\mathcal{H}(N_e^{(i+1)}) \leq \mathcal{H}(N_e^{(i)})$ since $\mathcal{H}(\cdot)$ is monotonic. Then $\gamma^{(i+1)}(e) \leq \gamma^{(i)}(e)$. □

Proof of Lemma 3.2

Proof. We first show by induction that $\forall G' \subseteq G, \forall e \in G'$, $\gamma^{(i)}(e) \geq \delta(G')$. For the base case, $\gamma^{(0)}(e) = |\bowtie_{e,G}| \geq |\bowtie_{e,G'}| \geq \delta(G')$. Now assume $\forall e \in G', \gamma^{(i)}(e) \geq \delta(G')$ at any i-th iteration where $i > 1$. Consider $(i+1)$-th iteration, since $\bowtie_{e,G'} \subseteq \bowtie_{e,G}$, then for each $\bowtie \in \bowtie_{e,G'}$, $\rho(e, \bowtie) = \min_{e' \in \bowtie, e' \neq e} \gamma^{(i-1)}(e') \geq \delta(G')$ by the induction hypothesis. Besides, $|\bowtie_{e,G'}| \geq \delta(G')$, thus $N_e^{(i+1)}$ has at least $\delta(G')$ elements whose value is at least $\delta(G')$. Then $\gamma^{(i+1)}(e) = \mathcal{H}(N_e^{(i+1)}) \geq \delta(G')$. Furthermore, since G' is chosen arbitrarily, by Lemma 2.1, $\gamma^{(i)}(e) \geq \max_{G' \subseteq G, e \in G'} \delta(G') = \phi_e$ for any $i \geq 0$. □

Proof of Theorem 3.3

Proof. Suppose at t-th iteration, $\forall e \in E(G), \gamma^{(t)}(e) = \gamma^{(t-1)}(e)$ (convergence). We show $\gamma^{(t)}(e) = \phi_e$ for all edges. For any specific e, where $\gamma^{(t)}(e) = b$, consider constructing the following subgraph G' from G: edges of G' are $\{e : e \in G \wedge \gamma^{(t)}(e) \geq b\}$. By the computation of ρ and $\mathcal{H}$ function, each edge in G' is contained in at least b such butterflies $\bowtie$ in G, that each $e' \in \bowtie$ satisfies $\gamma^{(t)}(e') \geq b$, and thus such $\bowtie$ also belongs to G'. Consequently, all edges in G' are contained at least b butterflies in G' ($\delta(G') \geq b$). By Lemma 2.1, $\phi_e \geq \delta(G') \geq b = \gamma^{(t)}(e)$. At the same time, $\gamma^{(t)}(e) \geq \phi_e$ (Lemma 3.2), thus $\phi_e = \gamma^{(t)}(e)$. □

Proof of Theorem 4.4

Proof. We prove by induction. For the base case, when $i = 0$, $\forall e \in S^{(0)}, \phi_e = |\bowtie_{e,G}| = \gamma^{(0)}(e)$. Now assume for $i > 0$, for any $e \in S^{(0)} \cup \cdots \cup S^{(i)}$, $\gamma^{(i)}(e) = \phi_e$. Then consider any edge $e \in S^{(i+1)}$, for the computation of $\gamma^{(i+1)}(e)$, we first analyse the butterflies which contain e and then the computation of $\mathcal{H}(N_e^{(i+1)})$. Let G' be the subgraph induced by $S^{(i+1)} \cup S^{(i+2)} \cup \cdots$, for any $e \in S^{(i+1)}$, $\bowtie_{e,G}$ can be divided in two non-overlapping subsets $\mathcal{B}_1 = \{\bowtie : \forall e \in \bowtie, e \in G'\}$ and $\mathcal{B}_2 = \{\bowtie : \exists e \in \bowtie, e \notin G'\}$. To compute $\mathcal{H}(N_e^{(i+1)})$, for any $\bowtie \in \bowtie_{e,G}$, there are two cases:

1. $\bowtie \in \mathcal{B}_1$. Because e is peeled at this iteration, then $|\mathcal{B}_1| = |\bowtie_{e,G'}| \leq MS = \phi_e$, by the execution of lines 5-5 of Algo. 3.
2. $\bowtie \in \mathcal{B}_2$. Then there exists at least one e' such that $e' \in \bowtie$ peeled in previous iterations. By Corollary 4.3, $\phi_{e'} \leq \phi_e$. By the induction hypothesis $\gamma^{(i+1)}(e')$ has already converged, *i.e.,* $\phi_{e'} = \gamma^{(i)}(e')$. Therefore, $\rho(e, \bowtie) \leq \phi_e$.

Since $|\mathcal{B}_1| \leq \phi_e$ and $\forall \bowtie \in \mathcal{B}_2, \rho(e, \bowtie) \leq \phi_e$, then $\gamma^{(i+1)}(e) = \mathcal{H}(N_e^{(i+1)}) \leq \phi_e$ by definition of H-function. Meanwhile, $\gamma^{(i+1)}(e) \geq \phi_e$ by Lemma 3.2, then $\gamma^{(i+1)}(e) = \phi_e$, indicating $\gamma^{(\cdot)}(e)$ has already converged within $i+1$ iterations. □

Proof of Theorem 4.2

Proof. We proof by induction. When $i = 0$, $S^{(0)}$ contains those edges with the minimum support $MS^{(0)}$ in G. For any $e \in S^{(0)}$, by Definition 2.3, $\phi_e \geq MS^{(0)}$. On the other hand, for any $G' \subseteq G$ such that $e \in G'$, $|\bowtie_{e,G'}| \leq |\bowtie_{e,G}| = MS^{(0)}$, then $\delta(G') \leq |\bowtie_{e,G'}| = MS^{(0)}$. According to Lemma 2.1, $\phi_e = \max_{G' \subseteq G, e \in G'} \delta(G') \leq MS^{(0)}$, and $\phi_e \leq MS^{(0)} \wedge \phi_e \geq MS^{(0)} \rightarrow \phi_e = MS^{(0)}$. Assume $\forall e_l \in S^{(l)}, \phi_{e_l} = MS^{(l)}$ holds for any $l \in [0, i]$ ($i > 0$). Now consider iteration $i+1$, let e_{i+1} be an arbitrary edge in $S^{(i+1)}$. We prove by showing $\phi_{e_{i+1}} \geq MB^{(i+1)}$ and $\phi_{e_{i+1}} \leq MB^{(i+1)}$. Let $j \in [1, i+1]$ be the first iteration such that $MS^{(j)} = MS^{(i+1)}$, consider the subgraph $L^{(j)}$, since $\delta(L^{(j)}) = MS^{(i+1)}$ and $e_{i+1} \in L^{(j)}$, we get $\phi_{e_{i+1}} \geq MS^{(i+1)}$ by Lemma 2.1. Next, we prove $\phi_{e_{i+1}} \leq MS^{(i+1)}$. To show this, we first show that if $j \geq 1$, for $l \in [0, j-1]$, $S^{(l)} \cap \Gamma_{\phi_{e_{i+1}}} = \emptyset$. If $\Gamma_{\phi_{e_{i+1}}}$ contains any $e_l \in S^{(l)}$, then $\delta(\Gamma_{\phi_{e_{i+1}}}) \leq \phi_{e_l} = MS^{(l)} < MS^{(i+1)}$ by Lemma 2.1 and induction hypothesis. At the same time, $\phi_{e_{i+1}} = \delta(\Gamma_{\phi_{e_{i+1}}}) \geq \delta(L^{(j)}) = MS^{(i+1)}$ by Lemma 2.1, which leads to contradiction. We then conclude $\Gamma_{\phi_{e_{i+1}}} \subseteq L^{(j)}$. Furthermore, there are two cases for $\Gamma_{\phi_{e_{i+1}}}$:

1. $\Gamma_{\phi_{e_{i+1}}} \subseteq L^{(i+1)}$, then by bitruss definition, $\phi_{e_{i+1}} = \delta(\Gamma_{\phi_{e_{i+1}}}) \leq |\bowtie_{e_{i+1}, \Gamma_{\phi_{e_{i+1}}}}|$. Since $\Gamma_{\phi_{e_{i+1}}} \subseteq L^{(i+1)}$ and e_{i+1} is removed in $i+1$-th iteration, we have $|\bowtie_{e_{i+1}, \Gamma_{\phi_{e_{i+1}}}}| \leq |\bowtie_{e_{i+1}, L^{(i+1)}}| \leq MS^{(i+1)}$. Consequently, $\phi_{e_{i+1}} \leq MS^{(i+1)}$;
2. $\Gamma_{\phi_{e_{i+1}}} \not\subseteq L^{(i+1)}$, then $\Gamma_{\phi_{e_{i+1}}}$ contains at least one edge e_l from $S^{(j)} \cup \cdots \cup S^{(i)}$. Suppose $e_l \in S^{(l)}$ for some $l \in [j, i]$. During iteration j to $i+1$, MS is unchanged. Then $\phi_{e_l} = MS^{(l)} = MS^{(i+1)}$ by induction hypothesis. Then by Lemma 2.1, $\phi_{e_{i+1}} = \delta(\Gamma_{\phi_{e_{i+1}}}) \leq \phi_{e_l} = MS^{(i+1)}$.

In both cases, $\phi_{e_{i+1}} \leq MS^{(i+1)}$, and our proof is complete. □

Proof of Theorem 4.5

Proof. When Algo. 4 terminates, $\forall e \in F_i, |\bowtie_{e,F_i^+}| > k$, this is guaranteed by consecutive peeling (line 16). After running SubPeel, for any butterfly $\bowtie \in \bowtie_{e,F_i^+}$, there are two cases:

- $\forall e' \in \bowtie$, e' is an inner edge. Since e' is not peeled locally on F_i, e' is reserved globally, *i.e.,* $e' \in G$, therefore $\bowtie \in G$;
- $\exists e' \in \bowtie$, e' is an external edge. Since e' is reserved on F_i^+, e' is reserved on some other fragment F_j^+ as an inner edge, then $e' \in G$ and $\bowtie \in G$.

Therefore, for any remaining inner edge $e \in F_i$, $|\bowtie_{e,G}| > k$, then $\delta(G) > k$. According to Lemma 2.1, $\phi_e > k$. □

Proof of Lemma 5.2

Proof. Consider a butterfly ${}^{u}_{v}\!\bowtie^{w}_{x}$ in $\bowtie_{F_i, F_i^+}$. Suppose $p_i(u) = \min\{p_i(u), p_i(v), p_i(w), p_i(x)\}$, then by definition of $\bowtie_i$, we can see ${}^{u}_{v}\!\bowtie^{w}_{x} \in \bowtie_i(u, w)$ and ${}^{u}_{v}\!\bowtie^{w}_{x} \notin \bowtie_i(v, x)$. That is, each butterfly in $\bowtie_{F_i, F_i^+}$ belongs to one and only one butterfly set of $\bowtie_i(u, w)$. □

Proof of Lemma 5.3

Proof. Denote by $\|{}_{v}\angle^{w}_{u}\|_{C(G)}$ the number of replicates of wedge ${}_{v}\angle^{w}_{u}$ in all F_i^+ induced by partition $C(G) = \{F_1, \cdots, F_p\}$. Then we can see over the pruned index, for each ${}_{v}\angle^{w}_{u}$ in $\mathbb{W}_G$,

$$\|{}_{v}\angle^{w}_{u}\|_{C(G)} = \begin{cases} 1, & (u,v) \text{ and } (w,v) \text{ are in the same } F_i \\ 2, & \text{otherwise} \end{cases}$$

That is, an inner wedge is at most replicated in two different fragments. Consequently,

$$\sum_{i \in [1,p]} |\mathbb{W}_{F_i^+}| = \sum_{{}_{v}\angle^{w}_{u} \in \mathbb{W}_G} \|{}_{v}\angle^{w}_{u}\|_{C(G)} \leq \sum_{{}_{v}\angle^{w}_{u} \in \mathbb{W}_G} 2 = 2|\mathbb{W}_G|$$

□

Proof of Theorem 6.1

Proof. The decision version of BABGP is BABGP-D:

- Input: $G(V,E), p, \epsilon, T$;
- Output: decides whether or not there exists such an $C(G)$, that for $\forall i \in [1,p]$, $|\bowtie_{F_i,F_i^+}| \le \epsilon \cdot B$ and $t \le T$.

Clearly, BABGP-D is in NP. The certificate is an $C(G)$, and checking whether $\forall i \in [1,p]$, $|\bowtie_{F_i,F_i^+}| \le \epsilon \cdot B$ and $t \le T$ can be done in polynomial time. Next, we show BABGP-D is NP-complete by reducing from 3-Partition. Given a multiset of positive integers $S = \{a_1, \cdots, a_n\}$ where $n = 3k$, $\sum_{i=1}^{n} a_i = kA$, and for $\forall i \in [1,n], a_i \in (\frac{A}{4}, \frac{A}{2})$, 3-Partition decides whether or not there exists a partition of S into k triplets $S_1, \cdots, S_k$ such that the sum of the numbers in each S_i is A. 3-Partition is strongly NP-complete, since it remains NP-complete when all numbers (a_i, A) are polynomially bounded. Given an instance Π_1 of 3-Partition. First, for each $a_i \in S$, we create an isolated bipartite graph $F_i(V_i, E_i)$, where $U(F_i) = \{x_0, \cdots, x_{a_i}\}, L(F_i) = \{y_0, \cdots, y_{a_i}\}$, and $E_i = \{(x_0, y_0)\} \cup \{(x_l, y_l), (x_l, y_{l-1}), (x_{l-1}, y_l) : l \in [1, a_i]\}$. It can be easily verified that each F_i is a single connected component of G. Therefore, $F_i = F_i^+$, and $|\bowtie_{F_i,F_i^+}| = |\bowtie_{F_i}| = |\bowtie_{F_i^+}| = a_i$. Next, we create an instance Π_2 of BABGP-D(G,k,1,0) by setting $G = F_i \cup \cdots \cup F_n, p = k, \epsilon = 1, T = 0$. We show $\Pi_1 \in$ 3-Partition iff $\Pi_2 \in$ BABGP-D:

- $\Rightarrow$: Suppose Π_1 is a "Yes" instance, then there exists a partition which contains k triplets and the sum of numbers in each triplet is A. Since each a_i has an one-to-one mapping to F_i. Consider such an $C(G)$ that partitions G by assigning F_i to the triplet a_i resides in, then for $\forall i \in [1,p]$, $|\bowtie_{F_i,F_i^+}| = |\bowtie_{F_i}| = A = \frac{kA}{k} = \frac{\sum_{i=1}^{p} |\bowtie_{F_i,F_i^+}|}{p}$. Besides, since each F_i is isolated, meaning that no butterfly is divided into different partitions, so $t = \sum_{i=1}^{p}(|F_i^+| - |F_i|) = 0$ and Π_2 is a "Yes" instance.
- $\Leftarrow$: Suppose Π_2 is a "Yes" instance, then there exists an $C(G)$ satisfies the constrains of Π_2. Since $0 \le t \le T = 0$, there is no butterfly that is divided across different partitions by $C(G)$, and $\sum_{i=1}^{p} |\bowtie_{F_i,F_i^+}| = \sum_{i=1}^{n} |\bowtie_{F_i}| = \sum_{i=1}^{n} a_i = kA$. At the same time, $\forall i \in [1,p], |\bowtie_{F_i,F_i^+}| \le \epsilon \frac{\sum_{i=1}^{p} |\bowtie_{F_i}|}{p} = \epsilon \cdot \frac{kA}{p} = A$. Therefore, $\forall i \in [1,p], |\bowtie_{F_i,F_i^+}| = A$. The fact $a_i \in (\frac{A}{4}, \frac{A}{2})$ leads to $|\bowtie_{F_i,F_i^+}| \in (\frac{A}{4}, \frac{A}{2})$, so each partition in $C(G)$ has exactly 3 connected components of G. Since each F_i has an one-to-one mapping to a_i, $C(G)$ derives a feasible partition for S, and Π_1 is a "Yes" instance.

□

Proof of Corollary 6.2

Proof. Given an instance Π_1 of 3-Partition, we can create an instance Π_3 of BABGP with the similar process of creating Π_2 in proof of Theorem 6.1. Instead of satisfying the threshold T for t in Π_2, Π_3 tries to find such an $C(G)$ to minimize t. Suppose there is an α-approximation algorithm $\mathcal{A}$ for BABGP $(G, p, 1)$. If Π_1 is a "Yes" instance, then OPT = 0. If Π_1 is a "No" instance, then there exists at least one butterfly is partitioned across different groups by $\mathcal{A}$ and OPT ≥ 1. Therefore, $\mathcal{A}$ can distinguish with these two cases, which contradicts with that 3-Partition is strongly NP-complete unless P=NP. □

Proof of Theorem 6.3

Proof. By definition of B_i, it is evident that $B_i \le \sum_{e \in F_i} |\bowtie_{e,G}|$. Next we prove $|\bowtie_{F_i,F_i^+}| \le B_i$. by showing the following:

$$\sum_{u,v \in V} \sigma_1(F_i, u, v) |_u\bowtie_v| \le |\bowtie^{(2)}_{F_i,F_i^+}| + |\bowtie^{(3)}_{F_i,F_i^+}| + |\bowtie^{(4)}_{F_i,F_i^+}| \quad (8)$$

$$\sum_{u,v \in V} \sigma_2(F_i, u, v) |_u\bowtie_v| \le |\bowtie^{(4)}_{F_i,F_i^+}| \quad (9)$$

Here $|\bowtie^{(j)}_{F_i,F_i^+}|$ denotes the set of butterflies in F_i^+ that consists of exact j edges in F_i, where $j \in [1,4]$. To see Equation 8 is correct, note that 1) butterfly sets $_u\bowtie_v$ with different pairs of (u,v) are disjoint and 2) for each of the counted sets with $\sigma_1(F_i, u, v) =$ true, the butterflies in the sets are all in $\bigcup_{j \ge 2} \bowtie^{(j)}_{F_i,F_i^+}$, since they all contain at least 2 edges in F_i. Putting this together, we have $\sum_{u,v \in V} \sigma_1(F_i, u, v) |_u\bowtie_v| \le |\bigcup_{j \ge 2} \bowtie^{(j)}_{F_i,F_i^+}|$ *i.e.,* Equation 8 holds. Similarly Equation 9 is true, because each of the counted sets with $\sigma_2(F_i, u, v) =$ true consist of butterflies with all 4 edges in F_i.

Next we replace the corresponding entry in Equation 1, we have

$$B_i \ge \sum_{e \in F_i} |\bowtie_{e,G}| - (|\bowtie^{(2)}_{F_i,F_i^+}| + |\bowtie^{(3)}_{F_i,F_i^+}| + 3|\bowtie^{(4)}_{F_i,F_i^+}|) \quad (10)$$

Note that

$$\sum_{e \in F_i} |\bowtie_{e,G}| = \sum_{j \in [1,4]} j \times |\bowtie^{(j)}_{F_i,F_i^+}| \quad (11)$$

Putting Equations 10 and 11 together, we can see

$$B_i \ge \sum_{j \in [1,4]} |\bowtie^{(j)}_{F_i,F_i^+}| + |\bowtie^{(3)}_{F_i,F_i^+}| \ge \sum_{j \in [1,4]} |\bowtie^{(j)}_{F_i,F_i^+}| = |\bowtie_{F_i,F_i^+}| \quad (12)$$

□

Proof of Theorem 7.1

Proof. By Lemma 2.1, since $F^B \subset G$, for any edge $e \in F$, we have $\max_{e \in G', G' \subset G} \delta(G') \ge \max_{e \in G', G' \subset F^B} \delta(G')$, *i.e.,* $\phi_e(G) \ge \phi_e(F^B)$. Next we prove $\phi_e(G) \le \phi_e(F^B)$ by contradiction. Suppose $\alpha = \phi_e(G) > \phi_e(F^B) = \beta \ge j$, then by Lemma 2.1, there exists some subgraph $G' \subset G$, such that 1) $e \in G'$; and 2) for all edges $e' \in G'$, $|\bowtie_{e',G'}| \ge \alpha > \beta \ge j$. Obviously there are only a finite set of such subgraphs G'. Denote by G^* the smallest one. Then we can see 1) $G^* \subset \Gamma_\alpha \subset \Gamma_j$ (by Definition 2.2); and 2) $G^* \setminus F^B \ne \emptyset$ (otherwise, $G^* \subset F^B$, and consequently by Lemma 2.1 $\alpha \le \phi_e(G^*) \le \phi_e(F^B) = \beta$, this contradicts to the assumption of $\alpha > \beta$). Now consider constructing a new subgraph $G^{**} = G^* \setminus F^B$, *i.e.,* we remove all edge in F^B from G^*. Then for any edge $e' \in G^{**}$, there is $|\bowtie_{e',G^{**}}| = |\bowtie_{e',G^*}|$. This is because by Definition 7.1 of F^B, edges outside of F^B in Γ_j do not form butterflies with edges in F^B. Consequently, we have a graph G^{**}, which satisfies the same condition of G^*, but with a smaller size $|G^{**}| < |G^*|$, as $G^* \setminus F^B \ne \emptyset$. This contradicts the fact that G^* is the smallest subgraph satisfying the given condition. □

Proof of Theorem 7.4

Proof. We prove by introducing a DP algorithm. First, we compute the bitruss number of each edge, taking $O(|\mathbb{W}_G| + |\bowtie_G|)$. Next, we build a bipartite graph G^+ where each node in upper/lower layer represents an edge e (a butterfly $\bowtie$), and there exists an edge between e and $\bowtie$ if e is contained in $\bowtie$. This can be done in $O(|\bowtie_G|)$.

Let $\Phi = [\phi_1, \cdots, \phi_z]$ be the sequence of z (bounded by $|E|$) distinct bitruss numbers of G with an increasing order, and E_{ϕ_i} be the set of edges with the bitruss number ϕ_i, and $c(\cdot,\cdot)$ be a cost function which maps any consecutive subsequence of Φ to a positive integer, *i.e.,* $c(i,j)$ is the number of butterflies in the bitruss butterfly-complete subgraph of $E(i,j) = E_{\phi_i} \cup \cdots \cup E_{\phi_j}$, *i.e.,* those edges which shared butterflies with $E(i,j)$ in Γ_{ϕ_i}. Note that $c(\cdot,\cdot)$ can be built in $O(z^2|\bowtie_G|)$ time given G^+. BHBP is equal to partition the ordered sequence Φ into p disjoint groups $S_1, \cdots, S_p$ with consecutive elements such that $\max_{i\in[1,p]} c(S(i))$ minimum, where $c(S(i)) = c(l,m)$ and ϕ_l and ϕ_m are the first and last elements of $S(i)$. The DP procedure is as follows. Let $M(z,p)$ be the minimum cost of partitioning the z elements of Φ into p groups, the transition equation is:

$$M(z,p) = \min_{1\le i\le z-1} \max\{M(i,p-1), c(i+1,z)\},$$

with the base cases of $M(i,1) = c(1,i)$ for $i \in [1,z]$ and $M(1,k) = c(1,1)$ for $k \in [1,p]$. Therefore, computing $M(z,p)$ requires $O(z^2p)$ time, and the optimal solution can be recovered by backtracking the process of computing $M(z,p)$. □

Proof of Lemma 10.1

Proof. To see that Lemma 10.1 is correct, we prove that: for an edge $e = (u,v)$, and a wedge ${}_v\angle_u^w$ (resp. ${}_u\angle_w^v$) in $\mathbb{W}_G$, $\phi_e \ge |H(u,w)| - 1$ (resp. $\phi_e \ge |H(w,v)| - 1$). Consider the subgraph G' formed by the collection of wedges in $H(u,w)$. We can verify that the subgraph is a biclique of K_x^2 with $x = |H(u,w)|$, since vertices u and w are connected to the pivots of all wedges in $H(u,w)$. Hence each of the edges e' in G' has the butterfly support of $|H(u,w)| - 1$, and $\phi_{e'}(G') = |H(u,w)| - 1$. By Lemma 2.1, we can see $\phi_e(G) \ge \phi_e(G')$, *i.e.,* $\phi_e \ge |H(u,w)| - 1$. In a similar way, we can prove $\phi_e \ge |H(w,v)| - 1$ for all wedges ${}_u\angle_w^v \in \mathbb{W}_G$. Put these together and we can see Lemma 10.1 is correct. □