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): 1889 - 1901, 2022.
doi:10.14778/3538598.3538610

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 [21] and k-truss [11, 25], 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 [23, 40]. 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 [34].

Recommendation System. 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 [27].

Anomaly Detection. 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 [4]. 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 [38]. This means the number of user-product relations grew extremely fast. According to [37], 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:10.14778/3538598.3538610

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 [23, 40] follow a similar sequential bottom-up peeling framework (details in [36]), *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. [34] 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 response 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. Proofs of the paper are in [36].

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 v and x, respectively, by edges (u, v), (u, x), (w, v) and (w, x).*

Denote by $\overset{u}{v}\bowtie\overset{w}{x}$ 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 [36].

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, [19] 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. [24] 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 [24], with the specialization to bitruss model.

Lemma 3.1 ([24]). *For all $i \geq 1$, $\forall e \in E, \gamma^{(i)}(e) \leq \gamma^{(i-1)}(e)$.*

Lemma 3.2 ([24]). *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 ([24]). *$\forall e \in E$, the sequence $\gamma^{(i)}(e)$ for $i \geq 0$ converges to ϕ_e.*

We show an upper bound for the number of iterations of H-BD in Sect. 4.1, which is tighter than [24]. Next, we introduce our distributed solution based on H-BD.

Algorithm 1 H-BD

Input: $G(V, E)$
Output: ϕ_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: $\quad i \leftarrow i+1$, $converge \leftarrow True$
5: $\quad$ **for all** $e \in E$ **in parallel do**
6: $\quad\quad N_e \leftarrow \emptyset$
7: $\quad\quad$ **for all** $\bowtie \in \bowtie_{e,G}$ **do**
8: $\quad\quad\quad \rho_{e,\bowtie} = \min_{e' \in \bowtie, e' \neq e} \gamma^{(i-1)}(e'),\ N_e \leftarrow N_e \cup \{\rho_{e,\bowtie}\}$
9: $\quad\quad \gamma^{(i)}(e) \leftarrow \mathcal{H}(N_e)$
10: $\quad\quad$ **if** $\gamma^{(i)}(e) \neq \gamma^{(i-1)}(e)$ **then** $converge \leftarrow False$
11: $\phi_e \leftarrow \gamma^{(i)}(e), \forall e \in E$

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 [36] 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).

Cost Analysis. 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 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

Algorithm 2 SC-HBD (Subgraph-Centric Decomposition)

Input: F_i^+, MSG_r
Output: ϕ_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

Algorithm 3 BatchPeel

Input: $G(V, E)$
Output: ϕ_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** ϕ_e for each e

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).

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;

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. □

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 [26], 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 ← ∅;
2: for all w such that (w, v) ∈ F_i^+ ∧ p(u) > p(w) do
3:     if (w, v) ∈ F_i ∨ (u, v) ∈ F_i then
4:         for all x∠_u^w ∈ H_i(u, w) such that x ≠ v do
5:             B_e ← B_e ∪ {v⋈x^w}
6:     else
7:         for all x∠_u^w ∈ H̄_i(u, w) do
8:             B_e ← B_e ∪ {v⋈x^w}
9: for all w such that (w, u) ∈ F_i^+ ∧ 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 [36]).

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

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

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 [36]. Note that current index such as the BE-Index [34] 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 [36].

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 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

Algorithm 6 BABP (Butterfly-Aware Balanced Partitioner)

Input: $G(V(U, L), E)$, a positive p, a user-defined threshold ϵ ($\epsilon > 1$)
Output: A p-way edge partitions $C(G) = \{F_1, F_2, \ldots, F_p\}$.

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;

$|\mathbb{W}_G| << |\bowtie_G|$ 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[10]. 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 [36].

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 [35], 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 $G \setminus \Gamma_t$. 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: ϕ_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: $\quad \Gamma_{t_i} \leftarrow$ 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: $\quad$ Construct subgraph F_i on worker-i
9: $\quad F_i^B \leftarrow$ the bitruss butterfly-complete graph of F_i in G
10: $\quad$ Initialize index $\bar{H}_i$ and butterfly support $sup(\cdot)$ over F_i^B
11: $\quad$ LocalPeel($sup(\cdot)$, $\bar{H}_i$, F_i^B)
12: **function** LocalPeel($sup(\cdot)$, $\bar{H}_i$, F_i^B)
13: $\quad$ **while** there exists some edge in F_i that is not peeled **do**
14: $\quad\quad k \leftarrow \min_{e \in F_i} sup(e)$
15: $\quad\quad$ **let** $Q = \{e \mid e \in F_i \wedge sup(e) = k\}$
16: $\quad\quad$ **while** $Q \neq \emptyset$ **do** Repeat lines 11-16 of Algo. 4
17: $\quad$ **return** ϕ_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 [36]) 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 ϕ_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: $\quad t_i \leftarrow$ EstimateBitruNum($\Gamma_{t_{i-1}}$, $B_{\max}$)
5: $\quad \Gamma_{t_i} \leftarrow k$-Bitruss($\Gamma_{t_{i-1}}$, t_i), $\quad F_i \leftarrow \Gamma_{t_i} \setminus \Gamma_{t_{i-1}}$
6: $\quad$ **while** $|\bowtie_{F_i}| > B_{\max}$ **do**
7: $\quad\quad t \leftarrow \min_{e\in F_i} sup(e)$
8: $\quad\quad$ 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 [36] 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 $|\bowtie_{F_i^B}| \le |\bowtie_G|/p$. 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 [36] 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

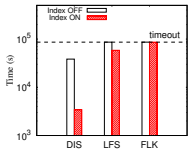
In function k-Bitruss (line 5 in Algo. 8), we are computing Γ_{t_i} from the current subgraph of $\Gamma_{t_{i-1}}$, hence it is overkill to directly plug in SC-PBD and compute ϕ_e for each peeled edge e in $\Gamma_{t_i} \setminus \Gamma_{t_{i-1}}$. To deal with this issue, we propose an optimization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

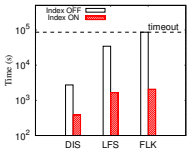
To see that this optimization works, observe that each time, $sup()$ is decreased by an accumulated delta count indicated by $\Delta H_i(u, v)$, instead of only 1. For instance, when set of edges to peel Q is the whole fragment F_i, peeling via explicitly enumerating butterflies will have to scan all $|\bowtie_{F_i,F_i^+}|$ 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

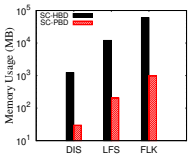
	Name	Abbr.	$\|G\|$	$\|\bowtie_G\|$	Network Type
Medium	Discog-lstyle	DIS	1.1×10^6	5.2×10^9	feature
	lasf.FM-song	LFS	4.4×10^6	3.2×10^{10}	interaction
	Flickr	FLK	8.5×10^6	3.5×10^{10}	affiliation
Large	Delicious	DEL	1.0×10^8	5.7×10^{10}	interaction
	Epinions	EPN	1.3×10^7	1.7×10^{11}	rating
	Jester150	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

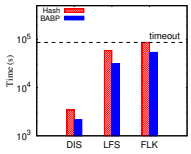


(b) Response Time of SC-PBD

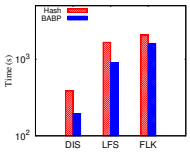


(c) Space Cost of Index

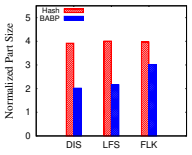
Figure 1: Effectiveness of Local Indexes



(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 [35], and the ParButterfly algorithm in [26]. Note that there are two BiT-BU methods described in [35], *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.

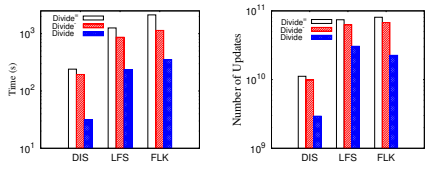
Impact on efficiency. Fixing $p = 96$, the total response time of SC-HBD (resp. SC-PBD) with and without local index is shown in Fig 1a (resp. Fig 1b). Note that the time for index construction is included in the total response time. We find that the local index significantly speeds up both SC-HBD and 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.

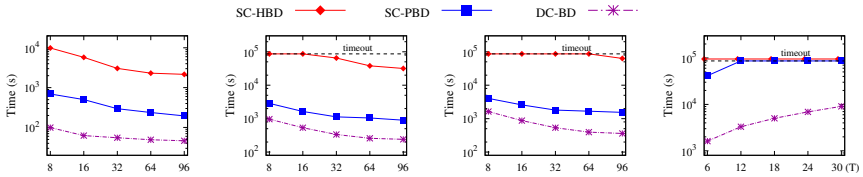
Impact on efficiency. Shown in Fig 2a-2b, we can see: (1) BABP boosts SC-HBD by 59% and 85% over DIS and LFS, respectively, compared with that running on hash partitions. (2) For SC-PBD, the gap of BABP and hash partitioning is $\geq 31\%$ over all three datasets. (3) The partitioning cost are included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

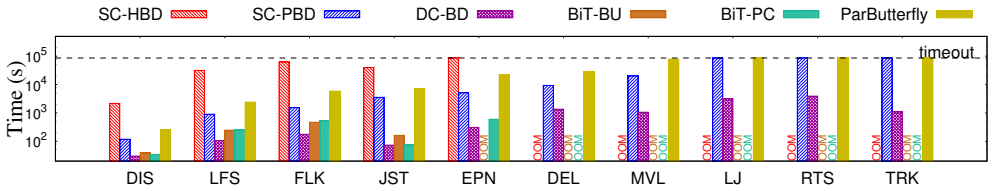


Figure 5: Comparison of Different Bitruss Decomposition Methods ($p = 96$, OOM denotes Out-Of-Memory Error)

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 $|\bowtie_G| \geq$ 12T 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 [15, 26, 35] on shared-memory machines. [26] 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, [35] extends [34] 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. [15] 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 [26], BE-Index in [15, 35]) 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 [18] is a maximal subgraph where each node in the upper/lower layer has at least α/β neighbors. Maximal biclique [1] and quasi-biclique [28] 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). [20] finds such a biclique in G whose (p, q)-biclique density is largest. A k-tip [23] 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 [5, 7]) are proposed for vertex-cut [6, 8, 14, 16, 22, 31] and edge-cut [2, 12, 13, 29, 30, 39]. Vertex-cut (resp. edge-cut) partitions edges (resp. vertices) into disjoint balanced subsets and reduce vertex (resp. edge) replication. As pointed out by [8, 33], 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. [9] and [8] bond vertex-cut with edge-cut by cutting high-degree nodes controlled by parameters. [17] 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 [3, 32]. [3] 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. [32] 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.

ACKNOWLEDGMENTS

Yue Wang is partially supported by China NSFC (No. 62002235) and Guangdong Basic and Applied Basic Research Foundation (No. 2019A1515110473). Ruiqi Xu is supported by the National Research Foundation, Singapore under its Strategic Capability Research Centres Funding Initiative. Lei Chen's work is partially supported by National Key Research and Development Program of China Grant No. 2018AAA0101100, the Hong Kong RGC GRF Project 16202218, CRF Project C6030-18G, C1031-18G, C5026-18G, CRF C2004-21GF, AOE Project AoE/E-603/18, RIF Project R6020-19, Theme-based project TRS T41-603/20R, China NSFC No. 61729201, Guangdong Basic and Applied Basic Research Foundation 2019B151530001, Hong Kong ITC ITF grants ITS/044/18FX and ITS/470/18FX, Microsoft Research Asia Collaborative Research Grant, HKUST-NAVER/LINE AI Lab, Didi-HKUST joint research lab, HKUST-Webank joint research lab grants and HKUST Global Strategic Partnership Fund (2021 SJTU-HKUST).

REFERENCES

[1] 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.

[2] 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.

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

[4] 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.

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

[6] Florian Bourse, Marc Lelarge, and Milan Vojnovic. 2014. Balanced graph edge partition. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*. 1456–1465.

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

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

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

[14] 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.

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

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] 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

[24] 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

[25] 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

[26] 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

[27] 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.

[28] 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.

[29] 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.

[30] 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.

[31] 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.

[32] 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.

[33] 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).

[34] 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

[35] 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.

[36] Yue Wang, Ruiqi Xu, Xun Jian, Alexander Zhou, and Lei Chen. 2022. Towards Distributed Bitruss Decomposition on Bipartite Graphs–Full Version. https://keithyue.github.io/files/vldb22-DBitruss.pdf [Online; accessed 5-May-2022].

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

[38] 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].

[39] 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.

[40] 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