I/O-Efficient Butterfly Counting at Scale

ZHIBIN WANG, Nanjing University, China
LONGBIN LAI, Alibaba Group, China
YIXUE LIU, Nanjing University, China
BING SHUI, Nanjing University, China
CHEN TIAN, Nanjing University, China
SHENG ZHONG, Nanjing University, China

Butterfly (a cyclic graph motif) counting is a fundamental task with many applications in graph analysis, which aims at computing the number of butterflies in a large graph. With the rapid growth of graph data, it is more and more challenging to do butterfly counting due to the super-linear time complexity and large memory consumption. In this paper, we study I/O-efficient algorithms for doing butterfly counting on hierarchical memory. Existing algorithms of the kind cannot guarantee I/O optimality. Observing that in order to count butterflies, it suffices to "witness" a subgraph instead of the whole structure, a new class of algorithms called semi-witnessing algorithm is proposed. We prove that a semi-witnessing algorithm is not restricted by the lower bound $\Omega(\frac{|E|^2}{MB})$ of a witnessing algorithm, and give a new bound of $\Omega(\min(\frac{|E|^2}{MB}, \frac{|E||V|}{\sqrt{MB}}))$. We further develop the IOBufs algorithm that manages to approach the I/O lower bound, and thus claim its optimality. Finally, we make efforts to parallelize IOBufs to further improve the performance and scalability. We show in the experiment that IOBufs significantly outperforms the state-of-the-art algorithms EMRC and BFC-EM. In addition, IOBufs can scale to conducting butterfly counting on the Clueweb graph with 37 billion edges and quintillions (10^{18}) of butterflies.

CCS Concepts: • **Mathematics of computing → Graph enumeration**; **Graph algorithms**; • **Hardware → External storage**; • **Computing methodologies → Shared memory algorithms**.

Additional Key Words and Phrases: Graph, butterfly, I/O-efficient algorithm, parallel algorithm.

ACM Reference Format:
Zhibin Wang, Longbin Lai, Yixue Liu, Bing Shui, Chen Tian, and Sheng Zhong. 2023. I/O-Efficient Butterfly Counting at Scale. *Proc. ACM Manag. Data* 1, 1, Article 34 (May 2023), 27 pages. https://doi.org/10.1145/3588714

1 INTRODUCTION

Butterfly (a.k.a., rectangle) is a cyclic motif[1] that is fundamental in graph analysis. Particularly, the butterfly is the smallest non-trivial cohesive motif [53–56] on a bipartite graph [2, 32, 45, 67], where vertices can be divided into two disjoint sets, and edges existing only between the two

[1]Following a convention, we call a small structure as a motif to avoid causing ambiguity with the data graph.

Authors' addresses: Zhibin Wang, Nanjing University, State Key Laboratory for Novel Software Technology, Nanjing, Jiangsu, China, wzbwangzhibin@gmail.com; Longbin Lai, Alibaba Group, Hangzhou, Zhejiang, China, Longbin.lailb@alibaba-inc.com; Yixue Liu, Nanjing University, State Key Laboratory for Novel Software Technology, Nanjing, Jiangsu, China, mf20330052@smail.nju.edu.cn; Bing Shui, Nanjing University, State Key Laboratory for Novel Software Technology, Nanjing, Jiangsu, China, 191098191@smail.nju.edu.cn; Chen Tian, Nanjing University, State Key Laboratory for Novel Software Technology, Nanjing, Jiangsu, China, tianchen@nju.edu.cn; Sheng Zhong, Nanjing University, State Key Laboratory for Novel Software Technology, Nanjing, Jiangsu, China, sheng.zhong@gmail.com.

2836-6573/2023/5-ART34 $15.00
https://doi.org/10.1145/3588714

sets of vertices. Consider a graph $G = (V, E)$, where V and E are the sets of vertices and edges, respectively. The problem of butterfly counting is to compute the total number of butterflies in G. Butterfly counting plays an important role in many applications, such as spam detection [16, 57, 58], recommendation systems [50], word-document clustering [13], research group identification [12], and link prediction according to transitivity theory [9]. Recently, Lyu et al. [34] have leveraged butterfly counting to prune infeasible vertices in a fraud-detection scenario of e-commerce.

A butterfly can be naturally decomposed into two wedges, where a wedge is an intersection of two edges, as demonstrated in Figure 1. Thus, it is a common practice to first count wedges between each pair of vertices as intermediate states and use the wedge count to further count butterflies. Thus, edges and wedges (as intermediate results) are two dominant types of data to materialize in memory for counting butterflies.

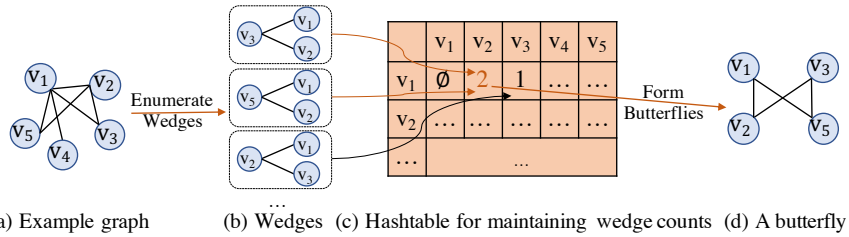


Fig. 1. An example of butterfly counting.

With the rapid growth of graph data, the computation resources for butterfly counting (e.g., processing capacity and memory) can quickly drain due to its super-linear time complexity and large memory consumption. Fortunately, the development of modern architectures brings in new opportunities. Regarding processing capacity, modern multi-core CPUs [19, 36, 49], GPUs [35, 39, 59], and FPGAs [15, 33, 63] have introduced massive parallelism that expands processing capacity to an unprecedented extent. Regarding memory configuration, people tend to leverage hierarchical memory, i.e., faster yet smaller on-board (main) memory as the primary data container at runtime and relatively slower yet larger secondary memory to hold data overflowed from the main memory. A typical practice adopts RAM as the main memory, and Solid State Drive (SSD), Hard Disk Drive as the secondary memory. A more recent work in the cloud [52] has used the local VM (virtual machine) memory as the main memory and the cloud storage such as S3 as the secondary memory.

1.1 Existing solutions and their weakness

In-memory algorithms. ExactBFC [47] is a sequential algorithm that optimized butterfly counting by using vertex priority to avoid duplicate calculation, which cannot fully utilize the processing capacity of modern hardware. Note that the primary computation of butterfly counting is to count the wedges between all pairs of vertices, and we denote the task as $T_{wc}\{V \times V\}$. It is straightforward to parallelize the subtask of $T_{wc}\{\{u\} \times V\}$ for $u \in V$, as proposed by BFC-VP++ [55]. Although BFC-VP++ was designed to carefully reuse the memory, it still incurs a $O(|E| + t|V|)$ space cost with t working threads. Memory quickly becomes insufficient when porting such an algorithm to the modern hardware, as it is not uncommon to configure hundreds of threads.

Hierarchical-memory algorithms. BFC-EM [55] and EMRC [68] have been proposed to leverage disk as the secondary memory to ease the main-memory shortage. Henceforth, we use M to denote the size of the main memory and B to denote the size of a data block that serves as the data unit exchanged between the main and secondary memories. Particularly, BFC-EM loads the edges from the disk in batches to compute wedges. After completing the current batch, the wedges are

immediately spilled to the disk to make room for the next batch. BFC-EM incurs $O(\frac{\Lambda}{B})$ I/Os, where Λ denotes the number of wedges in the graph. The I/O complexity is insensitive to the main memory size M, making it incapable of benefiting from more memory.

Alternatively, EMRC partitions the graph into p subgraphs to fit in the main memory, and processes each subgraph sequentially. The authors proved that no *witnessing* algorithm for butterfly counting could guarantee $o(\frac{|E|^2}{MB})$ I/Os. Here, a witnessing algorithm [22] terms a class of algorithms that must "see" all occurrences of the motif (here butterfly) in the main memory. They further showed that EMRC arrives at $O(\frac{|E|^2}{MB})$ I/Os in the worst case, thus claiming its I/O optimality. Nevertheless, we find out that a non-negligible $O(\frac{|V|^2}{p^2})$ space for materializing wedges has been overlooked in EMRC, which may cause memory overflow for processing large graphs. After consulting with the authors, we fix the issue in Section 4.3. However, this leads to larger I/Os of EMRC as $O(\frac{|E|^2}{MB} + \frac{|E||V|}{\sqrt{MB}})$ and overturns its I/O optimality.

Note that existing algorithms on hierarchical memory have been developed in the sequential CPU context, which fail to exploit the full potential of the modern multi-core processors.

1.2 Our contributions

In this paper, we study *I/O-efficient* and *parallel* butterfly counting algorithms in the hardware context that embodies 1) a shared-memory multi-core processor, including but not limited to CPU; 2) a hierarchical-memory configuration, in which the main memory size satisfies $M = o(|E|)$ and the secondary memory is arbitrarily large.

Our first contribution is the proposal of a new class of algorithms called the *semi-witnessing* algorithm, by observing that it suffices to see a subgraph of butterfly (e.g., wedge) in the main memory for counting butterflies, as opposed to the witnessing algorithm that has to see the whole butterfly. Based on the semi-witnessing algorithm, we derive a new I/O lower bound for butterfly counting as $\Omega(\min(\frac{|E||V|}{\sqrt{MB}}, \frac{|E|^2}{MB}))$. Note that when $\frac{2|E|}{|V|} < c_3\sqrt{M}$ for a constant c_3, in other words, the density (or average degree) of the graph is sufficiently small, our result degrades to $\Omega(\frac{|E|^2}{MB})$ as given in [68]. One may thus argue that our result is impractical, as it seems that "most" graphs under discussion are sparse graphs. Nevertheless, there actually exists a large spectrum of dense graphs, including but not limited to IoT (Internet of Things) [31], software function calls [20], transitive closure graph [25], cryptocurrency graph [64], and brain neural network [46].

Our second contribution is developing an algorithmic framework for doing butterfly counting on the hierarchical memory, called IOBufs, short for I/O-efficient Butterfly Counting at Scale, which is configurable to incorporate not only all our newly developed variants, but also existing algorithms including EMRC and BFC-VP++. We show that IOBufs ultimately arrives at the worst-case optimal I/O complexity of $O(\min(\frac{|E||V|}{\sqrt{MB}}, \frac{|E|^2}{MB}))$ with the adaptive configuration according to graph density. Particularly, IOBufs can adapt to the main-memory size M. In fact, when M is sufficiently large to accommodate all data required by the algorithm, a variant of IOBufs becomes BFC-VP++.

Our final contribution is parallelizing IOBufs in order to leverage the processing capacity of modern hardware. In the parallel context, given that each working thread may maintain its own state, the algorithm can consume more memory than a sequential counterpart. As an example, BFC-VP++ consumes $O(|E| + t|V|)$ space when there are t working threads. In the in-memory context, such an increment of space complexity can cause the algorithm to lower the DoP (degree of parallelism) to avoid overflowing the memory when processing large graphs. In the hierarchical-memory setting, it will lead to the increment of I/O cost and may compromise the I/O efficiency of the algorithm. We have identified that the dilemma of naïve approaches (including that of

BFC-VP++) lies in the *coarse-grained* parallelism, which means that an algorithm attempts to parallelize some relatively large subtasks, such as $T_{wc}\{\{u\} \times V\}$ in BFC-VP++. In response to this, we propose a more fine-grained approach that further divides the subtask into smaller pieces. Noticing that too fine a granularity can increase the scheduling overhead [44], we carefully tradeoff the I/O-efficiency and granularity of parallelism in IOBufs.

1.3 Organizations

The rest of the paper is organized as follows. Section 2 reviews the related works. Preliminary is presented in Section 3. Existing solutions are given in Section 4, in which we revisit the I/O complexity of EMRC. We propose the semi-witnessing algorithm and derive the I/O lower bound of butterfly counting in Section 5. Based on the semi-witnessing algorithm, we develop multiple variants of IOBufs in Section 6 in order to approach the I/O lower bound. Section 7 further discusses the fine-grained technique for parallelizing IOBufs. Section 8 reports the experimental results, and Section 9 concludes the paper.

2 RELATED WORK

2.1 Motif counting and subgraph matching

Motifs are small subgraphs in the data graph, and thus existing subgraph matching approaches can be utilized for motif counting. There are two mainstream subgraph matching approaches: backtracking-based [4, 5, 17] and join-based [3, 28, 29, 60]. These approaches were mainly developed for matching/enumerating subgraphs in general, and might not be the most suitable for butterfly counting. Recent literature [37, 43, 62] considered counting graphlets, i.e., all motifs up to k vertices, while butterfly counting is trivially a part of 4-vertex graphlet counting. However, the time complexity of 4-vertex graphlet counting is dominated by 4-clique (a complete graph with 4 vertices), making it sub-optimal for butterfly counting. As triangle and butterfly are the two most widely-studied motifs, we will next focus on reviewing the works that were explicitly developed for counting/listing the two motifs in massive graphs.

2.2 Massive triangle counting and listing

With graph data distributed into different machines, [18] was developed by synchronizing the intermediate states to list the triangles. The algorithm of [42] partitioned the graphs with replication in order to avoid communication. Modern hardware is also leveraged for counting triangles. The results of triangle counting in [23, 24, 41] have demonstrated the great potential of GPUs. With multiple load-balancing techniques, a recent work of [41] managed to scale the task to as many as 1024 GPU cards. Huang et al. considered triangle counting on FPGAs in [24] for better energy efficiency. In [61], the authors accelerated the intersection operation for triangle counting using both CPUs and GPUs. In the hierarchical-memory setting, [21, 40] proposed I/O-efficient algorithms for triangle counting by using disk as the secondary memory.

2.3 Butterfly counting

Apart from the works introduced in Section 1, ParButterfly [49] has been developed for butterfly counting with four variants called hashing, histogram, sorting, and batching, which mainly focus on parallelizing the most critical operation – wedge aggregating. As pointed out by [49], the most-optimized batching variant of ParButterfly is actually the parallel BFC-VP++ [55] algorithm, and thus we do not further discuss it in the paper. As the graph data becomes massive, researchers have studied to approximate the number of butterflies through sampling. The state-of-the-art (SOTA) sort in [47] considered vertex-, edge- and wedge-sampling algorithms, as well as the edge sparsification

technique to estimate the number of butterflies. It is interesting to compare the performance of the SOTA approximate algorithm with that of our IOBufs. On the same dataset Journal (Table 3), the authors reported in Figure 6(b) of [47] around 8s run time using $25\%|E|$ sparsification ($\geq 25\%|E|$ memory) to obtain a result with 99.9% accuracy. In comparison, IOBufs achieves better performance (5.7s in Figure 6) using $25\%|E|$ memory yet obtains the exact result. Note that our techniques are orthogonal to the approximate algorithm, given that IOBufs can be adopted on the graph that is still too large to fit in the main memory even after being sparsified. In the streaming setting, the FLEET algorithm [48] was proposed, which combines edge sampling, edge sparsification, and adaptive random sampling to estimate the number of butterflies in both infinite and windowed streams. Furthermore, [65, 66] considered counting butterflies on uncertain graphs, in which each edge has a probability of being present.

3 PRELIMINARY

Table 1. Frequently used notations.

Notation	Definition
$G(V, E)$, $G(V_G, E_G)$	graph with V (V_G) and E (E_G)
$N(u)$, $N_G(u)$	neighbors of u
$d(u)$, $d_G(u)$	degree of u
$\overline{d}$, $\overline{d_G}$	average degree of G
(u, v, w)	a wedge consists of u, v, w
(u, v, w, x)	a butterfly consists of u, v, w, x
Φ, Φ_G (Λ, Λ_G)	the number of butterflies (wedges) of G
$\mathcal{H}$	a set for maintaining wedge count
M	the size of the main memory
B	the size of a block of data
p, q	the partition numbers of graph data and fine-grained parallelism
c_1, c_2, c_3	constant values in complexity functions determined by the memory in bytes taken by an edge/wedge

Notations. In this paper, we consider an unlabelled, undirected simple graph[2] $G(V_G, E_G)$, where V_G and $E_G \subseteq V_G \times V_G$ denote the vertex and edge set, respectively. An undirected edge between two vertices u and v is denoted as (u, v), or equivalently (v, u). We let $N_G(u)$ (resp. $d_G(u) = |N_G(u)|$) denote the neighbors (resp. degree) of vertex u in G, i.e., $N_G(u) = \{v|(u, v) \in E_G\}$. We also use $\overline{d_G}$ to denote the average degree of the graph G, i.e., $\overline{d_G} = \frac{1}{|V_G|}\sum_{u \in V_G} d(u) = \frac{2|E_G|}{|V_G|}$. A graph $g(V_g, E_g)$ is called a *subgraph* of G, denoted as $g \subseteq G$, if $V_g \subseteq V_G$ and $E_g \subseteq E_G$. Given four vertices $u, v, w, x \in V$, a butterfly (u, v, w, x) is a 4 cycle formed by edges (u, v), (v, w), (w, x), and (x, u). A wedge (u, v, w) is formed by the two edges of (u, v) and (v, w), in which v is called the *center* vertex and u, w are called the *leaf* vertices. Let Φ_G and Λ_G be the number of butterflies and wedges in G, respectively. Besides, we denote $\mathcal{H}\{\mathcal{P} \rightarrow \mathbb{N}\}$ for $\mathcal{P} \subseteq V_G \times V_G$ as a set of key-value pairs for maintaining the wedge counting, where the key is a vertex pair (u, w) and the corresponding value $\mathcal{H}(u, w)$ is a natural number. For simplicity, we will omit the subscript of G in the above notations when G is clear in the context. We summarize frequently used notations in Table 1.

Butterfly counting. Butterfly counting aims to compute the number of butterflies in a given graph. Note that each butterfly instance (u, v, w, x) can appear 8 times in a graph due to automorphism.

[2]Note that our techniques apply seamlessly to a bipartite graph.

We follow [55] to deduplicate by using vertex priority. In this paper, we will not dive into the technique, and refer interested readers to [55] for further details.

Algorithm 1 In-memory butterfly counting framework

Input: graph G
Output: number of butterflies Φ
1: Initialize (0 for each entry) the hashtable $\mathcal{H}\{V \times V \rightarrow \mathbb{N}\}$
2: **for each** wedge (u, v, w) in G **do**
3: $\quad \Phi \leftarrow \Phi + \mathcal{H}(u, w)$
4: $\quad \mathcal{H}(u, w) \leftarrow \mathcal{H}(u, w) + 1$

Observe that a butterfly (u, v, w, x) is formed by two wedges (u, v, w) and (u, x, w). Thus, it is a common practice to count wedges as a preliminary step in butterfly counting. Specifically, if there exist k wedges between vertices u and w, as $\{(u, v_1, w), \ldots, (u, v_k, w)\}$, a total number of $\frac{k*(k-1)}{2}$ butterflies can be formed by combining any pair of wedges. Obviously, it suffices to maintain the number of wedges between all pairs of vertices $(u, w) \in (V \times V)$ for counting butterflies. In the following, we often write *wedges*, short for the number of wedges. Consequently, a general framework of in-memory butterfly counting is established, as presented in Algorithm 1, showing the general procedure of the in-memory butterfly counting. For each wedge (u, v, w) computed over the graph (line 2), the number of the entry (u, w) in $\mathcal{H}$ will be incremented by 1 (line 4). Note that a small trick in line 3 updates the current butterfly count Φ by adding the current value of $\mathcal{H}(u, w)$. This actually leverages the sum of an arithmetic sequence, namely $\frac{k*(k-1)}{2} = \sum_{i=0}^{k-1} i$. Taking the vertex pair (v_1, v_2) in Figure 1 as an example, we have $\mathcal{H}(v_1, v_2) = 2$ meaning that there are two wedges existing between v_1 and v_2, and one butterfly (v_1, v_3, v_2, v_5) is formed accordingly.

4 BUTTERFLY COUNTING ON HIERARCHICAL MEMORY

As the graph becomes large, people have studied to solve butterfly counting on the memory with two-level hierarchies, in which the main memory has a small capacity of M but is fast to access, while the secondary memory has a large capacity but is relatively slower. As a common practice to compute the I/O cost, we consider a *block* of size B as the unit data exchanged between the main and secondary memory.

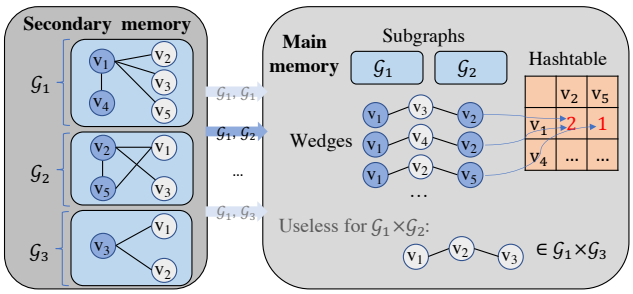


Fig. 2. The execution of IOBufs-naïve on hierarchical memory.

4.1 Existing solutions

BFC-EM [55] was proposed to load the edges in a batching manner and conduct wedge computing sequentially for each batch of edges. The wedges will be constantly spilled to the secondary memory in order to avoid overflowing the main memory. BFC-EM can do butterfly counting with

constant space complexity, while it renders an I/O cost insensitive to M. We show in the experiment (Section 8) that BFC-EM can barely benefit from more memory. Alternatively, the authors developed EMRC and proved its "I/O optimality" in [68]. Following EMRC to partition the graph, we propose the algorithmic framework of IOBufs, which provides an interface that can be further implemented to realize different variants of the algorithm. Note that the framework also incorporates EMRC, and one of the variants actually becomes the in-memory BFC-VP++ [55] if the main memory is sufficiently large to accommodate the edges and wedges.

4.2 The partition-based framework

Graph partition. We first randomly partition the vertices V into p disjoint subsets satisfying

$$V = \bigcup_{i=1}^{p} \mathcal{V}_i \text{ ,with } \forall 1 \leq i \neq j \leq p, \mathcal{V}_i \cap \mathcal{V}_j = \emptyset,$$

and then construct each partitioned subgraph as $\mathcal{G}_i(\mathcal{V}_i, \mathcal{E}_i)$, where $\mathcal{E}_i = \{(u,v) | (u \in \mathcal{V}_i \vee v \in \mathcal{V}_i) \wedge (u,v) \in E\}$. Note that alternative partition strategies may be considered, which should have little impact on our algorithm as long as it produces a balanced number of edges across partitions, as will be empirically studied in Section 8.6. Given that we must fit the largest partition in the main memory, the imbalanced partition may result in a larger p, and accordingly the increment of time and I/O complexity of the algorithm.

The framework. Algorithm 2 demonstrates the algorithmic framework of IOBufs. Lines 1-2 first partition the graph into p parts, which can actually be preprocessed as will be discussed. For every two partitioned subgraphs (lines 3-4), it launches the interface IOBufs-interface() in line 4 to count the butterflies between two subgraphs. The interface is the key to configure different variants of the algorithm.

Algorithm 2 The algorithmic framework of IOBufs

Input: graph G
Output: number of butterflies Φ
1: Configure the partition number p
2: Partition G into p parts, as $\{\mathcal{G}_1, \ldots, \mathcal{G}_p\}$
3: **for** $i \in \{1, ..., p\}$ **do**
4: **for** $j \in \{1, ..., p\}$ **do**
5: $\Phi \leftarrow \Phi$ + IOBufs-interface$(\mathcal{G}_i, \mathcal{G}_j)$

Algorithm 3 IOBufs-naïve (a.k.a. EMRC)

1: **function** IOBufs-naïve$(\mathcal{G}_i, \mathcal{G}_j)$
2: Load $\mathcal{G}_i, \mathcal{G}_j$ and initialize $\mathcal{H}\{\mathcal{V}_i \times \mathcal{V}_j \rightarrow \mathbb{N}\}$ in the main memory
3: **for** wedges (u, v, w) satisfying $u \in \mathcal{V}_i, v \in V, w \in \mathcal{V}_j$ **do**
4: $\Phi \leftarrow \Phi + \mathcal{H}(u, w)$
5: $\mathcal{H}(u, w) \leftarrow \mathcal{H}(u, w) + 1$
6: **return** Φ

A naïve variant of IOBufs is given with the corresponding implementation of IOBufs-interface in Algorithm 3. This algorithm is actually EMRC, but the partition number p may be configured differently (Section 4.3). The algorithm has the same skeleton as Algorithm 1, but places a constraint in line 3 to guarantee each butterfly to be counted exactly once.

Example 4.1. Figure 2(a) demonstrates partitioning the graph in Figure 1 into three parts, and Figure 2(b) gives a running example of IOBufs-naïve. In the two partitioned subgraphs $\mathcal{G}_1$ and $\mathcal{G}_2$, we can locate two wedges between (v_1, v_2), and thus one butterfly is recorded. The constraint of line 3 ensures the correctness of the algorithm. Consider the wedge (v_1, v_2, v_3). If without the constraint, it will be counted multiple times; otherwise, it will only be counted while processing $(\mathcal{G}_1, \mathcal{G}_3)$.

The authors of [68] derived the I/O complexity of EMRC as $O(\frac{|E|^2}{MB})$, and proved that EMRC achieves the optimal I/O. However, we observe that the authors overlooked the space cost of wedges, which leads to the increment of I/O cost of EMRC. In the following, we rectify the result after consulting with the authors.

4.3 Revisit the I/O complexity of EMRC

According to the random partition strategy, each partition has $O(\frac{|E|}{p})$ edges by expectation with high possibility [21]. Moreover, the set $\mathcal{H}$ now only needs to maintain the wedges of vertex pair $(u, w) \in \mathcal{V}_i \times \mathcal{V}_j$ each time. As both $\mathcal{V}_i$ and $\mathcal{V}_j$ are a fraction of $1/p$ of the vertices, $\mathcal{H}$ consumes $O(\frac{|V|^2}{p^2})$ space, which is unfortunately ignored in [68]. As a result, the space complexity becomes $O(\frac{|E|}{p} + \frac{|V|^2}{p^2})$ in Algorithm 3.

We next analyze the actual I/O complexity. According to [68], the I/O cost of Algorithm 3 is dominated by line 4, which continuously loads the subgraphs from the secondary memory. By summing all pairs of subgraphs, we obtain

$$\sum_{i=1}^{p}\sum_{j=1}^{p}\frac{|\mathcal{E}_i| + |\mathcal{E}_j|}{B} = O(\frac{p|E|}{B}). \tag{1}$$

Note that the required data of the algorithm must fit in the main memory. Given the space complexity of Algorithm 3, we have

$$\begin{aligned} M \geq c_1\frac{|E|}{p} + c_2\frac{|V|^2}{p^2} \;\Rightarrow \\ Mp^2 - c_1|E|p - c_2|V|^2 \geq 0, \end{aligned} \tag{2}$$

where c_1 and c_2 are constant values determined by the size (in bytes) of an edge and a wedge in the main memory.

By quadratic formula, we further have

$$p \geq \frac{c_1|E| + \sqrt{(c_1|E|)^2 + 4c_2M|V|^2}}{2M}. \tag{3}$$

Let $p = \lceil\frac{2c_1|E|+\sqrt{4c_2M|V|^2}}{2M}\rceil = \lceil\frac{c_1|E|}{M} + \frac{\sqrt{c_2}|V|}{\sqrt{M}}\rceil$, the main memory is sufficient to maintain both edges and wedges. Together with Equation 1, we can derive the I/O complexity of IOBufs-naïve (and EMRC) as $O(\frac{|E||V|}{\sqrt{M}B} + \frac{|E|^2}{MB})$.

Accordingly, we revisit the time complexity. The algorithm must enumerate each wedge exactly once, which costs $O(\Lambda)$; the cost of processing the edges of the graph is $O(p|E|)$ according to Equation 1. Putting them together, we have the time complexity of $O(\Lambda + \frac{|E||V|}{\sqrt{M}} + \frac{|E|^2}{M})$. Surprisingly, given that $\Lambda = O(|E|^{1.5})$ [10] and $M = o(|E|)$, this is tighter than $O(\frac{|E|^2}{\sqrt{M}})$ as given in the paper [68].

Remark 4.1. From the above analysis, we summarize the *requirement* and *configuration* that should be applied through this paper:

- **Requirement**: For an algorithm to work on the hierarchical memory, the main memory must have sufficient capacity to accommodate the data required by the algorithm.
- **Configuration**: In practice, we can configure the minimum possible p such that the above space requirement is satisfied. As in Equation 3, once the main memory size and the graph statistics are given, p can be configured prior to running the algorithm, and thus we can preprocess partitioning the graph.

Particularly for EMRC, if we configure $p = \lceil \frac{c_1|E|}{M} \rceil$ according to the paper [68], it is unable to meet the above memory requirement for processing large graphs. If we configure p as Equation 3, the I/O cost is larger than the optimal bound (will be derived in Section 5). In other words, we conclude that the algorithm of EMRC is not I/O-optimal.

5 I/O LOWER BOUND OF BUTTERFLY COUNTING

Zhu et al. [68] derived $\Omega(\frac{|E|^2}{MB})$ as the I/O lower bound of butterfly counting based on the *witnessing* algorithm [22]. The witnessing algorithm requires the algorithm to see all butterflies in the main memory, which is too strict for butterfly counting. We show that it suffices to witness a subgraph of the butterfly (e.g., wedge), if the task is to count rather than enumerate butterflies. In response to this, we propose a new class of algorithms called *semi-witnessing* algorithm, and prove that any semi-witnessing algorithm for butterfly counting must incur $\Omega(\min(\frac{|E|^2}{MB}, \frac{|E||V|}{\sqrt{MB}}))$ I/Os, which is lower than existing results.

5.1 The semi-witnessing algorithm

Note that Algorithm 1 leverages a fact that a butterfly (u, v, w, x) can be decomposed into two wedges by a pair of vertices (u, w). To correctly count butterflies, the algorithm only needs to record the number of wedges between the leaves u and w without materializing the center vertices. The center vertices in this case are nonessential for the counting. Without the center vertices (and the associated edges), the algorithm cannot completely witness a butterfly. We hence believe that the I/O bound [68] derived from the witnessing algorithm may leave room for improvement.

Although Algorithm 1 does not witness butterflies, it does witness all wedges as shown in line 2. Inspired by this, we propose a new class of algorithms called *semi-witnessing* algorithm for counting a given motif, by allowing the algorithm to witness only a subgraph instead of the whole motif. Formally,

Definition 5.1. Given a graph G, a small motif g, and a subgraph $g' \subseteq g$, we denote $A_{g'}$ as a semi-witnessing algorithm regarding g' for counting the occurrences of g in G subject to

(1) $A_{g'}$ must witness all occurrences of g' in the main memory;
(2) there exists no g'' where $g' \subset g''$ such that $A_{g'}$ witnesses all occurrences of g''.

Obviously, A_g is a witnessing algorithm.

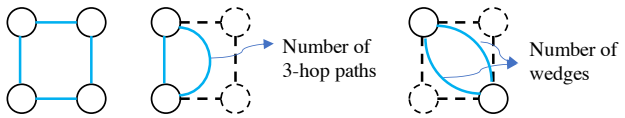


Fig. 3. Three types of semi-witnessing algorithms for butterfly counting, in which the vertices/edges outlined with dotted lines are nonessential.

As shown in Figure 3, we can develop three types of semi-witnessing algorithms for butterfly counting, namely A_{BF}, A_{3P}, and A_{WG}, where the subgraphs are butterfly, 3-hop path (a path

connected by 3 consecutive paths), and wedge, respectively. Note that semi-witnessing algorithms regarding an edge and two parallel edges, denoted respectively as A_E and A_{PE}, are not listed in Figure 3. The following lemma rules out their existence.

Lemma 5.2. *There exists neither A_{PE} nor A_E for butterfly counting.*

Proof. Consider a butterfly (u, v, w, x). If there is an A_{PE} algorithm, after witnessing the parallel edges, such as (u, v) and (w, x) without loss of generality (w.l.g.), it is essential to also check the existence of the edges (u, x) and (v, w) for the butterfly. In this case, the whole butterfly has already been witnessed. If there is an A_E algorithm, after witnessing an edge, such as (u, v) w.l.g., it is infeasible to simultaneously witness either (u, x) or (v, w), otherwise, it is at least an A_{WG} algorithm by Definition 5.1. Moreover, co-witnessing the edge (w, x) parallel to (u, v) ends up with an infeasible A_{PE} algorithm. □

Our next move is to provide the I/O lower bound for the three types of semi-witnessing algorithms. Prior to that, we introduce some useful notations. Given $\mathcal{E} \subseteq E$ and $u, v \in V$, we denote $\mathcal{E}_v$ as the edges in $\mathcal{E}$ that must contain v, and $\mathcal{E}(u, v)$ as an indicator that returns 1 when the edge $(u, v) \in \mathcal{E}$ and 0 otherwise. Moreover, the following I/O inequality obviously holds for any motif:

$$\#\text{I/Os} \geq \frac{\#\text{ occurrences of the motif}}{\text{maximum of \# the motif counted per I/O}}. \tag{4}$$

5.2 The I/O lower bound of A_{BF}

This is the case where Zhu et al. [68] derived the bound of $\Omega(\frac{|E|^2}{MB})$. While finding their proof complicated to follow, we propose a more succinct version in this paper. We first have

Lemma 5.3. *Given any $\mathcal{E} \in E$, the number of butterflies $\Phi_{\mathcal{E}}$ that can be witnessed in $\mathcal{E}$ satisfies*

$$\Phi_{\mathcal{E}} \leq |\mathcal{E}|^2.$$

Proof.

$$\begin{aligned}\Phi_{\mathcal{E}} &= \sum_{u \in V} \sum_{v \in V \setminus \{u\}} \sum_{w \in V \setminus \{u,v\}} \sum_{x \in V \setminus \{u,v,w\}} \mathcal{E}(u, v) \times \mathcal{E}(v, w) \times \mathcal{E}(w, x) \times \mathcal{E}(x, u) \\ &\leq \sum_{(u,v) \in \mathcal{E}} \sum_{(w,x) \in \mathcal{E}} \mathcal{E}(v, w) \times \mathcal{E}(x, u) \leq |\mathcal{E}|^2.\end{aligned}$$

□

Theorem 5.4. *No A_{BF} algorithm for butterfly counting can guarantee $o(\frac{|E|^2}{MB})$ I/Os.*

Proof. The main memory can hold at most $O(M)$ edges. After conducting one I/O, $O(B)$ more edges will be loaded into the main memory, making $O(M+B)$ edges in total. According to Lemma 5.3, there are at most $O((M + B)^2)$ butterflies witnessed by A_{BF}. Note that we cannot guarantee that the butterflies are all *newly* discovered while conducting each I/O. After conducting sufficient numbers of I/Os, it is possible that some butterflies are counted more than once. Therefore, if we immediately utilize Equation 4 based on the result of a single I/O, we will obtain an I/O lower bound that is too loose to be practical. Inspired by [20], we can conduct s consecutive I/Os before launching one butterfly counting to mitigate the impact of duplicate counting. By doing so, there are at most $O((M + sB)^2)$ butterflies witnessed by A_{BF}, i.e., $O(\frac{(M+sB)^2}{s})$ per I/O by average, which is $O(MB)$ by setting $s = O(\frac{M}{B})$. Besides, the number of butterflies in a graph can arrive at $\Theta(|E|^2)$. Putting them into Equation 4, we can derive the I/O lower bound of $\Omega(\frac{|E|^2}{MB})$ for any A_{BF}, which is exactly the bound given in [68]. □

5.3 The I/O lower bound of A_{WG}

Recall that we use $\mathcal{H}$ as a set of key-value pairs to record the wedge count, where the keys are the leaves of the wedge. Henceforth, we will use $\mathcal{H} \subseteq V \times V$ to indicate the (number of) wedges materialized in $\mathcal{H}$. We further say a wedge (u, v, w) is subject to $\mathcal{H}$, if $(u, w) \in \mathcal{H}$. We have

Lemma 5.5. *For any graph G, given $\mathcal{E} \subseteq E$ and $\mathcal{H} \subseteq V \times V$, the number of new wedges subject to $\mathcal{H}$ that can be witnessed by $\mathcal{E}$, denoted as $\Lambda_{\mathcal{E}}[\mathcal{H}]$, satisfies*

$$\Lambda_{\mathcal{E}}[\mathcal{H}] \leq \sqrt{|\mathcal{H}||\mathcal{E}|}.$$

Proof.

$$\begin{aligned}\Lambda_{\mathcal{E}}[\mathcal{H}] &= \sum_{(u,w)\in\mathcal{H}} \overbrace{\sum_{v\in V} \mathcal{E}_v(u,v) \times \mathcal{E}_v(v,w)}^{\text{number of new wedges witnessed by } \mathcal{E}} \\ &= \sum_{v\in V} \left(\sum_{(u,w)\in\mathcal{H}} \mathcal{E}_v(u,v) \times \mathcal{E}_v(v,w) \right).\end{aligned}$$

Given any $v \in V$, on the one hand, there form at most $|\mathcal{E}_v|^2$ new wedges; on the other hand, the wedges subject to $\mathcal{H}$ are bounded by $|\mathcal{H}|$. Consequently,

$$\begin{aligned}\Lambda_{\mathcal{E}}[\mathcal{H}] \leq& \sum_{v\in V} \min(|\mathcal{H}|, |\mathcal{E}_v|^2) \leq \sum_{v\in V} \sqrt{|\mathcal{H}|} \times |\mathcal{E}_v| \\ =& \sqrt{|\mathcal{H}|} \times \sum_{v\in V} |\mathcal{E}_v| = \sqrt{|\mathcal{H}|}|\mathcal{E}|.\end{aligned} \tag{5}$$

□

Note that an A_{WG} algorithm only needs to deal with wedges and edges, as the butterflies can be immediately counted once the number of wedges between all pairs of vertices has been registered. It is then critical for an A_{WG} to consider how to exploit the main and secondary memory to handle wedges and edges, respectively. To ease the discussion, we start with a simple case called *wedge-only*. In this case, the algorithm loads edges in a streaming manner to count wedges, and the main memory only maintains wedges among vertex pairs. The wedges, after being used to count butterflies, do not need to be spilled to the secondary memory. For further reference, the IOBufs-wedge algorithm in Section 6 is one such algorithm.

Lemma 5.6. *In the wedge-only case, no A_{WG} algorithm for butterfly counting can guarantee $o(\frac{|E||V|}{\sqrt{MB}})$ I/Os.*

Proof. The wedges are maintained in the main memory with $|\mathcal{H}| = O(M)$, and we load $O(B)$ edges per I/O to update the wedge counting. According to Lemma 5.5, at most $O(\sqrt{MB})$ new wedges will be witnessed by conducting one I/O. Given that the total number of wedges can be $\Theta(|E||V|)$, a direct application of Equation 4 gives an I/O lower bound of $\Omega(\frac{|E||V|}{\sqrt{MB}})$. □

Obviously, there are some constraints in the wedge-only case, while it paves the way for discussing the general *wedge-edge-shared* case, in which the main memory can simultaneously materialize both wedges and edges, and the wedges *can* be spilled to the secondary memory for further use. As a matter of fact, "wedge-only" is a special case of wedge-edge-shared. We conclude:

Theorem 5.7. *No A_{WG} algorithm for butterfly counting can guarantee $o(\frac{|E||V|}{\sqrt{MB}})$ I/Os.*

Proof. In the wedge-edge-shared case, the main memory should be divided to store wedges and edges. Therefore, both edges and wedges take up $O(M)$ space. Considering performing an I/O to load $O(B)$ edges, we have $|\mathcal{E}| = O(M+B)$ and $|\mathcal{H}| = O(M)$. According to Lemma 5.5, $O(\sqrt{M}(M+B))$ new wedges can be witnessed. Similarly, if we conduct an I/O to load $O(B)$ more wedges, $O(\sqrt{M+B}M)$ new wedges can be witnessed. Directly applying Equation 4 by only considering one I/O leads to a loose bound. Hence, we conduct $s = s_1 + s_2$ consecutive I/Os as Theorem 5.4, where s_1 I/Os are for loading edges and s_2 I/Os are for loading wedges. As a result, a total number of $O(M + s_1B)$ edges and $O(M + s_2B)$ wedges are now in the main memory, which gives the newly witnessed wedges as at most

$$O((M + s_1B)\sqrt{M + s_2B}) = O((M + sB)\sqrt{M + sB}).$$

The number of wedges that are witnessed on average along the process is $O(\sqrt{MB})$ by setting $s = O(\frac{M}{B})$. Considering the number of wedges can be as many as $\Theta(|E||V|)$, we derive the I/O bound of A_{WG} as $\Omega(\frac{|E||V|}{\sqrt{MB}})$ in the general wedge-edge-shared case. □

5.4 The I/O lower bound of A_{3P}

Theorem 5.8. *No A_{3P} algorithm for butterfly counting can guarantee $o(\frac{|E||V|}{\sqrt{MB}})$ I/Os.*

Proof. A_{3P} requires witnessing the 3-hop paths between all pairs of vertices. There are two ways to do so:

- **Directly computing all the 3-hop paths in the main memory.** The case is not viable based on Definition 5.1. Note that if all 3-hop paths have been witnessed, together with the fact that the last edge must be checked to close a butterfly, the algorithm already witnesses all butterflies.
- **Dividing a 3-hop path into a wedge concatenating with an edge.** The I/O cost of such an algorithm is at least that of computing the wedges, and thus it cannot render I/Os in $o(\frac{|E||V|}{\sqrt{MB}})$ by Theorem 5.7.

According to the above analysis, the theorem holds. □

5.5 Final result and discussions

Theorem 5.9. *No semi-witnessing algorithm for butterfly counting can guarantee $o(\min(\frac{|E|^2}{MB}, \frac{|E||V|}{\sqrt{MB}}))$ I/Os.*

Proof. Recall from the discussions in Lemma 5.2 and Figure 3 that any semi-witnessing algorithm for butterfly counting must belong to either A_{BF}, A_{3P} and A_{WG}. Thus, this theorem holds by summarizing Theorem 5.4, Theorem 5.7, and Theorem 5.8. □

The result in Theorem 5.9 clearly guides us to design the algorithm according to the density of the graph. Based on the size of $\overline{d}$ and $\sqrt{M}$, an A_{BF} algorithm is favored if the graph is sufficiently sparse, while an A_{WG} algorithm is a better choice if the graph is dense. In Section 6, we will develop two variants of the algorithm and make them adaptive to the graph density accordingly.

Moreover, we believe that the semi-witnessing algorithm in Definition 5.1 has paved a new way for developing I/O-efficient algorithms for general motif counting. We focus on butterfly in this paper as a pioneering step, and will dive into general motifs in the future.

6 THE I/O-OPTIMAL ALGORITHM

Based on Algorithm 2, we design our IOBufs that can achieve the I/O lower bound in Theorem 5.9. We first point out the key observation to lower I/O cost for butterfly counting. We then optimize Algorithm 3 based on the observation by proposing two variants of the algorithm, namely IOBufs-edge

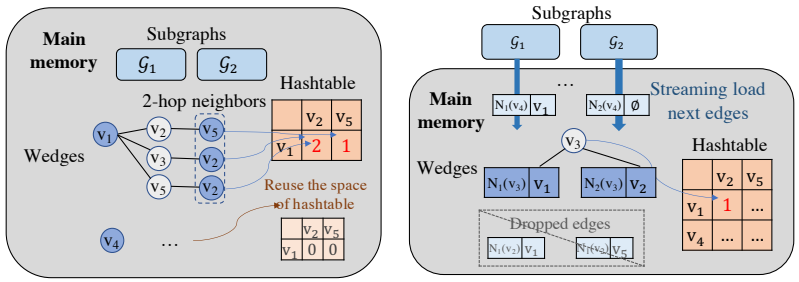
and IOBufs-wedge. We show that IOBufs-edge belongs to A_{BF} and IOBufs-wedge belongs to A_{WG} (Definition 5.1), and they can also approach the I/O lower bound of Theorem 5.4 and Theorem 5.7, respectively. IOBufs can hence adaptively select IOBufs-edge and IOBufs-wedge based on whether the graph is sparse or dense. Finally, we claim that IOBufs is I/O-optimal according to Theorem 5.9.

6.1 The key observation

As we have discussed in Section 4.3, the main reason that Algorithm 3 cannot achieve I/O optimality is that it must maintain both edges and wedges in the main memory, which, however, is not a necessity according to a key observation:

> The number of butterflies can be *correctly* derived from either edges or wedges *individually*.

If we have only edges in the main memory, we can clearly try all combinations of 4 *vertices* and check whether they can form a butterfly. Obviously, such a naïve approach cannot work in practice due to the large time complexity. On the other hand, if we already get the number of wedges between each pair of vertices, the number of butterflies can be computed as we have shown in Algorithm 1. However, if the wedges are randomly computed with edges kept in the secondary memory, the discovery of each wedge may involve I/O cost for two edges, which will lead to intolerable I/O cost. The above observation does inspire us to consider keeping only edges or wedges in the main memory, but it is non-trivial to develop such an algorithm with proper time and I/O complexity. The main idea is that we can *randomly* access the in-memory data, while *sequentially* processing the other data in a streaming manner. This naturally leads to two variants of the algorithm, namely IOBufs-edge and IOBufs-wedge, standing for the IOBufs algorithm with only edges and wedges maintained in the main memory, respectively.



(a) IOBufs-edge (b) IOBufs-wedge

Fig. 4. The execution of IOBufs-edge and IOBufs-wedge on hierarchical memory.

6.2 The IOBufs-edge variant

Algorithm 4 presents the IOBufs-edge implementation. Apart from IOBufs-naïve, IOBufs-edge does not allocate a set to record the wedge count for all pairs of $\mathcal{V}_i \times \mathcal{V}_j$ in the first place. Instead, it does so sequentially for each vertex in $\mathcal{V}_i$ (lines 2-3). Note that, after each vertex u is processed, the number of wedges of $\{u\} \times \mathcal{V}_j$ has already contributed to the final result by lines 6-7. Consequently, the memory can be reused to process the following vertices.

Example 6.1. As illustrated in Figure 4(a), after two subgraphs $\mathcal{G}_1$ and $\mathcal{G}_2$ are loaded into the main memory, the set $\mathcal{H}\{\{v\} \times \mathcal{V}_2\}$ will be allocated. $\mathcal{H}$ will first record the wedge count between $v_1 \in \mathcal{V}_1$ and all vertices in $\mathcal{V}_2$. Once the process of v_1 is completed, $\mathcal{H}$ will be reused for v_4.

Algorithm 4 IOBufs-edge

1: **function** IOBufs-edge($\mathcal{G}_i, \mathcal{G}_j$)
2: Load subgraph $\mathcal{G}_i, \mathcal{G}_j$ into the main memory
3: **for** $u \in \mathcal{V}_i$ **do**
4: Initialize $\mathcal{H}\{\{u\} \times \mathcal{V}_j \to \mathbb{N}\}$ in the main memory
5: **for** wedges (u, v, w) satisfying $v \in N_{\mathcal{G}_i}(u), w \in \mathcal{V}_j$ **do**
6: $\Phi \leftarrow \Phi + \mathcal{H}(u, w)$
7: $\mathcal{H}(u, w) \leftarrow \mathcal{H}(u, w) + 1$
8: **return** Φ

Witnessing butterflies. We show that IOBufs-edge actually belongs to A_{BF}. At first glance, as IOBufs-edge follows the framework of Algorithm 1 to derive butterfly counting from wedges, it may not witness all butterflies. Let us take a closer look at Algorithm 4. For any butterfly (u, v, w, x), while locating the wedge of (u, v, w) in line 5, we must have all nonessential vertices v and x and their associated edges in the main memory due to line 2. As a result, the butterfly has already been witnessed in the main memory.

Complexity Analysis. The reuse of memory for wedges gives the space complexity of IOBufs-edge as $O(\frac{|E|}{p} + \frac{|V|}{p}) = O(\frac{|E|}{p})$. According to the space requirement of Remark 4.1, we can get $p = O(\frac{|E|}{M})$, and by Equation 1, we can derive the I/O cost of IOBufs-edge as $O(\frac{|E|^2}{MB})$. Given that IOBufs-edge belongs to A_{BF}, we claim its I/O optimality according to Theorem 5.4. The time complexity is $O(\Lambda + \frac{|E|^2}{M})$ according to Section 4.3.

6.3 The IOBufs-wedge variant

IOBufs-wedge is given in Algorithm 5 which only maintains wedges in the main memory. The algorithm first initializes a memory space of $\mathcal{H}(\mathcal{V}_i \times \mathcal{V}_j)$ for recording the wedges between the two partitioned graphs. While organizing the graph as a sequence of the adjacent lists of vertices in the secondary memory, the algorithm can load the data of $N_i(v)$ ($N_{\mathcal{G}_i}(v) \cap \mathcal{V}_i$) and $N_j(v)$ sequentially for all $v \in V$ in a streaming manner (line 4).

Example 6.2. Figure 4(b) shows the process of IOBufs-wedge for $\mathcal{G}_1, \mathcal{G}_2$. The set of $\mathcal{H}\{\mathcal{V}_1 \times \mathcal{V}_2\}$ will be allocated in the first place. Then wedges are counted with edges sequentially loaded into the main memory. For example, when counting the wedges with v_3 as the center vertex, only edges connecting v_3 in the two subgraphs ((v_1, v_3) and (v_3, v_2)) will be loaded. In this case, a wedge (v_1, v_3, v_2) is counted. As these edges will not be used in later computation, they can be dropped to make room for the edges connecting next center vertex v_4.

Algorithm 5 IOBufs-wedge

1: **function** IOBufs-wedge($\mathcal{G}_i, \mathcal{G}_j$)
2: Initialize $\mathcal{H}\{\mathcal{V}_i \times \mathcal{V}_j \to \mathbb{N}\}$ in the main memory
3: **for** $v \in V$ **do**
4: Load $N_i(v) = N_{\mathcal{G}_i}(v) \cap \mathcal{V}_i$ and $N_j(v) = N_{\mathcal{G}_j}(v) \cap \mathcal{V}_j$
5: **for** wedges (u, v, w) satisfying $u \in N_i(v), w \in N_j(v)$ **do**
6: $\Phi \leftarrow \Phi + \mathcal{H}(u, w)$
7: $\mathcal{H}_(u, w) \leftarrow \mathcal{H}_(u, w) + 1$
8: **return** Φ

Witnessing wedges. Obviously, IOBufs-wedge must witness all wedges. We show how it differs from the IOBufs-edge variant without witnessing the butterflies. Consider a butterfly (u, v, w, x).

In one iteration that processes v (line 3), it witnesses all vertices but x; in another iteration that processes x, it may fail to witness v. Overall, IOBufs-wedge cannot guarantee simultaneously witnessing all vertices of a butterfly in the main memory. As a result, IOBufs-wedge belongs to A_{WG}. More specifically, as it runs by only maintaining the wedges in the main memory, it is a wedge-only A_{WG}.

Complexity analysis. IOBufs-wedge consumes $O(\frac{|V|^2}{p^2})$ memory space to maintain $\mathcal{H}(\mathcal{V}_i \times \mathcal{V}_j)$, which gives $p = O(\frac{|V|}{\sqrt{M}})$ according to Remark 4.1. By Equation 1, we have the I/O cost of IOBufs-wedge as $O(\frac{|E||V|}{\sqrt{M}B})$, which is optimal according to Theorem 5.7. Similarly, we can derive the time complexity as $O(\Lambda + \frac{|E||V|}{\sqrt{M}})$.

Table 2. Comparison of IOBufs-edge and IOBufs-wedge.

Variant	p	Time complexity	Space complexity	I/O complexity
IOBufs-edge	$O(\frac{\lvert E\rvert}{M})$	$O(\Lambda + \frac{\lvert E\rvert^2}{M})$	$O(\frac{\lvert E\rvert}{p})$	$O(\frac{\lvert E\rvert^2}{MB})$
IOBufs-wedge	$O(\frac{\lvert V\rvert}{\sqrt{M}})$	$O(\Lambda + \frac{\lvert E\rvert\lvert V\rvert}{\sqrt{M}})$	$O(\frac{\lvert V\rvert^2}{p^2})$	$O(\frac{\lvert E\rvert\lvert V\rvert}{\sqrt{M}B})$

6.4 The adaptive algorithm

Table 2 quickly compares IOBufs-edge and IOBufs-wedge. Obviously, we can adaptively choose between the two variants based on whichever yields lower I/O cost, as:

$$\frac{c_1|E|^2}{MB} < \frac{\sqrt{c_2}|E||V|}{\sqrt{M}B} \quad \Rightarrow \bar{d}(= \frac{2|E|}{|V|}) < c_3\sqrt{M}, \tag{6}$$

where $c_3 = \frac{2\sqrt{c_2}}{c_1}$. In other words, we should use IOBufs-edge when the graph is sufficiently sparse, namely $\bar{d} < c_3\sqrt{M}$, and use IOBufs-wedge otherwise. Therefore, we have IOBufs adaptively configured as:

Algorithm 6 IOBufs, the adaptive variant

```
1: if d̄ < c3√M then
2:     apply IOBufs-edge in Algorithm 2
3: else
4:     apply IOBufs-wedge in Algorithm 2
```

Obviously, the I/O complexity of IOBufs is $O(\min(\frac{|E|^2}{MB}, \frac{|E||V|}{\sqrt{M}B}))$. According to Theorem 5.9, we claim that IOBufs is I/O-optimal. Besides, the time complexity is $O(\Lambda + \min(\frac{|E|^2}{M}, \frac{|E||V|}{\sqrt{M}}))$.

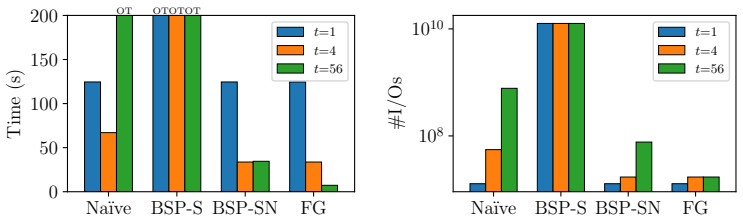
7 PARALLELIZATION AND IMPLEMENTATION

In this section, we focus on the parallelization of IOBufs. Henceforth, we use t to denote the degree of parallelism (DoP). Note that it is trivial to parallelize IOBufs-wedge: we simply replace the **for** loop in line 3 of Algorithm 5 with a **parfor**, and all working threads can share a common $\mathcal{H}$. However, it is non-trivial to do so for IOBufs-edge without compromising its I/O efficiency. Observing that the main issue of the naïve approaches is the coarse-grained parallelism, we propose a more fine-grained approach to better tradeoff parallelism and I/O efficiency of IOBufs-edge.

7.1 Naïve coarse-grained approaches

A "Naïve" approach for parallelizing IOBufs-edge in Algorithm 2 is to assign the subtask of each pair of $(\mathcal{G}_i, \mathcal{G}_j)$ to a working thread. However, this will cause the space complexity to increase by t times, which consequently leads to the increment in I/O cost. For IOBufs-edge, as the total space cost becomes $O(t \cdot \frac{|E|}{p})$ and must still be accommodated by the main memory of size M, we can derive that $p = O(t \cdot \frac{|E|}{M})$ and the I/O cost as $O(t \cdot \frac{|E|^2}{MB})$.

We next discuss a bulk-synchronous parallel (BSP) [51] mechanism by developing an interactive workflow for Algorithm 2, where each iteration handles a pair of partitioned subgraphs. The parallelization is applied within each iteration, and all working threads will synchronize at the end of the current iteration before moving to the next. For IOBufs-edge, it is rather parallelizing lines 3-7 in Algorithm 5. We denote $T_{wc}\{U_1 \times U_2\}$ for any $U_1, U_2 \subseteq V$ as the task of counting wedges between the pairs of vertices in $U_1 \times U_2$. As BFC-VP++, we conduct a **parfor** in line 3 of Algorithm 4 to parallelize the subtask of $T_{wc}\{\{u\} \times \mathcal{V}_j\}$ for all $u \in \mathcal{V}_i$. Critically, the allocation of $\mathcal{H}$ in line 4 must be carefully considered. If all working threads share a common $\mathcal{H}\{\mathcal{V}_i \times \mathcal{V}_j\}$, which is called "BSP-Shared" ("BSP-S" for short), the algorithm degrades to IOBufs-naïve that must have large I/O and time complexity. Alternatively, a solution called "BSP-Shared/nothing" ("BSP-SN" for short) makes each working thread to maintain a local $\mathcal{H}$ in order to emancipate from data races. It is trivial to derive the space cost as $O(\frac{|E|}{p} + \frac{t|V|}{p})$, and the I/O cost as $O(\frac{|E|^2}{MB} + \frac{t|V||E|}{MB})$. Such I/O cost may be huge, as it is likely to have $t > \bar{d}$ when processing a sparse graph on the modern hardware.



(a) Wall clock time. (b) I/O cost.

Fig. 5. Performance of four parallel techniques on Delicious (refers to Table 3) with $t = 1, 4, 56$.

To reveal that none of the above alternatives - namely "Naïve", "BSP-S", and "BSP-SN" - scale well, we conducted a micro benchmark that compares alternatives with our "Fine-Grained" ("FG" for short) solution, and the results of overall running time and numbers of I/Os are shown in Figure 5. For a test that cannot terminate within one hour, we mark OT for running time, and manually calculate its I/O cost. "BSP-S" fails to complete all test cases. "Naïve" scales poorly, and it even runs OT when $t = 56$. "BSP-SN" still performs similarly as "FG" when $t = 4$, but it cannot further benefit from more parallelism and is outperformed by "FG" by a large margin when $t = 56$. The corresponding I/O cost supports the performance results in all tests. Noticeably, when t increases, the I/O cost of "FG" almost remains fixed, while those of the alternative solutions increase significantly.

7.2 Fine-grained parallelism

We observe that the dilemma of trading off the parallelism and the I/O efficiency of all approaches in Section 7.1 lies in the coarse-grained parallelism, namely attempting to parallelize a subtask of $T_{wc}\{U_1 \times U_2\}$ with some large U_1 or U_2. In response to this, we propose a more fine-grained

approach based on "BSP-shared-nothing"[3] by further dividing the subtask of $T_{wc}\{\{u\} \times \mathcal{V}_j\}$ into q disjoint pieces of

$$\{T_{wc}\{\{u\} \times \mathcal{V}_{j,1}\}, T_{wc}\{\{u\} \times \mathcal{V}_{j,2}\}, \ldots, T_{wc}\{\{u\} \times \mathcal{V}_{j,q}\}\},$$

where $\mathcal{V}_j = \mathcal{V}_{j,1} \cup \mathcal{V}_{j,2} \cup \cdots \cup \mathcal{V}_{j,q}$ and $\mathcal{V}_{j,i_x} \cap \mathcal{V}_{j,i_y} = \emptyset$ for any $1 \le i_x \ne i_y \le q$. We will see that the configuration of the value q is key to the parallelism-I/O tradeoff, but let us first check out the parallel IOBufs-edge algorithm in Algorithm 7. Line 4 is responsible for further dividing the task. In line 6, a concurrent queue is initialized to hold all vertices in $\mathcal{V}_i$, which is popped out in line 8 for each working thread to initiate processing a subtask of $T_{wc}\{\{u\} \times \mathcal{V}_{j,k}\}$. Lines 7-13 embody the sub-routine to run in parallel. The process will not stop until the query Q becomes empty. Each working thread of id τ ($1 \le \tau \le t$) maintains a local Φ_τ for the butterflies already counted. Finally, an aggregation is conducted to summarize the Φ_τ for all working threads as the final result. Note that by letting each thread pop vertices from a concurrent queue rather than iterate over $\mathcal{V}_j$ in a round-robin fashion, we actually adopt the dynamic-scheduling strategy that is shown to have better load balance in [55].

Algorithm 7 IOBufs-edge, the parallelization

```
1:  function IOBufs-edge(G_i, G_j)
2:      Load subgraph G_i, G_j into the main memory
3:      Configure q according to Equation 7
4:      Partition V_j into q disjoint parts as {V_{j,1}, V_{j,2} ..., V_{j,q}}
5:      for k in {1 ... q} do
6:          Place all vertices of V_i in a concurrent queue Q
7:          parfor an idle thread τ ∈ {1, ..., t} do
8:              u = Q.pop()
9:              Initialize H{{u} × V_{j,k} → ℕ}
10:             for wedges (u, v, w) satisfying v ∈ N_{G_i}(u), w ∈ V_{j,k} do
11:                 Φ_τ ← Φ_τ + H(u, w)
12:                 H(u, w) ← H(u, w) + 1
13:         until Q is empty
14:     Aggregate Φ = Σ_{τ=1}^{τ=t} Φ_τ
15:     return Φ
```

Our "Fine-grained" technique renders space and I/O complexity as $O(\frac{|E|}{p} + \frac{t}{q} \cdot \frac{|V|}{p})$ and $O(\frac{|E|^2}{MB} + \frac{t}{q} \cdot \frac{|V||E|}{MB})$, respectively. We immediately re-approach I/O optimality by setting $q = \Theta(t)$. However, a large q may result in a subtask being too fine-grained to run efficiently in parallel [44] due to the scheduling overhead. Therefore, we consider how to tradeoff the I/O cost and scheduling cost. Let $C_{\text{I/O}}$ and C_{sched} denote the cost of conducting one I/O, and schedule one subtask, respectively. While partitioning a graph into p parts and a subtask into q parts, we have $\frac{2p|E|}{B}$ total I/Os to conduct, and pq^2 subtasks to schedule. As a result, we have the following optimization problem:

$$\begin{aligned} &\textbf{Minimize } \frac{|E|}{B}\frac{C_{\text{I/O}}}{C_{\text{sched}}}p + p^2q, \\ &\textbf{subject to } c_1\frac{|E|}{p} + c_2\frac{t|V|}{pq} \le M, \text{ and } p, q \in \mathbb{N}^+ . \end{aligned} \tag{7}$$

[3]We also study both "Naïve" and "BSP-S", but find little room for improvement.

Given p, it is obvious that q must be as small as possible while satisfying the space requirement. We then iterate $p = \lceil \frac{c_1|E|}{M} \rceil$ (when $q \to \infty$) to $p = \lceil \frac{c_1|E|+c_2t|V|}{M} \rceil$ (when $q = 1$) and derive the corresponding q till we obtain a pair of p and q that can minimize f. In practice, $\frac{C_{\text{I/O}}}{C_{\text{sched}}}$ can be measured as follows: we first drive p by q according to the memory constraint and make the objective function a function of q, namely $f(q)$; then we iterate the q value to do some preliminary tests on certain graphs to obtain a $q*$ value that renders the best performance; we finally obtain the $\frac{C_{\text{I/O}}}{C_{\text{sched}}}$ value by letting $f'(q*) = 0$, where f' denotes the differential of f.

Remark 7.1. When $M >> |E|$, namely the main memory is arbitrarily large, we can leverage Equation 7 to derive an interesting result. Given that the lower bound of p as $\lceil \frac{c_1|E|}{M} \rceil = 1$ and the upper bound $\lceil \frac{c_1|E|+c_2t|V|}{M} \rceil = 1$, we must have $p = 1$ and $q = 1$ to minimize Equation 7. In this case, IOBufs-edge becomes the parallel BFC-VP++ [55].

8 EVALUATION

All evaluated algorithms are implemented in C++ and compiled with g++ 7.5.0. The optimization flag is set as "-O3". We run all tests using an Intel Xeon Gold 6330 CPU of 56 physical cores, together with a 128GB DDR4 RAM and a 1TB SSD disk. Unless otherwise specified, we set the main memory capacity to 25% of the graphics size by default, i.e. $M = 25\%|E|$. Note that we control memory usage using cgroup configuration [1] (a Linux kernel feature) in order to diminish the impact of caching for properly benchmarking the I/O cost. Throughout the experiments, we mark OT if an algorithm cannot terminate in one hour. For an algorithm running OT, the presented #I/Os is manually computed according to Equation 1.

Table 3. Graph statistics.

Graph	Source	$\|V\|$	$\|E\|$	$\overline{d}$	Φ	Sizes
MANN	[46]	3.32E+03	5.51E+06	3,316	1.51E+13	44M
Flickr	[26]	5.00E+05	8.55E+06	34	3.53E+10	72M
Journal	[26]	1.07E+07	1.12E+08	21	3.30E+12	1.0G
Delicious	[26]	3.46E+07	1.02E+08	6	5.69E+10	1.1G
Tracker	[26]	4.04E+07	1.41E+08	7	2.01E+13	1.4G
Orkut	[26]	1.15E+07	3.27E+08	57	2.21E+13	2.7G
Bi-twitter	[55]	4.17E+07	6.02E+08	29	6.30E+13	5.1G
Bi-sk	[55]	5.06E+07	9.11E+08	36	1.22E+14	7.7G
Bi-uk	[55]	7.77E+07	1.33E+09	34	4.89E+14	11G
Clueweb	[11]	9.78E+08	3.74E+10	76	1.49E+18	286G

Datasets. The statistics and sources of the graphs are summarized in Table 3. Particularly, MANN is a real dense graph provided by DIMACS [14] for graph challenges. Journal, Delicious, and Orkut are social networks representing user-group memberships of online communities. Flickr [38] shows the user-photo relationship on an online photo-sharing website. Tracker is the web tracking dataset representing the relationship between the internet domains and the trackers they contain. Bi-twitter, Bi-sk, and Bi-uk are subgraphs of large real social graph twitter [27] and web graph sk-2005 and uk-2006-05 from WebGraph [6–8]. We follow [55] to construct bipartite graphs from them. Among the graphs, we use MANN, Journal, Delicious, and Orkut as default datasets in some tests. Clueweb is a gigantic web graph whose size is larger than the configured memory of our machine. It is used to test the real-life performance of our algorithm without manually controlling the memory. In addition, we apply the kronecker generator [30] to generate synthetic graphs to purposely control the statistics. All graphs have been preprocessed into undirected simple graphs, and organized

in the format of compressed sparse row (CSR). Each vertex is identified by a 4-byte integer, and their ids are rearranged according to the degree (priority) following [55]. The size of each graph is reported accordingly. We by default apply the "Radix" strategy [29] to partition the graph, as will be discussed in Section 8.6.

Algorithms. Our empirical studies are conducted against the following algorithms:

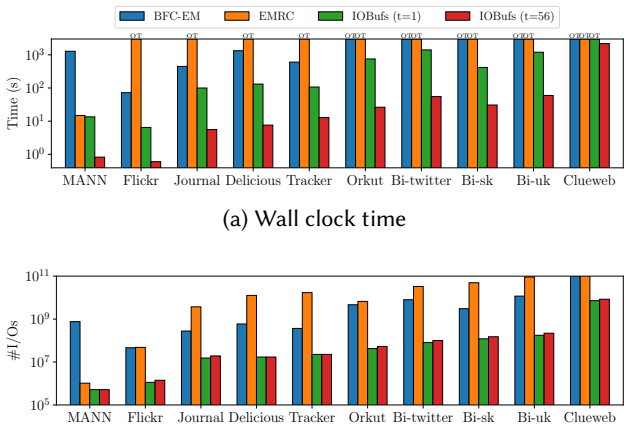
The IOBufs *variants.* IOBufs-naïve, IOBufs-edge, and IOBufs-wedge are our algorithms given in Algorithm 3, Algorithm 4, and Algorithm 5, respectively. IOBufs denotes the I/O-optimal algorithm in Algorithm 6 that can adaptively choose between IOBufs-edge and IOBufs-wedge based on graph density. In the sequential context, the partition number p will be configured according to Remark 4.1 for each variant. Additionally, IOBufs-edge, IOBufs-wedge, and IOBufs can run in parallel according to Section 7. We use the DoP $t = 56$ by default. p and q will be either configured according to Equation 7 by default or specified otherwise. Given the graph storage, the constant values of c_1, c_2 are determined by the memory usage of the edges and wedges, respectively, and c_3 is computed in Equation 6. We set $c_1 = 16$ as it takes 4 bytes to store the end vertex of an edge, and each edge must be stored in two directions for two partitioned subgraphs; $c_2 = 4$ as we record the wedge count between two vertices as an integer; and immediately $c_3 = 0.25$.

The BFC variants. We compare two BFC variants [55], namely BFC-VP++ and BFC-EM. BFC-VP++ is the most optimized variant of the kind developed in the parallel (in-memory) context, while BFC-EM stands for the variant leveraging disk as the secondary memory. The authors have not made the source codes publicly accessible. Thus, we apply our IOBufs-edge running in parallel with arbitrarily large M as BFC-VP++ (Remark 7.1). Since BFC-EM cannot fit into our framework, we implement the algorithm by referencing the paper and consulting with the authors.

EMRC. The EMRC algorithm is introduced in [68]. The authors do not provide the source codes, but it is actually IOBufs-naïve with $p = \lceil \frac{|E|}{M} \rceil$ according to Section 4.3. However, it runs OOM in most cases under this original setting. Thus, we set $p = \lceil \frac{c_1|E|}{M} + \frac{c_2|V|}{\sqrt{M}} \rceil$ for EMRC.

Note that the authors of BFC and EMRC agree that we faithfully reproduced their results.

8.1 Compare with the state of the arts



(a) Wall clock time

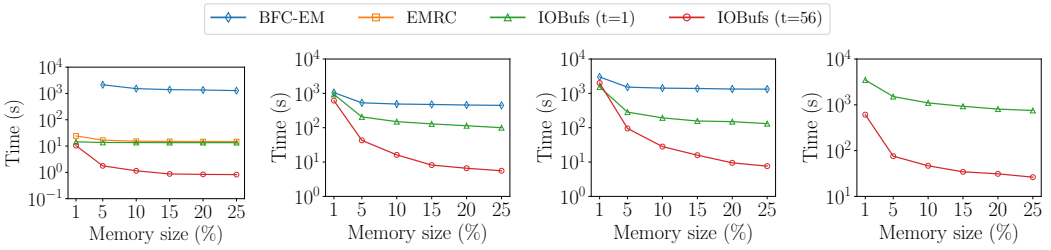
(b) I/O cost

Fig. 6. The performance of IOBufs, BFC-EM, and EMRC.

In this section, we compare IOBufs with the hierarchical-memory algorithms: BFC-EM and EMRC. Figure 6 shows the results of wall clock time and #I/Os on all graphs in Table 3. For EMRC, it suffers from huge I/O cost and thus runs OT in all cases except MANN. According to our analysis in Section 4.3, EMRC must maintain both wedges ($O(\frac{|V|^2}{p^2})$) and edges ($O(\frac{|E|}{p})$) in the main memory, while IOBufs only maintains one of them. In a dense graph such as MANN whose wedges and edges are relatively close in volume (by $|E| \approx |V|^2/p$ in each partition), EMRC can perform similarly to IOBufs. While in other sparse datasets which contain much more wedges than edges, performance of EMRC downgrades dramatically. Let us look into BFC-EM whose I/O complexity is determined by the number of wedges in the graph. Observe that it performs worse on the graphs with more wedges. Particularly on MANN, a small but dense graph, BFC-EM performs the worst among all algorithms. Overall, based on the available cases, IOBufs (single thread) outperforms BFC-EM by 25× on average in term of wall clock time. Regarding I/O cost, it incurs 209× and 364× less I/Os than BFC-EM and EMRC, respectively. Moreover, we can further speed up IOBufs via parallelism. Observe that the I/O cost only slightly increases in the parallel cases, mostly due to the fine-grained parallelism. Henceforth, unless otherwise specified, one may be aware that the IOBufs-wedge variant is chosen for the dense graph MANN, while IOBufs-edge variant is used for all other graphs.

Furthermore, to reveal the ability of IOBufs on the huge real-world graph, we run IOBufs on the gigantic Clueweb [6–8] containing almost 1 billion vertices and 37 billion edges, a total of 286GB, using all the configured memory (128GB) of our machine. The experimental results show that IOBufs successfully counts butterflies at the scale of quintillions (10^{18}) on Clueweb in 2182s, while BFC-EM and EMRC run OT, and their I/O costs are extremely large (more than 10^{11} I/Os), and thus omitted in Figure 6(b).

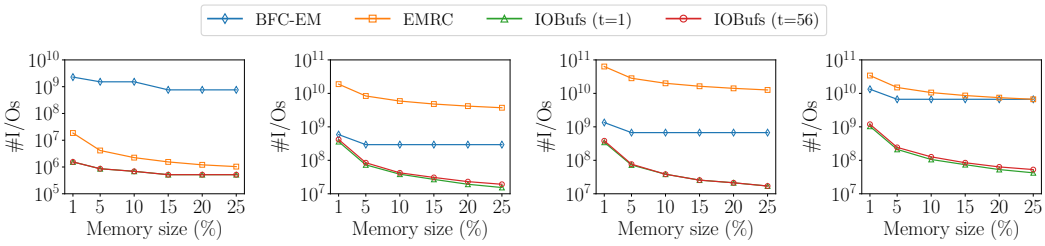
8.2 Impact of memory



(a) MANN. (b) Journal. (c) Delicious. (d) Orkut.

Fig. 7. Memory size impact (Wall clock time).

As indicated in [21], memory size is critical to the performance of hierarchical-memory algorithms. We hence vary M from $1\%|E|$ to $25\%|E|$ to evaluate the algorithms of BFC-EM, EMRC, and IOBufs, and report the results in Figure 7 and Figure 8. Clearly, BFC-EM cannot benefit from a larger memory configuration, as its performance almost remains fixed while increasing the memory size. EMRC can still only handle MANN, but it shows performance improvement in this case as the growth of memory. Our IOBufs demonstrates an obvious dropping trend for both wall clock time and #I/Os when there is larger memory to run the algorithm. Such a trend is more notable in the parallel cases because there are larger subtasks to run rendering lower scheduling overhead when the memory is sufficient (according to space requirement). Notice that the single-threaded IOBufs performs better than the multi-threaded version with $M = 1\%|E|$ in Figure 7(c). With such

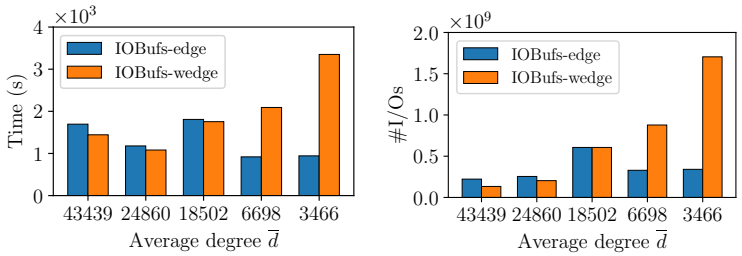


(a) MANN. (b) Journal. (c) Delicious. (d) Orkut.

Fig. 8. Memory size impact (I/O cost).

small memory, each partitioned subgraph and the corresponding workload become too small to benefit from parallelism. Noticeably, IOBufs consistently outperforms the competitors throughout all memory configurations. Its performance is already reasonably good even with $1\%|E|$ memory, while we recommend at least $5\%|E|$ memory if possible to better exploit parallelism.

8.3 IOBufs-edge **vs.** IOBufs-wedge

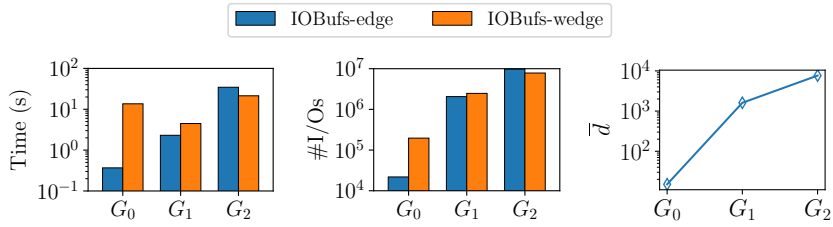


(a) Wall clock time. (b) I/O cost.

Fig. 9. Cost of IOBufs-edge and IOBufs-wedge on five generated graphs with different average degrees.

Varying degree: As discussed in Section 6, the average degree $\overline{d}$ will affect the choice of IOBufs-edge and IOBufs-wedge of our algorithm. We show the rationale for this design choice. To do so, we generate 5 graphs using kronecker [30] with (almost) fixed number of edges and vary $\overline{d}$ as shown in Figure 9. We use the default memory of $25\%|E| \approx 5GB$. Therefore, the dividing point of the degree is roughly $0.25\sqrt{M} \approx 18000$. As shown in Figure 9, the two variants of the algorithm perform comparatively around the dividing point. On the left-hand side where the degrees are larger, IOBufs-wedge performs better than IOBufs-edge. On the other side where the degrees are smaller, IOBufs-edge outperforms IOBufs-wedge in turn. The I/O cost reflects the same trend as running time. The results are consistent with our discussions in Section 6.

Transitivity closure analysis: In the network analysis, it is interesting to construct a *TC* graph from the base graph by adding edges between vertices that are originally disconnected but form transitive closures (i.e., wedges) [25]. Let the base graph be G_0, G_1 be the TC graph constructed from G_0, and G_2 be the one constructed from G_1, and so forth. This process can increase the density of a graph, and thus is used to study the performance of IOBufs-edge and IOBufs-wedge. We generate a graph with $\overline{d} = 16$ and $|V| = 2^{14}$ as G_0, based on which G_1 and G_2 are constructed. Figure 10 illustrates the results on these graphs. As the average degree increases, IOBufs-edge

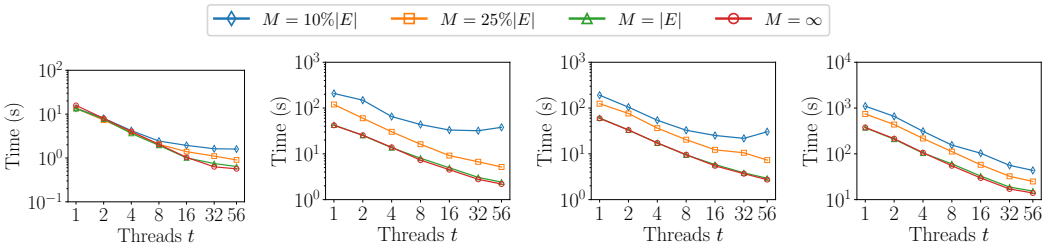


(a) Wall clock time. (b) I/O cost. (c) Degree.

Fig. 10. Performance of IOBufs-edge and IOBufs-wedge on TC graphs

initially outperforms IOBufs-wedge by a large margin on G_0, and is caught up with and eventually overturned by IOBufs-wedge on G_1 and G_2.

8.4 Scalability

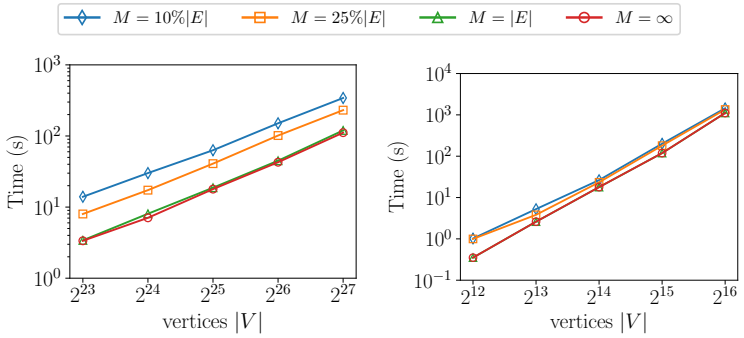


(a) MANN. (b) Journal. (c) Delicious. (d) Orkut.

Fig. 11. Scale-up performance of IOBufs, varying t.

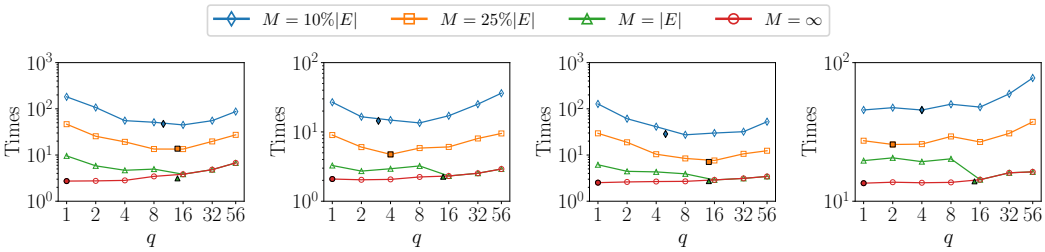
Scale-up. To reveal the scale-up performance of our fine-grained parallel technique, we vary the number of threads t and run IOBufs in different memory settings on the default graphs. Note that we include the case that $M = \infty$, in which IOBufs-edge becomes BFC-VP++ according to Remark 7.1. Observe that the cases of smaller memory typically have worse scale-up performance. In these cases, there are more subtasks to run, and the scheduling cost increases accordingly. Nevertheless, it already scales almost as well as the in-memory algorithm BFC-VP++ by slightly increasing the memory to $25\%|E|$. Overall, the performance of the algorithms with $M = 10\%|E|, 25\%|E|, |E|$, and ∞ are improved by 11×, 21×, 20× and 23×, respectively, while increasing the working threads from 1 to 56. The case of $M = |E|$ is also an interesting baseline. Note that the in-memory BFC-VP++ cannot run in this case because there is no room for maintaining the wedges, while our IOBufs can achieve competitive performance.

Data-scale. To study the data-scale performance, we apply kronecker generator and generate: 1) 5 sparse graphs with fixed $\overline{d}$ as 16 and varying $|V|$ from 2^{23} to 2^{27}; 2) 5 dense graphs with $\overline{d} = 50\%|V|$ and varying $|V|$ from 2^{12} to 2^{16}. The results of running the algorithm in different memory settings are reported in Figure 12. According to Section 6, the IOBufs-edge variant will be adopted except for the cases of testing on the dense graphs with $M = 10\%|E|$ and $M = 25\%|E|$. Using $M = \infty$ as the baseline, IOBufs scales pretty well in all cases, even when configuring $10\%|E|$ memory size.



(a) Sparse graphs with $\overline{d} = 16$. (b) Dense graphs with $\overline{d} = 50\%|V|$.

Fig. 12. Data-scale performance of IOBufs, varying graph sizes.



(a) Tracker. (b) Journal. (c) Delicious. (d) Orkut.

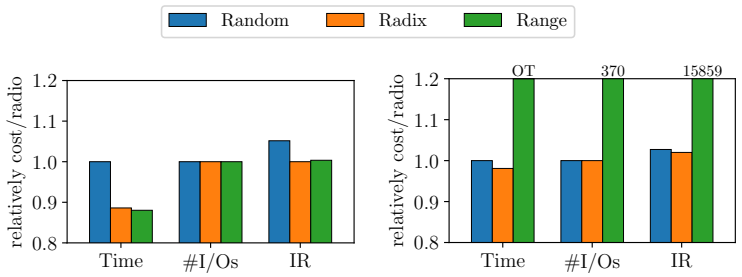
Fig. 13. Performance of fine-grained parallelism of IOBufs, varying q.

8.5 The fine-grained parallelism

We evaluate the effectiveness of the fine-grained parallelism proposed in Section 7. We run IOBufs (or more specifically IOBufs-edge) in different memory settings by varying the q value from 1 to 56, and show the results of the four graphs in Figure 13. Note that here we replace MANN in the default graphs as Tracker because IOBufs-wedge does not need fine-grained parallelism. On each line in Figure 13, there is a solid-filled point that represents the derived value of q in Equation 7, which aligns very well with the q that actually gives the best performance. Observe that except $M = \infty$, the performance of all cases demonstrates a trend of first descending and then ascending as the increment of q. In the beginning, the performance is improved because a larger q can result in a smaller I/O cost. After passing the optimal value, the performance declines in turn as a larger q introduces more cost for scheduling than benefit. Note that for $M = \infty$, the best configuration of q (as well as p) is always 1, as we analyzed in Remark 7.1.

8.6 Impact of partition strategy

Given a vertex of id i, we compare three commonly-used partition strategies as mentioned in Section 4.2: 1) *Random*: place the vertex in the partition of $rand(i)\%p$, where $rand()$ is a pseudo-random number generator; 2) *Radix [29]*: place the vertex in the partition of $i\%p$; 3) *Range [68]*: place the vertex according to which range its id belongs to, or more specifically, in the partition of $\lfloor i \times p/|V| \rfloor$. We evaluate the degree of balance achieved by a partitioning strategy via the *imbalance*



(a) MANN. (b) Journal.

Fig. 14. Performance of three partitioning approaches.

ratio [41], denoted as $IR = \frac{\max_i(|\mathcal{E}_i|)}{\min_i(|\mathcal{E}_i|)}$. Figure 14 reports the results of the three partition strategies on a dense graph MANN and a sparse graph Journal, in which Time and #I/Os are relative to the "Random" strategy. The results of "Random" and "Radix" are very similar. To be precise, "Radix" is slightly more balanced than "Random", and on top of which the algorithm also performs slightly better. Because of this, we use "Radix" as the default strategy. The "Range" strategy performs very differently on MANN and Journal, which is resulted from the rearrangement of vertex ids by the degrees. On MANN where all vertices roughly have the same degree, the partition is balanced, but on Journal, the partition in preceding ranges clearly involves vertices of larger degree, and thus more edges. The poor performance of the algorithm is observed with such an imbalanced partition. In summary, the partition strategy should have little impact on IOBufs as long as it produces a balanced number of edges across partitions.

9 CONCLUSION

We study the I/O-efficient algorithm for butterfly counting at scale in this paper. Observing that it suffices to witness only a subgraph rather than the whole structure in the main memory for counting butterflies, we propose the semi-witnessing algorithm and prove that no semi-witnessing algorithm for butterfly counting can guarantee $o(\min(\frac{|E|^2}{MB}, \frac{|E||V|}{\sqrt{MB}}))$ I/Os. We then develop the IOBufs algorithm that can arrive at the I/O lower bound, and parallelize the algorithm with a fine-grained technique to tradeoff the I/O and computation efficiency. The experimental results have verified the effectiveness of all our proposed techniques, which makes IOBufs outperform the state of the arts by orders of magnitude.

ACKNOWLEDGMENTS

Sheng Zhong was supported, in part, by NSFC-62272215, Leading-edge Technology Program of Jiangsu NSF under Grant BK20202001, and National Key R&D Program of China under Grants 2020YFB1005900. We also thank the authors of EMRC and BFC for their invaluable help in reproducing their results and insightful discussions, and all anonymous reviewers for their constructive feedback that helped us refine our ideas and arguments.

REFERENCES

[1] [n.d.]. cgroup. https://www.kernel.org/doc/Documentation/cgroup-v2.txt.
[2] Sinan G Aksoy, Tamara G Kolda, and Ali Pinar. 2017. Measuring and modeling bipartite graphs with community structure. *Journal of Complex Networks* 5, 4 (2017), 581–603.
[3] Khaled Ammar, Frank McSherry, Semih Salihoglu, and Manas Joglekar. 2018. Distributed evaluation of subgraph queries using worstcase optimal lowmemory dataflows. *arXiv preprint arXiv:1802.03760* (2018).

[4] Bibek Bhattarai, Hang Liu, and H Howie Huang. 2019. Ceci: Compact embedding cluster index for scalable subgraph matching. In *Proceedings of the 2019 International Conference on Management of Data*. 1447–1462.

[5] Fei Bi, Lijun Chang, Xuemin Lin, Lu Qin, and Wenjie Zhang. 2016. Efficient subgraph matching by postponing cartesian products. In *Proceedings of the 2016 International Conference on Management of Data*. 1199–1214.

[6] Paolo Boldi, Andrea Marino, Massimo Santini, and Sebastiano Vigna. 2014. BUbiNG: Massive Crawling for the Masses. In *WWW*. 227–228.

[7] Paolo Boldi, Marco Rosa, Massimo Santini, and Sebastiano Vigna. 2011. Layered Label Propagation: A MultiResolution Coordinate-Free Ordering for Compressing Social Networks. In *WWW*. ACM Press, 587–596.

[8] Paolo Boldi and Sebastiano Vigna. 2004. The WebGraph Framework I: Compression Techniques. In *WWW*. ACM Press, 595–601.

[9] Hsinchun Chen, Xin Li, and Zan Huang. 2005. Link prediction approach to collaborative filtering. In *Proceedings of the 5th ACM/IEEE-CS Joint Conference on Digital Libraries (JCDL'05)*. IEEE, 141–142.

[10] Norishige Chiba and Takao Nishizeki. 1985. Arboricity and Subgraph Listing Algorithms. *SIAM J. Comput.* 14, 1 (feb 1985), 210–223. https://doi.org/10.1137/0214017

[11] Charles L Clarke, Nick Craswell, and Ian Soboroff. 2009. *Overview of the trec 2009 web track*. Technical Report. WATERLOO UNIV (ONTARIO).

[12] Graham Cormode, Divesh Srivastava, Ting Yu, and Qing Zhang. 2008. Anonymizing bipartite graph data using safe groupings. *Proceedings of the VLDB Endowment* 1, 1 (2008), 833–844.

[13] Inderjit S. Dhillon. 2001. Co-Clustering Documents and Words Using Bipartite Spectral Graph Partitioning. In *Proceedings of the Seventh ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '01)*. Association for Computing Machinery, New York, NY, USA, 269–274. https://doi.org/10.1145/502512.502550

[14] DIMACS. [n.d.]. DIMACS Challenge. http://dimacs.rutgers.edu/Challenges/.

[15] Eric Finnerty, Zachary Sherer, Hang Liu, and Yan Luo. 2019. Dr. BFS: Data Centric Breadth-First Search on FPGAs. In *2019 56th ACM/IEEE Design Automation Conference (DAC)*. 1–6.

[16] David Gibson, Ravi Kumar, and Andrew Tomkins. 2005. Discovering Large Dense Subgraphs in Massive Graphs. In *Proceedings of the 31st International Conference on Very Large Data Bases* (Trondheim, Norway) *(VLDB '05)*. VLDB Endowment, 721–732.

[17] Huahai He and Ambuj K Singh. 2008. Graphs-at-a-time: query language and access methods for graph databases. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*. 405–418.

[18] Loc Hoang, Vishwesh Jatala, Xuhao Chen, Udit Agarwal, Roshan Dathathri, Gurbinder Gill, and Keshav Pingali. 2019. DistTC: High performance distributed triangle counting. In *2019 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 1–7.

[19] Sungpack Hong, Tayo Oguntebi, and Kunle Olukotun. 2011. Efficient Parallel Graph Exploration on Multi-Core CPU and GPU. In *2011 International Conference on Parallel Architectures and Compilation Techniques*. 78–88. https://doi.org/10.1109/PACT.2011.14

[20] Xin Hu, Tzi-cker Chiueh, and Kang G. Shin. 2009. Large-Scale Malware Indexing Using Function-Call Graphs. In *Proceedings of the 16th ACM Conference on Computer and Communications Security* (Chicago, Illinois, USA) *(CCS '09)*. Association for Computing Machinery, New York, NY, USA, 611–620. https://doi.org/10.1145/1653662.1653736

[21] Xiaocheng Hu, Yufei Tao, and Chin-Wan Chung. 2013. Massive graph triangulation. In *Proceedings of the 2013 ACM SIGMOD international conference on Management of data*. 325–336.

[22] Xiaocheng Hu, Yufei Tao, and Chin-Wan Chung. 2014. I/O-efficient algorithms on triangle listing and counting. *ACM Transactions on Database Systems (TODS)* 39, 4 (2014), 1–30.

[23] Yang Hu, Hang Liu, and H Howie Huang. 2018. Tricore: Parallel Triangle Counting on GPUs. In *SC*. IEEE, 171–182.

[24] Sitao Huang, Mohamed El-Hadedy, Cong Hao, Qin Li, Vikram S Mailthody, Ketan Date, Jinjun Xiong, Deming Chen, Rakesh Nagi, and Wen-mei Hwu. 2018. Triangle Counting and Truss Decomposition using FPGA. In *HPEC*. IEEE, 1–7.

[25] H. V. Jagadish. 1990. A Compression Technique to Materialize Transitive Closure. *ACM Trans. Database Syst.* 15, 4 (dec 1990), 558–598. https://doi.org/10.1145/99935.99944

[26] Jérôme Kunegis. 2013. Konect: the koblenz network collection. In *Proceedings of the 22nd international conference on world wide web*. 1343–1350.

[27] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. 2010. What is Twitter, a Social Network or a News Media?. In *WWW*. 591–600.

[28] Longbin Lai, Lu Qin, Xuemin Lin, and Lijun Chang. 2015. Scalable subgraph enumeration in mapreduce. *Proceedings of the VLDB Endowment* 8, 10 (2015), 974–985.

[29] Longbin Lai, Zhu Qing, Zhengyi Yang, Xin Jin, Zhengmin Lai, Ran Wang, Kongzhang Hao, Xuemin Lin, Lu Qin, Wenjie Zhang, et al. 2019. Distributed subgraph matching on timely dataflow. *Proceedings of the VLDB Endowment* 12, 10 (2019), 1099–1112.

[30] Jure Leskovec, Deepayan Chakrabarti, Jon Kleinberg, Christos Faloutsos, and Zoubin Ghahramani. 2010. Kronecker graphs: an approach to modeling networks. *Journal of Machine Learning Research* 11, 2 (2010).

[31] Hongjun Li, Fanyu Kong, and Jia Yu. 2021. Secure Outsourcing for Normalized Cuts of Large-scale Dense Graph in Internet of Things. *IEEE Internet of Things Journal* (2021), 1–1. https://doi.org/10.1109/JIOT.2021.3138103

[32] Pedro G Lind, Marta C Gonzalez, and Hans J Herrmann. 2005. Cycles and clustering in bipartite networks. *Physical review E* 72, 5 (2005), 056127.

[33] Chenhao Liu, Zhiyuan Shao, Kexin Li, Minkang Wu, Jiajie Chen, Ruoshi Li, Xiaofei Liao, and Hai Jin. 2021. ScalaBFS: A Scalable BFS Accelerator on FPGA-HBM Platform. In *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (Virtual Event, USA) *(FPGA '21)*. Association for Computing Machinery, New York, NY, USA, 147. https://doi.org/10.1145/3431920.3439463

[34] Bingqing Lyu, Lu Qin, Xuemin Lin, Ying Zhang, Zhengping Qian, and Jingren Zhou. 2020. Maximum biclique search at billion scale. *Proceedings of the VLDB Endowment* (2020).

[35] Lingxiao Ma, Zhi Yang, Youshan Miao, Jilong Xue, Ming Wu, Lidong Zhou, and Yafei Dai. 2019. NeuGraph: Parallel Deep Neural Network Computation on Large Graphs. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. USENIX Association, Renton, WA, 443–458. https://www.usenix.org/conference/atc19/presentation/ma

[36] Son T. Mai, Martin Storgaard Dieu, Ira Assent, Jon Jacobsen, Jesper Kristensen, and Mathias Birk. 2017. Scalable and Interactive Graph Clustering Algorithm on Multicore CPUs. In *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*. 349–360. https://doi.org/10.1109/ICDE.2017.94

[37] Ine Melckenbeeck, Pieter Audenaert, Thomas Van Parys, Yves Van De Peer, Didier Colle, and Mario Pickavet. 2019. Optimising orbit counting of arbitrary order by equation selection. *BMC bioinformatics* 20, 1 (2019), 1–13.

[38] Alan Mislove, Massimiliano Marcon, Krishna P Gummadi, Peter Druschel, and Bobby Bhattacharjee. 2007. Measurement and analysis of online social networks. In *Proceedings of the 7th ACM SIGCOMM conference on Internet measurement*. 29–42.

[39] Amir Hossein Nodehi Sabet, Junqiao Qiu, and Zhijia Zhao. 2018. Tigr: Transforming Irregular Graphs for GPU-Friendly Graph Processing. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems* (Williamsburg, VA, USA) *(ASPLOS '18)*. Association for Computing Machinery, New York, NY, USA, 622–636. https://doi.org/10.1145/3173162.3173180

[40] Rasmus Pagh and Francesco Silvestri. 2014. The input/output complexity of triangle enumeration. In *Proceedings of the 33rd ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*. 224–233.

[41] Santosh Pandey, Zhibin Wang, Sheng Zhong, Chen Tian, Bolong Zheng, Xiaoye Li, Lingda Li, Adolfy Hoisie, Caiwen Ding, Dong Li, et al. 2021. TRUST: Triangle Counting Reloaded on GPUs. *IEEE Transactions on Parallel and Distributed Systems* 32, 11 (2021), 2646–2660.

[42] Ha-Myung Park and Chin-Wan Chung. 2013. An efficient mapreduce algorithm for counting triangles in a very large graph. In *Proceedings of the 22nd ACM international conference on Information & Knowledge Management*. 539–548.

[43] Ali Pinar, C Seshadhri, and Vaidyanathan Vishal. 2017. Escape: Efficiently counting all 5-vertex subgraphs. In *Proceedings of the 26th international conference on world wide web*. 1431–1440.

[44] Zhengping Qian, Chenqiang Min, Longbin Lai, Yong Fang, Gaofeng Li, Youyang Yao, Bingqing Lyu, Xiaoli Zhou, Zhimin Chen, and Jingren Zhou. 2021. GAIA: A System for Interactive Analysis on Distributed Graphs Using a High-Level Language. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. USENIX Association, 321–335. https://www.usenix.org/conference/nsdi21/presentation/qian-zhengping

[45] Garry Robins and Malcolm Alexander. 2004. Small worlds among interlocking directors: Network structure and distance in bipartite graphs. *Computational & Mathematical Organization Theory* 10, 1 (2004), 69–94.

[46] Ryan A. Rossi and Nesreen K. Ahmed. 2015. The Network Data Repository with Interactive Graph Analytics and Visualization. In *AAAI*. https://networkrepository.com

[47] Seyed-Vahid Sanei-Mehri, Ahmet Erdem Sariyuce, and Srikanta Tirthapura. 2018. Butterfly counting in bipartite networks. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 2150–2159.

[48] Seyed-Vahid Sanei-Mehri, Yu Zhang, Ahmet Erdem Sariyüce, and Srikanta Tirthapura. 2019. FLEET: butterfly estimation from a bipartite graph stream. In *Proceedings of the 28th ACM International Conference on Information and Knowledge Management*. 1201–1210.

[49] Jessica Shi and Julian Shun. 2020. Parallel algorithms for butterfly computations. In *Symposium on Algorithmic Principles of Computer Systems*. SIAM, 16–30.

[50] Xiaoyuan Su and Taghi M. Khoshgoftaar. 2009. A Survey of Collaborative Filtering Techniques. *Adv. in Artif. Intell.* 2009, Article 4 (Jan. 2009), 1 pages. https://doi.org/10.1155/2009/421425

[51] Leslie G. Valiant. 1990. A Bridging Model for Parallel Computation. *Commun. ACM* 33, 8 (aug 1990), 103–111. https://doi.org/10.1145/79173.79181

[52] Midhul Vuppalapati, Justin Miron, Rachit Agarwal, Dan Truong, Ashish Motivala, and Thierry Cruanes. 2020. Building An Elastic Query Engine on Disaggregated Storage. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. USENIX Association, Santa Clara, CA, 449–462. https://www.usenix.org/conference/nsdi20/presentation/vuppalapati

[53] Jia Wang, Ada Wai-Chee Fu, and James Cheng. 2014. Rectangle counting in large bipartite graphs. In *2014 IEEE International Congress on Big Data*. IEEE, 17–24.

[54] Kai Wang, Yiheng Hu, Xuemin Lin, Wenjie Zhang, Lu Qin, and Ying Zhang. 2021. A Cohesive Structure Based Bipartite Graph Analytics System. In *Proceedings of the 30th ACM International Conference on Information & Knowledge Management*. 4799–4803.

[55] Kai Wang, Xuemin Lin, Lu Qin, Wenjie Zhang, and Ying Zhang. 2019. Vertex priority based butterfly counting for large-scale bipartite networks. *Proceedings of the VLDB Endowment* 12, 10 (2019), 1139–1152.

[56] Kai Wang, Xuemin Lin, Lu Qin, Wenjie Zhang, and Ying Zhang. 2022. Accelerated butterfly counting with vertex priority on bipartite graphs. *The VLDB Journal* (2022), 1–25.

[57] Kai Wang, Xuemin Lin, Lu Qin, Wenjie Zhang, and Ying Zhang. 2022. Towards efficient solutions of bitruss decomposition for large-scale bipartite graphs. *The VLDB Journal* 31, 2 (2022), 203–226.

[58] Kai Wang, Wenjie Zhang, Ying Zhang, Lu Qin, and Yuting Zhang. 2021. Discovering significant communities on bipartite graphs: An index-based approach. *IEEE Transactions on Knowledge and Data Engineering* (2021).

[59] Yangzihao Wang, Andrew Davidson, Yuechao Pan, Yuduo Wu, Andy Riffel, and John D. Owens. 2016. Gunrock: A High-Performance Graph Processing Library on the GPU. *SIGPLAN Not.* 51, 8, Article 11 (Feb. 2016), 12 pages. https://doi.org/10.1145/3016078.2851145

[60] Zhengyi Yang, Longbin Lai, Xuemin Lin, Kongzhang Hao, and Wenjie Zhang. 2021. Huge: An efficient and scalable subgraph enumeration system. In *Proceedings of the 2021 International Conference on Management of Data*. 2049–2062.

[61] Abdurrahman Yaşar, Sivasankaran Rajamanickam, Jonathan Berry, Michael Wolf, Jeffrey S Young, and Ümit V ÇatalyÜrek. 2019. Linear Algebra-Based Triangle Counting via Fine-Grained Tasking on Heterogeneous Environments: (Update on Static Graph Challenge). In *HPEC*. 1–4.

[62] Hao Zhang, Jeffrey Xu Yu, Yikai Zhang, Kangfei Zhao, and Hong Cheng. 2020. Distributed subgraph counting: a general approach. *Proceedings of the VLDB Endowment* 13, 12 (2020), 2493–2507.

[63] Jialiang Zhang and Jing Li. 2018. Degree-Aware Hybrid Graph Traversal on FPGA-HMC Platform. In *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (Monterey, CALIFORNIA, USA) *(FPGA '18)*. Association for Computing Machinery, New York, NY, USA, 229–238. https://doi.org/10.1145/3174243.3174245

[64] Chen Zhao and Yong Guan. 2015. A GRAPH-BASED INVESTIGATION OF BITCOIN TRANSACTIONS. In *Advances in Digital Forensics XI*, Gilbert Peterson and Sujeet Shenoi (Eds.). Springer International Publishing, Cham, 79–95.

[65] Gengda Zhao, Kai Wang, Wenjie Zhang, Xuemin Lin, Ying Zhang, and Yizhang He. 2022. Efficient Computation of Cohesive Subgraphs in Uncertain Bipartite Graphs. In *2022 IEEE 38th International Conference on Data Engineering (ICDE)*. IEEE, 2333–2345.

[66] Alexander Zhou, Yue Wang, and Lei Chen. 2021. Butterfly counting on uncertain bipartite graphs. *Proceedings of the VLDB Endowment* 15, 2 (2021), 211–223.

[67] Qiuyu Zhu, Jiahong Zheng, Hao Yang, Chen Chen, Xiaoyang Wang, and Ying Zhang. 2020. Hurricane in Bipartite Graphs: The Lethal Nodes of Butterflies. In *32nd International Conference on Scientific and Statistical Database Management* (Vienna, Austria) *(SSDBM 2020)*. Association for Computing Machinery, New York, NY, USA, Article 18, 4 pages. https://doi.org/10.1145/3400903.3400916

[68] Rong Zhu, Zhaonian Zou, and Jianzhong Li. 2018. Fast rectangle counting on massive networks. In *2018 IEEE International Conference on Data Mining (ICDM)*. IEEE, 847–856.

Received April 2022; revised July 2022; accepted August 2022