Noname manuscript No.
(will be inserted by the editor)

Parallelization of Butterfly Counting on Hierarchical Memory

Zhibin Wang · Longbin Lai · Yixue Liu · Bing Shui · Chen Tian · Sheng Zhong

Received: date / Accepted: date

Abstract 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 this 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 $\Omega(\min(\frac{|E|^2}{MB}, \frac{|E||V|}{\sqrt{MB}}))$. Subsequently, we develop the IOBufs algorithm that manages to approach the I/O lower bound, and thus claim its optimality. Finally, we investigate the parallelization of IOBufs to improve its performance and scalability. To support various hardware configurations, we introduce a general parallel framework, PIOBufs. Our analysis indicates that the key to implementing PIOBufs on multi-core CPUs lies in the fine-grained task division. Furthermore, we extend the CPU-tailored PIOBufs to harness the extensive parallelism that GPUs provide. Our experimental results show that IOBufs performs better than established algorithms such as EMRC, BFC-EM and G-BFC. Thanks to its I/O-efficient design, IOBufs can handle large graphs that exceed the main memory capacity on both CPUs and GPUs. A significant result is that IOBufs can manage butterfly counting on the Clueweb graph, which has 37 billion edges and quintillions (10^{18}) of butterflies.

Keywords Graph; butterfly counting; hierarchical memory; parallel algorithm; GPGPU

Z. Wang, Y. Liu, B. Shui, C. Tian, S. Zhong
Nanjing University, State Key Laboratory for Novel Software Technology, Nanjing, Jiangsu, China
E-mail: {wzbwangzhibin, sheng.zhong}@gmail.com, {mf20330052,191098191}@smail.nju.edu.cn, tianchen@nju.edu.cn

L. Lai
Alibaba Group, Hangzhou, Zhejiang, China
E-mail: Longbin.lailb@alibaba-inc.com

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 [78,80, 77,76] on a bipartite graph [41,61,3,97], where vertices can be divided into two disjoint sets, and edges exist only between the two 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 [19,81,82], recommendation systems [70], word-document clustering [16], research group identification [15], and link prediction according to transitivity theory [11]. Recently, Lyu et al. [46] 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

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

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.

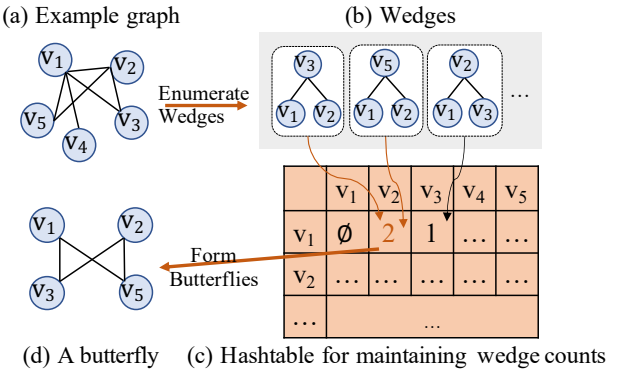


Fig. 1: An example of butterfly counting.

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 architecture brings in new opportunities. Regarding processing capacity, modern multi-core CPUs [24,48,67], GPUs [83,47,54], and FPGAs [92,18,42] 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. More recent work in the cloud [74] 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

We review existing methods for butterfly counting and their weakness in three settings: in-memory sequential, hierarchical-memory, and parallel.

In-memory sequential setting. ExactBFC [65] is a sequential algorithm that optimizes butterfly counting by using vertex priority to avoid duplicate calculations. However, the sequential algorithm cannot fully utilize the processing capacity of modern hardware and can not scale to large graphs. Note that the primary computation of butterfly counting is to count the wedges (butterflies) between all pairs of vertices, and we denote the task as $T_{wc}\{V \times V\}$.

Hierarchical-memory setting. BFC-EM [78] and EMRC [98] 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, producing a total of p^2 tasks for the wedge counting between each pair of subgraphs. The authors proved that no *witnessing* algorithm for butterfly counting could guarantee $o(\frac{|E|^2}{MB})$ I/Os. Here, a witnessing algorithm [28] 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.

Parallel setting. Given the core task of $T_{wc}\{V \times V\}$ for butterfly counting, it is straightforward to parallelize the subtasks of $T_{wc}\{\{u\} \times V\}$ for $\forall u \in V$, as proposed by BFC-VP++ [78]. While BFC-VP++ is specifically designed to optimize memory usage through reuse, it still demands a space complexity of $O(|E| + W|V|)$ given W working threads. When adapting this algorithm to modern hardware, especially GPUs that often deploy thousands of threads, memory constraints become pronounced. In the context of GPUs, G-BFC [86] was introduced to enhance butterfly counting by leveraging the massive parallelism of these devices. However, G-BFC still faces challenges related to high memory consumption – a significant concern given the generally smaller memory capacity of GPUs. Additionally, while G-BFC

aims to optimize for the hierarchical parallelism of a GPU, it inadvertently incorporates algorithms with increased time complexity.

It is important to highlight that existing algorithms for hierarchical memory have been exclusively designed for sequential execution. In contrast, all existing parallel algorithms are confined to a main memory environment. Currently, there is an absence of a solution that addresses butterfly counting in a hierarchical-memory context with parallelism. In this work, we bridge this gap, significantly enhancing the scalability of butterfly counting and tapping into the full capacity of modern multi-core processors.

1.2 Our contributions

In this paper, we study *I/O-efficient* and *parallel* butterfly counting algorithms on static graphs. Specifically, we consider the hardware context that embodies 1) a hierarchical memory configuration, in which the main memory size satisfies $M = o(|E|)$ and the secondary memory is arbitrarily large; 2) a shared-memory multi-core processor, e.g., multi-core CPUs and GPUs.

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 the 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 $\Omega(\min(\frac{|E||V|}{\sqrt{M}B}, \frac{|E|^2}{MB}))$. 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 [98]. 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) [40], software function calls [26], transitive closure graph [31], cryptocurrency graph [93], and brain neural network [62].

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 $O(\min(\frac{|E||V|}{\sqrt{M}B}, \frac{|E|^2}{MB}))$ 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 third contribution is the proposal of a parallelization framework PIOBufs in order to support various hardware configurations. PIOBufs includes `TaskDivider`, `TaskScheduler` and `TaskRunner`. In the parallel context, given that each worker may need to maintain its own state for the task assigned by `TaskScheduler`, the algorithm can consume more memory than a sequential counterpart. As an example, BFC-VP++ consumes $O(|E| + W|V|)$ space when there are W workers each corresponding to a working thread. In a hierarchical-memory environment, the data needs further partition to accommodate such increased memory demands. This increases the I/O costs and potentially breaks the I/O optimality. We recognize the underlying issue with most straightforward methods (including BFC-VP++) is their reliance on *coarse-grained* parallelism. In these methods, the algorithm tries to parallelize relatively large subtasks produced by `TaskDivider`. To address this, we introduce FG, a more fine-grained solution based on the PIOBufs framework. Its `TaskDivider` breaks the subtask into smaller pieces. However, we are also aware that very fine granularity can amplify the scheduling overhead, as pointed out in [60]. As a result, in FG, we strike a balance between I/O efficiency and the granularity of parallelism.

Our final contribution involves adapting the FG method to GPUs, which inherently support a high degree of parallelism (e.g., encompassing hundreds of thousands of threads). To fully utilize the processing potential of GPUs, it is vital to account for hierarchical parallelism and memory access patterns. Observing that only "warps" and "blocks" are suitable candidates for serving as workers in FG, we propose two distinct kernels, i.e., FG-BaaW and FG-WaaW. The FG-BaaW kernel employs a block as its worker (BaaW). This design choice is made to ensure the efficient handling of *large* workloads by harnessing the parallel processing capability of numerous threads within a block. Moreover, strategic optimizations are incorporated to enhance the memory-access efficiency associated with FG-BaaW. On the other hand, the FG-WaaW kernel assigns a warp as its worker (WaaW), tailored for *small* workloads. The FG-WaaW allows a larger number of workers to operate in parallel, which may cause a surge of memory consumption as mentioned earlier. This leads to the development of a novel butterfly-counting algorithm grounded in the routine of "merge-sort". This method negates the necessity of extra memory required by the hashtable, instead harnessing warp-level primitives for efficiently counting wedges. By adaptively selecting between FG-BaaW and FG-WaaW, our proposed

algorithm outperforms the state-of-the-art G-BFC [86] by a factor of 30×.

1.3 Roadmap

The rest of the paper is organized as follows. Section 2 reviews the related works. Preliminary is presented in Section 3. Our primary focus is on handling the large-scale graphs in the hierarchical-memory setting. Existing hierarchical-memory solutions are given in Section 4, in which we revisit the I/O complexity of EMRC. Observing that existing solutions suffer from high I/O costs, we establish a new, more efficient lower bound anchored in the principle of semi-witnessing algorithms in Section 5. Based on the semi-witnessing algorithm, we develop two variants of IOBufs in Section 6 in order to approach the I/O lower bound. After achieving the goal of minimizing I/O costs, we focused on boosting performance by parallelizing the algorithm in Section 7. Our studies on multi-core CPUs highlight the importance of a "fine-grained" solution (Section 7.2) that allows for massive parallelism without compromising the efficiency of the algorithm regarding I/Os. Building on this "fine-grained" solution, we have developed algorithms to effectively utilize GPU technology for processing large-scale butterfly counting tasks in Section 8. Section 9 sets the experimental configurations, while Section 10 and Section 11 report the experimental results on CPUs and GPUs, respectively. Finally, Section 13 concludes the paper.

2 Related Work

2.1 Motif counting and subgraph matching

Motifs are small subgraphs in the data graph, and thus existing approaches for subgraph matching can be utilized for motif counting. There are two mainstream subgraph matching approaches: backtracking-based [7,21, 6] and join-based [87,36,5,37]. These approaches were mainly developed for matching/enumerating subgraphs in general, and might not be the most suitable for butterfly counting. Recent literature [91,50,58,12] considered counting graphlets, i.e., all motifs up to k vertices, while butterfly counting is trivially a part of 4-vertex graphlet counting. Notice, that 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. Nevertheless, there are several works dedicated to counting small motifs, which explicitly develop algorithms tailored for each type of motif, including the butterfly. For instance, instead of setting vertex priority to avoid duplication [78], Pinar et al. [58] incorporate a graph orientation optimization that removes half of the edges by orienting edges in a degeneracy order. To prevent the omission of potential butterflies, they enumerate both out-wedges and inout-wedges, forming all possible acyclic orientations of a butterfly by two kinds of wedges. Recently, GraphSet [68] further optimizes graph mining through equivalent set transformation, which aims to eliminate most control flow and reduce computation overhead. 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, [22] was developed by synchronizing the intermediate states to list the triangles. The algorithm of [57] partitioned the graphs with replication in order to avoid communication. Modern hardware is also leveraged for counting triangles. The results of triangle counting in [56,29,30] have demonstrated the great potential of GPUs. With multiple load-balancing techniques, a recent work of [56] managed to scale the task to as many as 1024 GPU cards. Huang et al. considered triangle counting on FPGAs in [30] for better energy efficiency. In [88], the authors accelerated the intersection operation for triangle counting using both CPUs and GPUs. In the hierarchical-memory setting, [55,27] 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 [67] 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 [67], the most-optimized batching variant of ParButterfly is actually the parallel BFC-VP++ [78] 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. For instance, [65] 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 approx-

imate algorithm with that of our IOBufs. On the same dataset Journal (Table 5), the authors reported in Figure 6(b) of [65] 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 10) 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 [66] 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, [96,94] considered counting butterflies on uncertain graphs, in which each edge has a probability of being present. Recently, G-BFC [86] further accelerated butterfly counting via GPUs.

Optimization on bipartite graphs. Butterfly counting is particularly prevalent in bipartite graphs, such as buyer-product networks [95,100]. Unlike unipartite graphs where triangles are common, in bipartite graphs, butterflies are the primary patterns of interest since there are no triangles. Consequently, butterflies in bipartite graphs are analogous to triangles in unipartite graphs. Researchers also try to optimize the butterfly counting based on the nature of the bipartite graph. [65] chooses the vertex side with the smaller complexity to count the butterfly. Recently, [90] proposes one-side sampling, a method capitalizing on the observation that the leaf vertices of a wedge in a bipartite graph belong to the same partition. Again, our work is orthogonal to these optimizations and can incorporate them when dealing with bipartite input graphs.

2.4 Graph processing systems

Researchers have developed flexible graph processing systems to simplify the implementation of intricate graph algorithms. These systems frequently embrace a high-level programming model, notably the vertex-centric model and its variants [49,69,20]. Moreover, advancements have been made to adapt these systems for hierarchical memory architectures [35,64,63,99,72,73] and GPU utilization [83,32,54,85,59,23,29], showcasing their adaptability across diverse computing environments. These graph processing systems are predominantly tailored for graph algorithms, where each vertex maintains a state of constant space and its computation is bound to immediate neighbors. In contrast, butterfly counting demands that each vertex records the wedge count, resulting in $O(|V|^2)$ ($|V|$ is the number of vertices in the graph) space requirements and necessitating access to 2-hop neighbors. This divergence makes generic graph processing systems inefficient for butterfly counting tasks and thus motivates our development of IOBufs and its parallelization.

We will show in Section 7.1 that the straightforward implementation of butterfly counting using the high-level programming model in graph processing systems actually mirrors the alternative BSP-S and BSP-SN solutions, which are less efficient compared to our FG.

Regarding the utilization of hierarchical memory, X-Stream [64] streams the edges from disk to memory (a.k.a. semi-streaming), while GridGraph [99] partitions the graph into small subgraphs to fit into the memory. These ideas also inspire the design of IOBufs, which partitions the graph into subgraphs to fit into the memory and streaming loads the edges from disk to memory to reduce space usage. However, IOBufs further partitions the graph with more considerations on wedges in addition to solely on edges considered in GridGraph. In addition to the concept of semi-streaming, IOBufs also needs to enforce proper loading sequence of edges in the streaming process for efficient wedge enumeration.

When it comes to GPU optimization, techniques for coalesced memory access [85,32,54,83], shared memory utilization [56,29,44,43], and subwarp parallelism [59, 23] are widely used. Existing systems leverage coalesced memory access primarily to optimize direct neighbor access. In contrast, IOBufs must consider 2-hop neighbor access, necessitating more complex optimization strategies. Similarly, while shared memory in these systems is typically allocated for managing simple vertex states such as integers [43], IOBufs requires a nuanced approach to efficiently handle wedge count maintenance, highlighting its unique optimization challenges.

3 Background

3.1 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,

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

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

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 $\mathcal{H}\{\mathcal{P} \to \mathbb{N}\}$ 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.

3.2 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. We follow [78] to deduplicate by using vertex priority. In this paper, we will not dive into the technique, and refer interested readers to [78] 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 $\mathcal{H}\{V \times V \to \mathbb{N}\}$
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 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 added 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.

3.3 GPU architecture

The massive parallelism and high-bandwidth memory capabilities of GPU make it the ideal hardware for accelerating butterfly counting. To fully exploit the potential of GPU, it is crucial to leverage the hierarchical parallelism and the memory configuration for programming GPU (more specifically, CUDA [2]) kernels, as depicted in Figure 2. We briefly introduce the architecture in the following.

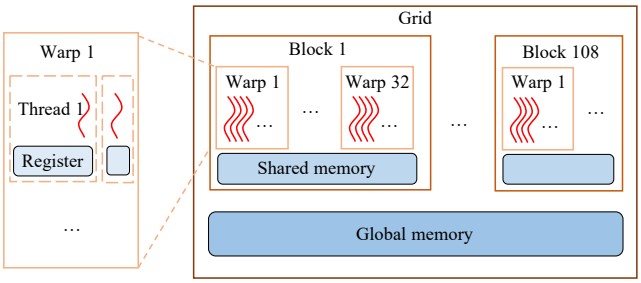


Fig. 2: Hierarchical parallelism and the corresponding memory configuration on GPUs.

Thread. The smallest unit of execution, a thread, represents the lowest level of parallelism. A GPU core ex-

ecutes each thread, which has access to its registers for ongoing data processing.

Block. A block, or a thread block, consists of a group of threads that are assigned to a Stream Multiprocessor (SM) for execution. Each SM is equipped with shared memory, enabling all threads within the block to cache data for efficient data interchange. A synchronization primitive named `__syncthreads()` allows explicit synchronization amongst the threads in a block. It is also crucial to mention **Warp** in the hierarchy, which is a group of typically 32 threads in a thread block that simultaneously execute the same instruction following a Single Instruction Multiple Threads (SIMT) pattern, and synchronization happens implicitly. Despite having no physical memory configuration, GPUs provide high-performance warp-level primitives for data exchange among threads within a warp.

Grid. A grid represents the highest level of parallelism. It is a collection of blocks working collectively to process a larger workload, harnessing the complete computational power of a GPU device. In our hierarchical memory model, the global memory allocated to all threads in a grid can act as the primary memory, while the CPU memory can function as secondary memory to handle overflowed data.

4 Butterfly Counting on Hierarchical Memory

As the graph becomes large, people have studied butterfly counting on the hierarchical memory, 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.

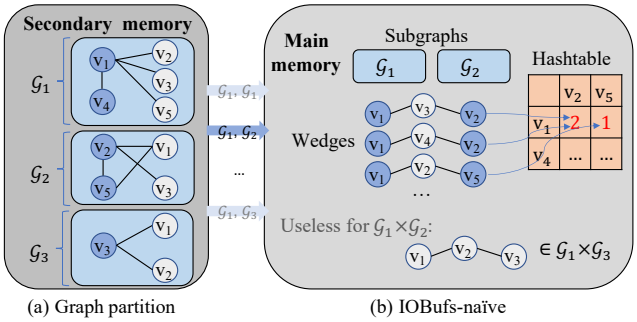


Fig. 3: The execution of IOBufs-Naïve on hierarchical memory.

4.1 Existing solutions

Semi-streaming [35,64,63,99] is a popular technique for processing large graphs on hierarchical memory. It maintains partial data (e.g., vertex states) in the memory and streaming loads the edges from the secondary memory. Regarding butterfly counting, BFC-EM [78] follows the semi-streaming model, in which it streaming loads the edges and conducts wedge computing sequentially for loaded edges. The wedges will be constantly spilled to the secondary memory 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 9) that BFC-EM can barely benefit from more memory. Alternatively, the authors developed EMRC and proved its "I/O optimality" in [98]. 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++ [78] 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

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

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 10.6. Given that we must fit the largest partition in the main memory, the imbalanced partition may result in a larger p, consequently increasing the 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 configuring 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 1 Figure 3(a) demonstrates partitioning the graph in Figure 1 into three parts, and Figure 3(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 [98] 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 increases the 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 [27]. 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 [98]. 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 [98], 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 $\Lambda = O(|E|^{1.5})$ [13] and $M = o(|E|)$, this is tighter than $O(\frac{|E|^2}{\sqrt{M}})$ as given in the paper [98].

Remark 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 the partitioning of the graph.

Particularly for EMRC, if we configure $p = \lceil \frac{c_1|E|}{M} \rceil$ according to the paper [98], 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. [98] derived $\Omega(\frac{|E|^2}{MB})$ as the I/O lower bound of butterfly counting based on the *witnessing* algorithm [28]. 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 [98] 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 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''.

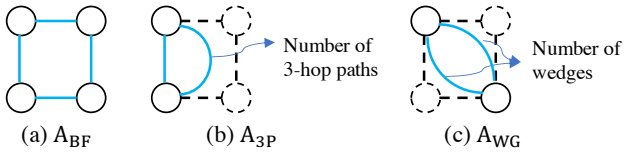


Fig. 4: Three types of semi-witnessing algorithms for butterfly counting, in which the vertices/edges outlined with dotted lines are nonessential.

Obviously, A_g is a witnessing algorithm.

As shown in Figure 4, 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 4. The following lemma rules out their existence.

Lemma 1 *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 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:

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

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

This is the case where Zhu et al. [98] 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 2 *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

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

Theorem 1 *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 2, 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, some butterflies may be 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 [28], 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 [98].

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 3 *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*

$\Lambda_{\mathcal{E}}[\mathcal{H}] \leq 2\sqrt{|\mathcal{H}|}|\mathcal{E}|$.

Proof

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

Given any $v \in V$, on the one hand, they 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,

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

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 4 *In the wedge-only case, no A_{WG} algorithm for butterfly counting can guarantee $o(\frac{|E||V|}{\sqrt{M}B})$ 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 3, at most $O(\sqrt{M}B)$ 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 $\Omega(\frac{|E||V|}{\sqrt{M}B})$.

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 2 *No A_{WG} algorithm for butterfly counting can guarantee $o(\frac{|E||V|}{\sqrt{M}B})$ 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 3, $O(\sqrt{M}(M + B))$ 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 1, 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 $O(\sqrt{M}B)$ by setting $s = O(\frac{M}{B})$. Considering that the number of wedges can be as many as $\Theta(|E||V|)$, we derive the I/O bound of A_{WG} as $\Omega(\frac{|E||V|}{\sqrt{M}B})$ in the general wedge-edge-shared case.

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

Theorem 3 *No A_{3P} algorithm for butterfly counting can guarantee $o(\frac{|E||V|}{\sqrt{M}B})$ 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 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 $o(\frac{|E||V|}{\sqrt{M}B})$ by Theorem 2.

According to the above analysis, the theorem holds.

5.5 Final result and discussions

Theorem 4 *No semi-witnessing algorithm for butterfly counting can guarantee $o(\min(\frac{|E|^2}{MB}, \frac{|E||V|}{\sqrt{M}B}))$ I/Os.*

Proof Recall from the discussions in Lemma 1 and Figure 4 that any semi-witnessing algorithm for butterfly counting must belong to either A_{BF}, A_{3P} or A_{WG}. Thus, this theorem holds by summarizing Theorem 1, Theorem 2, and Theorem 3.

The result in Theorem 4 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.

6 The I/O-optimal Algorithm

Based on Algorithm 2, we design our IOBufs that can achieve the I/O lower bound in Theorem 4. 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 1), and they can also approach the I/O lower bound of Theorem 1 and Theorem 2, 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 4.

6.1 The key observation

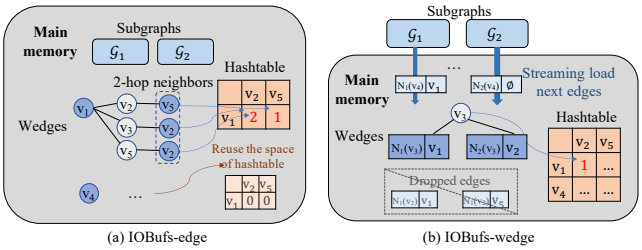
As we have discussed in Section 4.3, the main reason that Algorithm 3 cannot achieve I/O optimality is that it must maintain both edges and wedges in the main memory, which, however, is not a necessity according to a key observation:

> 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 directly 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, which is also called *semi-streaming* [52]. However, semi-streaming in general graph processing systems [35,64,63,99,72,73] primarily focuses on simple vertex states (e.g., integer) and edges, while in butterfly counting, the main research subject shifts to edges and wedges. In general graph processing systems, the space cost of edges is much more than vertex states, which makes these systems focus on optimizing the storage and memory access of edges. In contrast, regarding butterfly counting, the importance of edges and wedges may vary in different input graphs, 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. 5: 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 2 As illustrated in Figure 5(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 sight, 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 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 1. 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 3 Figure 5(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 the 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 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 2. 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:

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

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 $\overline{d} < c_3\sqrt{M}$, and use IOBufs-wedge otherwise. Therefore, we have IOBufs adaptively configured as:

Algorithm 6 IOBufs, the adaptive variant

1: **if** $\overline{d} < c_3\sqrt{M}$ **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 4, 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

After the I/O cost is minimized, the next step is to parallelize the algorithm to further leverage the massive parallelism supported by modern hardware. 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 $\mathcal{H}$. In the following, we will focus on discussing the non-trivial parallel techniques for IOBufs-edge.

Let us represent $T_{wc}\{V \times V\}$ as the process of calculating the wedges between all pairs of vertices, a core operation in butterfly counting as mentioned before. In addition, we take into account W workers in the runtime environment to execute the task. To facilitate the discussion of various solutions for parallelizing IOBufs-edge across varied hardware types, including multi-core CPUs and GPUs, we introduce a generic parallel framework named PIOBufs. The framework can be adapted to different scenarios using three critical interfaces, namely `TaskDivider`, `TaskScheduler`, and `TaskRunner`.

- `TaskDivider` handles dividing the main task ($T_{wc}\{V \times V\}$) into subtasks that can be processed by the workers in parallel. We denote the subtask as $T_{wc}\{\mathcal{V}_1 \times \mathcal{V}_2\}$ for any non-empty $\mathcal{V}_1 \subseteq V$ and $\mathcal{V}_2 \subseteq V$. Given that each worker might require maintaining memory space for processing the subtask, it is vital to carefully divide the subtasks and distribute subtasks among the workers without compromising the I/O efficiency of the algorithm.
- `TaskScheduler` involves determining the order in which the subtasks should be executed by the workers. This scheduling should take into account factors such as the availability of workers and the execution order of subtasks.
- `TaskRunner` focuses on how each subtask should be programmed to run on a worker. The programming should leverage any device-specific techniques or op-

timizations to achieve high performance, such as the hierarchical architecture of GPUs.

With the uniform framework, we can discuss the key factors to consider when implementing PIOBufs across various hardware configurations.

Adapting PIOBufs to sequential I/O-optimal algorithm. When operating with a single worker, the I/O-optimal algorithm discussed in the previous section can be seamlessly integrated into PIOBufs by following implementations:

- `TaskDivider`: According to Algorithm 2, divide the task $T_{wc}\{V \times V\}$ into subtasks of $T_{wc}\{\mathcal{V}_i \times \mathcal{V}_j\}$, with $\mathcal{V}_i$ and $\mathcal{V}_j$ denoting the vertex sets of the two partitioned graphs.
- `TaskScheduler`: Sequentially schedule the subtasks to the worker.
- `TaskRunner`: The worker directly calls the IOBufs-edge (Algorithm 4) to process $T_{wc}\{\mathcal{V}_i \times \mathcal{V}_j\}$ for a pair of partitioned subgraphs $(\mathcal{G}_i, \mathcal{G}_j)$.

Multi-core CPUs. In the upcoming subsections of Section 7.1 and Section 7.2, we delve into CPU environments, where it's straightforward to assign each worker to a CPU thread. In this context, the role of `TaskRunner` becomes trivial, steering our focus towards `TaskDivider` and `TaskScheduler`. A few candidate CPU parallelization solutions have been explored, including Naïve, which is a straightforward extension of the sequential algorithm of IOBufs; BSP-S, which can be recognized as an implementation of parallel butterfly counting upon the graph processing system [35,64,63, 99,72,73,69]; and BSP-SN, which actually represents existing parallel butterfly counting algorithms [78,67]. Recognizing the constraints of existing methods stemming from their coarse-grained task division, we introduce FG, which features a `TaskDivider` designed to dissect the task of butterfly counting into more fine-grained subtasks. FG lays the groundwork for the efficient parallelization of IOBufs in this paper.

GPUs. In the forthcoming Section 8, building directly on the `TaskDivider` and `TaskScheduler` implementations from FG, we explore a two-layer parallelism strategy for the `TaskRunner` within the GPU setting. This strategy is specifically engineered to capitalize on the GPU's massive parallelism, aiming for optimal utilization of its hundreds of thousands of working threads to enhance processing efficiency.

7.1 CPU parallelization: alternative solutions

We present three alternative parallelization solutions on a CPU, namely Naïve, BSP-S, and BSP-SN, and analyze their drawbacks, which motivate us to develop our "fine-grained" solution.

"Naïve" solution. As depicted in Figure 6(a), a naïve way to parallelize IOBufs-edge is to allocate individual threads to handle the subtasks in the I/O-optimal algorithm. We only need to adjust the `TaskScheduler` as follows:

- `TaskScheduler`: Schedule the subtasks in a round-robin fashion from a thread pool of W workers.

However, Naïve will cause the space complexity to increase by W times, consequently increasing the I/O cost. For IOBufs-edge, as the total space cost becomes $O(W \cdot \frac{|E|}{p})$ and must still be accommodated by the main memory of size M, we can derive that $p = O(W \cdot \frac{|E|}{M})$ and the I/O cost as $O(W \cdot \frac{|E|^2}{MB})$.

"BSP" solutions. We next discuss a bulk-synchronous parallel (BSP) [71] mechanism by developing an iterative workflow for Algorithm 2, where each iteration handles a pair of partitioned subgraphs $(\mathcal{G}_i, \mathcal{G}_j)$. The parallelization is applied within each iteration by further dividing the subtask of $T_{wc}\{\mathcal{V}_i \times \mathcal{V}_j\}$. This is equivalent to conducting a **parfor** in line 3 of Algorithm 4 to parallelize the subtask of $T_{wc}\{\{u\} \times \mathcal{V}_j\}$ for all $u \in \mathcal{V}_i$, which is exactly what BFC-VP++ does. Specifically, "BSP" implements PIOBufs as:

- `TaskDivider`: On the basis of $T_{wc}\{\mathcal{V}_i \times \mathcal{V}_j\}$ in the "Naïve" solution, it further divides it into subtasks of $T_{wc}\{\{u\} \times \mathcal{V}_j\}$ for all $u \in \mathcal{V}_i$.
- `TaskScheduler`: The execution is scheduled iteratively. In a certain iteration for processing the partitioned subgraphs $(\mathcal{G}_i, \mathcal{G}_j)$, we adopt the dynamic scheduling technique proposed in [80] to parallelize the execution of all $T_{wc}\{\{u\} \times \mathcal{V}_j\}$. Specifically, it arranges all subtasks of $T_{wc}\{\{u\} \times \mathcal{V}_j\}$ in a concurrent queue by the descending order of $d(u)$. With W workers in the pool, each worker, once idle, will be scheduled to dequeue the top-ordered subtask to run. Synchronization is conducted at the end of each iteration to guarantee that all workers have finished their tasks and that the new graph data has supplanted the old data before proceeding to the subsequent iteration.
- `TaskRunner`: Each worker simply runs lines 4-7 of Algorithm 4.

Given how $\mathcal{H}$ in line 4 of Algorithm 4 is maintained, the "BSP" solution has two variants, namely BSP-S and BSP-SN. As illustrated in Figure 6(b), BSP-S requires that all workers share a common $\mathcal{H}\{\mathcal{V}_i \times \mathcal{V}_j\}$.

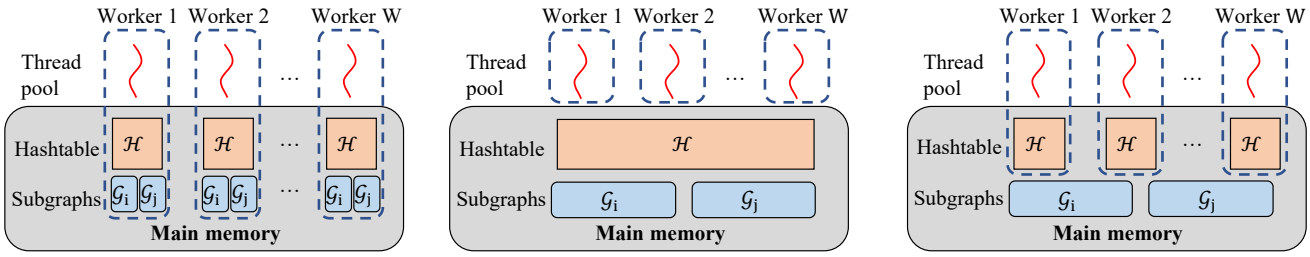


Fig. 6: Solutions of parallelizing IOBufs-edge on a CPU.

In fact, a straightforward implementation[3] of butterfly counting in graph processing systems [35,64,63,99,72, 73,69] is BSP-S: each vertex $v \in V$ maintains a local vertex state, i.e, a hashtable $\mathcal{H}\{\{v\} \times \mathcal{V}\}$, resulting in an overall hashtable $\mathcal{H}\{V \times \mathcal{V}\}$; then all vertices send their neighbors to the two-hop neighbors such that the wedge counts can be updated in the hashtable; note that the process can be conducted in iterations if V is partitioned accordingly. However, it's not hard to see that the algorithm has now degraded to IOBufs-Naïve (Algorithm 3), which has large I/O and time complexity. Alternatively, BSP-SN (BSP Shared Nothing) lets each worker maintain a local $\mathcal{H}$, which is similar to the idea of ParButterfly [67] with batching wedge aggregation. We can derive the space cost as $O(\frac{|E|}{p} + \frac{W|V|}{p})$, and the I/O cost as $O(\frac{|E|^2}{MB} + \frac{W|V||E|}{MB})$. Such I/O cost may be huge, as it is likely to have $W > \overline{d}$ when processing a sparse graph on the modern multi-core CPU.

7.2 The fine-grained solution

We observe that the dilemma of trading off the parallelism and the I/O efficiency of all solutions in Section 7.1 lies in the coarse-grained `TaskDivider`. In other words, the `TaskDivider` produces $T_{wc}\{\mathcal{V}_1 \times \mathcal{V}_2\}$ (requiring hashtable $\mathcal{H}\{\mathcal{V}_1 \times \mathcal{V}_2\}$) with some large $\mathcal{V}_1$ or $\mathcal{V}_2$, increasing the space complexity. In response, we propose the "fine-grained" (FG) solution based on BSP-SN[4]. Notice, GridGraph [99] also exploits the fine-grained partition for edges, while we focus on how to partition the subtask, in other words, the wedges represented by the hashtable.

As shown in Algorithm 7, the `TaskDivider` of FG (line 1-6) further divides the subtask of $T_{wc}\{\{u\} \times \mathcal{V}_j\}$ in BSP-SN into q disjoint parts (therefore, more fine-grained) of

$$\{T_{wc}\{\{u\}\times\mathcal{V}_{j,1}\}, T_{wc}\{\{u\}\times\mathcal{V}_{j,2}\}, \ldots, T_{wc}\{\{u\}\times\mathcal{V}_{j,q}\}\}, \tag{7}$$

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 I/O-parallelism tradeoff, but let us first check out the other components of FG. In line 7-8, `TaskScheduler` simply pops out the first task in the concurrent queue, while the dynamic scheduling technique in [80] is implicitly implemented as:

- The execution order is determined during enqueueing in line 3,
- The assignment of a subtask to an idle worker is processed naturally in line 19,20.

`TaskRunner` in lines 9-15 handles running the given subtask. It is important to note that the hashtable initialized in line 10 now has a space cost of $\frac{|\mathcal{V}_j|}{q}$ as opposed to $|\mathcal{V}_j|$ in the BSP-SN solution, which is the key to reducing the I/O cost. Accordingly, when enumerating the vertex w of a wedge (u, v, w) in line 12, the intersection $N_{\mathcal{G}_2}(v) \cap \mathcal{V}$ ensures that only the wedges with $w \in \mathcal{V}$ are counted to avoid duplicate counting, where $\mathcal{V}$ is a subset of $\mathcal{V}_j$ that is assigned to the worker. The process will not stop until the query Q becomes empty (line 22). Each worker with id τ ($1 \le \tau \le W$) maintains a local Φ_τ for the butterflies already counted. Finally, an aggregation is conducted in line 23 to summarize the Φ_τ for all workers as the final result.

Our FG solution renders space and I/O complexity as $O(\frac{|E|}{p} + \frac{W}{q} \cdot \frac{|V|}{p})$ and $O(\frac{|E|^2}{MB} + \frac{W}{q} \cdot \frac{|V||E|}{MB})$, respectively. We immediately re-approach I/O optimality by setting $q = \Theta(W)$. However, a large q may result in a subtask being too fine-grained to run efficiently in parallel [60] due to the scheduling overhead.

Configuration of q. We study the configuration of q to trade the I/O cost and scheduling cost. Let $C_{\text{I/O}}$ and

[3] Here, we use vertex-centric model as an example, other programming model such as edge-centric has similar approach.

[4] We also tried with both Naïve and BSP-S but found limited scope for enhancement in either solution.

Algorithm 7 FG: The "fine-grained" implementation of PIOBufs

```
1: function TaskDivider(T_wc{V_1 × V_2})
2:     Configure q according to Equation 8
3:     for u ∈ V_1 in descending order of d(u) do
4:         Divide T_wc{{u} × V_2} into q parts by Equation 7
5:         Push the subtasks into the concurrent queue Q
6:     return Q
7: function TaskScheduler(Q)
8:     return Q.pop()
9: function TaskRunner(G_1, G_2, T_wc{{u} × V})
10:    Initialize H{{u} × V → ℕ}
11:    for v ∈ N_{G_1}(u) do
12:        for w ∈ N_{G_2}(v) ∩ V do
13:            Φ ← Φ + H(u, w)
14:            H(u, w) ← H(u, w) + 1
15:    return Φ
16: function IOBufs-edge(G_i, G_j)
17:    Load partitioned graphs G_i, G_j into the main memory
18:    Q ← TaskDivider(T_wc{V_i × V_j})
19:    parfor an idle worker τ ∈ {1, ..., W} do
20:        T_wc ← TaskScheduler(Q)
21:        Φ_τ ← Φ_τ + TaskRunner(G_i, G_j, T_wc)
22:    until Q is empty
23:    Aggregate Φ ← Σ_{τ=1}^{τ=W} Φ_τ
24:    return Φ
```

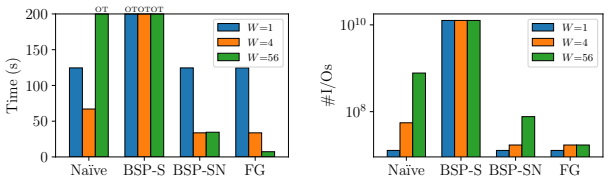
C_{sched} represent the cost of one I/O and one task synchronization, respectively. When partitioning a graph into p parts and a subtask into q parts, the total number of required I/Os is $\frac{2p|E|}{B}$, while the total number of required task synchronizations is p^2q. As a result, we have the following optimization problem:

$$\begin{aligned} \min_{p,q\in\mathbb{N}} \quad & \frac{|E|}{B}\frac{C_{\text{I/O}}}{C_{\text{sched}}}p + p^2q, \\ \text{s.t.} \quad & c_1\frac{|E|}{p} + c_2\frac{W|V|}{pq} \le M, \text{ and } p, q \in \mathbb{N}^+ . \end{aligned} \tag{8}$$

The optimization problem aims to find the optimal partition setting, i.e., p and q, that can minimize overall cost. Given p, it is obvious that q must be as small as possible while satisfying the space requirement. Accordingly, the solver of the optimization problem involves iterating $p = \lceil \frac{c_1|E|}{M} \rceil$ (when $q \to \infty$) to $p = \lceil \frac{c_1|E|+c_2T|V|}{M} \rceil$ (when $q = 1$) and deriving the corresponding q till we obtain a pair of p and q that can minimize f.

However, although the constant value $|E|$, $|V|$, B, W, M can be directly obtained from the input graph and the hardware specification, the $\frac{C_{\text{I/O}}}{C_{\text{sched}}}$ ratio, needs further benchmark tests to determine. In practice, it 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 a small graph to obtain a $q*$ value that renders the best performance; we finally obtain the $\frac{C_{\text{I/O}}}{C_{\text{sync}}}$ value by letting $f'(q*) = 0$, where f' denotes the differential of f.

With the solver and available constant values for the given hardware, we can derive the optimal configuration of q for any new graph.



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

Fig. 7: Performance of four parallel techniques on Delicious (refers to Table 5) with $W = 1, 4, 56$.

Micro benchmark. We conducted a micro benchmark that compares these alternatives, namely Naïve, BSP-S, and BSP-SN, with our FG solution, and the results of overall running time and I/Os are shown in Figure 7. 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 $W = 56$. BSP-SN still performs similarly to FG when $W = 4$, but it cannot further benefit from more parallelism and is outperformed by FG by a large margin when $W = 56$. Noticeably, when W increases, the I/O cost of FG almost remains constant, while those of the alternative solutions increase significantly.

Remark 2 When $M >> |E|$, namely the main memory is arbitrarily large, we can leverage Equation 8 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 8. In this case, FG becomes the parallel BFC-VP++ [78].

8 GPU Parallelization

Beyond the CPU-based parallelization, GPUs may be suitable for accelerating the computationally intensive butterfly counting task due to their capacity for massive parallelism and high memory access bandwidth. For instance, the NVIDIA A100 model that we use in this work contains 108 streaming multiprocessors (SM), each equipped with 64 CUDA cores and producing a total of 19.5 TFLOPS peak performance while providing up to 2 TB/s of HBM2 memory bandwidth. However, the shortage of on-board memory in GPUs significantly

Table 3: Variances of W and t while configuring different kinds of *worker* on an A100 GPU (110,592 working threads in total).

Worker	Grid	Block	Warp	Thread
W	1	108	3,456	110,592
t	110,592	1,024	32	1

limits its applicability to large graphs. Therefore, we propose to leverage the memory hierarchy to perform butterfly counting on GPUs, where the GPU memory will operate as the main memory, whereas CPU memory or disk storage can be utilized as the secondary memory.

In this section, we will investigate the implementation of the PIOBufs framework on a GPU. In Section 7.1, we have demonstrated that the I/O complexity can escalate due to an increase in the degree of parallelism (i.e., W in CPU). This has led us to design FG on CPUs for further subdivision of the subtask. On GPUs, a significantly larger degree of parallelism can be leveraged (hundreds of thousands) in comparison to CPUs (hundreds). As a result, we base the GPU implementation on FG as outlined in Algorithm 7. Specifically, we preserve the `TaskDivider` and `TaskScheduler` designs from the FG blueprint, and pay attention to crafting the `TaskRunner` tailored for the GPU environment. To fully leverage the capabilities of GPUs, it is imperative to address these pivotal considerations:

- Hierarchical Parallelism: As introduced in Section 3.3, it is crucial to exploit hierarchical parallelism on a GPU. When setting up the PIOBufs worker, aligning it with the right levels of parallelism becomes a pivotal factor.
- Single Instruction Multiple Threads (SIMT): Adhering to the SIMT computational paradigm, the GPU, when optimally harnessed, can markedly amplify algorithmic performance.
- Coalesced Memory Access: Given the distinctive memory access pattern of GPUs, which diverges from that of CPUs, the design of our parallel algorithms must prioritize coalesced memory access. This ensures full utilization of the memory bandwidth.

8.1 What can be the worker on a GPU

The GPU programming has demonstrated a hierarchical parallelism (Section 3.3) consisting of four levels of abstraction, arranged from lowest to highest: thread, warp, block and grid. A question arises: *to which level of parallelism should be used as a worker of* PIOBufs *on GPUs*. In Table 3, we present the number of workers

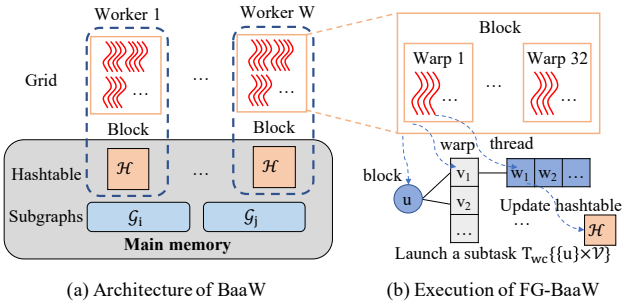


Fig. 8: The process of `TaskRunner` in the optimized block kernel.

(W) for various worker configurations on the cutting-edge A100 GPU. Each worker on a GPU now manages a group of working threads, given as t in Table 3. Note that the total degree of parallelism (110,592) is equivalent to $W \cdot t$. While the exact values for W and t might differ across other (NVIDIA) GPU models, the overall patterns remain largely consistent.

It is straightforward to rule out two undesirable design choices.

- Thread as a Worker: The straightforward porting of the FG from CPU to GPU is typically undesirable in GPU programming due to its inability to leverage the SIMT (single instruction multiple threads) nature of GPU.
- Grid as a Worker: When a large number of working threads in a grid are engaged in processing a fine-grained subtask, there is a potential for underutilization, resulting from a substantial number of idle threads.

Next, we will examine the remaining two potential solutions in detail, namely FG-BaaW (BaaW for "Block as a Worker") and FG-WaaW (WaaW for "Warp as a Worker").

8.2 Block as a worker

Figure 8(a) illustrates the architecture of FG-BaaW that considers a block, corresponding to a physical SM in a GPU, as a worker for the PIOBufs framework. According to Table 3, the number of workers W in FG-BaaW is 108, closely matching that in the CPU scenario. Consequently, we can directly implement the `TaskRunner` of FG-BaaW in accordance with Algorithm 8 by inheriting the I/O-parallelism tradeoff from Section 7.2.

Naïve block kernel. It is straightforward to adapt the FG's `TaskRunner` (refer to Algorithm 8 for convenience) as the block kernel for GPUs. This essentially

Algorithm 8 FG's `TaskRunner`.

```
1: function TaskRunner(G1, G2, Twc{{u} × V})
2:     Initialize H{{u} × V → N}
3:     for v ∈ NG1(u) do
4:         for w ∈ NG2(v) ∩ V do
5:             Φ ← Φ + H(u, w)
6:             H(u, w) ← H(u, w) + 1
7:     return Φ
```

means replacing the "**for**" directive in line 3 of Algorithm 8 with a "**parfor**". Consequently, the instructions between lines 4-6 will be concurrently executed in a round-robin fashion by every thread within a block. Notably, as the count of vertices within a partition usually surpasses the number of blocks, we can employ the kernel fusion technique [75]. This method combines multiple subtasks, specifically $T_{wc}\{\{u\} \times \mathcal{V}\}$ for several u, into a singular block kernel, diminishing the frequency of kernel launches and synchronizations. Moreover, the initialization of the hashtable in line 2 of Algorithm 8 is conducted only once at the initialization, rather than for every individual subtask as seen in the original algorithm. However, it is crucial that the hashtable must be reset after the completion of each subtask.

Coalesced memory access [85, 32, 54, 83]. The above naïve block kernel does not fully harness optimized memory access. We also notice there are several GPU graph processing systems [32, 54] targeting the coalesced memory access in neighbor list processing. Butterfly counting, however, presents a greater challenge due to its inherent requirement to process 2-hop neighbors, which complicates the attainment of coalesced memory access. While prior work has focused on optimizing graph layout to enhance memory access patterns [85, 32, 54], our approach extends beyond graph structure to place a greater emphasis on thread organization and the patterns in which 2-hop neighbors are accessed. Considering the naïve kernel directly employs the vertex-centric fashion, when assigning individual GPU threads for computing the 2-hop neighbors w of a vertex u as illustrated in line 4 of Algorithm 8, each thread individually triggers distinct memory transactions to retrieve $N_{\mathcal{G}_1}(v)$ for different vertices v (line 3). Due to the non-contiguous nature of memory storing $N_{\mathcal{G}_1}(v)$ in graph data, this method can incur a surge in memory transactions from the global memory, thereby compromising efficiency. A solution to this bottleneck is the coalesced memory access technique, facilitated by assigning a warp (each having 32 threads) in the block to process a singular vertex v. As demonstrated in Figure 8(b), each thread within this warp then handles a distinct vertex w in parallel — all of whom are neighbors of v which are maintained continuously in the memory — facilitating simultaneous and contiguous memory accesses. By aligning the memory accesses of threads in a warp, the coalesced approach helps reduce scattered memory transactions and results in boosted performance. A potential issue is, with 32 threads in a warp, under-utilization might occur if there are fewer than 32 neighboring vertices w to process in line 4. To navigate this, we consider further segmenting a warp into smaller sub-warps [59, 23]. Clearly, when a warp gets divided into 32 single-threaded sub-warps, the program reverts to the initial, less efficient implementation. Our experiment indicates that sub-warps consisting of 8-16 threads offer optimal performance for most tested graphs. It is noteworthy that G-BFC [86] has overlooked this coalesced memory access optimization.

Shared memory [56, 29, 44, 43]. As introduced in Section 3.3, threads within a block have the capability to access a rapid, localized memory space termed shared memory. This access path offers a bandwidth considerably greater than when tapping into the global memory. However, a challenge emerges due to the limited capacity of shared memory, which is constrained to a default 48 KB per block even in advanced GPUs like the A100. Notably, vertices with a high degree often have a pronounced likelihood of being 2-hop neighbors to other vertices. This observation leads us to segment the hashtable, as outlined in line 2 of Algorithm 8, into two distinct segments: $\mathcal{H}_1\{\{u\} \times \mathcal{V}_{large}\}$ and $\mathcal{H}_2\{\{u\} \times \mathcal{V}_{small}\}$. The segment $\mathcal{V}_{large}$ encompasses the top-s high-degree vertices from $\mathcal{V}$, where s represents the capacity of the shared memory. Conversely, $\mathcal{V}_{small}$ denotes the remaining vertices in $\mathcal{V}$ after removing those in $\mathcal{V}_{large}$. Throughout computational processes, the section $\mathcal{H}_1\{\{u\} \times \mathcal{V}_{large}\}$ remains cached in shared memory, whereas $\mathcal{H}_2\{\{u\} \times \mathcal{V}_{small}\}$ is stored in the global memory. Given the prior arrangement of $\mathcal{V}$ by vertex degree [78], dividing it into $\mathcal{V}_{large}$ and $\mathcal{V}_{small}$ is straightforward.

We observe that G-BFC [86] also endeavors to enhance hashtable access through shared memory, but its strategy diverges from ours significantly. Specifically, G-BFC divides the hashtable into $\frac{|\mathcal{H}|}{s}$ chunks, ensuring each chunk fits within the shared memory. The algorithm then proceeds in $\frac{|\mathcal{H}|}{s}$ rounds. In each round, it updates the wedge counts exclusively for the vertex pairs (u, w) where w resides in the current chunk, leading to redundant wedge computations. Consequently, this approach amplifies the time complexity of the algorithm by a factor of $\frac{|\mathcal{H}|}{s}$. In practice, the size of hashtable $\mathcal{H}$ is contingent on the number of vertices, potentially extending into the tens of millions (e.g., the Twitter graph). In contrast, the shared memory has a capacity of merely tens of thousands of buckets. Accordingly,

the amplification factor can reach up to thousands, inevitably resulting in inefficiency.

Algorithm 9 FG-BaaW's `TaskRunner`: the optimized block kernel.

```
1:  /* Cache H1{{u} × V_large} in the shared memory */
2:  /* Store H2{{u} × V_small} in the global memory */
3:  /* Set NumThreadsPerSubWarp as size of sub-warp and
    NumSubWarpsPerBlock the number of sub-warps in the
    thread block */
4:  function TaskRunner(G1, G2, T_wc{{u} × V})
5:      i ← subWarpId
6:      while i < |N_G1(u)| do
7:          v ← N_G1(u)[i]
8:          W-list ← N_G2(v) ∩ V
9:          j ← threadId
10:         while j < |W-list| do
11:             w ← W-list[j]
12:             if w ∈ V_large then
13:                 Φ ← Φ + atomicAdd(H1(u, w), 1)
14:             else
15:                 Φ ← Φ + atomicAdd(H2(u, w), 1)
16:             j ← j + NumThreadsPerSubWarp
17:         i ← i + NumSubWarpsPerBlock
18:     __syncthreads()
19:     Reset H1 and H2
20:     return Φ
```

Optimized block kernel. Algorithm 9 presents FG-BaaW's `TaskRunner` as the optimized block kernel. It is essential to note that the kernel fusion is already managed by the `TaskScheduler`, and hence is not discussed here. Lines 1-2 partition the hashtable into two segments that reside in shared and global memory, respectively. The `TaskRunner` then seamlessly processes each neighbor v of u using all available warps within the thread block (line 6). Here, the symbol $A[i]$ references the i^{th} item in a collection A. Each thread within a warp simultaneously calculates the wedge count for v (lines 8-16), with each thread handling a distinct vertex w. Keep in mind that warps are subdivided into sub-warps, with the division controlled by the parameter of `NumThreadsPerSubWarp`. Importantly, the value of `NumSubWarpsPerBlock` is determined by the value of `NumThreadsPerSubWarp`, as their multiplication yields the total number of threads within a block. Unlike its CPU counterpart, hashtables for both $\mathcal{H}_1$ (in shared memory) and $\mathcal{H}_2$ (in global memory) are accessed concurrently by multiple threads in a block. We thus employ the `atomicAdd` primitive provided by CUDA to avoid contentions (lines 13 and 15). In our implementation, `atomicAdd` reads a counter at the hashtable in global or shared memory, adds a number to it, and writes the result back to the counter. Moreover, `atomicAdd` returns the original value of the counter, which can be added to count Φ. To conclude the kernel, it is essential to reset both hashtables once all threads have confirmed the completion of computations, as fulfilled by lines 18-19.

8.3 Warp as a worker

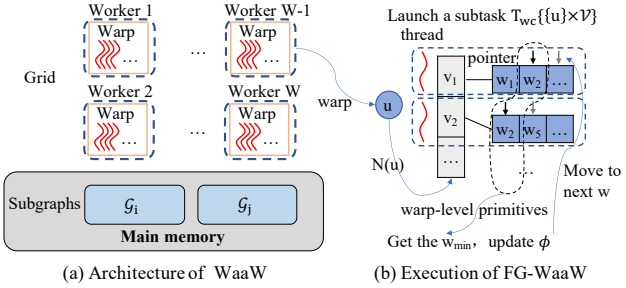


Fig. 9: The process of `TaskRunner` in the warp kernel.

The FG-WaaW approach, shown in Figure 9(a), leverages a warp as a worker to in parallel execute the `TaskRunner` in the PIOBufs framework. The number of workers W now matches the total number of warps on the GPU. In our case, with an A100 GPU, $W = 3,456$ by Table 3. Sticking to the `TaskRunner` implementation in Algorithm 8 requires each of the 3,456 workers to allocate memory for a hashtable to capture wedge counts. According to Equation 8, this could compel us to use a large value for either p or q to satisfy the memory requirements. Such a decision would increase the I/O or scheduling costs. Therefore, it is critical to design an algorithm that circumvents the necessity of utilizing a hashtable.

Sort-and-scan. In contrast to employing the hashtable to record the wedges, we introduce an alternative "sort-and-scan" option for doing wedge counting (and thus butterfly counting). As illustrated in Figure 9(b), for each neighbor v of u, we need to process the specific W-list (the list that maintains w, namely $N_{\mathcal{G}_2}(v) \cap \mathcal{V}$) to determine the wedge count. The "sort-and-scan" method can be conceptualized (without directly implementing it in this manner) as follows:

1. Concatenate the W-lists of all neighbors v of u into a singular list.
2. Sort the concatenated list by the vertex w.
3. Scan the sorted list to determine the wedge count.

Let us look into an illustrative example.

Example 4 Given a present vertex u, which contains three neighbors v_1, v_2 and v_3. The W-lists of v_1, v_2 and

v_3 are $\{w_1, w_2, w_3\}$, $\{w_2, w_3, w_4\}$ and $\{w_1, w_3, w_4\}$, respectively. After concatenating and sorting these three lists (assuming the subscripts of w indicate their order), we end up with a sorted list of $\{w_1, w_1, w_2, w_2, w_3, w_3, w_4, w_4\}$. Using w_1 as an example, it appears twice in the sorted list. Given that we are dealing with a simple graph, these two instances of w_1 must be adjacent to two different vertices (specifically v_1 and v_3). This implies that there is a butterfly of (u, v_1, w_1, v_2). Actually, the presence of n such consecutive vertices of w in the sort list results in the inclusion of $\frac{n(n-1)}{2}$ butterflies.

Evidently, a direct implementation of the "sort-and-scan" method, which requires the materialization of the merged list, is not feasible. Such an approach would require retaining all 2-hop neighbors w (that can be repeated) in memory. This could be even more resource-intensive than maintaining the hashtable in Algorithm 8, which only logs a distinct number for each w.

Fortunately, we may not need to actually materialize and sort the merged list due to the following two observations:

1. Given that the neighbors of a vertex have already been pre-sorted, we can efficiently merge these multiple W-lists using the well-established "merge-sort" algorithm.
2. For wedge counting, our primary concern is determining the frequency of each specific w in the merged list. This eliminates the need to pre-materialize the entire list.

Merge-sort transformation. We introduce a novel `TaskRunner` design for FG-WaaW that is inspired by the k-way "merge-sort" algorithm. The new `TaskRunner` leverages warp-level primitive operations for merging the W-lists and counting wedges. As depicted in Figure 9(b), when we delegate a warp to undertake the wedge-counting task of $T_{wc}\{\{u\} \times \mathcal{V}\}$, each of the 32 threads within the warp will concurrently handle a distinct vertex $v \in N_{\mathcal{G}_1}(u)$ to traverse its *W-list*. Given $d(u)$ as the number of vertices v in the present task, for simplicity, we assume $d(u) \leq 32$ such that each thread in the warp is responsible for no more than one W-list. We will look into the scenarios where $d(u) > 32$ in subsequent discussions.

Before delving into the warp kernel design, let us first revisit the high-performance warp-level primitives provided by CUDA. Notably, we capitalize two such primitives to fulfill the "merge-sort" approach for wedge counting.

- `__reduce_min_sync(val)`: When invoked, each thread within a warp simultaneously provides its distinct input value as `val`. This operation then identifies the minimum among these values and returns it to every thread in the warp.
- `__reduce_sum_sync(val)`: Similar to `__reduce_min_sync`, this function computes and returns the aggregate (sum) of all `val` values presented by the threads in the warp.

Algorithm 10 FG-WaaW's `TaskRunner`: the warp kernel

```
1: function TaskRunner(G1, G2, T_wc{{u} × V})
2:     i ← threadId
3:     v ← N_G1(u)[i]
4:     W-list ← N_G2(v) ∩ V
5:     w_local ← W-list.pop()
6:     while W-list ≠ ∅ do
7:         w_min ← __reduce_min_sync(w_local)
8:         if w_local = w_min then
9:             c ← 1
10:            w_local ← W-list.pop()
11:        else
12:            c ← 0
13:        n ← __reduce_add_sync(c)
14:        if i % 32 = 0 then
15:            Φ ← Φ + n(n−1)/2
16:    __syncwarp().
17:    return Φ
```

In Algorithm 10, we outline the warp kernel as FG-WaaW's `TaskRunner`. Within each warp, the 32 threads operate in parallel, each processing the W-list of a distinct vertex v (lines 2-3). Given that the W-list has been pre-sorted, retrieving the local minimum vertex w_{local} is as straightforward as popping the first element of the list (line 4). While processing the `while` loop, each thread always holds the local minimum vertex w_{local} from the unprocessed vertices in its respective W-list. With this in mind, the `__reduce_min_sync` operation allows us to pinpoint the global minimum vertex w_{min} among all threads in the warp (line 7). Given that the graph is simple and devoid of duplicate edges, the minimum value stands as a singular entity within each list. Consequently, when the local minimum w_{local} within a thread aligns with the value of the global minimum w_{min}, it signifies a local count of 1. This event prompts progression towards the subsequent local minimum w_{local} (line 9, 10). Conversely, the local count is set to 0 (line 12). Subsequently, using the `__reduce_sum_sync` function, the aggregate count n for the entire warp is computed (line 13). The first thread in the warp (line 14) then calculates the updated wedge count, augmenting it by the value of $\frac{n(n-1)}{2}$ (line 15).

Example 5 Figure 9(b) demonstrates the process of a subtask $T_{wc}\{\{u\} \times \mathcal{V}\}$ by a warp. During each iteration,

__reduce_min_sync are employed to obtain the w_{min} (i.e., w_2 as highlighted in the black dotted box) among the threads. As the first two threads observe their current w_{local} matching the global minimum w_{min}, they advance the pointers to the subsequent elements in the lists.

Complexity analysis. The space complexity for FG-WaaW's TaskRunner is $O(\frac{|E|}{p})$ as no hashtable is required. It is worth noting that the assignment of W-list in line 4 is just for ease of presentation, which does not incur any additional space consumption. Correspondingly, the I/O complexity aligns with that of sequential algorithm Algorithm 4. Regarding the time complexity, it increases the time complexity of IOBufs-edge by a factor of r, namely $O(r(\Lambda + \frac{|E|^2}{M}))$, where r is the complexity inherent in the reduce primitives __reduce_min_sync and __reduce_sum_sync. In practice, r can be recognized as a small value, as these primitives are executed directly within GPU registers, boasting a significantly enhanced efficiency.

Handling $d(u) > 32$. To extend Algorithm 10 to vertices u with $d(u) > 32$, we can assign each thread to handle $k = \lceil \frac{d(u)}{32} \rceil$ W-lists in line 4. However, this requires executing the __reduce_min_sync in line 7 and __reduce_sum_sync line 13 for k times, further increasing the time complexity by a factor of k. In the worst case, k can be $O(|V|)$, which will be further discussed in Section 8.4.

8.4 The adaptive kernel

We have presented two distinct GPU implementations, namely FG-BaaW and FG-WaaW, with each possessing its unique merits and limitations. The FG-BaaW approach, following the FG method on CPUs, readily adopts the I/O-parallelism tradeoff discussed in Section 7.2. Yet, with 1024 threads in a block, there is a potential issue of under-utilization. While our introduction of sub-warping in Section 8.2 aims to ameliorate the issue, it does not eradicate the problem. For instance, with NumThreadsPerSubWarp set at 16, a block contains only 64 sub-warps. If $d(u) < 64$, it inevitably results in idle threads while processing the task.

Conversely, with a novel TaskRunner design in FG-WaaW, there is no need to allocate a hashtable for each worker, which presents a notable advantage in terms of conserving space and minimizing I/O complexity. Nevertheless, as explored in Section 8.3, the time complexity of FG-WaaW can escalate by a factor of $r \cdot \lceil \frac{d(u)}{32} \rceil$. This makes it less efficient when dealing with vertices of a higher degree.

It is thus intuitive to adaptively combine FG-BaaW and FG-WaaW, making the choice contingent upon $d(u)$. Specifically, with a predetermined degree threshold d_t, we opt for FG-WaaW for processing a vertex u when $d(u) \leq d_t$ and resort to FG-BaaW otherwise. While G-BFC implements a comparable adaptive kernel, our approach distinctly leverages more efficient algorithms for both block and warp kernels. Unlike G-BFC [86], which also includes a thread kernel, we refrain from incorporating such a kernel as it significantly increases the time complexity. In addition, kernel selection of G-BFC relies on the size of 2-hop neighbors of u, a metric that is costly to compute and often necessitates approximation. In contrast, our criterion, $d(u)$, can be effortlessly retrieved from the graph data.

9 Experimental Setup

All evaluated algorithms are implemented in C++/CUDA, and compiled with g++ 7.5.0 and CUDA Toolkit 11.6. The optimization flag is set as "-O3". We deployed two hardware contexts with the hardware configurations listed in Table 4. Unless otherwise specified, we set the main memory capacity to 25% of the graph 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 4: Hardware configurations.

Items	CPU Context	GPU Context
Hardware	Intel Xeon Gold 6330	NVIDIA A100 GPU
Processors	56 cores	108 SMs
Main Memory	128GB CPU memory	40GB GPU memory
Sec. Memory	1TB SSD	128GB CPU memory

Datasets. The statistics and sources of the graphs are summarized in Table 5. Particularly, MANN is a real dense graph provided by DIMACS [17] for graph challenges. Journal, Delicious, and Orkut are social networks representing user-group memberships of online communities. Flickr [51] shows the user-photo relationships on an online photo-sharing website. Tracker is the web tracking dataset representing the relationships between the internet domains and the trackers they contain. Bi-twitter, Bi-sk, and Bi-uk are subgraphs of large real social graph Twitter [34] and web graph sk-2005

Table 5: Graph statistics.

Graph	Src.	$\|V\|$	$\|E\|$	$\overline{d}$	Φ	Sizes
MANN	[62]	3.32E+03	5.51E+06	3,316	1.51E+13	44M
Flickr	[33]	5.00E+05	8.55E+06	34	3.53E+10	72M
Journal	[33]	1.07E+07	1.12E+08	21	3.30E+12	1.0G
Delicious	[33]	3.46E+07	1.02E+08	6	5.69E+10	1.1G
Tracker	[33]	4.04E+07	1.41E+08	7	2.01E+13	1.4G
Orkut	[33]	1.15E+07	3.27E+08	57	2.21E+13	2.7G
Bi-twitter	[78]	4.17E+07	6.02E+08	29	6.30E+13	5.1G
Bi-sk	[78]	5.06E+07	9.11E+08	36	1.22E+14	7.7G
Bi-uk	[78]	7.77E+07	1.33E+09	34	4.89E+14	11G
Clueweb	[14]	9.78E+08	3.74E+10	76	1.49E+18	286G

and uk-2006-05 from WebGraph [10, 9, 8]. We follow [78] 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 [39] to generate synthetic graphs to purposely control the statistics.

Configuration of IOBufs. 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 [78]. The size of each graph is reported accordingly. By default, we apply the "Radix" strategy [37] to partition the graph, as will be discussed in Section 10.6. 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 $c_3 = 0.25$.

Table 6 summarizes the default configuration of IOBufs utilized in the experiments.

Table 6: Default configuration of IOBufs.

Parameter	Value
c_1	16
c_2	4
c_3	0.25
p, q	According to Equation 8
M	25%$\|E\|$
W, t	CPU: 56, 1; GPU: refer to Table 3
Shared memory for $\mathcal{H}$	32KB per block
`NumThreadsPerSubWarp`	16

10 Evaluation on CPU Context

The empirical studies concerning the CPU context 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 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 8 by default or specified otherwise.

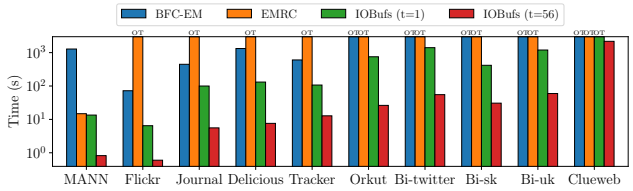
The BFC variants. We compare two BFC variants [78], namely BFC-VP++ and BFC-EM. BFC-VP++ is the most optimized variant of the kind developed in the parallel (in-memory) context, while BFC-EM stands for the variant leveraging disk as the secondary memory. The authors have not made the source codes publicly accessible. Thus, we apply our IOBufs-edge running in parallel with arbitrarily large M as BFC-VP++ (Remark 2). 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 [98]. 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.

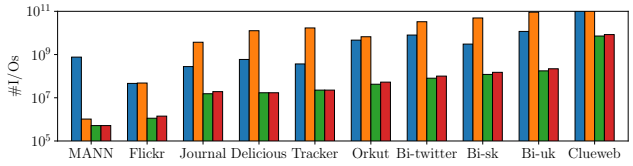
Note that the authors of BFC and EMRC agree that we faithfully reproduced their results.

10.1 Comparison with the SOTA

In this section, we compare IOBufs with the hierarchical-memory algorithms: BFC-EM and EMRC. Figure 10 shows the results of wall clock time and #I/Os on all graphs in Table 5. For EMRC, it suffers from huge I/O costs 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, the performance of EMRC downgrades dramatically. Let us



(a) Wall clock time.



(b) I/O cost.

Fig. 10: The performance of IOBufs, BFC-EM, and EMRC.

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 [10,9,8] 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 10(b).

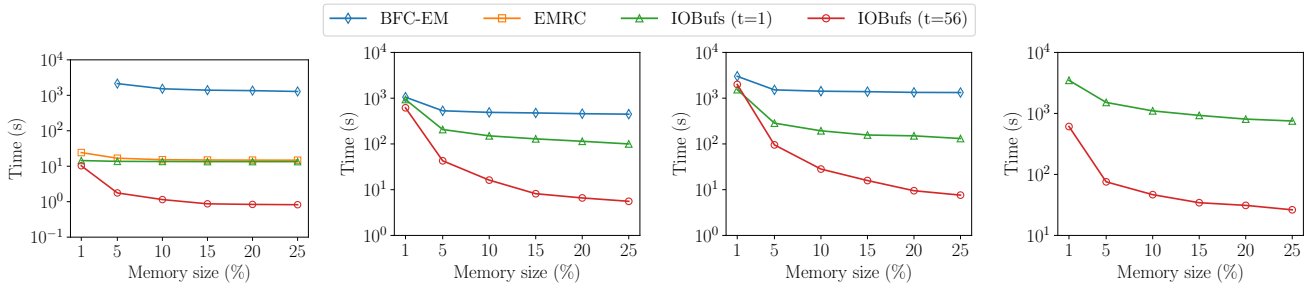
10.2 Impact of memory

As indicated in [27], 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 BFC-EM, EMRC, and IOBufs, and report the results in Figure 11. 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 a 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 11(c). With such 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.

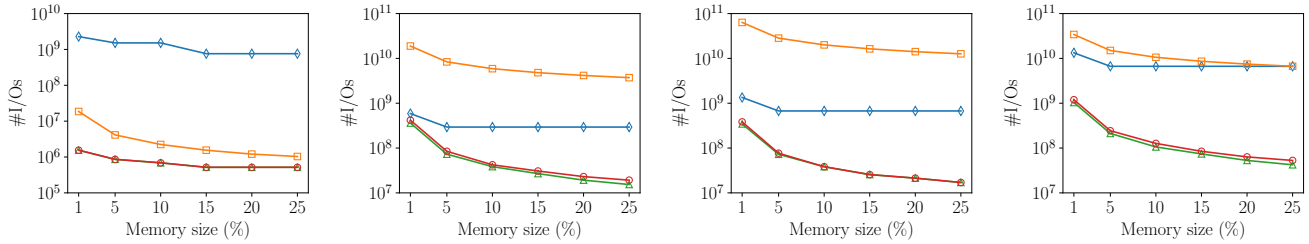
10.3 IOBufs-edge vs. IOBufs-wedge

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 [39] with (almost) fixed number of edges and vary $\overline{d}$ as shown in Figure 12. 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 12, 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) [31]. 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 13 illustrates the results on these graphs. As the aver-

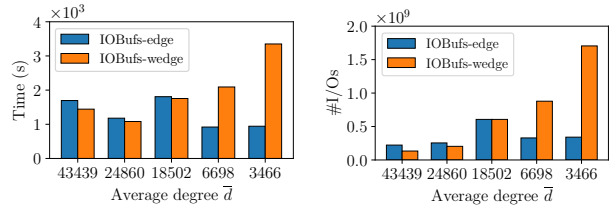


(a) MANN (Wall clock time). (b) Journal (Wall clock time). (c) Delicious (Wall clock time). (d) Orkut (Wall clock time).



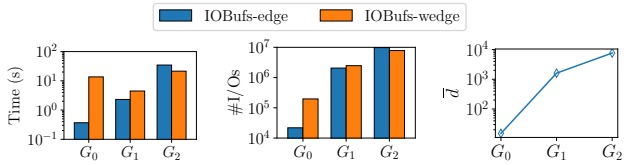
(e) MANN (I/O cost). (f) Journal (I/O cost). (g) Delicious (I/O cost). (h) Orkut (I/O cost).

Fig. 11: Memory size impact.



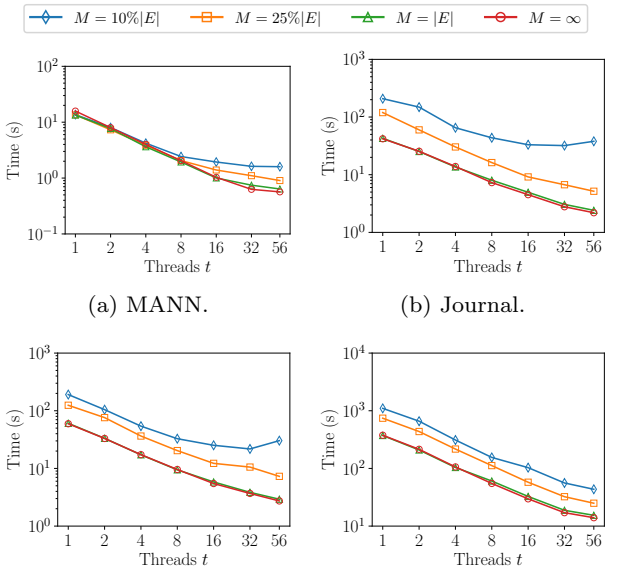
(a) Wall clock time. (b) I/O cost.

Fig. 12: Cost of IOBufs-edge and IOBufs-wedge on five generated graphs with different average degrees.



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

Fig. 13: Performance of IOBufs-edge and IOBufs-wedge on TC graphs



(a) MANN. (b) Journal.

(c) Delicious. (d) Orkut.

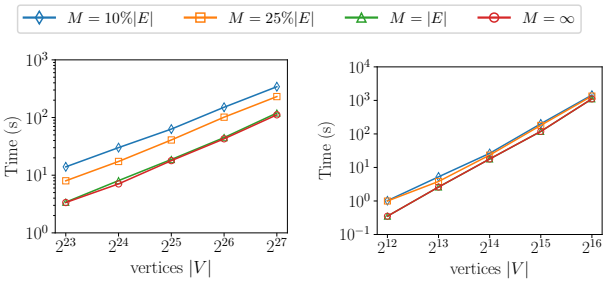
Fig. 14: Scale-up performance of IOBufs, varying t.

age degree increases, IOBufs-edge 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.

10.4 Scalability

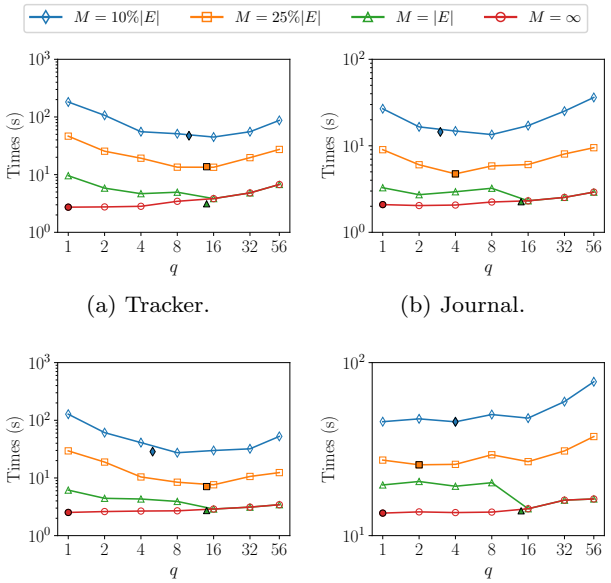
Scale-up. To reveal the scale-up performance of our fine-grained parallel technique, we vary the number of threads t and run IOBufs in different memory settings on the default graphs. Note that we include the case that $M = \infty$, in which IOBufs-edge becomes BFC-VP++ according to Remark 2. As shown in Figure 14, we 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 $11\times$, $21\times$, $20\times$ and $23\times$, 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.



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

Fig. 15: Data-scale performance of IOBufs, varying sizes.



(a) Tracker. (b) Journal.

(c) Delicious. (d) Orkut.

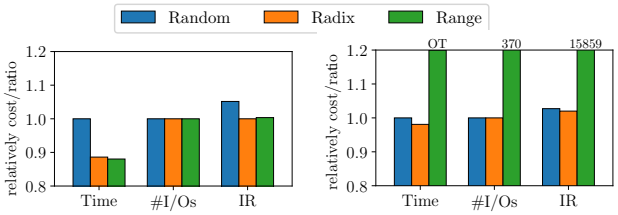
Fig. 16: Performance of fine-grained parallelism of IOBufs, varying q.

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 15. 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.

10.5 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 16. Note that here we replace MANN in the default graphs with Tracker because IOBufs-wedge does not need fine-grained parallelism. On each line in Figure 16, there is a solid-filled point that represents the derived value of q by the solver of Equation 8, 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 with the increase 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 2.

10.6 Impact of partition strategy



(a) MANN. (b) Journal.

Fig. 17: Performance of three partitioning approaches.

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 [37]*: place the vertex in the partition of $i\%p$; 3) *Range [98]*: 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 ratio* [56], denoted as IR $= \frac{\max_i(|\mathcal{E}_i|)}{\min_i(|\mathcal{E}_i|)}$. Figure 17 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 that 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 results from the rearrangement of vertex ids by 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.

11 Evaluation on GPU Context

In this section, we evaluate the performance of IOBufs on GPUs. As the I/O and IOBufs have already been evaluated in Section 9, we focus on the performance of the optimization techniques in GPUs (Section 8). To facilitate our discussion, we use IOBufs to represent the adaptive kernel of FG-WaaW and FG-BaaW. By default, we set `NumThreadsPerSubWarp` to 16, and the shared memory size to 32KB, which yields the optimal performance for IOBufs.

11.1 Comparison with the SOTA

We compare G-BFC [86] that implements butterfly counting on GPUs. However, as of now, the authors have not provided access to the code of G-BFC for our evaluation. Additionally, it is essential to acknowledge that various design elements, including the use of specialized data structures like hashtables on GPUs [56], have a substantial impact on the performance of G-BFC. These intricacies make it challenging to reproduce their code and results accurately.

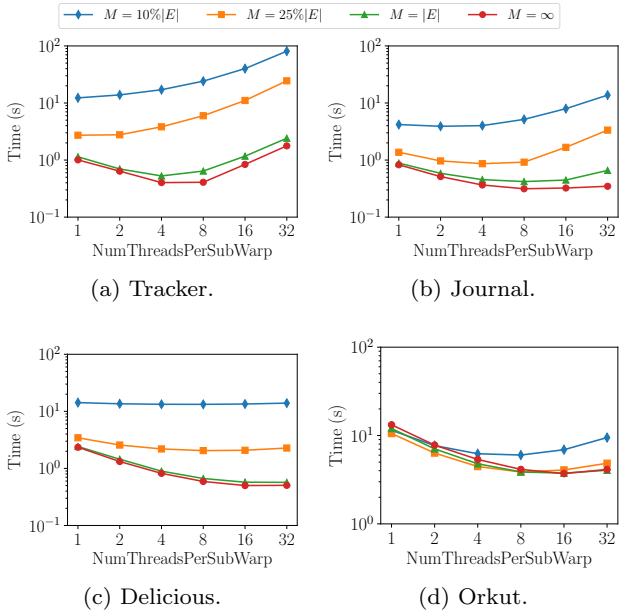
Table 7: Performance of G-BFC and IOBufs on Journal.

Alg.	Memory capacity	GPUs	Memory bandwidth	FP32 (float) performance	Time
G-BFC	∞	3090	936 GB/s	35.6 TFLOPS	30 s
IOBufs	∞	A100	1555 GB/s	19.5 TFLOPS	0.6 s
IOBufs	$25\%\|E\|$	A100	1555 GB/s	19.5 TFLOPS	4.2 s

Table 7 presents the performance of G-BFC and IOBufs on the Journal dataset (Table 5), which is the largest graph evaluated by G-BFC. It should be noted that the performance metrics for G-BFC are sourced from their original publication [86]. We utilized a different GPU card A100 while G-BFC is evaluated on 3090. As A100 is equipped with enhanced memory bandwidth, we adjusted IOBufs kernel time by a factor of $\frac{1555}{936} = 1.7$ to ensure a fair comparison. When harnessing the full memory capacity, IOBufs performs approximately 30× faster than G-BFC. This trend persists even when memory is restricted to $25\%|E|$, with IOBufs still achieving speeds nearly 4× faster than G-BFC. The advantage of IOBufs over G-BFC can be traced back to the foundational design principles of both algorithms. First, although both leverage the hierarchical parallelism of GPU programming, IOBufs maintains an optimized time complexity, while G-BFC adopts a less optimal algorithm. Second, IOBufs exploits more GPU features, such as coalesced memory access, sub-warp optimization, and warp-level primitive operations.

11.2 The design choice of FG-BaaW

We evaluate the performance of FG-BaaW kernel with different configurations. Notably, the FG-WaaW kernel consistently outperforms the FG-BaaW kernel when dealing with small workloads of $T_{wc}\{\{u\} \times V\}$ with $d(u) \leq 32$ (as will be analyzed in the subsequent section). Thus, we focus our performance evaluation of FG-BaaW specifically on workloads with $d(u) > 32$, as otherwise FG-WaaW is the preferred kernel.



(a) Tracker. (b) Journal.

(c) Delicious. (d) Orkut.

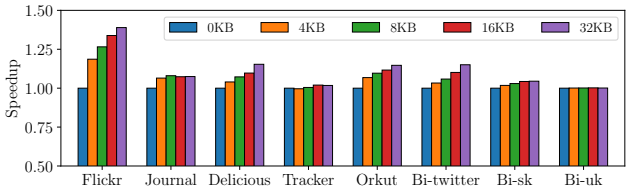
Fig. 18: Impact of `NumThreadsPerSubWarp`.

Impact of `NumThreadsPerSubWarp`. As discussed in Section 8.2, the `NumThreadsPerSubWarp` presents a trade-off between memory access and thread utilization. A smaller `NumThreadsPerSubWarp` may lead to uncoalesced memory access, whereas a larger `NumThreadsPerSubWarp` can result in idle threads. Therefore, we conduct a performance evaluation of FG-BaaW by varying the `NumThreadsPerSubWarp` to determine the optimal configuration. Notice, by setting the `NumThreadsPerSubWarp` equivalent to 1, FG-BaaW reduces to the block kernel in G-BFC.

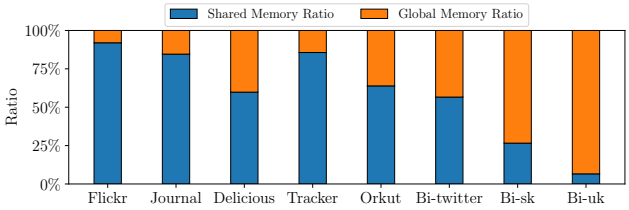
Figure 18 demonstrates the performance of FG-BaaW with varying `NumThreadsPerSubWarp` on the default graphs. With the increasing `NumThreadsPerSubWarp`, the performance shows a decreasing trend followed by an increasing trend, while the best performance is achieved when the `NumThreadsPerSubWarp` is around 8, 16. Accordingly, we have established a default `NumThreadsPerSubWarp` of 16 in our experiments. Moreover, the experiments also reveal several interesting insights. Firstly, the density of the graph plays a pivotal role: dense graphs with sufficient workload in each subtask tend to benefit from a larger `NumThreadsPerSubWarp`, as it mitigates uncoalesced memory access, while sparse graphs with relatively small workload in each subtask often favor smaller subwarps to enable better thread utilization. Secondly, the available main memory size also influences the optimal `NumThreadsPerSubWarp`. In scenarios with limited memory capacity, smaller subwarps are preferable, as the workload is partitioned into smaller pieces, aligning well with the characteristics of small subwarps. Conversely, in settings with sufficient main memory resources, larger subwarps can be effectively utilized by processing larger workloads to improve coalesced memory access.

Leveraging the shared memory. Another optimization for FG-BaaW is to incorporate the shared memory to cache the hashtable. We observe that setting limited GPU memory size will result in the caching of the hashtable and graph in the L1 cache or L2, and manually caching the hashtable in the shared memory only has a marginal performance improvement. Thus, our experiments focus on the scenario with full memory capacity.

Figure 19(a) demonstrates the performance of FG-BaaW with different shared memory sizes across various graphs with sufficient GPU memory. The experimental results indicate a consistent trend of improved performance as the shared memory allocation increases. On average, utilizing 32KB shared memory yields 12% speedup. This performance improvement is mainly attributed to the reduction of global memory access. To



(a) Speedup of Shared Memory Utilization.



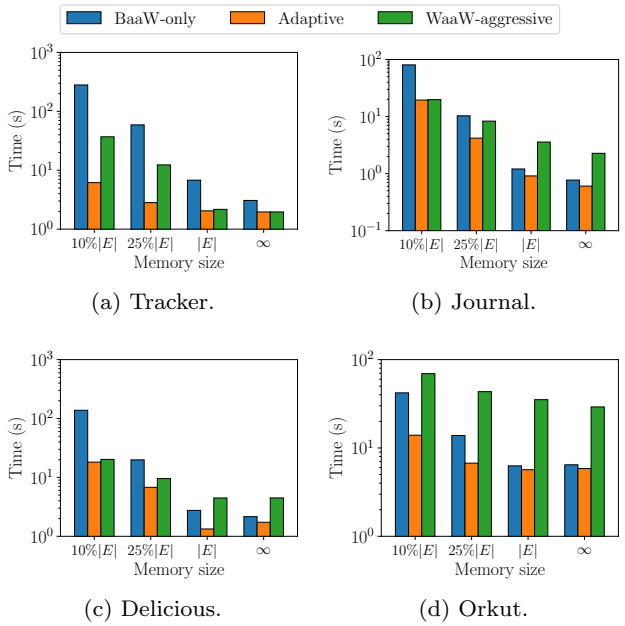
(b) Shared Memory to Global Memory Access Ratio.

Fig. 19: Shared memory size impact.

gain deeper insights, we investigate the distribution of hashtable accesses within the shared memory, as depicted in Figure 19(b). The experimental results reveal that, on average, approximately 59% of hashtable accesses occur within the shared memory when utilizing 32KB of shared memory. Interestingly, we also observe that larger graphs, such as Bi-uk, experience limited performance improvements. This phenomenon can be attributed to the larger hashtable requirements of such graphs, which significantly surpass the available shared memory whose size is constrained.

11.3 FG-BaaW vs. FG-WaaW

As previously mentioned, employing the FG-BaaW kernel will lead to idle threads for workloads with smaller-degree vertices, while employing the FG-WaaW kernel will increase the complexity for vertices with larger degrees. A better solution is adaptively configuring the kernel according to the vertex degree. Figure 20 demonstrates the performance of three variants: BaaW-only, adaptive, and WaaW-aggressive, where BaaW-only variant always chooses FG-BaaW kernel for each kind of workload, adaptive variant configures the faster variant for a different kind of workload, and WaaW-aggressive variant aggressively utilizes the FG-WaaW kernel while possible. Note that it is often not possible to only adopt FG-WaaW, which can only support the workload of vertex u with $d(u) \leq 1024$. This limitation arises from the constrained register capacity within a warp on A100 GPUs. The adaptive variant outperforms BaaW-only and WaaW-aggressive variants by up to 46× and 6.4×, respectively. By comparing the BaaW-only with WaaW-



(a) Tracker. (b) Journal.

(c) Delicious. (d) Orkut.

Fig. 20: The performance of FG-BaaW, FG-WaaW and hybrid of two levels of parallelism.

aggressive, we observe that the density of the graph makes a difference. The BaaW-only variant always performs better in the Orkut graph with $\overline{d} = 57$, while the WaaW-aggressive alternative has better performance in the sparse Tracker graph with $\overline{d} = 7$.

To further reveal the differences of the kernels of FG-BaaW and FG-WaaW, we divide the workloads based on the degree ranges of $[1, 32], [33, 64], \ldots, [1025, \infty]$. Figure 21 presents the performance results among these ranges with varying memory sizes on the Journal graph. As the workload gets larger, FG-WaaW exhibits a decreasing performance trend, while FG-BaaW exhibits an increasing performance trend, which is consistent with our analysis. Another observation is that the choice of kernel for the same set of vertices is also affected by memory size. Specifically, when operating with a small memory capacity ($10\%|E|$ and $25\%|E|$), FG-WaaW may outperform FG-BaaW even on larger workloads. This is because when memory is limited, the graph must be further partitioned, resulting in a smaller workload for each vertex, which benefits the FG-WaaW kernel. Conversely, with a large memory capacity ($100\%|E|$ and ∞), FG-WaaW only performs better than FG-BaaW at the smallest degree range.

11.4 CPU or GPU?

The decision between utilizing the CPU or the GPU for computational tasks hinges on several factors, including available main memory and the characteristics of the graph. The performance of IOBufs is notably influenced by the main memory capacity (M) of the selected hardware (either GPU or CPU) and the size of the graph ($|E|$). Additionally, our experiments indicate that the graph's density also plays a crucial role in hardware selection. As shown in Figure 22, we conducted an evaluation of IOBufs's execution times on the GPU across a range of GPU memory sizes, from sufficiently large down to $10\%|E|$, and compared these against the CPU baseline. For the densely connected Orkut graph, characterized by an average degree $\overline{d}$ of 57, we observed that the algorithm performed better on the GPU even when memory M was limited to just $25\%|E|$. On the other hand, for the more sparsely connected Delicious graph, which has an average degree of 6, GPU performance fell short of CPU performance once M was reduced to $90\%|E|$.

In theory, given a fixed value of $|E|$, the computation workload is proportional to $\overline{d}$, while the I/O is proportional to $\frac{1}{M}$. To this end, we propose a decision-making criterion for hardware selection as follows:

$$\begin{cases} \text{Choose GPU} & \text{if } \frac{\overline{d}}{c_4} > \frac{|E|}{M}, \\ \text{Choose CPU} & \textit{otherwise}. \end{cases} \tag{9}$$

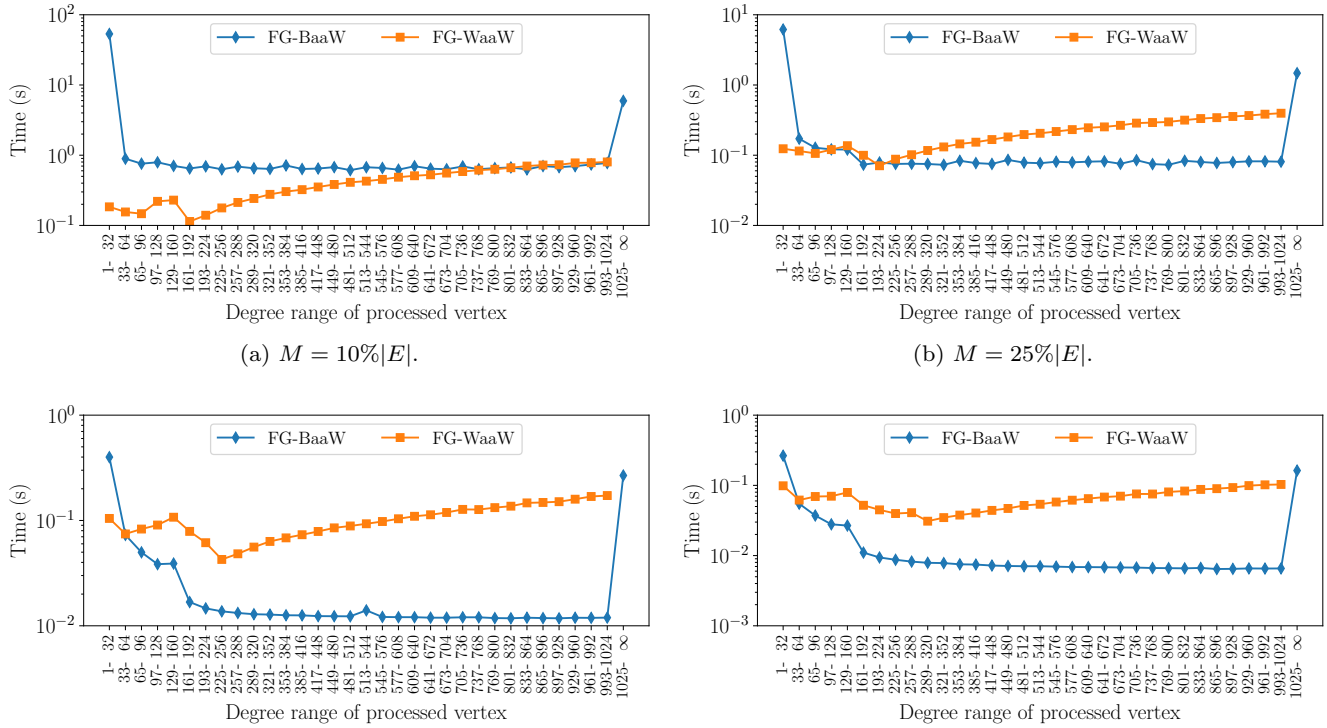
Here, c_4 represents a hardware-specific constant. According to our experiments as shown in Figure 22, we set $c_4 = 15$ in our evaluation. It's important to note that this constant is adjustable based on the hardware environment and does not adhere to a strict universal standard.

12 From butterfly to general motif

In this section, we first introduce how to count k-wedges, a kind of motif generalized from butterfly, under the framework of IOBufs. Additionally, we explore the potential of adopting the techniques of IOBufs for counting a broader spectrum of motifs. These include I/O bound analysis based on semi-witness algorithms, fine-grained parallel execution, and the merge-sort transformation in GPUs.

12.1 Counting k-wedges

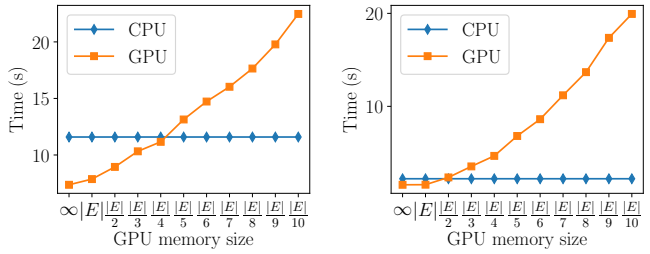
A k-wedge represents a general form of a butterfly, consisting of k interconnected wedges sharing common leaf



(a) $M = 10\%|E|$. (b) $M = 25\%|E|$.

(c) $M = |E|$. (d) $M = \infty$.

Fig. 21: The performance of FG-BaaW and FG-WaaW on Journal graph, varying the workload.



(a) Orkut. (b) Delicious.

Fig. 22: The performance IOBufs on CPUs vs GPUs. We configure sufficiently large (i.e. ∞) memory on CPUs and use the performance as baseline.

vertices. In this context, the butterfly is precisely a 2-wedge. The counting of k-wedges is crucial for analyzing community structures [4,38,45], especially within bipartite graphs [46,79].

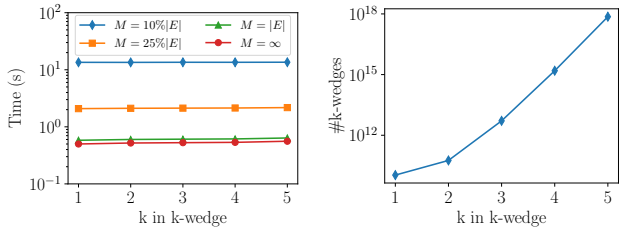
Before delving into k-wedge counting, it's instructive to review butterfly counting via wedge enumeration. For any two vertices, IOBufs calculates the number of wedges connecting them. If n wedges exist between these vertices, the resulting butterfly count is determined by $\binom{n}{2}$.

Similarly, we can adapt the calculation as $\binom{n}{k}$ for counting k wedges. It's important to note that the strategy of decomposing the counting process into incremental steps, as given in line 3 of Algorithm 1, remains effective. This is supported by the following equation:

$$\binom{n}{k} = \sum_{i=0}^{n-1} \Delta_{i\to i+1}, \text{ where}$$
$$\Delta_{i\to i+1} = \binom{i}{k} - \binom{i+1}{k} = \binom{i}{k-1}. \tag{10}$$

We simply increment the k-wedge count by $\binom{\mathcal{H}(u,w)}{k-1}$, each time when updating the wedge count in $\mathcal{H}(u,w)$. All techniques in IOBufs are thus seamlessly applicable to k-wedge counting.

Empirical study. We conduct an empirical study to compare the performance of counting k-wedges and butterflies. Utilizing the Delicious dataset and adhering to the experimental setup described in Section 9, we present our findings in Figure 23. Regarding execution performance, there demonstrates a slightly increasing trend with the increment of k, which is consistent with the computational complexity of calculating $\binom{n}{k}$. The number of k-wedge increases exponentially as k grows, and it can overflow a 64-bit integer when $k > 5$.



(a) Wall clock time. (b) #k-wedges.

Fig. 23: The performance and the number of k-wedges in the Delicious dataset.

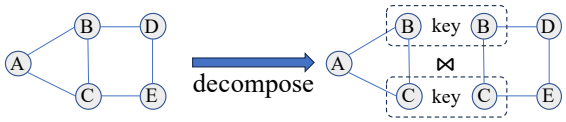


Fig. 24: The house motif: its decomposition and the process of counting via join operations.

12.2 Counting general motifs

While IOBufs is initially designed with the specific purpose of butterfly counting, we also explore the possibilities of adopting IOBufs's techniques to counting general motifs.

Semi-witnessing. Semi-witnessing algorithms present a novel perspective for reducing I/O complexity in motif counting, which suggests that fully witnessing a motif is not a prerequisite for its counting, thereby decreasing I/O complexities by avoiding the materialization of unnecessary intermediate data. In order to apply semi-witness algorithms, it is crucial that the motif can be formed by joining subgraphs, and each subgraph contains *free* vertices that do not serve as join keys. For example, in the house motif shown in Figure 24, given the triangle (A, B, C) and butterfly (B, C, D, E) that are joined by keys B and C to form the motif, the vertices A, D, and E are now the free vertices. As a result, the counting of house motifs can benefit from the semi-witnessing algorithms, which require witnessing the triangles and butterflies instead of the whole structure.

It's obvious that there are different ways of decomposing the motifs into subgraphs for joining. Exploring how to determine the optimal decomposition strategy, such that the resulting semi-witnessing algorithms align with the lowest possible I/O complexities, is a promising future direction.

Fine-grained parallelization strategy. Given an optimal (regarding I/O bound) decomposition of the motif, where the join keys are consisted of k vertices, we can model the motif counting problem as a task of $T_{\text{mc}}\{\underbrace{V \times \cdots \times V}_{k \text{ times}}\}$. For example, the house counting problem can be modeled as $T_{\text{mc}}\{V \times V\}$, where the task aims to count the number of triangles and butterflies joined by each pair of vertices B and C. Observing that intermediate results in motif counting may exceed the size of the input graph by a considerable margin [38, 60], the fine-grained parallelism of FG can be employed to address this issue by partitioning the task as well as the intermediate results into smaller parts to meet the memory constraints.

We also notice several GPU graph mining systems [12,89,84,25] adopt the DFS traversal for motif counting to control the intermediate memory usage. However, these systems are founded on enumeration-based algorithms, which may not be optimal for counting motifs with high complexities. It will be interesting to explore how the fine-grained parallelization strategy can be integrated into these systems.

Merge-sort transformation. The challenge of counting motifs often involves dealing with substantial memory demands to store intermediate results, a situation that becomes particularly acute in GPU environments where memory is limited. To mitigate these memory constraints, the merge-sort transformation offers a viable solution in two cases:

- **Worst-case optimal join algorithms [53]**: These algorithms, which are crucial in motif counting [38, 87], rely on the intersection of multiple neighbor lists. The merge-sort transformation helps reduce memory consumption during the process of the list intersection.
- **Decomposition into symmetrical subgraphs**: For motifs that can be broken down into symmetrical (more formally, isomorphic) subgraphs (e.g., the butterfly can be decomposed into two wedges that are identical), we can do motif counting by partially counting the subgraphs, and then leverage the merge-sort transformation to efficiently "merge" the results.

13 Conclusion

We finally summarize the process for selecting the optimal configurations for IOBufs, considering both graph characteristics and hardware specifications. The process, as shown in Figure 25, is unfolded in three steps:

- **Hardware Selection**: The initial decision between utilizing CPUs or GPUs is guided by Equation 9.

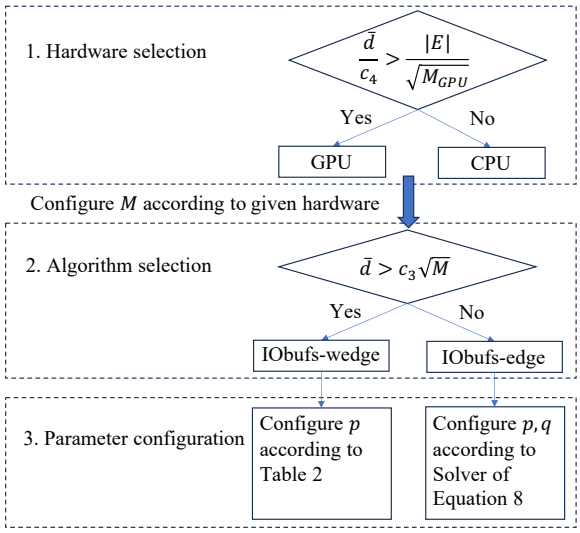


Fig. 25: The process for selecting the optimal configurations for IOBufs.

– **Algorithm Selection**: Next, the choice between algorithm variants, specifically IOBufs-edge and IOBufs-wedge, is made following the guidelines in Section 6.4.
– **Parameter Configuration**: If IOBufs-edge is selected, parameters such as p and q are configured using the solver introduced in Section 7.2. For GPU configurations, the kernel choice between FG-BaaW or FG-WaaW is adaptively made based on vertex degree, as elaborated in Section 8.4. Otherwise, if IOBufs-wedge is selected, parameter p is configured according to Table 2.

To conclude, 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 $o(\min(\frac{|E|^2}{MB}, \frac{|E||V|}{\sqrt{M}B}))$ I/Os. We then develop the IOBufs algorithm that can arrive at the I/O lower bound. Finally, we present a framework PIOBufs to parallelize the algorithm as well as a fine-grained technique to tradeoff the I/O and computation efficiency. The PIOBufs is further extended to GPUs. The experimental results have verified the effectiveness of all our proposed techniques, which makes IOBufs outperform the state of the art by orders of magnitude.

14 Compliance with Ethical Standards

Disclosure of Potential Conflicts of Interest. The authors declare that they have no conflict of interest.

Research Involving Human Participants and/or Animals. This article does not contain any studies with human participants or animals performed by any of the authors.

Informed consent. As this study does not involve human participants, informed consent is not applicable.

References

1. cgroup. `https://www.kernel.org/doc/Documentation/cgroup-v2.txt`
2. Cuda toolkit. URL `https://developer.nvidia.com/cuda-toolkit`
3. Aksoy, S.G., Kolda, T.G., Pinar, A.: Measuring and modeling bipartite graphs with community structure. Journal of Complex Networks **5**(4), 581–603 (2017)
4. Allahbakhsh, M., Ignjatovic, A., Benatallah, B., Beheshti, S.M.R., Bertino, E., Foo, N.: Collusion detection in online rating systems. In: Web Technologies and Applications: 15th Asia-Pacific Web Conference, APWeb 2013, Sydney, Australia, April 4-6, 2013. Proceedings 15, pp. 196–207. Springer (2013)
5. Ammar, K., McSherry, F., Salihoglu, S., Joglekar, M.: Distributed evaluation of subgraph queries using worstcase optimal lowmemory dataflows. arXiv preprint arXiv:1802.03760 (2018)
6. Bhattarai, B., Liu, H., Huang, H.H.: Ceci: Compact embedding cluster index for scalable subgraph matching. In: Proceedings of the 2019 International Conference on Management of Data, pp. 1447–1462 (2019)
7. Bi, F., Chang, L., Lin, X., Qin, L., Zhang, W.: Efficient subgraph matching by postponing cartesian products. In: Proceedings of the 2016 International Conference on Management of Data, pp. 1199–1214 (2016)
8. Boldi, P., Marino, A., Santini, M., Vigna, S.: BUbiNG: Massive Crawling for the Masses. In: WWW, pp. 227–228 (2014)
9. Boldi, P., Rosa, M., Santini, M., Vigna, S.: Layered Label Propagation: A MultiResolution Coordinate-Free Ordering for Compressing Social Networks. In: WWW, pp. 587–596. ACM Press (2011)
10. Boldi, P., Vigna, S.: The WebGraph Framework I: Compression Techniques. In: WWW, pp. 595–601. ACM Press (2004)
11. Chen, H., Li, X., Huang, Z.: Link prediction approach to collaborative filtering. In: Proceedings of the 5th ACM/IEEE-CS Joint Conference on Digital Libraries (JCDL'05), pp. 141–142. IEEE (2005)
12. Chen, X., et al.: Efficient and scalable graph pattern mining on {GPUs}. In: 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22), pp. 857–877 (2022)
13. Chiba, N., Nishizeki, T.: Arboricity and subgraph listing algorithms. SIAM J. Comput. **14**(1), 210–223 (1985). DOI 10.1137/0214017. URL `https://doi.org/10.1137/0214017`
14. Clarke, C.L., Craswell, N., Soboroff, I.: Overview of the trec 2009 web track. Tech. rep., WATERLOO UNIV (ONTARIO) (2009)
15. Cormode, G., Srivastava, D., Yu, T., Zhang, Q.: Anonymizing bipartite graph data using safe groupings. Proceedings of the VLDB Endowment **1**(1), 833–844 (2008)

16. Dhillon, I.S.: 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, p. 269–274. Association for Computing Machinery, New York, NY, USA (2001). DOI 10.1145/502512.502550. URL https://doi.org/10.1145/502512.502550
17. DIMACS: Dimacs challenge. Http://dimacs.rutgers.edu/Challenges/
18. Finnerty, E., Sherer, Z., Liu, H., Luo, Y.: Dr. bfs: Data centric breadth-first search on fpgas. In: 2019 56th ACM/IEEE Design Automation Conference (DAC), pp. 1–6 (2019)
19. Gibson, D., Kumar, R., Tomkins, A.: Discovering large dense subgraphs in massive graphs. In: Proceedings of the 31st International Conference on Very Large Data Bases, VLDB '05, p. 721–732. VLDB Endowment (2005)
20. Gonzalez, J.E., Low, Y., Gu, H., Bickson, D., Guestrin, C.: {PowerGraph}: Distributed {Graph-Parallel} computation on natural graphs. In: 10th USENIX symposium on operating systems design and implementation (OSDI 12), pp. 17–30 (2012)
21. He, H., Singh, A.K.: 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, pp. 405–418 (2008)
22. Hoang, L., Jatala, V., Chen, X., Agarwal, U., Dathathri, R., Gill, G., Pingali, K.: Disttc: High performance distributed triangle counting. In: 2019 IEEE High Performance Extreme Computing Conference (HPEC), pp. 1–7. IEEE (2019)
23. Hong, S., Kim, S.K., Oguntebi, T., Olukotun, K.: Accelerating CUDA Graph Algorithms at Maximum Warp. Acm Sigplan Notices **46**(8), 267–276 (2011)
24. Hong, S., Oguntebi, T., Olukotun, K.: Efficient parallel graph exploration on multi-core cpu and gpu. In: 2011 International Conference on Parallel Architectures and Compilation Techniques, pp. 78–88 (2011). DOI 10.1109/PACT.2011.14
25. Hu, L., Zou, L.: A gpu-based graph pattern mining system. In: Proceedings of the 31st ACM International Conference on Information & Knowledge Management, pp. 4867–4871 (2022)
26. Hu, X., Chiueh, T.c., Shin, K.G.: Large-scale malware indexing using function-call graphs. In: Proceedings of the 16th ACM Conference on Computer and Communications Security, CCS '09, p. 611–620. Association for Computing Machinery, New York, NY, USA (2009). DOI 10.1145/1653662.1653736. URL https://doi.org/10.1145/1653662.1653736
27. Hu, X., Tao, Y., Chung, C.W.: Massive graph triangulation. In: Proceedings of the 2013 ACM SIGMOD international conference on Management of data, pp. 325–336 (2013)
28. Hu, X., Tao, Y., Chung, C.W.: I/o-efficient algorithms on triangle listing and counting. ACM Transactions on Database Systems (TODS) **39**(4), 1–30 (2014)
29. Hu, Y., Liu, H., Huang, H.H.: Tricore: Parallel Triangle Counting on GPUs. In: SC, pp. 171–182. IEEE (2018)
30. Huang, S., El-Hadedy, M., Hao, C., Li, Q., Mailthody, V.S., Date, K., Xiong, J., Chen, D., Nagi, R., Hwu, W.m.: Triangle Counting and Truss Decomposition using FPGA. In: HPEC, pp. 1–7. IEEE (2018)
31. Jagadish, H.V.: A compression technique to materialize transitive closure. ACM Trans. Database Syst. **15**(4), 558–598 (1990). DOI 10.1145/99935.99944. URL https://doi.org/10.1145/99935.99944
32. Khorasani, F., Vora, K., Gupta, R., Bhuyan, L.N.: Cusha: Vertex-centric graph processing on gpus. In: Proceedings of the 23rd International Symposium on High-Performance Parallel and Distributed Computing, HPDC '14, p. 239–252. Association for Computing Machinery, New York, NY, USA (2014). DOI 10.1145/2600212.2600227. URL https://doi.org/10.1145/2600212.2600227
33. Kunegis, J.: Konect: the koblenz network collection. In: Proceedings of the 22nd international conference on world wide web, pp. 1343–1350 (2013)
34. Kwak, H., Lee, C., Park, H., Moon, S.: What is Twitter, a Social Network or a News Media? In: WWW, pp. 591–600 (2010)
35. Kyrola, A., Blelloch, G., Guestrin, C.: {GraphChi}:{Large-Scale} graph computation on just a {PC}. In: 10th USENIX symposium on operating systems design and implementation (OSDI 12), pp. 31–46 (2012)
36. Lai, L., Qin, L., Lin, X., Chang, L.: Scalable subgraph enumeration in mapreduce. Proceedings of the VLDB Endowment **8**(10), 974–985 (2015)
37. Lai, L., Qing, Z., Yang, Z., Jin, X., Lai, Z., Wang, R., Hao, K., Lin, X., Qin, L., Zhang, W., et al.: Distributed subgraph matching on timely dataflow. Proceedings of the VLDB Endowment **12**(10), 1099–1112 (2019)
38. Lai, L., Yang, Y., Wang, Z., Liu, Y., Ma, H., Shen, S., Lyu, B., Zhou, X., Yu, W., Qian, Z., et al.: {GLogS}: Interactive graph pattern matching query at large scale. In: 2023 USENIX Annual Technical Conference (USENIX ATC 23), pp. 53–69 (2023)
39. Leskovec, J., Chakrabarti, D., Kleinberg, J., Faloutsos, C., Ghahramani, Z.: Kronecker graphs: an approach to modeling networks. Journal of Machine Learning Research **11**(2) (2010)
40. Li, H., Kong, F., Yu, J.: Secure outsourcing for normalized cuts of large-scale dense graph in internet of things. IEEE Internet of Things Journal pp. 1–1 (2021). DOI 10.1109/JIOT.2021.3138103
41. Lind, P.G., Gonzalez, M.C., Herrmann, H.J.: Cycles and clustering in bipartite networks. Physical review E **72**(5), 056,127 (2005)
42. Liu, C., Shao, Z., Li, K., Wu, M., Chen, J., Li, R., Liao, X., Jin, H.: Scalabfs: A scalable bfs accelerator on fpga-hbm platform. In: The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, FPGA '21, p. 147. Association for Computing Machinery, New York, NY, USA (2021). DOI 10.1145/3431920.3439463. URL https://doi.org/10.1145/3431920.3439463
43. Liu, H., Huang, H.H.: Enterprise: breadth-first graph traversal on gpus. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, pp. 1–12 (2015)
44. Liu, H., Huang, H.H.: {SIMD-X}: Programming and processing of graph algorithms on {GPUs}. In: 2019 USENIX Annual Technical Conference (USENIX ATC 19), pp. 411–428 (2019)
45. Liu, J., Wang, W.: Op-cluster: Clustering by tendency in high dimensional space. In: Third IEEE international conference on data mining, pp. 187–194. IEEE (2003)
46. Lyu, B., Qin, L., Lin, X., Zhang, Y., Qian, Z., Zhou, J.: Maximum biclique search at billion scale. Proceedings of the VLDB Endowment (2020)
47. Ma, L., Yang, Z., Miao, Y., Xue, J., Wu, M., Zhou, L., Dai, Y.: Neugraph: Parallel deep neural network computation on large graphs. In: 2019

USENIX Annual Technical Conference (USENIX ATC 19), pp. 443–458. USENIX Association, Renton, WA (2019). URL https://www.usenix.org/conference/atc19/presentation/ma
48. Mai, S.T., Dieu, M.S., Assent, I., Jacobsen, J., Kristensen, J., Birk, M.: Scalable and interactive graph clustering algorithm on multicore cpus. In: 2017 IEEE 33rd International Conference on Data Engineering (ICDE), pp. 349–360 (2017). DOI 10.1109/ICDE.2017.94
49. Malewicz, G., Austern, M.H., Bik, A.J., Dehnert, J.C., Horn, I., Leiser, N., Czajkowski, G.: Pregel: a system for large-scale graph processing. In: Proceedings of the 2010 ACM SIGMOD International Conference on Management of data, pp. 135–146 (2010)
50. Melckenbeeck, I., Audenaert, P., Van Parys, T., Van De Peer, Y., Colle, D., Pickavet, M.: Optimising orbit counting of arbitrary order by equation selection. BMC bioinformatics **20**(1), 1–13 (2019)
51. Mislove, A., Marcon, M., Gummadi, K.P., Druschel, P., Bhattacharjee, B.: Measurement and analysis of online social networks. In: Proceedings of the 7th ACM SIGCOMM conference on Internet measurement, pp. 29–42 (2007)
52. Muthukrishnan, S., et al.: Data streams: Algorithms and applications. Foundations and Trends® in Theoretical Computer Science **1**(2), 117–236 (2005)
53. Ngo, H.Q., Porat, E., Ré, C., Rudra, A.: Worst-case optimal join algorithms. Journal of the ACM (JACM) **65**(3), 1–40 (2018)
54. Nodehi Sabet, A.H., Qiu, J., Zhao, Z.: 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, ASPLOS '18, p. 622–636. Association for Computing Machinery, New York, NY, USA (2018). DOI 10.1145/3173162.3173180. URL https://doi.org/10.1145/3173162.3173180
55. Pagh, R., Silvestri, F.: The input/output complexity of triangle enumeration. In: Proceedings of the 33rd ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems, pp. 224–233 (2014)
56. Pandey, S., Wang, Z., Zhong, S., Tian, C., Zheng, B., Li, X., Li, L., Hoisie, A., Ding, C., Li, D., et al.: Trust: Triangle counting reloaded on gpus. IEEE Transactions on Parallel and Distributed Systems **32**(11), 2646–2660 (2021)
57. Park, H.M., Chung, C.W.: An efficient mapreduce algorithm for counting triangles in a very large graph. In: Proceedings of the 22nd ACM international conference on Information & Knowledge Management, pp. 539–548 (2013)
58. Pinar, A., Seshadhri, C., Vishal, V.: Escape: Efficiently counting all 5-vertex subgraphs. In: Proceedings of the 26th international conference on world wide web, pp. 1431–1440 (2017)
59. Polak, A.: Counting Triangles in Large Graphs on GPU. In: IPDPSW, pp. 740–746. IEEE (2016)
60. Qian, Z., Min, C., Lai, L., Fang, Y., Li, G., Yao, Y., Lyu, B., Zhou, X., Chen, Z., Zhou, J.: 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), pp. 321–335. USENIX Association (2021). URL https://www.usenix.org/conference/nsdi21/presentation/qian-zhengping
61. Robins, G., Alexander, M.: Small worlds among interlocking directors: Network structure and distance in bipartite graphs. Computational & Mathematical Organization Theory **10**(1), 69–94 (2004)
62. Rossi, R.A., Ahmed, N.K.: The network data repository with interactive graph analytics and visualization. In: AAAI (2015). URL https://networkrepository.com
63. Roy, A., Bindschaedler, L., Malicevic, J., Zwaenepoel, W.: Chaos: Scale-out graph processing from secondary storage. In: Proceedings of the 25th Symposium on Operating Systems Principles, pp. 410–424 (2015)
64. Roy, A., Mihailovic, I., Zwaenepoel, W.: X-stream: Edge-centric graph processing using streaming partitions. In: Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles, pp. 472–488 (2013)
65. Sanei-Mehri, S.V., Sariyuce, A.E., Tirthapura, S.: Butterfly counting in bipartite networks. In: Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, pp. 2150–2159 (2018)
66. Sanei-Mehri, S.V., Zhang, Y., Sariyüce, A.E., Tirthapura, S.: Fleet: butterfly estimation from a bipartite graph stream. In: Proceedings of the 28th ACM International Conference on Information and Knowledge Management, pp. 1201–1210 (2019)
67. Shi, J., Shun, J.: Parallel algorithms for butterfly computations. In: Symposium on Algorithmic Principles of Computer Systems, pp. 16–30. SIAM (2020)
68. Shi, T., Zhai, J., Wang, H., Chen, Q., Zhai, M., Hao, Z., Yang, H., Chen, W.: Graphset: High performance graph mining through equivalent set transformations. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, pp. 1–14 (2023)
69. Shun, J., Blelloch, G.E.: Ligra: a lightweight graph processing framework for shared memory. In: Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming, pp. 135–146 (2013)
70. Su, X., Khoshgoftaar, T.M.: A survey of collaborative filtering techniques. Adv. in Artif. Intell. **2009** (2009). DOI 10.1155/2009/421425. URL https://doi.org/10.1155/2009/421425
71. Valiant, L.G.: A bridging model for parallel computation. Commun. ACM **33**(8), 103–111 (1990). DOI 10.1145/79173.79181. URL https://doi.org/10.1145/79173.79181
72. Vora, K.: {LUMOS}:{Dependency-Driven} disk-based graph processing. In: 2019 USENIX Annual Technical Conference (USENIX ATC 19), pp. 429–442 (2019)
73. Vora, K., Xu, G., Gupta, R.: Load the edges you need: A generic {I/O} optimization for disk-based graph processing. In: 2016 USENIX Annual Technical Conference (USENIX ATC 16), pp. 507–522 (2016)
74. Vuppalapati, M., Miron, J., Agarwal, R., Truong, D., Motivala, A., Cruanes, T.: Building an elastic query engine on disaggregated storage. In: 17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20), pp. 449–462. USENIX Association, Santa Clara, CA (2020). URL https://www.usenix.org/conference/nsdi20/presentation/vuppalapati
75. Wahib, M., Maruyama, N.: Scalable kernel fusion for memory-bound gpu applications. In: SC'14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, pp. 191–202. IEEE (2014)
76. Wang, J., Fu, A.W.C., Cheng, J.: Rectangle counting in large bipartite graphs. In: 2014 IEEE International Congress on Big Data, pp. 17–24. IEEE (2014)

77. Wang, K., Hu, Y., Lin, X., Zhang, W., Qin, L., Zhang, Y.: A cohesive structure based bipartite graph analytics system. In: Proceedings of the 30th ACM International Conference on Information & Knowledge Management, pp. 4799–4803 (2021)
78. Wang, K., Lin, X., Qin, L., Zhang, W., Zhang, Y.: Vertex priority based butterfly counting for large-scale bipartite networks. Proceedings of the VLDB Endowment **12**(10), 1139–1152 (2019)
79. Wang, K., Lin, X., Qin, L., Zhang, W., Zhang, Y.: Efficient bitruss decomposition for large-scale bipartite graphs. In: 2020 IEEE 36th International Conference on Data Engineering (ICDE), pp. 661–672. IEEE (2020)
80. Wang, K., Lin, X., Qin, L., Zhang, W., Zhang, Y.: Accelerated butterfly counting with vertex priority on bipartite graphs. The VLDB Journal pp. 1–25 (2022)
81. Wang, K., Lin, X., Qin, L., Zhang, W., Zhang, Y.: Towards efficient solutions of bitruss decomposition for large-scale bipartite graphs. The VLDB Journal **31**(2), 203–226 (2022)
82. Wang, K., Zhang, W., Zhang, Y., Qin, L., Zhang, Y.: Discovering significant communities on bipartite graphs: An index-based approach. IEEE Transactions on Knowledge and Data Engineering (2021)
83. Wang, Y., Davidson, A., Pan, Y., Wu, Y., Riffel, A., Owens, J.D.: Gunrock: A high-performance graph processing library on the gpu. SIGPLAN Not. **51**(8) (2016). DOI 10.1145/3016078.2851145. URL `https://doi.org/10.1145/3016078.2851145`
84. Wang, Z., Meng, Z., Li, X., Lin, X., Zheng, L., Tian, C., Zhong, S.: Smog: Accelerating subgraph matching on gpus. In: 2023 IEEE High Performance Extreme Computing Conference (HPEC), pp. 1–7. IEEE (2023)
85. Wu, B., Zhao, Z., Zhang, E.Z., Jiang, Y., Shen, X.: Complexity analysis and algorithm design for reorganizing data to minimize non-coalesced memory accesses on gpu. ACM SIGPLAN Notices **48**(8), 57–68 (2013)
86. Xu, Q., Zhang, F., Yao, Z., Lu, L., Du, X., Deng, D., He, B.: Efficient load-balanced butterfly counting on gpu. Proceedings of the VLDB Endowment **15**(11), 2450–2462 (2022)
87. Yang, Z., Lai, L., Lin, X., Hao, K., Zhang, W.: Huge: An efficient and scalable subgraph enumeration system. In: Proceedings of the 2021 International Conference on Management of Data, pp. 2049–2062 (2021)
88. Yaşar, A., Rajamanickam, S., Berry, J., Wolf, M., Young, J.S., ÇatalyÜrek, Ü.V.: Linear Algebra-Based Triangle Counting via Fine-Grained Tasking on Heterogeneous Environments: (Update on Static Graph Challenge). In: HPEC, pp. 1–4 (2019)
89. Zeng, L., Zou, L., Özsu, M.T., Hu, L., Zhang, F.: Gsi: Gpu-friendly subgraph isomorphism. In: 2020 IEEE 36th International Conference on Data Engineering (ICDE), pp. 1249–1260. IEEE (2020)
90. Zhang, F., Chen, D., Wang, S., Yang, Y., Gan, J.: Scalable approximate butterfly and bi-triangle counting for large bipartite networks. Proceedings of the ACM on Management of Data **1**(4), 1–26 (2023)
91. Zhang, H., Yu, J.X., Zhang, Y., Zhao, K., Cheng, H.: Distributed subgraph counting: a general approach. Proceedings of the VLDB Endowment **13**(12), 2493–2507 (2020)
92. Zhang, J., Li, J.: Degree-aware hybrid graph traversal on fpga-hmc platform. In: Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, FPGA '18, p. 229–238. Association for Computing Machinery, New York, NY, USA (2018). DOI 10.1145/3174243.3174245. URL `https://doi.org/10.1145/3174243.3174245`
93. Zhao, C., Guan, Y.: A graph-based investigation of bitcoin transactions. In: G. Peterson, S. Shenoi (eds.) Advances in Digital Forensics XI, pp. 79–95. Springer International Publishing, Cham (2015)
94. Zhao, G., Wang, K., Zhang, W., Lin, X., Zhang, Y., He, Y.: Efficient computation of cohesive subgraphs in uncertain bipartite graphs. In: 2022 IEEE 38th International Conference on Data Engineering (ICDE), pp. 2333–2345. IEEE (2022)
95. Zhao, T., Malir, M., Jiang, M.: Actionable objective optimization for suspicious behavior detection on large bipartite graphs. In: 2018 IEEE International Conference on Big Data (Big Data), pp. 1248–1257 (2018). DOI 10.1109/BigData.2018.8621975
96. Zhou, A., Wang, Y., Chen, L.: Butterfly counting on uncertain bipartite graphs. Proceedings of the VLDB Endowment **15**(2), 211–223 (2021)
97. Zhu, Q., Zheng, J., Yang, H., Chen, C., Wang, X., Zhang, Y.: Hurricane in bipartite graphs: The lethal nodes of butterflies. In: 32nd International Conference on Scientific and Statistical Database Management, SSDBM 2020. Association for Computing Machinery, New York, NY, USA (2020). DOI 10.1145/3400903.3400916. URL `https://doi.org/10.1145/3400903.3400916`
98. Zhu, R., Zou, Z., Li, J.: Fast rectangle counting on massive networks. In: 2018 IEEE International Conference on Data Mining (ICDM), pp. 847–856. IEEE (2018)
99. Zhu, X., Han, W., Chen, W.: {GridGraph}:{Large-Scale} graph processing on a single machine using 2-level hierarchical partitioning. In: 2015 USENIX Annual Technical Conference (USENIX ATC 15), pp. 375–386 (2015)
100. Zweig, K.A., Kaufmann, M.: A systematic approach to the one-mode projection of bipartite graphs. Social Network Analysis and Mining **1**(3), 187–218 (2011)