Practicably Boosting the Processing Performance of BFS-like Algorithms on Semi-External Graph System via I/O-Efficient Graph Ordering

Tsun-Yu Yang, Yuhong Liang, and Ming-Chang Yang,
The Chinese University of Hong Kong

https://www.usenix.org/conference/fast22/presentation/yang

This paper is included in the Proceedings of the
20th USENIX Conference on File and Storage Technologies.
February 22–24, 2022 • Santa Clara, CA, USA
978-1-939133-26-7

Open access to the Proceedings
of the 20th USENIX Conference on
File and Storage Technologies
is sponsored by USENIX.

Practicably Boosting the Processing Performance of BFS-like Algorithms on Semi-External Graph System via I/O-Efficient Graph Ordering

Tsun-Yu Yang, Yuhong Liang, and Ming-Chang Yang
The Chinese University of Hong Kong

Abstract

As graphs continue to grow to have billions of vertices and edges, the attention of graph processing is shifted from in-memory graph system to external graph system. Of the two the latter offers a cost-effective option for processing large-scale graphs on a single machine by holding the enormous graph data in both memory and storage. Although modern external graph systems embrace many advanced I/O optimization techniques and can perform well in general, graph algorithms that build upon Breadth-First Search (BFS) (a.k.a. BFS-like algorithms) still commonly suffer poor processing performance. The key reason is that the recursive vertex traversal nature of BFS may lead to poor I/O efficiency in loading the required graph data from storage for processing.

Thus, this paper presents I/O-Efficient Graph Ordering (IOE-Order) to pre-process the graph data, while better I/O efficiency in loading storage-resident graph data can be delivered at runtime to boost the processing performance of BFS-like algorithms. Particularly, IOE-Order comprises two major pre-processing steps. The first is Breadth-First Degree-Second (BFDS) Ordering, which exploits both graph traversal pattern and degree information to store the vertices and edges which are most likely to be accessed together for I/O efficiency improvement. The second is Out-Degree Binning, which splits the BFDS-ordered graph into multiple sorted bins based on out-degrees of vertices so as to 1) further increase I/O-efficiency for runtime graph processing and 2) deliver high flexibility in pre-caching vertices based on the memory availability. In contrast to the state-of-the-art pre-processing techniques for BFS-like algorithms, IOE-Order demonstrates better efficiency and practicability: It delivers higher processing performance by achieving higher I/O efficiency but entails much lower pre-processing overhead.

1 Introduction

Breadth-First Search (BFS) is the foundation of many popular and important graph algorithms (a.k.a. BFS-like algorithms) that share a common feature called recursive graph traversal. That is, given a starting set of vertices, their adjacent vertices (i.e., neighbors) will be explored recursively until all the connected vertices are visited. Due to this feature of exploration, BFS-like algorithms are useful in various domains, such as networking [13], bioinformatics [20, 32], social media [8, 23, 51], and others [24, 31, 38]. In addition, based on the survey [42], BFS-like algorithms are popular. Particularly, among 13 typical graph algorithms, Connected-Component is most widely used, and Shortest-Path and Betweenness-Centrality are also within the top five: They are all BFS-like. Moreover, the BFS-like recursive graph traversal also plays a critical role in many important graph mining algorithms such as Subgraph Searching and Pruning [26, 36].

However, BFS-like algorithms generally have poor locality of access. Specifically, compared with other graph algorithms such as PageRank [18, 39] and Sparse Matrix-Vector Multiplication [29] where all vertices are regularly visited, BFS-like algorithms only visit a subset of vertices at any given time. More seriously, it is very challenging to predict how the vertices are going to be visited given the fact that the BFS can start with any vertex. As a result, even till today, how to efficiently process graphs using BFS-like algorithms continues drawing a lot of attention in both academia and industry.

On the other hand, as graphs continue to grow and cannot fit in the memory of a machine, people start to leverage the massive storage to keep the enormous graph data for graph processing at low cost. Among several feasible solutions (which will be presented in Section 2.1 in details), the *semi-external graph system* is a popular option that demonstrates its capability of efficiently processing the large-scale graph in a single machine [53]. Due to the complex relationships (i.e., edges) among entities (i.e., vertices) in real-world graphs, the number of edges is typically significantly larger than that of vertices [53]. Thus, semi-external graph systems propose to keep the large-sized edge data in the massive storage for holding a large-scale graph at low cost, but maintain the small-sized vertex data in the faster memory for offering better performance of graph processing (that typically generates lots of small and random updates to the attributes of

vertices). Fortunately, since the memory space in commodity PCs nowadays is generally large enough to hold the vertices of most of large-scale graphs [2], semi-external graph system is regarded as a cost-effective model in graph processing and several excellent semi-external graph systems have been developed [25, 29, 44, 53].

To further improve the performance of loading edges from the slower storage, various effective I/O optimization techniques have been suggested and integrated in modern semi-external graph systems (which will be introduced in Section 2.1 in detail). However, these general techniques could only bring limited improvement to BFS-like algorithms due to the lack of consideration of the BFS's recursive graph traversal nature. Thus, Lee et al. try to tackle the poor performance issue of BFS-like algorithms via *pre-processing optimizations* of *ordering* and *pre-caching*. Specifically, ordering is a technique to re-order the graph to improve the locality of access, whereas pre-caching is to pre-load the data in memory which will not be changed during the entire execution (see Section 2.2 for details). Nevertheless, based on our evaluations, their designs for BFS-like algorithms still leave a substantial room for improvement due to the limited *I/O efficiency*; furthermore, they may even suffer the critical issue of limited practicability for spending considerable time on pre-processing compared to the improvement that they bring (see Section 2.3 for details).

To boost the processing performance of BFS while deliver high practicability, this paper proposes *I/O-Efficient Graph Ordering (IOE-Order)*, which comprises two steps to pre-process the graph, while better I/O efficiency in loading edge data can be achieved during graph processing. The first step is called Breadth-First Degree-Second (BFDS) Ordering. Specifically, BFDS not only exploits the graph traversal pattern to capture the global structure of a graph, but also, based on the global structure, keeps the neighbors of high in-degree vertices together so that more I/O requests with high I/O efficiency can be issued during graph processing.

The second step is Out-Degree (OutD) Binning. Under BFDS-ordered graph, OutD Binning further splits the edge data into multiple bins based on the sizes of edge lists (i.e., out-degrees of vertices). Additionally, all the bins are sorted and stored sequentially on the graph according to their average out-degrees. In this way, the vertices of small out-degree can be physically separated and then efficiently pre-cached, while loading data from the rest of bins could enjoy much higher I/O efficiency. Furthermore, the design of multiple bins in the graph provides high flexibility. That is, semi-external graph systems can easily pre-cache edge data starting from the bin with the smallest average out-degree based on the different amounts of memory in various machines.

We implement IOE-Order in C++ and evaluate its effectiveness on Graphene [29], which is an open-sourced, state-of-the-art semi-external graph system. In particular, we enrich Graphene to support pre-caching. Our evaluations based on billion-scale graphs reveal that, compared with the integrated solution of state-of-the-art pre-processing optimizations [28], IOE-Order delivers better efficiency and practicability. In terms of efficiency, IOE-Order can improve the processing time of various BFS-like algorithms by 18.8% on average and even up to 36.1%, thanks to its efficacy in increasing/optimizing I/O efficiency from 70.9% to 82.1% on average and even up to 98.8%. As for the practicability, IOE-Order entails much lower (i.e., 815.2× lower) online pre-processing overhead by enabling a flexible and efficient way to pre-cache edges with a holistic graph ordering.

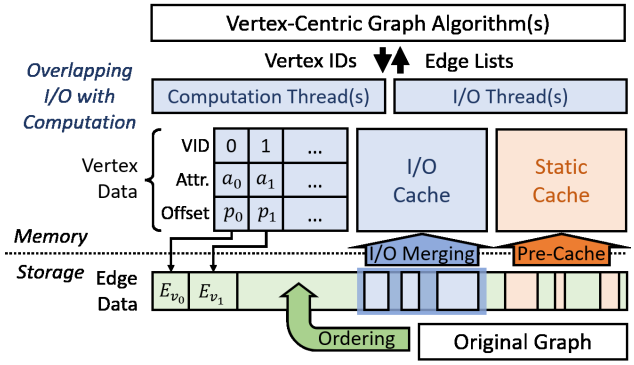
The rest of this paper is organized as follows: Section 2 presents the background and motivation regarding this work. Section 3 introduces the main design of I/O-Efficient Graph Ordering. Next, Section 4 demonstrates the evaluation results. Finally, Section 5 discusses the related work and Section 6 concludes this work.

2 Background and Motivation

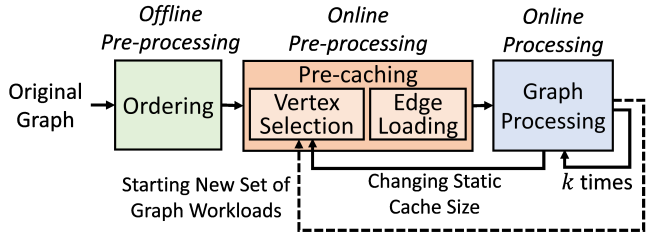
2.1 Semi-External Graph Processing

A graph generally comprises two sets of data: *vertex data* that consist of a set of vertices along with vertex attributes and *edge data* that describe the set of edges linking two vertices along with edge properties. That is, an edge describes the neighboring relationship of two connected vertices, and for an edge $e = (u, v)$ in a directed graph, v is referred to as the *out-neighbor* of u while u is referred to as the *in-neighbor* of v. The terms *out-degree* and *in-degree* further indicate the number of out-neighbors and in-neighbors for a given vertex respectively. In practice, in the vertex data, each vertex is assigned with a distinct value, called *vertex ID*, for identification purpose. Edge data, on the other hand, represent all the edges in the form of *edge list* that enumerates the neighbors' vertex IDs for a specific vertex, and all the edge lists are further sorted by vertex IDs. Thus, the edge list of a given vertex in the edge data can be easily indexed by the vertex ID.

As depicted in Figure 1(a), the semi-external graph system is introduced to cost-effectively process the graph data by 1) keeping the small-sized but frequently-updated vertex data in the faster memory while 2) storing the large-sized but mostly-read-only edge data in the cheaper storage. Particularly, in modern semi-external graph systems [25, 29, 44, 53], a vertex's attribute and file offset to its edge list are maintained in memory, and the entire edge data are stored in storage as file(s). On the other hand, the modern semi-external graph systems usually support the *push-style, vertex-centric programming model* [33] because of its ability to express a lot of graph algorithms and its ease for distributed and parallelized execution [21]. Under such programming model, graph algorithms are designed to iteratively specify and activate a subset of vertices (a.k.a. *active vertices*) that need to be processed in the following iteration. Thus, the modern semi-external graph systems typically provide the API to load all the edge lists of



(a) Typical System Architecture of Semi-External Graph System.



(b) Graph Processing Flow with Ordering and Pre-Caching.

Figure 1: An Overview of Semi-External Graph System.

active vertices from the edge data, so that graph algorithms can, based on the loaded edge data, efficiently update the vertex attributes in memory and generate a new set of active vertices for the next iteration of processing.

As presented in Figure 1(a) as well, to further improve the efficiency of loading edge data from storage, from bottom to top, modern semi-external graph systems also introduce several general I/O optimization techniques as follows:

I/O Merging. Solid-state drives (SSDs), which are adopted in most state-of-the-art semi-external graph systems, deliver better I/O throughput under I/O requests of larger sizes [29, 53]. Therefore, when the I/O requests are issued to retrieve the blocks in SSD, several small requests are typically merged into a large request for higher I/O throughput [29, 53]. This technique is called I/O merging. Taking FlashGraph [53] as an example, it merges 4KB I/O requests to consecutive blocks if possible, so an I/O request actually issued by FlashGraph could typically range from 4 KB up to many MBs. By contrast, Graphene [29] issues 512 B I/Os and aggressively merges them to close (but not necessary to be consecutive) blocks and forms a larger I/O request (up to 16 KB) and submits a great amount of asynchronous I/Os to saturate I/O throughput.

I/O Cache. After the completion of I/O requests, the loaded edge lists are typically kept in the I/O cache, which is a small amount of user-space memory managed by the semi-external graph system, and wait for being processed. Different systems usually have their strategies to manage the I/O cache. For instance, FlashGraph [53] adopts traditional page cache management strategy (i.e., g-clock algorithm) to evict the less-frequently-accessed data from the I/O cache. On the other hand, since Graphene [29] issues fine-grained 512 B I/Os, it directly discards the loaded edge lists, instead of keeping them for future use, in I/O cache after being processed to better utilize I/O cache.

Overlapping I/O with Computation. The mainstream semi-external graph systems typically overlap I/O with computation to have a significant improvement in performance. To realize this functionality, *asynchronous I/O* is one key technique since it can allow a thread to do the computation while there are several ongoing I/Os in the background [53]. Another key technique is to leverage the *multi-thread programming* that enables the separation of computation jobs and I/O jobs in different threads so that both type of jobs can be done parallelly [29].

2.2 Pre-processing for BFS-like Algorithms

Although modern semi-external graph systems integrate several general I/O optimization techniques, BFS-like algorithms are still notorious for their poor processing performance [28]. The reason is twofold: First, the I/O optimization techniques introduced in Section 2.1 are for general graph algorithms. Therefore, their designs do not particularly favor the recursive graph traversal nature of BFS. Second, real-world graphs usually have irregular structure and follow power-law distribution [16]. In other words, the edge lists of a vertex's neighbors tend to be scattered across the edge data, and their actual sizes are much smaller than the block granularity of I/O requests.

To alleviate the poor processing performance of BFS-like algorithms on semi-external graph system, Lee et al. look for the opportunity of *pre-processing* the graph data [28]. As shown in Figure 1(b), before processing the graph by BFS-like graph algorithms, they propose to first pre-process the graph by two stages: *ordering* and *pre-caching*.

Ordering. In general, ordering is a common technique to convert a graph into a new one by re-assigning vertex IDs and re-ordering the edge lists in the edge data based on the newly assigned vertex IDs [28]. Due to this nature, typically, ordering only needs to be applied once per graph and can be done by using a powerful server. Thus, we refer to the ordering stage as *offline pre-processing*.

In contrast to the existing ordering techniques that are mainly designed for in-memory graph systems [33, 37], Lee et al. introduce *Neighborhood Ordering (Norder)* [28] to achieve better access locality for semi-external graph systems. Its objective is to assign neighboring vertices with close vertices IDs so that the edge lists of neighboring vertices can be thereby stored closely in the edge data for the reduction of I/O cost. In practice, Norder minimizes the standard deviation of the neighboring vertex IDs by performing the BFS with depth level bound [28] starting from the highest in-degree vertex. The depth level is bounded to two because they empirically found it to be effective for overall performance.

Pre-caching. Static cache is a common technique, which can be found in many system designs [15, 30, 47], to reduce the number of issued I/Os. Unlike traditional cache which replaces data upon cache miss, the data in static cache are pre-loaded and will not be evicted during the whole execution.

Lee et al. utilize the static cache to selectively pre-cache some edge data before processing the graph [28]. Since a graph is usually analyzed many times (i.e., number k in Figure 1(b)) for obtaining various information [52], the pre-cached data can benefit the graph processing for multiple rounds until all the graph workloads are completed. Please note that, since the edge data must be adaptively pre-cached based on the static cache size and graph workloads at runtime, the pre-caching stage needs to be re-performed whenever the available memory space for static cache changes or a new set of graph workloads launches. Thus, we refer to the pre-caching stage as *online pre-processing*.

As illustrated in Figure 1(b), the pre-caching stage can be further divided into two steps: *vertex selection* and *edge loading*. Particularly, the step of vertex selection has a direct impact on how I/Os can be reduced, since this step determines which edge data concerned with the selected vertices are going to be pre-cached (from the storage) during the step of edge loading. Thus, to reduce small and random I/Os during graph processing, Lee et al. propose *Greedy Vertex Selection (GVS)* to pick out the vertices whose edge lists are not stored in the same I/O block as their siblings (i.e., vertices sharing the same in-neighbor) [28].

2.3 Motivation: I/O Efficiency of BFS-like Algorithms

Despite the fact that Norder and GVS indeed achieve noticeable performance improvement for BFS-like algorithms, the question of *how close we are from the optimal processing performance* still remains. Thus, this section will answer this question, through a series of theoretical modelling and practical evaluations, from a new perspective: *I/O efficiency*.

Since BFS-like algorithms generally show higher demands for I/O than computation [29] and modern semi-external graph systems typically overlap the I/O with computation [29, 53], the I/O performance basically dominates the overall processing performance of BFS-like algorithms. Thus, the *processing performance* (denoted as *Proc. Perf.*) of BFS-like algorithm on a semi-external graph system can be first expressed as Equation 1: That is, the processing performance is proportional to the number of bytes actually processed by the BFS-like algorithm (denoted as *Proc. Bytes*) but is in inverse propotation to the total time spent on I/O (denoted as denotes the number of bytes actually processed by the BFS-like algorithm and *I/O Time*).

$$\textit{Proc. Perf.} \propto \frac{\textit{Proc. Bytes}}{\textit{I/O Time}} = \frac{\textit{Trans. Bytes}}{\textit{I/O Time}} \times \frac{\textit{Proc. Bytes}}{\textit{Trans. Bytes}} \tag{1}$$

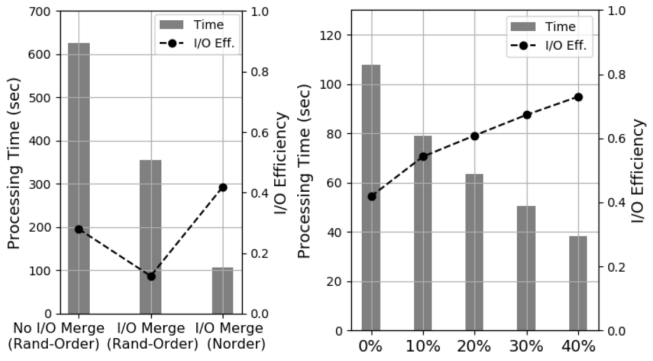
Where:

$$\textit{I/O Throughput} = \frac{\textit{Trans. Bytes}}{\textit{I/O Time}}, \tag{2}$$

$$\textit{I/O Efficiency} = \frac{\textit{Proc. Bytes}}{\textit{Trans. Bytes}}. \tag{3}$$

If we introduce the total number of transferred bytes (denoted as *Trans. Bytes*) into Equation 1, the processing performance can be further expressed into the product of two critical components: *I/O throughput* and *I/O efficiency*. As expressed in Equations 2 and Equation 3, I/O throughput represents the total number of transferred bytes (*Trans. Bytes*) within the total time spent on I/O (*I/O Time*), while I/O efficiency indicates the ratio of the total number of processed bytes (*Proc. Bytes*) to the total number of transferred bytes (*Trans. Bytes*).

From Equation 1, we can learn that *the key to optimize processing performance is by simultaneously maintaining high I/O throughput and high I/O efficiency*. However, in practice, there usually exists a trade-off between them. Taking Graphene [29] as an example, on one hand, it proposes to exploit fine-grained I/O granularity to avoid low I/O efficiency, but on the other hand, leverages I/O merging to achieve high I/O throughput. Thus, to see how the I/O merging affects the overall processing performance of BFS-like algorithms in practice, Figure 2(a) presents the results of performing a BFS algorithm on Graphene [29] using the large-scale uk2007 graph [48], which comprises nearly four billions of edges. In this figure, the x-axis indicates whether the I/O merging is enabled and the ordering algorithm used to re-order the evaluated graph, while the y-axis demonstrates the overall processing performance (in terms of the total processing time) and the I/O efficiency. It can be clearly observed that although the I/O merging can effectively reduce the total processing time by 43%, it also degrades the I/O efficiency by 12.4% when the evaluated graph is randomly re-ordered (denoted as Rand-Order).



(a) I/O Merging and Ordering. (b) Norder and GVS (with I/O Merging).

Figure 2: A Series of Evaluations of Running BFS Algorithm on Graphene [29] with `uk2007` Graph [48].

Figure 2(a) also reveals how the state-of-the-art Norder algorithm helps to improve the I/O efficiency with the I/O

merging enabled. Particularly, we can clearly observe that Norder indeed shows its efficacy to further reduce the total processing time by 69.7% against Rand-Order. Nevertheless, the actual I/O efficiency under Norder is still quite low, which is only 41.9%; that is, almost 60% of the bytes are loaded but useless over all the transferred bytes.

To further understand whether the state-of-the-art GVS can help improving the I/O efficiency, we vary the static cache size to be 0% (i.e., no static cache), 10%, ..., to 40% of the edge data size in uk2007 graph [48] and have the graph re-ordered by Norder. As shown in Figure 2(b), as the static cache size keeps increasing, GVS can not only keep decreasing the total processing time but also have the potential to improve the I/O efficiency. This is because GVS aims to reduce the number of small and random I/Os during graph processing. In other words, when more edge data are pre-cached by GVS in a static cache of larger sizes, more small and random I/Os to the pre-cached edge data can be completed in the faster static cache for preventing incurring I/Os with low I/O efficiency. Nevertheless, such improvement comes at a huge cost and demand for the static cache size. As shown in Figure 2(b), only when the static cache size is up to 40% of the evaluated edge data size, a nearly 75% (specifically, 73.1%) of the I/O efficiency can be achieved eventually. This not only exposes the ineffectiveness of GVS in utilizing the static cache, but even makes semi-external graph system still costly to process BFS-like algorithms on large-scale graphs.

Table 1: Pre-processing Time of Norder and GVS (seconds).

Static Cache Size	Ordering (Norder)	Vertex Selection (GVS)	Edge Loading (based on GVS)
10%	51.7	2,914.7	16.3
20%		4,708.3	17.6
30%		6,061.0	18.7
40%		7,053.4	19.6

There is even one more thing that may further limit the practicability of the existing the pre-processing techniques, particularly GVS, if the pre-processing overhead is considered. Table 1 shows the pre-processing time of Norder, GVS, and the edge loading time based on GVS. It can be clearly observed that, although the pre-processing time of Norder is about 51.7 seconds, the ordering stage is actually an offline preprocessing optimization which only introduces an one-shot overhead. By contrast, GVS, which is online pre-processing optimization, requires up to 7,053 seconds (which is almost 67.17× of the total processing time of BFS) when the static cache size is as large as 40% of the edge data. Given the fact that GVS needs to be re-performed whenever the size of static cache changes or a new set of graph workloads launches, such high time complexity may make GVS fail to reduce the end-to-end graph processing time especially when the number of workloads (i.e., number k in Figure 1(b)) is not large enough to amortize the huge overhead of GVS.

3 I/O-Efficient Graph Ordering

3.1 Overview

Based on the investigations presented in Section 2.3, we realize that optimizing the I/O efficiency of loading storage-resident edge data is crucial to boost the processing performance of BFS-like algorithms on semi-external graph system. Thus, this section introduces a new *I/O-Efficient Graph Ordering* that not only enables higher I/O efficiency for better processing performance by pre-processing the graph but also demonstrates great practicability by entailing low pre-processing overhead.

As shown in Figure 3, the proposed IOE-Order consists of two major steps to re-order the graph. The first step is *Breadth-First Degree Second (BFDS) Ordering* (see Section 3.2) that exploits both graph traversal pattern and in-degrees of vertices to re-assign the vertex IDs such that the edge lists of vertices, which are most likely to be together traversed by BFS-like algorithms, can be thereby closely stored in the edge data. This step can guarantee that higher I/O efficiency in loading the edge data involved in graph traversals can be achieved, even when the I/O merging is also enabled for high I/O throughput.

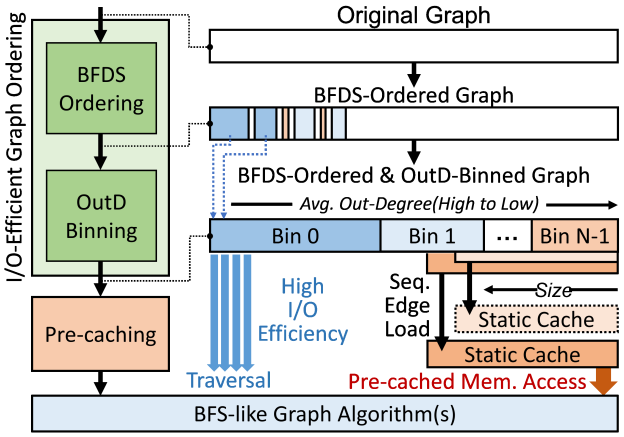


Figure 3: An Overview of I/O-Efficient Graph Ordering.

The second step, *Out-Degree (OutD) Binning* (see Section 3.3), is then introduced to split the BFDS-ordered graph into multiple bins based on the sizes of edge lists (out-degree), while edge lists within each bin are still sorted based on the BFDS-suggested order. Moreover, all these bins are sorted and stored sequentially on the edge data based on the their average out-degrees in descending order to form the final graph ordering. This step ensures that the small edge lists can be physically separated and then pre-cached, so that the I/O efficiency of loading those bins of large edge lists can be thereby improved.

Last but not the least, as illustrated in Figure 3, the final BFDS-ordered and OutD-binned graph also suggests a flexible and efficient way to utilize the static cache. Particularly,

we can easily and sequentially load the edge lists of bin(s) in the reverse order based on the available size of static cache. In other words, the final graph is a holistic graph ordering which cleverly embeds the suggestion of vertex selection to completely eliminate the online pre-processing overhead of selecting vertices for pre-caching edges.

3.2 Breadth-First Degree-Second Ordering

3.2.1 Design Concept

The state-of-the-art ordering (i.e., Norder [28]) for BFS-like algorithms mainly relies on the intuition that the neighboring vertices will be typically traversed together. This intuition, although it is generally correct, it may overlook the importance of graph traversal nature of BFS. Particularly, in contrast to the neighboring information which could only provide the local information regarding vertices, the graph traversal pattern can offer the global and structural view about the whole graph and imply that which vertices may have higher probabilities to be traversed earlier/late than others.

To examine how the graph traversal pattern can help with the improvement in I/O efficiency, let us consider a typical BFS starting with a given vertex v^*. Suppose that the graph is "coincidentally" re-ordered according to the BFS traversal pattern starting with v^*, the I/O efficiency in performing the BFS (starting with the vertex v^*) can theoretically reach the optimal (i.e., 100%). In such case, the I/O accesses to the storage-resident edge data will be like sequentially sliding over the storage space from the smallest vertex ID to the largest vertex ID.

However, the optimal ordering might not always be the case in practice, because BFS can start with any vertex. Fortunately, we observe that graph traversal pattern, along with the in-degree information of vertices, can provide us the following probabilistic hints about how vertices are going to be traversed earlier/late than others. First, high in-degree vertices have higher possibility to be visited earlier than low in-degree ones. Second, during graph traversal, the vertices which are recursively connected to high in-degree vertices are also likely to be traversed earlier. Thus, not only the high in-degree vertices but also the vertices recusively connected to high in-degree vertices shall be best stored with their respective neighbors closely in storage for delivering higher I/O efficiency.

Based on the above insights, we introduce the *Breadth-First Degree-Second (BFDS) Ordering* that performs the graph traversal with the consideration of the in-degree information of vertices to order a graph. Particularly, BFDS performs the graph traversal, starting with the highest in-degree vertex (which demonstrates the highest probability of being traversed earlier) but iteratively traverses the out-neighbors based on their in-degrees. In contrast to the conventional BFS, BFDS further sorts the active vertices in an iteration according to their in-degrees in descending order so that the out-neighbors of the "sorted" active vertices are assigned with consecutive vertex IDs in order. In a nutshell, BFDS not only orders the vertices based on the graph traversal pattern (i.e., "Breadth-First") but further gives higher priority to keep the neighbors of high in-degree vertices together (i.e., "Degree-Second"). Consequently, BFDS maximally keeps vertices which are likely to be traversed together, making the I/Os issued by BFS-like algorithms enjoy higher I/O efficiency.

3.2.2 Design Details

Algorithm 1 shows the proposed BFDS ordering. Lines 1-3 are for the initialization of BFDS. Specifically, Line 1 finds out the maximum in-degree vertex, and push it to be starting vertex in Line 2. Line 3 is the new re-assigned vertex ID.

Algorithm 1 Breadth-First Degree-Second Ordering

Input: Graph $G = (V, E)$;
Output: Mapping function Map

```
1:  Find out v_max_in to be the max in-degree vertex
2:  Active ← {v_max_in}
3:  NewID ← 0
4:  while Active is not empty do
5:      Sort Active based on in-degree in descending order
6:      for v ∈ Active do
7:          for u ∈ v.neighbors & u is not visited do
8:              NextActive ← u
9:              Map.add(u, NewID)
10:             NewID ← NewID + 1
11:         end for
12:     end for
13:     Active ← NextActive
14: end while
15: Assign the rest of unvisited vertices new IDs
```

Lines 4-14 are the core function of BFDS. To begin with, Line 5 will sort all the active vertices (denoted as *Active*) based on their in-degree in descending order. Line 6 iterates all the vertices from the highest in-degree one and tries to assign consecutive IDs to its un-visited neighbors from Lines 7-11. Particularly, if a neighboring vertex is not assigned to a new ID yet, it gets a new ID and becomes the active vertex of next iteration (denoted as *NextActive*) in Line 8. Finally, the while loop will end if *Active* is empty. Nevertheless, several isolated vertices that cannot be reached by the graph traversal are still not assigned with new IDs. As a result, Line 15 will assign the rest of unvisited vertices with new IDs.

As we can see, the time complexity of BFDS ordering is $O(|E|+|V|)$ combined with the overhead incurred by Line 5. Suppose the total iteration is n and there are V_i active vertices in iteration i, The total number of visited vertices is $\sum_{i=1}^{n} V_i \leq V$. The time incurred by Line 5 shows in the following.

$$\sum_{i=1}^{n} V_i \log V_i < \sum_{i=1}^{n} V_i \log V \leq V \log V$$

Therefore, the time complexity of BFDS is $O(|E|+|V|+|V|\log|V|)$. On the other hand, the time complexity of Norder [28] is roughly $O(|E|+|V|)$, which is slightly faster than BFDS. Nevertheless, BFDS shows significant improvement in performance against Norder in Section 4. Furthermore, ordering only needs to be done once per graph and can be completed in offline [50]. We believe BFDS is an effective ordering overall in practice.

3.3 Out-Degree Binning

3.3.1 Design Concept

Although BFDS improves the I/O efficiency of BFS-like algorithms, it is still hard to reach the optimum due to the irregular structure of real-world graphs. To further optimize I/O efficiency while offering the efficient pre-caching, we propose a simple yet effective design, namely *Out-Degree (OutD) Binning*, to refine the BFDS-ordered graph. Its key idea is to isolate those edge lists, which might be harder to be properly re-ordered by the BFDS Ordering and could be the root cause of lower I/O efficiency; however, the BFDS-suggested ordering is still maximally preserved within each bin to have high I/O efficiency of loading the rest of edge lists.

Our intuition, in practice, is that *the I/O efficiency of loading a vertex's edge list might relate to its number of out-neighbors (out-degree)*, since the out-neighbors of an active vertex will typically be processed by BFS-like algorithms at a time, which thereby leads to larger number of processed bytes (i.e., numerator) in Equation 3. By contrast, as the issued I/O size is generally large for achieving higher bandwidth, loading smaller number of out-neighbors of an active vertex from storage may most likely result in lower I/O efficiency.

With such intuition in mind, Figure 4 illustrates why and how OutD Binning avoids low I/O efficiency. Particularly, we consider a simple scenario that the BFDS-ordered graph is split into two bins: Large-OutD bin (for containing larger edge lists such as E_{L_0} and E_{L_1}) and Small-OutD bin (for containing smaller edge lists such as E_{S_0} and E_{S_1}), and the edge lists of Small-OutD bin are pre-cached in the static cache. In contrast to the existing GVS which may just simply pre-cache E_{S_0} and E_{S_1} into the static cache (without binning them), the potential benefits of OutD Binning are twofold: First, since E_{S_0} and E_{S_1} are already pre-cached in the static cache, when loading the block containing E_{L_0}, OutD Binning effectively avoids the redundancy in loading E_{S_0}; instead, E_{L_1} can be actively loaded with E_{L_0} to achieve higher I/O efficiency, since E_{L_0} and E_{L_1} are supposed to be traversed together based on the BFDS ordering. Second, OutD Binning enables a more efficient pre-caching process. That is, rather than executing time-consuming vertex selection and loading the edge lists randomly and inefficiently (as the existing GVS does), OutD Binning suggests that the pre-caching process can be efficiently performed by sequentially loading the edge lists from the Small-OutD bin. Moreover, it will be more beneficial to pre-cache the edge lists from the Small-OutD bin than that from the Large-OutD bin. This is because a larger number of vertices can be pre-cached in the same size of static cache, and each vertex is typically visited the same number of times by BFS-like algorithms.

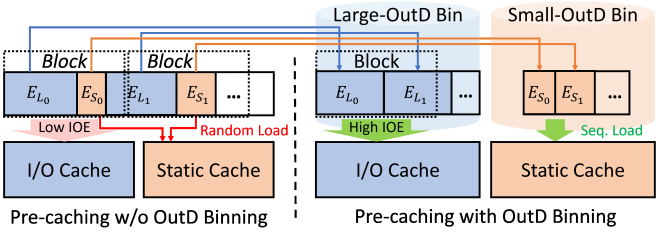


Figure 4: The Benefits of OutD-Binning.

3.3.2 Design Details

To make this design concept more practical, this section extends OutD Binning to split the BFDS-ordered graph into multiple bins. Furthermore, all bins are sorted and stored sequentially on the graph based on their average out-degrees in descending order. Thus, static cache can simply pre-cache the edge lists starting from the bin of the smallest average out-degree based on the available memory, making the pre-caching stage easy and efficient.

Nevertheless, bin size is important for overall performance. If the bin size is too small, the structure of BFDS-ordered graph could be destroyed, making the ordering less effective. If the bin size is much larger than the size of static cache, we could fail to pre-cache the critical edge lists, making static cache less effective. Thus, we propose to configure the bin sizes following the exponential decay of edge data size (as illustrated in Figure 5) since they are appropriate for large-scale graphs and perform well with or without static cache as shown in Section 4.2. Furthermore, we suggest to set the smallest bin to a size that is affordable by most of the computers (e.g., less than 2 GB) so that the whole smallest bin can be entirely pre-cached and the total number of bins could vary for graphs with different sizes.

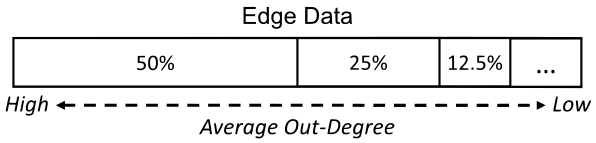


Figure 5: Layout of Exponential Decay Bin Sizes.

To order a graph with OutD Binning, we first sort all the vertices in ascending order based on their out-degrees so as to easily identify the small vertices. Next, starting from the smallest size of bin, we create bin by marking the vertices into the current bin. If the current bin is full, we will re-assign the vertex IDs based on the ordering of BFDS-optimized graph. This process keeps doing until all the bins are created.

Therefore, the time complexity of OutD Binning is $O(|V| \cdot log|V| + |V| + N \cdot |V|)$, where N is the total number of bins.

Besides the good performance, the advantage of utilizing the OutD Binning is two-fold. First, the graph optimized by OutD Binning provides high flexibility. That is, by providing multiple bins which demonstrate different I/O efficiency, OutD Binning embeds the results of vertex selection in graph; therefore, users can still easily pre-cache again without re-running OutD Binning because selection information preserves in the graph. On the contrary, GVS requires recomputation every time if the size of static cache changes or new set of graph workloads launches [28].

Second, OutD Binning enables efficient pre-caching. Particularly, OutD Binning stored the small edge lists together using bins. Thus, users can easily and efficiently pre-cache the edge lists based on the available memory in static cache. By contrast, GVS requires high complexity to select vertices in the pre-caching stage. The time complexity of GVS is $O(|E| + |V| + \sqrt{K \cdot M} + M \cdot log|V|)$, where the M is the static cache size and K is the parameter which is usually set to be hundreds or thousands [28]. Such time-consuming preprocessing might fail to reduce the end-to-end graph processing time especially when users do not have many workloads to amortize the huge overhead of GVS.

4 Evaluations

4.1 Evaluational Setup

This section conducts a series of evaluations to demonstrate the effectiveness of IOE-Order on Graphene. There are generally six classic BFS-like algorithms in the field [28], which are Breadth-First Search (BFS) [35], Single-Source Shortest Paths (SSSP) [19], All-Pair Shortest Path (APSP) [45], Weakly Connected Components (WCC) [19], Diameter Measurement (DIAM) [12], and Betweenness Centrality (BC) [7]. Nevertheless, some of them are similar to each others. For example, SSSP is implemented based on BFS, and DIAM consists of multiple rounds of BFS; besides, both of APSP and BC require the shortest paths from all vertices. Therefore, instead of evaluating all these six classical BFS-like algorithms, we only select BFS, APSP, and WCC, which demonstrate different behaviors of graph traversal. Details of the chosen algorithms are illustrated in the following:

Breadth-First Search (BFS) [35] is a typical algorithm for graph traversal. BFS begins with a user-input vertex and visits its neighbors by marking them to be the active vertices of next iteration. A visited vertex will not be visited again in the future. This procedure keeps going recursively until there is no more active vertex.

All-Pair Shortest Path (APSP) [45] computes the shortest paths from all the vertices in the graph. Due to the high complexity of computing SSSP from all vertices, an approximate approach is to randomly sample 32 source vertices, and calculate the distance by performing multi-source traversals from the sampled vertices.

Weakly Connected Components (WCC) [19] finds out the subgraphs that the vertices are all connected to each other. One way to implement WCC is to use BFS to detect the largest WCC first, and then explore the rest of smaller WCCs by exploiting label propagation.

We implement IOE-Order in C++ and evaluate its effectiveness using Graphene [29]. Notably, to the best of our knowledge, although there are several open-sourced, semi-external systems [25, 29, 53], Graphene [29] is the most state-of-the-art one that integrates multiple techniques for optimizing the graph processing on SSD; according to its paper [29], it performs the best against other existing semi-external systems (and even approaches the performance of in-memory systems such as Ligra [46] and Galois [37]). However, IOE-Order can be easily applied to other semi-external systems to further improve the performance of running BFS-like algorithms. Moreover, in contrast to the other systems [25, 53], it tackles the low I/O efficiency problem by leveraging 512-byte fine-grained I/O to read only the necessary data. Based on their designs, our method can further boost the processing performance of BFS-like algorithms by improving the I/O efficiency. Nevertheless, since Graphene does not support static cache, we realize the static cache as shown in Figure 1(a), where the static cache is implemented to be a separate memory space in addition to I/O cache.

To have a thorough comparison, we compare IOE-Order against the other three orderings incorporated with two vertex selection algorithms. In addition to Norder [28], we also add Rand-Order to be the baseline and In-degree Order (denoted as InD-Order), which re-assigns vertex IDs starting from the highest in-degree vertex. On the other hand, in addition to GVS [28], the other vertex selection algorithm is Out-Degree (denoted as OutD), which selects the vertices with small out-degree to keep as many edge lists in the static cache as possible. Because of the complexity to show many combinations of different designs, we use the notation Norder+GVS to represent the graph is ordered by Norder and the vertices are selected by GVS. By contrast, since the proposed method is a holistic optimization, we simply use IOE-Order to represent.

To show the accurate result, we run each graph algorithm five times and demonstrate the average result. However, BFS, in contrast to the other two algorithms, requires the user to input starting vertex. Thus, we randomly sample 32 starting vertices for BFS and report the average execution time. Table 2 lists all the graphs used for evaluation in this work. All of them are billion-scale, real-world graphs from webgraph [5,6]. Particularly, uk2007 [48] and gsh2015 [10] are web crawler graphs, while Twitter [11] is social graph. The largest graph in our experiment is gsh2015 [10], which contains 988 millions of vertices and 33.88 billions of edges.

All the experiments are conducted on HPE ProLiant DL560

Table 2: The Evaluated Graph Datasets.

Graph Name	Number of Vertices	Number of Edges
Twitter	42 M	1.4 B
uk2007	108 M	3.93 B
gsh2015	988 M	33.88 B

Gen10 server with Intel Xeon Platinum 8160 CPU and 1 TB DDR4-2666 memory on Debian GNU/Linux 9, and the storage device is Samsung SSD 860 EVO 1TB with SATA protocol. Please note that the actual memory used by Graphene is configured to be related to the size of edge data. Particularly, we fix the size of I/O cache to be 5% of the edge data, and vary the sizes of static cache to show the different results under various memory resources.

4.2 Evaluation of IOE-Order

We compare IOE-Order against all state-of-the-art optimizations. Figure 6 shows the overall comparison. Specifically, Figures 6(a), 6(c), and 6(b) depict the processing time of running BFS, APSP, and WCC, respectively, while Figures 6(d), 6(f), and 6(e) show the I/O efficiency of running the three algorithms in the same order. In each sub-figure, y-axis denotes the processing time or I/O efficiency and x-axis denotes the size of static cache, which ranges from 0% to 40% to the edge data size. Please note that 0% static cache means that there is no static cache but only I/O cache in Graphene. Since the loaded edge lists in static cache could be used for many times to amortize the pre-caching overhead, the results presented in Figure 6 skip the time spending on pre-caching stage, which will be discussed in detail in Section 4.3.

Overall speaking, ordering (0% static cache) has significant impact on performance. Not surprisingly, Rand-Order demonstrates the worst performance, while InD-Order averagely improves from Rand-Order by 43.6%. Norder, which is the state-of-the-art ordering optimization for BFS-like algorithms, performs better than InD-Order by 30.8%. IOE-Order further outperforms Norder by 16.5% and improve the I/O efficiency from 59.4% to 72.4% on average. Such improvement is due to BFDS ordering, which not only exploits high in-degree information but also traversal pattern to further effectively optimize the I/O efficiency.

Furthermore, we can generally observe the trend that ordering almost dominates the overall performance. That is, for most cases, given the same size of static cache, a better ordering will win no matter GVS or OutD is used. Nevertheless, GVS still outperforms OutD if the same ordering is adopted. Take Norder as an example, Norder+GVS averagely improves Norder+OutD by 10.2% in processing time. On the other hand, IOE-Order also demonstrates the effectiveness in pre-caching. Specifically, IOE-Order steadily improves Norder+GVS by 18.4%, 20.7%, 20.0%, and 18.2% regarding processing time on average as the static cache increases from 10% to 40%.

Since IOE-Order and Norder+GVS are generally the top two combinations which perform better than the others, we mainly focus on the comparison of these two in the following.

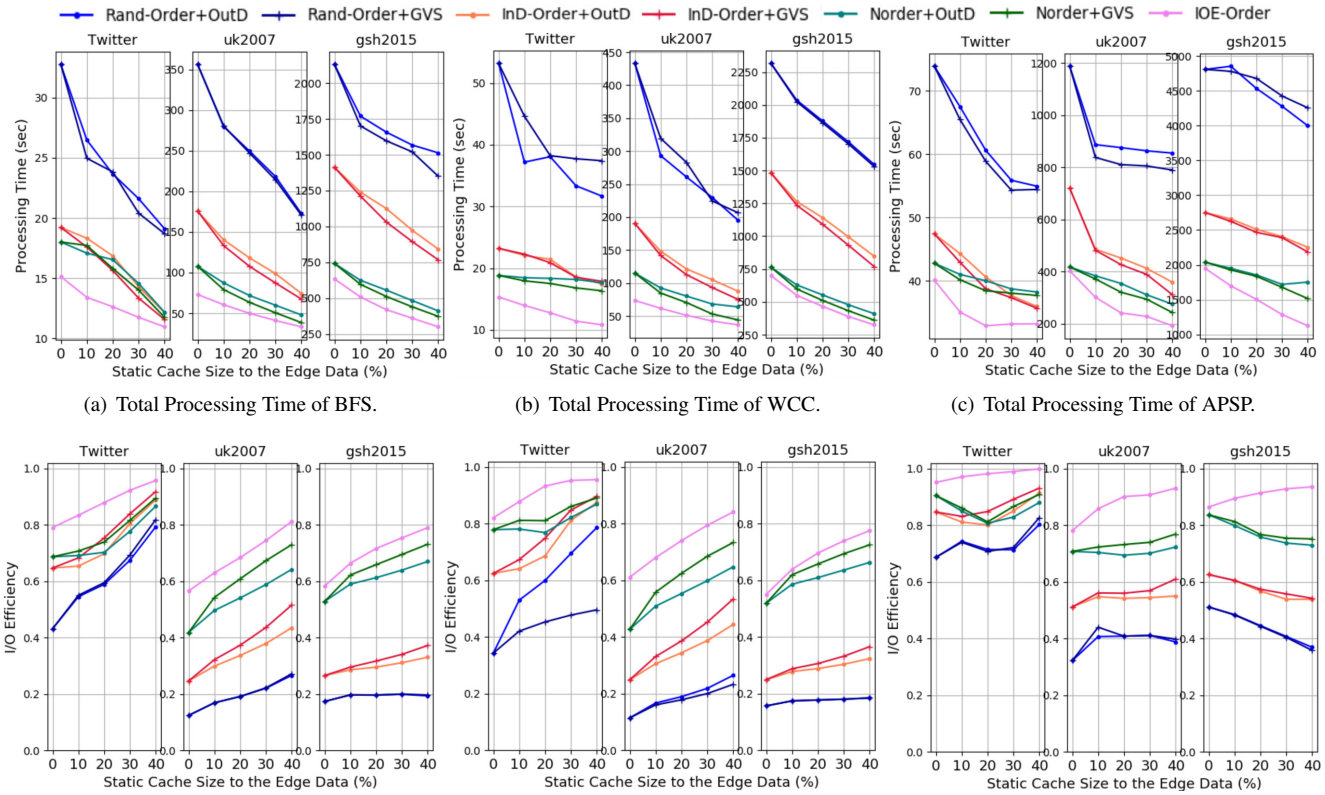
Details of BFS Evaluation. BFS, which is the foundation for all BFS-like algorithms, performs well with IOE-Order under all graphs. Averagely, IOE-Order improves 22.3%, 21.7%, 22.0%, 20.7%, and 17.3% against Norder+GVS when the size of static cache ranges from 0% to 40%. On the other hand, based on the various sizes of static cache, IOE-Order also provides the high I/O efficiencies, which averagely improves from 64.0% to 85.5%. We can see that the performance difference between IOE-Order and Norder+GVS becomes slightly smaller as the size of static cache increases. The reason is that, as more data are pre-cached, the performance gradually approaches optimum, making the optimization harder.

Details of WCC Evaluation. Generally speaking, in comparison with Norder+GVS, the processing time of IOE-Order improves averagely from 17.8% to 22.8% for various sizes of static cache. Since WCC can be decoupled into a BFS starting from a random vertex followed by label propagation, the results of WCC show a similar trend with BFS. Please refer to the discussion of BFS for more details.

Details of APSP Evaluation. Different from the other two algorithms, APSP randomly selects 32 source vertices for multi-source traversal, which requires more data from storage. Thus, APSP naturally shows higher I/O efficiency than the other two algorithms. Compared with Norder+GVS, IOE-Order averagely improves 4.7%, 14.4%, 19.0%, 20.5%, and 19.4% regarding processing time as static cache size ranges from 0% to 40%. Due to the nature of APSP, the performance difference between IOE-Order and Norder is small when there is no static cache. Nevertheless, as the size of static cache increases, GVS brings limited help in improving I/O efficiency because the I/O block issued by the graph system could contain the data which is already pre-cached in the static cache, which is likely to happen especially for the application like APSP requiring more data. Therefore, even if the processing time of Norder+GVS can still improve due to the more pre-cached data, the improvement of Norder+GVS is smaller than the improvement of IOE-Order. By contrast, since the pre-cached data are stored together in IOE-Ordered graph, the issued I/O blocks during graph processing will not contain those data, making I/O efficiency higher.

4.3 Overhead of Pre-processing

Table 3 shows the online pre-caching times of all graphs. First of all, GVS requires extremely long time to select vertices. For gsh2015 graph, it requires up to 83,131 seconds (which is around 112.19× of the total processing time of BFS) when the static cache size is 40% of the edge data. Such high time complexity might fails to reduce the end-to-end graph processing time especially when the number of graph workloads



(a) Total Processing Time of BFS. (b) Total Processing Time of WCC. (c) Total Processing Time of APSP.

(d) I/O Efficiency of BFS. (e) I/O Efficiency of WCC. (f) I/O Efficiency of APSP.

Figure 6: Overall Comparison among IOE-Order and The Other Methods.

Table 3: Online Pre-processing Time (seconds).

Graph	Twitter				uk2007				gsh2015			
Static Cache Size	10%	20%	30%	40%	10%	20%	30%	40%	10%	20%	30%	40%
Vertex Selection (OutD)	15.6	16.3	16.9	17.3	39.2	41.6	44.8	46.1	355.5	361.7	364.2	366.3
Edge Loading (OutD)	5.6	6.2	6.8	7.0	16.1	17.4	18.7	19.8	169.2	171.4	174.3	179.0
Vertex Selection (GVS)	1,181.5	1,750.3	2,096.9	2,350.3	2,914.7	4,708.3	6,061.0	7,053.4	31,781.2	53,491.6	70,388.6	83,131.4
Edge Loading (GVS)	5.5	5.9	6.4	6.6	16.3	17.6	18.7	19.6	173.2	180.3	185.6	191.4
Vertex Selection (IOE-Order)	*N/A (embedded in the step of OutD Binning)*											
Edge Loading (IOE-Order)	1.1	2.1	3.1	4.2	2.8	5.6	8.4	11.2	24.1	47.9	72.0	96.1

is not large enough to amortize the huge overhead of GVS. On the other hand, OutD demonstrates an efficient way for pre-caching, which only requires averagely 361.9 seconds for vertex selection and 173.5 seconds for edge loading for gsh2015. However, as shown in Section 4.2, OutD is less effective to benefit the graph processing than GVS. It might bring limited help when the number of graph workloads is large. Compared with the other methods, IOE-Order not only performs the best in graph processing but also enables very efficient pre-caching by eliminating the need of vertex selection. As a result, IOE-Order offers a more practical solution by bringing the greatest benefit regardless of the number of graph workloads.

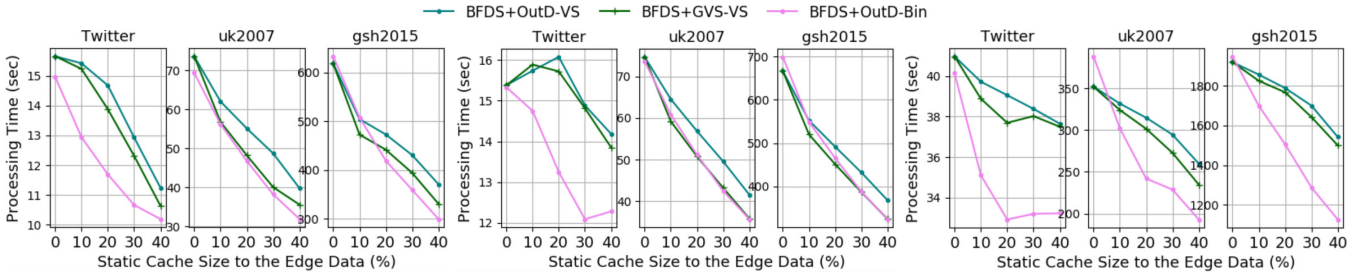
Table 4 shows the offline pre-processing times of all graphs. Since IOE-Order contains two steps to order a graph, the overhead of IOE-Order is around two to three times larger than the overhead of Norder. Fortunately, IOE-Order is inexpensive and brings more benefits than Norder as shown in Section 4.2. In particular, IOE-Order is not only 16.5% better on average but also embeds the result of vertex selection. Furthermore, since ordering is offline pre-processing, we regard IOE-Order as a more effective design than Norder.

Table 4: Offline Pre-processing Time (seconds).

Graph	Twitter	uk2007	gsh2015
Norder	24	52	483
IOE-Order (BFDS+OutD Binning)	57 (35+22)	130 (67+62)	1,580 (877+703)

4.4 Binning versus Vertex Selection

The following section will justify the benefit brought by the binning. Specifically, we show that the effectiveness of OutD Binning is better than OutD vertex selection. Although both



(a) Processing Time of BFS. (b) Processing Time of WCC. (c) Processing Time of APSP.

Figure 7: Comparison among OutD Binning, GVS-VS, and OutD-VS based on BFDS-ordered Graphs.

of designs are based on out-degree, binning can physically separate the pre-cached data, making the improvement better. Furthermore, compared with the time-consuming GVS, OutD Binning is not only much more efficient in pre-processing but even demonstrates obvious improvements for some specific cases. Please note that, since GVS needs to know how the graph is stored in storage to select vertices, it is extremely difficult to combine GVS with binning since binning will modify the order of graph all the time.

For better clarity, we denote OutD Binning as OutD-Bin, while GVS (OutD) vertex selection as GVS-VS (OutD-VS). Moreover, to have a fair comparison, we compare OutD-Bin against GVS-VS and OutD-VS based on the same ordering (BFDS ordering) with the same experiment setting as Section 4.2. Figure 7 depicts the processing times of all three algorithms, where x-axis denotes the static cache size and y-axis demonstrates the processing time. Because the space is limited and the trend of processing time and I/O efficiency is similar, we only show the results of processing time.

In general, OutD-Bin improves OutD-VS by -1.06%, 7.43%, 15.0%, 17.7%, and 17.0% as the size of static cache ranges from 0% to 40%. Since the binning will slightly damage the BFDS-ordered structure, BFDS with OutD-Bin performs marginally worse than BFDS when there is no static cache. Nevertheless, due to the effectiveness of physical separation, OutD-Bin performs gradually better than OutD-VS as the size of static cache increases. On the other hand, compared with GVS-VS, OutD-Bin improves 3.43%, 9.21%, 10.8%, and 10.0% on average as the size of static cache ranges from 10% to 40%. The major contribution of this improvement is due to APSP, where GVS-VS only brings limited help but OutD-Bin is still effective under the application with natural high I/O efficiency as discussed in Section 4.2. Particularly, the improvement can be up to 18.0% when static cache size is 40%. Furthermore, OutD-Bin performs better on Twitter, which also naturally shows higher I/O efficiency than the other graphs (as shown in Figure 6). Thus, we can also observe great improvement from GVS-VS to OutD-Bin under Twitter.

4.5 Impact of Bin Layout and Bin Sizes

As illustrated in Section 3.3, bin size for OutD Binning is crucial for overall performance. The following section shows the binning strategy of exponential decay (denoted as Exp. Decay) of the edge data size is generally better than the naïve binning strategy, which divides the graph into multiple equal-sized bins. Particularly, for naïve binning strategy, we create two graphs by setting the bin sizes to be 5% and 10% of edge data size; thus, there are totally 20 bins (denoted as 20-Bin) and 10 bins (denoted as 10-Bin) on the graphs. Due to page limitations, we only show the results of gsh2015, which is the largest graph evaluated in this paper.

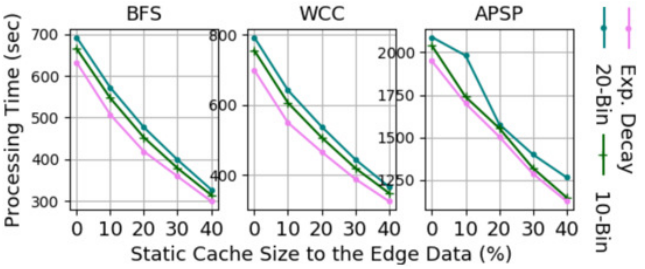


Figure 8: Gsh2015 with Different Binning Strategies.

Figure 8 depicts the processing times based on gsh2015, where x-axis denotes the static cache size to edge data and y-axis demonstrates the processing time. In general, since all binning strategies are able to capture the critical vertices, we can observe a similar improvement as the size of static cache increases. Thus, the performance under 0% static cache dominates the overall effectiveness. Particularly, Exp. Decay improves 10-Bin (20-Bin) by 5.7% (10.1%) on average. Furthermore, as shown in Section 4.4, Exp. Decay only slightly degrades by -1.06% compared with non-binning BFDS. As a result, binning strategy with exponential decay demonstrates a great way to not only preserve the structure of BFDS but also capture the critical vertices.

4.6 Evaluation on Non-BFS-like Algorithms

To discuss the impact of IOE-Order on non-BFS-like algorithms, this section further evaluates IOE-Order against other graph orderings with non-BFS-like algorithms. Due to page limitations, we only evaluate two popular non-BFS-like algorithms (i.e., PageRank and K-core), which represent two completely different access patterns, and only show the re-

sults of gsh2015 [10]. Specifically, PageRank [18, 39] ranks the importance of a webpage by repeatedly processing all vertices, whereas K-core [34, 43] detects the clustering structure of a graph by returning a maximal subgraph that consists of vertices of degree larger than K.

Table 5: Evaluation of Non-BFS-like Algorithms on Gsh2015.

Algo. \ Order	Random	In-Degree	Norder	IOE-Order
Pagerank	927.7	257.9	249.7	255.5
K-Core	795.7	534.2	371.1	314.1

Table 5 shows the total processing time (measured in seconds) based on gsh2015 when four different graph orderings are utilized with no static cache. Particularly, we can observe that IOE-Order also has the potential to improve (or at least not degrade) the performance of the two evaluated non-BFS-like algorithms. For PageRank, IOE-Order delivers similar processing time as in-degree ordering and Norder as all of them are designed to increase the locality of access for BFS-like algorithms instead of optimizing PageRank. As for K-core, IOE-Order outperforms the other three orderings (e.g., 15.4% better than Norder). This is because OutD-Binning of IOE-Order keeps the vertices of similar degrees together, resulting in good locality of access for K-core in selectively removing the vertices of less-than-K degrees.

5 Related Work

Fully External Graph Systems. Besides the semi-external systems presented in Section 2.1, many efforts have also been devoted to the development of fully external graph systems [1, 14, 17, 21, 27, 41, 49, 54]. In contrast to semi-external systems, fully external systems are very low-cost for storing *both* vertex and edge data in storage, and only require a small amount of memory for runtime graph processing. For example, GraphChi [27], which is the first fully external graph system, divides the whole graph into several partitions and loads them from storage into memory for further processing. Xstream [41] proposes to exploit edge-centric computation model to sequentially streams the edge data into memory. Based on edge-centric computation model, GridGraph [54] proposes 2D edge partitioning to further improve the performance by selectively accessing based on partition-level granularity.

Nevertheless, as these systems tend to issue large and sequential I/O to eliminate random access, they often suffer from the low utilization of loaded data when the graph algorithm only requires a small number of bytes. To tackle this issue, CLIP [1] proposes to increase the utilization of the loaded data by performing out-of-order execution to compute across future values. Particularly, CLIP re-computes the loaded partition to squeeze out all potential vertex updates [1]. LUMOS [49], on the other hand, further provides synchronous processing semantics for supporting synchronous graph algorithms based on the concept of future value computation. However, these works are proposed to increase the utilization of the loaded data for the systems leveraging sequential I/O. By contrast, IOE-Order is proposed to improve the I/O efficiency for semi-external systems that load only the necessary data on demand, and thus is orthogonal to the optimization of CLIP and LUMOS.

Graph Ordering. Ordering has been studied for a long time [3, 4, 9, 22] for in-memory graph systems [33, 37], which keep the whole graph in memory for processing, and is an important technique to improve the locality of access. For example, Pinar et al. leverage hypergraph to represent temporal or spatial localities so as to improve the performance of Sparse Matrixvector Multiplication [40]. Gorder optimizes the locality of vertex attribute updates to speed up CPU computation for general in-memory graph processing [50]. Nevertheless, they aims to optimize the computation order or vertex order, which are different from I/O optimization.

On the other hand, as discussed in Section 2.2, Norder [28] is proposed to optimize the processing performance of BFS-like graph algorithms on semi-external graph systems for reducing the I/O cost. IOE-Order has the same goal, but it further improves the I/O efficiency by exploiting the traversal pattern; moreover, OutD-Binning facilitates the pre-caching with lower pre-processing cost and higher flexibility.

6 Conclusion

This paper presents IOE-Order to practicably boost the processing performance of running BFS-like algorithms on semi-external graph systems by presenting a holistic graph ordering with two major pre-processing optimizations. Specifically, BFDS is proposed to leverage both graph traversal pattern and in-degree information to re-order the graph for higher I/O efficiency. Moreover, OutD-Binning is further introduced to refine and split the BFDS-ordered graph into multiple bins with different average out-degrees. Since all bins are further sorted and stored sequentially to form the edge data, OutD Binning not only enables high flexibility and efficiency in pre-caching the small edge lists but further increases the I/O efficiency in loading data from the rest of bins during graph processing. The evaluations show that, compared with the integrated solution of state-of-the-art pre-processing optimizations, IOE-Order delivers both high efficiency (by improving the processing time up to 36.1%) and high practicability (by entailing much lower pre-processing overhead) for various BFS-like algorithms.

Acknowledgments

We thank our shepherd, Nathan Beckmann, and all the anonymous reviewers for their valuable comments and suggestions. This work is supported in part by The Research Grants Council of Hong Kong SAR (Project No. CUHK14208521).

References

[1] Zhiyuan Ai, Mingxing Zhang, Yongwei Wu, Xuehai Qian, Kang Chen, and Weimin Zheng. Squeezing out all the value of loaded data: An out-of-core graph processing system with reduced disk i/o. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 125–137, Santa Clara, CA, July 2017. USENIX Association.

[2] Zhiyuan Ai, Mingxing Zhang, Yongwei Wu, Xuehai Qian, Kang Chen, and Weimin Zheng. Clip: A disk i/o focused parallel out-of-core graph processing system. *IEEE Transactions on Parallel and Distributed Systems*, 30(1):45–62, 2019.

[3] Junya Arai, Hiroaki Shiokawa, Takeshi Yamamuro, Makoto Onizuka, and Sotetsu Iwamura. Rabbit order: Just-in-time parallel reordering for fast graph analysis. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 22–31, 2016.

[4] Ümit V. Çatalyürek, Mehmet Deveci, Kamer Kaya, and Bora Uçar. Multithreaded clustering for multi-level hypergraph partitioning. In *2012 IEEE 26th International Parallel and Distributed Processing Symposium*, pages 848–859, 2012.

[5] Paolo Boldi, Marco Rosa, Massimo Santini, and Sebastiano Vigna. Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks. In Sadagopan Srinivasan, Krithi Ramamritham, Arun Kumar, M. P. Ravindra, Elisa Bertino, and Ravi Kumar, editors, *Proceedings of the 20th international conference on World Wide Web*, pages 587–596. ACM Press, 2011.

[6] Paolo Boldi and Sebastiano Vigna. The WebGraph framework I: Compression techniques. In *Proc. of the Thirteenth International World Wide Web Conference (WWW 2004)*, pages 595–601, Manhattan, USA, 2004. ACM Press.

[7] Ulrik Brandes. A faster algorithm for betweenness centrality. The Journal of Mathematical Sociology, 25(2):163–177, 2001.

[8] Wei Chen and Shang-Hua Teng. Interplay between social influence and network centrality: A comparative study on shapley centrality and single-node-influence centrality. In *Proceedings of the 26th International Conference on World Wide Web*, WWW '17, page 967–976, Republic and Canton of Geneva, CHE, 2017. International World Wide Web Conferences Steering Committee.

[9] E. Cuthill and J. McKee. Reducing the bandwidth of sparse symmetric matrices. In *Proceedings of the 1969 24th National Conference*, ACM '69, page 157–172, New York, NY, USA, 1969. Association for Computing Machinery.

[10] Gsh2015 dataset from WebGraph. http://law.di.unimi.it/webdata/gsh-2015/ , 2015.

[11] Twitter dataset from WebGraph. http://law.di.unimi.it/webdata/twitter-2010/ , 2010.

[12] Piotr Indyk Donald Aingworth, Chandra Chekuri and Rajeev Motwani. Fast estimation of diameter and shortest paths (without matrix multiplication). SIAM Journal on Computing, 28(4):1167–1181, 1999.

[13] Devdatt Dubhashi, Alessandro Mei, Alessandro Panconesi, Jaikumar Radhakrishnan, and Aravind Srinivasan. Fast distributed algorithms for (weakly) connected dominating sets and linear-size skeletons. *Journal of Computer and System Sciences*, 71, 03 2003.

[14] Nima Elyasi, Changho Choi, and Anand Sivasubramaniam. Large-scale graph processing on emerging storage devices. In *Proceedings of the 17th USENIX Conference on File and Storage Technologies*, FAST'19, page 309–316, USA, 2019. USENIX Association.

[15] Tiziano Fagni, Raffaele Perego, Fabrizio Silvestri, and Salvatore Orlando. Boosting the performance of web search engines: Caching and prefetching query results by exploiting historical usage data. *ACM Trans. Inf. Syst.*, 24(1):51–78, January 2006.

[16] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 17–30, Hollywood, CA, October 2012. USENIX Association.

[17] Wook-Shin Han, Sangyeon Lee, Kyungyeol Park, Jeong-Hoon Lee, Min-Soo Kim, Jinha Kim, and Hwanjo Yu. Turbograph: A fast parallel graph engine handling billion-scale graphs in a single pc. New York, NY, USA, 2013. Association for Computing Machinery.

[18] Taher H. Haveliwala. Topic-sensitive pagerank. In *Proceedings of the 11th International Conference on World Wide Web*, WWW '02, page 517–526, New York, NY, USA, 2002. Association for Computing Machinery.

[19] John Hopcroft and Robert Tarjan. Algorithm 447: Efficient algorithms for graph manipulation. Communications of the ACM, 16(6):372–378, 1973.

[20] H. Jeong, S. P. Mason, A.-L. Barabási, and Z. N. Oltvai. Lethality and centrality in protein networks. *Nature*, 411(6833):41–42, May 2001.

[21] Sang-Woo Jun, Andy Wright, Sizhuo Zhang, Shuotao Xu, and Arvind. Grafboost: Using accelerated flash storage for external graph analytics. In *Proceedings of the 45th Annual International Symposium on Computer Architecture*, ISCA '18, page 411–424. IEEE Press, 2018.

[22] Konstantinos I. Karantasis, Andrew Lenharth, Donald Nguyen, Marás J. Garzarán, and Keshav Pingali. Parallelization of reordering algorithms for bandwidth and wavefront reduction. In *SC '14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 921–932, 2014.

[23] Jooho Kim and Makarand Hastak. Social network analysis: Characteristics of online social networks after a disaster. *International Journal of Information Management*, 38(1):86–96, 2018.

[24] Alec Kirkley, Hugo Barbosa, Marc Barthelemy, and Gourab Ghoshal. From the betweenness centrality in street networks to structural invariants in random planar graphs. *Nature Communications*, 9(1):2501, Jun 2018.

[25] Pradeep Kumar and H. Howie Huang. G-store: High-performance graph store for trillion-edge processing. In *SC '16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 830–841, 2016.

[26] Chun-Yen Kuo, Ching Nam Hang, Pei-Duo Yu, and Chee Wei Tan. Parallel counting of triangles in large graphs: Pruning and hierarchical clustering algorithms. In *2018 IEEE High Performance extreme Computing Conference (HPEC)*, pages 1–6, 2018.

[27] Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. Graphchi: Large-scale graph computation on just a PC. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 31–46, Hollywood, CA, October 2012. USENIX Association.

[28] Eunjae Lee, Junghyun Kim, Keunhak Lim, Sam H. Noh, and Jiwon Seo. Pre-select static caching and neighborhood ordering for bfs-like algorithms on disk-based graph engines. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 459–474, Renton, WA, July 2019. USENIX Association.

[29] Hang Liu and H. Howie Huang. Graphene: Fine-grained IO management for graph computing. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*, pages 285–300, Santa Clara, CA, February 2017. USENIX Association.

[30] Zhen Liu, Philippe Nain, Nicolas Niclausse, and Don Towsley. Static caching of Web servers. In Kevin Jeffay, Dilip D. Kandlur, and Timothy Roscoe, editors, *Multimedia Computing and Networking 1998*, volume 3310, pages 179 – 190. International Society for Optics and Photonics, SPIE, 1997.

[31] Damien Magoni and Jean Jacques Pansiot. Analysis of the autonomous system network topology. *SIGCOMM Comput. Commun. Rev.*, 31(3):26–37, July 2001.

[32] Vladimir V. Makarov, Maxim O. Zhuravlev, Anastasija E. Runnova, Pavel Protasov, Vladimir A. Maksimenko, Nikita S. Frolov, Alexander N. Pisarchik, and Alexander E. Hramov. Betweenness centrality in multiplex brain network during mental task evaluation. *Phys. Rev. E*, 98:062413, Dec 2018.

[33] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: A system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, page 135–146, New York, NY, USA, 2010. Association for Computing Machinery.

[34] Alberto Montresor, Francesco De Pellegrini, and Daniele Miorandi. Distributed k-core decomposition. *CoRR*, abs/1103.5320, 2011.

[35] Edward F. Moore. The shortest path through a maze. In Proceedings of the International Symposium on the Switching Theory, 1959, pages 285–292.

[36] Aida Mrzic, Pieter Meysman, Wout Bittremieux, Pieter Moris, Boris Cule, Bart Goethals, and Kris Laukens. Grasping frequent subgraph mining for bioinformatics applications. *BioData Mining*, 11(1):20, Sep 2018.

[37] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. A lightweight infrastructure for graph analytics. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, page 456–471, New York, NY, USA, 2013. Association for Computing Machinery.

[38] Akinniyi Ojo, Ngok-Wa Ma, and Isaac Woungang. Modified floyd-warshall algorithm for equal cost multipath in software-defined data center. In *2015 IEEE International Conference on Communication Workshop (ICCW)*, pages 346–351, 2015.

[39] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking: Bringing order to the web. Technical Report 1999-66, Stanford InfoLab, November 1999. Previous number = SIDL-WP-1999-0120.

[40] Ali Pinar and Michael T. Heath. Improving performance of sparse matrix-vector multiplication. In *Proceedings of the 1999 ACM/IEEE Conference on Supercomputing*, SC '99, page 30–es, New York, NY, USA, 1999. Association for Computing Machinery.

[41] Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. X-stream: Edge-centric graph processing using streaming partitions. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, page 472–488, New York, NY, USA, 2013. Association for Computing Machinery.

[42] Siddhartha Sahu, Amine Mhedhbi, Semih Salihoglu, Jimmy Lin, and M. Tamer Özsu. The ubiquity of large graphs and surprising challenges of graph processing. *Proc. VLDB Endow.*, 11(4):420–431, December 2017.

[43] Ahmet Erdem Saríyüce, Buğra Gedik, Gabriela Jacques-Silva, Kun-Lung Wu, and Ümit V. Çatalyürek. Streaming algorithms for k-core decomposition. *Proc. VLDB Endow.*, 6(6):433–444, apr 2013.

[44] Zhiyuan Shao, Jian He, Huiming Lv, and Hai Jin. Fog: A fast out-of-core graph processing framework. *International Journal of Parallel Programming*, 45(6):1259–1272, Dec 2017.

[45] Alfonso Shimbel. Structural parameters of communication networks. The bulletin of mathematical biophysics, 15(4):501–507, 1953.

[46] Julian Shun and Guy E. Blelloch. Ligra: A lightweight graph processing framework for shared memory. *SIGPLAN Not.*, 48(8):135–146, feb 2013.

[47] I. Tatarinov, A. Rousskov, and V. Soloviev. Static caching in web servers. In *Proceedings of Sixth International Conference on Computer Communications and Networks*, pages 410–417, 1997.

[48] uk-2007 dataset from WebGraph. http://law.di.unimi.it/webdata/uk-2007-01/ , 2007.

[49] Keval Vora. LUMOS: Dependency-driven disk-based graph processing. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 429–442, Renton, WA, July 2019. USENIX Association.

[50] Hao Wei, Jeffrey Xu Yu, Can Lu, and Xuemin Lin. Speedup graph processing by graph ordering. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, page 1813–1828, New York, NY, USA, 2016. Association for Computing Machinery.

[51] Junlong Zhang and Yu Luo. Degree centrality, betweenness centrality, and closeness centrality in social network. In *Proceedings of the 2017 2nd International Conference on Modelling, Simulation and Applied Mathematics (MSAM2017)*, pages 300–303. Atlantis Press, 2017/03.

[52] Yu Zhang, Xiaofei Liao, Hai Jin, Lin Gu, Ligang He, Bingsheng He, and Haikun Liu. Cgraph: A correlations-aware approach for efficient concurrent iterative graph processing. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 441–452, Boston, MA, July 2018. USENIX Association.

[53] Da Zheng, Disa Mhembere, Randal Burns, Joshua Vogelstein, Carey E. Priebe, and Alexander S. Szalay. Flashgraph: Processing billion-node graphs on an array of commodity ssds. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, pages 45–58, Santa Clara, CA, February 2015. USENIX Association.

[54] Xiaowei Zhu, Wentao Han, and Wenguang Chen. Gridgraph: Large-scale graph processing on a single machine using 2-level hierarchical partitioning. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pages 375–386, Santa Clara, CA, July 2015. USENIX Association.