Seraph: Towards Scalable and Efficient Fully-external Graph Computation via On-demand Processing

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

https://www.usenix.org/conference/fast24/presentation/yang-tsun-yu

This paper is included in the Proceedings of the 22nd USENIX Conference on File and Storage Technologies.

February 27–29, 2024 • Santa Clara, CA, USA

978-1-939133-38-0

Open access to the Proceedings of the 22nd USENIX Conference on File and Storage Technologies is sponsored by

NetApp®

Seraph: Towards Scalable and Efficient Fully-external Graph Computation via On-demand Processing

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

Abstract

Fully-external graph computation systems exhibit optimal scalability by computing the ever-growing, large-scale graph with constant amount of memory on a single machine. In particular, they keep the entire massive graph data in storage and iteratively load parts of them into memory for computation. Nevertheless, despite the merit of optimal scalability, their unreasonably-low efficiency often makes them uncompetitive, and even unpractical, to the other types of graph computation systems. The key rationale is that most existing fully-external graph computation systems over-emphasize retrieving graph data from storage through sequential access. Although this principle achieves high storage bandwidth, it often causes reading excessive and irrelevant data, which can severely degrade their overall efficiency.

Therefore, this work presents Seraph, a fully-external graph computation system that achieves optimal Scalability while toward satisfactory Efficiency improvement. Particularly, inspired by the modern storage offering comparable sequential and random access speeds, Seraph adopts the principle of *on-demand processing* to access the necessary graph data for saving I/O while enjoying the decent speed in random access. On the basis of this principle, Seraph further devises three practical designs to bring excellent performance leap to fully-external graph computation: 1) the hybrid format to represent the graph data for striking a good balance between I/O amount and access locality, 2) the vertex passing to enable efficient vertex updates on top of hybrid format, and 3) the selective pre-computation to re-use the loaded data for I/O reduction. Our evaluations reveal that Seraph notably outperforms other state-of-the-art fully-external systems under all the evaluated billion-scale graphs and representative graph algorithms by up to two orders of magnitude.

1 Introduction

Graphs have been broadly used in many fields, such as networking [10], social media [6, 19, 45], and bioinformatics [16, 26], for their attractive structure to represent the entities as *vertices* and relations between entities as *edges*. In practice, a graph is represented by two data structures: *vertex data* (denoted as V) holding the attributes of vertices, and *edge data* (denoted as E) comprising the edge lists, each of which enumerates the destination vertices connected with the same source vertex. Typically, *graph computation* involves reading the edge data for neighboring vertices and updating the vertices' attributes from/to their neighbors' attributes.

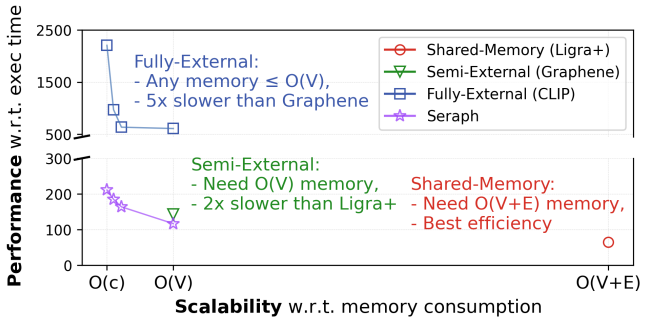


Figure 1: The performance-scalability spectrum among different state-of-the-art graph computation systems. The presented times are the average results of the evaluated graph systems with different amounts of memory described in §4.3.

Many single-machine *graph computation systems* have been developed to automate and optimize the process of graph computation, with the aim of high performance (i.e., low execution time). Recently, as graphs exponentially grow to have billions of vertices and edges, *scalability* is also essential for such systems. In this context, scalability refers to the capacity of a system to compute ever-growing, large-scale graphs within a single machine of common memory capacity. Therefore, how to design a scalable graph computation system that is also performant is the primary objective in this field.

In the following, we examine different kinds of single-machine graph systems from the aspects of scalability and performance. First, shared-memory graph systems (e.g., Ligra [36], Galois [32], Ligra+ [37]) require the entire graph data to be in memory (i.e., $O(V+E)$) for computing graphs with high performance. However, when targeting large-scale graphs, this approach is high-cost and difficult to scale as it

necessitates the machine with huge memory capacity.

To alleviate the issue of memory requirement, external-based graph computation systems, which can be further divided into *semi-external* and *fully-external* systems, are proposed to exploit the storage drives for graph computation at the cost of performance sacrifice. Semi-external graph computation systems (e.g., FlashGraph [47], Graphene [25]) are proposed to trade performance for alleviating the memory overhead by keeping the edge data in the massive-and-cheap storage while maintaining the vertex data in the small-and-expensive memory (i.e., $O(V)$). However, these systems only have limited scalability as their memory requirements still increase proportionally with graph sizes. Specifically, they cannot handle large-scale graphs with vertex data that exceeds the machine's memory capacity. Further, as graphs continue to grow to have billions of vertices, even keeping the vertex data in memory is not cost-effective [11, 17].

On the other hand, fully-external graph computation systems (e.g., GridGraph [50], CLIP [1], Lumos [42], V-Part [11]) further trade performance to offer the merit of optimal scalability; they can compute the large-scale graphs with a small amount of memory, which is independent from the graph data sizes. To accomplish this, they divide the large-scale graph into multiple subgraphs and keep them in storage; during runtime, each subgraph is iteratively handled so that the memory requirement can be effectively confined to computing only one subgraph. In other words, given c to be the available memory capacity of a machine, a fully-external graph computation system can use c to compute any size of large-scale graph by controlling the number of created subgraphs (i.e., $O(c)$). Thus, besides the edge I/O, fully-external systems require some additional I/Os to read/write the small-sized vertex data to establish each subgraph in memory by turns.

Fig. 1 summarizes the trade-off between scalability (i.e., memory consumption) and performance (i.e., execution time) among different kinds of state-of-the-art graph computation systems. Specifically, shared-memory-based Ligra+ demands a massive $O(V+E)$ memory for the highest performance. Semi-external-based Graphene requires $O(V)$ memory; compared to Ligra+, it needs 7.2x less memory yet exhibits 2x performance degradation. Fully-external-based CLIP demonstrates optimal scalability to compute the graph with any amount of memory that is smaller than or equal to $O(V)$; thus, it is able to use significantly less memory than the other types of systems. The minimum memory in Fig. 1 (i.e. $O(c)$) that CLIP uses is 17x less than Graphene and 127x less than Ligra+. However, we also observe a vast performance degradation: CLIP is significantly slower than Graphene regardless of the memory consumption; even given $O(V)$ memory, CLIP is around 5x slower than Graphene. In conclusion, although fully-external schemes offer the merit of optimal scalability, their severely-degraded performances often make them uncompetitive with other graph computation systems.

To fill this void, this work presents *Seraph*, a fully-external graph computation system that substantially boosts efficiency while offering optimal scalability.

To build such a system, we first recognize that most existing fully-external systems over-emphasize retrieving graph data from storage via sequential access. While this principle achieves high storage bandwidth, it also causes reading excessive-and-irrelevant data, particularly as many graph algorithms often exhibit sparse access patterns [15, 24, 29, 30]. Moreover, given that modern storage (e.g., solid-state drive (SSD)) offers comparable speeds for sequential and random access [38, 40], we seek the different principle of *on-demand processing* to access the necessary data with fine-grained I/O to save transferred data while exploiting the decent speed in random access. To investigate whether on-demand processing is promising for fully-external framework, we realize a baseline system to support on-demand processing and compare it against the state-of-the-art fully-external systems. Our evaluations, based on four types of storage devices, demonstrate the strong motivation that developing fully-external system with on-demand processing is a promising direction (see §2.3).

In light of this observation, we build Seraph upon *on-demand processing*. Moreover, we propose *three practical designs specially tailored for the framework of on-demand processing* to achieve further performance improvement. First, we observe that the traditional method for representing edge data has its pros and cons: it creates a good locality for accessing vertices yet increases the overhead of locating and reading edges. To this end, we present a new format, called *hybrid format*, to store the graph data by striking a good balance between locality for vertices and overhead for edges (see §3.2). Second, based on the hybrid format, we further propose *vertex passing* to enable efficient vertex updates by delaying and aggregating the vertex updates to the same subgraph via in-memory buffers (see §3.3). Third, although on-demand processing reads the necessary data, a common I/O block is typically way larger than an edge list. This mismatch inspires us with the opportunity of I/O re-using and the proposing of *selective pre-computation* to asynchronously compute the current and future vertices on the fly (see §3.4).

We implement Seraph in C++ and compare it against several state-of-the-art fully-external graph systems. Our evaluations, based on billion-scale graphs and representative graph algorithms, reveal that Seraph significantly outperforms the existing systems by up to two orders of magnitude. Further, with an increasing memory amount, Seraph also performs well and exhibits an up to 1.6x improvement over a recent semi-external graph system. Besides, we conduct investigation to justify each proposed design's effectiveness.

2 Background and Motivation

2.1 Background of Fully-external Systems

For large-scale graph computation on a single machine of limited memory capacity, fully-external graph computation systems necessarily divide the graph into multiple subgraphs

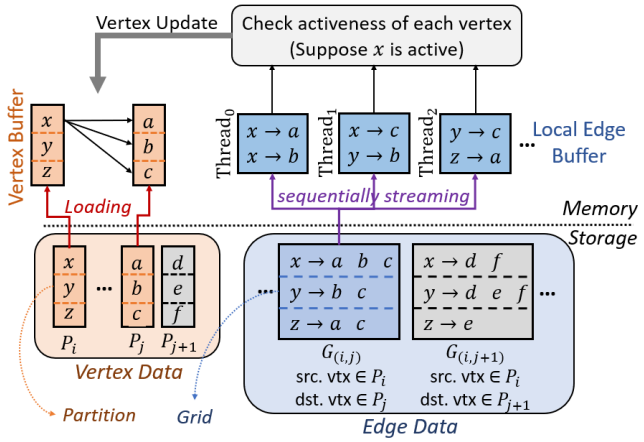


Figure 2: Typical system architecture of fully-external graph computation systems.

and handle them one at a time. In practice, as illustrated in Fig. 2, the vertices are generally divided into multiple disjoint *partitions*, and when processing the edges between two partitions (i.e., a subgraph), the memory requirement can be thus effectively limited to two vertex partitions involved (e.g., P_i and P_j) instead of the entire vertex data.

Based on the fundamental design described above, two common techniques are further adopted by the existing fully-external graph computation systems (such as [1,20,34,42,50]) for performance enhancement:

Grid Format for Edge Data. According to the results of vertex partitioning, existing fully-external graph computation systems typically store the edge data into *grid format*. In particular, a grid determines a set of edges by specifying one partition as source vertices and another (can be also the same one) as destination vertices. As shown in Fig. 2, the grid $G_{(i,j)}$ stores the set of edges whose source and destination vertices belong to partitions P_i and P_j, respectively. In other words, under the grid format, the edges of a source vertex can be split into multiple *segments* and are stored within different grids based on their destination vertices. As illustrated by Fig. 2, the edge list of the vertex x are split into two segments, which are (a, b, c) and (d, f), and are stored with grids $G_{(i,j)}$ and $G_{(i,j+1)}$, respectively.

Therefore, compared with storing the whole edge list of a vertex consecutively together (i.e., *row format* [28,31]), processing the edge lists in grid format enjoys good locality of vertex access. This is because the required vertex attributes can be limited to only two partitions of vertices, which can entirely fit in the limited memory of fully-external graph computation systems. For instance, GridGraph [50] proposes 2D edge partitioning to split the edge data into smaller grids where the edges in the same grid share similar locality in accessing source and destination vertices.

Streaming-based Processing. On top of the grid format, existing fully-external graph systems generally apply *streaming-based processing*: they compute a graph by streaming the edge data from storage grid-by-grid. Given that all storage drives typically deliver high bandwidth with sequential access, streaming-based processing shows the generality of accommodating all types of storage.

As illustrated in Fig. 2, to compute a grid (e.g., $G_{(i,j)}$), the system first loads the two corresponding vertex partitions (e.g., P_i and P_j) into *vertex buffers* in memory. Next, as graph systems typically run in multi-threads for better performance, every thread sequentially streams edges from different parts of grid into its *local edge buffer* in memory. Following, the system identifies the vertices that need to be computed (i.e., *active vertices*), and then produces updates to their neighboring vertices based on the attributes of active vertices. For instance, GridGraph [50] checks the activeness of every grid by turns and sequentially streams the entire active grid from storage, in the granularity of 24 MB, to perform vertex update.

2.2 Existing Fully-external Systems

Many fully-external graph computation systems have been proposed for large-scale graph computation on a single machine. For example, GraphChi [20], which is the first fully-external graph computation system, divides a graph into multiple disjoint shards and sequentially load each shard into memory for computation. X-stream [34] proposes edge-centric processing model to stream and compute every edge for achieving high speed of sequential access. Inspired by X-stream, GridGraph [50] demonstrates a representative framework by splitting a graph with a smaller granularity (called a grid) to improve data locality; it achieves significant performance enhancement over GraphChi and X-stream.

Later, many systems propose optimization based on the framework of GridGraph. For instance, CLIP [1] introduces state-of-the-art optimization for streaming-based processing by asynchronously re-computing the loaded grid multiple times to increase data utilization and accelerate the convergence of graph algorithms. This feature makes CLIP specialized for *asynchronous graph algorithms* (e.g., vertex values following monotonicity [43]). Similar to CLIP, Lumos [42] also attempts to re-compute the loaded grid, but it aims to optimize *synchronous graph algorithms* which require synchronous semantics: a vertex can only observe the values from the last iteration. Thus, Lumos performs future computation on a vertex only if it receives all the updates from its neighbors. Due to this strict requirement, Lumos is more suitable for optimizing the algorithms which naturally demand synchronous semantics. Therefore, although both CLIP and Lumos improve over GridGraph with future computation, they are specialized for different sets of algorithms, respectively. On the other hand, Wonderland [46] applies the graph abstraction technique from visualization systems to streaming-based processing. However, the abstraction-guided processing only works for accelerating the convergence of path-based algorithms such as shortest path.

V-Part [11], a recent fully-external system, proposes a novel

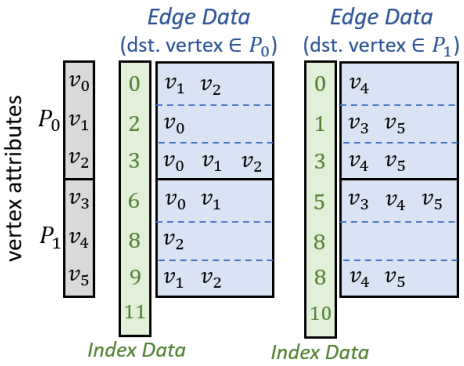


Figure 3: Data structure of GridGraph-ODP.

framework different from GridGraph. V-Part stores the destination and source vertices into different structures. Each destination contains only attributes, while each source is associated with vertex ID, attribute, and offset pointing to its edge list, forming the source vertex table. To process a partition, V-Part first loads the destination vertices and then streams the source vertex table of the same partition into memory. Next, V-Part on-demand reads the active edge lists based on the offsets and updates the destination vertices. Finally, V-Part requires a stage, called mirror update, to synchronize the values/attributes between source and destination to facilitate vertex updates.

Nevertheless, V-Part fails to utilize on-demand processing thoroughly. Although V-Part on-demand accesses the edge data, it still streams the source vertex table from storage in a partition-based granularity, severely impacting its performance. Moreover, V-Part requires an extra overhead of mirror update, making its performance often worse than the other system adopting streaming-based processing (which will be shown in §4.1). Thus, to study which processing principle is more suitable for fully-external framework, we realize our own baseline system with on-demand processing and compare it against the state-of-the-art system with streaming-based processing in the next section.

2.3 Motivation: Streaming-based Processing versus On-demand Processing

Although streaming-based processing often loads excessive-and-irrelevant data as it streams an entire grid even if there are only a few active vertices/edges, it retains the advantage of achieving high storage bandwidth. Thus, many prior work have attempted to optimize streaming-based processing from various perspectives based on fully-external framework [1, 42, 44]. However, since modern storage provides comparable sequential and random access speeds, we attempt to look for the other principle, on-demand processing, to save I/O while exploiting the decent speed of random access. To study which principle is more suitable for fully-external framework, this section aims to compare the baseline of on-demand processing against the optimized streaming-based processing.

To realize a baseline system with on-demand processing

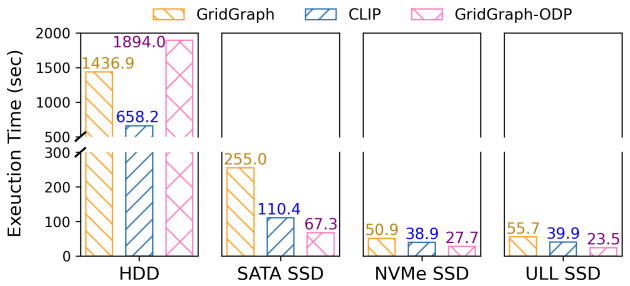


Figure 4: Evaluation of BFS on Twitter graph [9] with different storage devices.

under fully-external framework, we follow the traditional grid format to store edge data but replace streaming-based processing with the on-demand one. To this end, we revamp a representative, fully-external system GridGraph [50] to support on-demand processing (called GridGraph-ODP). Please note that we did not make CLIP support on-demand processing because the CLIP's re-computation only provides a coarse-grained, grid-based control, which contradicts the core concept of on-demand processing.

Compare to GridGraph, we add *index data* into GridGraph-ODP to record the offset of each segmented edge list, as shown in Fig. 3. These data enable on-demand processing for edge data because GridGraph-ODP can easily locate and read the required edge lists into memory. For example, suppose v is vertex and i is index, the edge list of v_s is located between i_s and i_{s+1}. Besides, since the vertex attributes are all sequentially stored based on their vertex IDs[1] in the file, it is easy to locate and access an attribute via its ID. Therefore, GridGraph-ODP can on-demand access both vertex and edge data.

We compare GridGraph-ODP against GridGraph [50] and CLIP [1]. In particular, GridGraph represents the baseline of streaming-based processing, and CLIP [1] stands for the state-of-the-art optimization for streaming-based processing. We use the famous breath-first search (BFS) [30] as a case study because it incurs both dense and sparse access patterns during computation [2]. We evaluate BFS on the three systems with the traditional hard-disk drive (HDD) as well as three different types of modern solid-state drives (SSD): SATA SSD [39], NVMe SSD [40], and ULL SSD [38]. The experiments are conducted in the same environment as that described in §4, and the number of threads is set to four, which is the same configuration as GridGraph [50].

Fig. 4 depicts the results. First, we can observe that HDD negatively impacts GridGraph-ODP. Compared with GridGraph, although GridGraph-ODP can save around 8.7x in loading edge data, its performance degrades by -24.1%. This result implies that the random I/O severely degrades the speed of HDD, and streaming-based processing is an effective principle to obtain high storage bandwidth. Moreover, CLIP improves GridGraph and GridGraph-ODP by 53.7% and 65.2%. The reason is that CLIP leverages streaming-based processing

[1] Each vertex is assigned with a distinct value to be its identity.

and further accelerates the convergence of graph algorithms by re-computing the loaded chunks multiple times. Thus, CLIP vastly outperforms the other two systems on HDD.

The results on modern SSDs show a different trend. Although both GridGraph and CLIP perform better on a faster drive, GridGraph-ODP improves more outstandingly because on-demand processing can save I/Os while enjoying the decent speed in random access. Moreover, we can observe an inspiring fact that, even if CLIP applies state-of-the-art optimization to streaming-based processing, GridGraph-ODP, which represents the baseline of on-demand processing, works remarkably better. Specifically, GridGraph-ODP outperforms GridGraph and CLIP by 59.0% and 36.3% on average. As a result, given that SSDs have prevailed nowadays, these experiments strongly motivate us that *advancing the development of fully-external system tailored for on-demand processing* is a promising direction.

3 Seraph

3.1 Overview

Motivated by §2.3, we realize that on-demand processing is a promising direction to build fully-external graph computation system. This inspires this work to develop a new fully-external graph computation system, Seraph, based on the principle of on-demand processing. Moreover, Seraph incorporates three main designs that are specially tailored for the framework of on-demand processing to pursue further performance improvement.

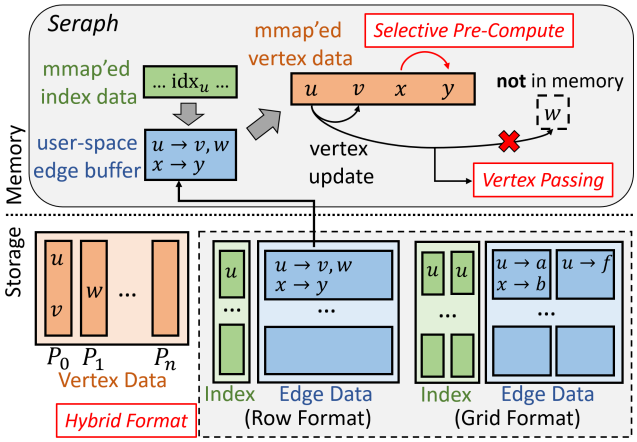


Figure 5: Architecture of Seraph.

Fig. 5 depicts Seraph's architecture, which combines the frameworks of fully-external graph computation and on-demand processing. To support fully-external framework, Seraph follows several traditional fundamentals such as keeping the entire graph data in storage and dividing the vertices into disjoint partitions. To reduce memory consumption, Seraph mmaps each vertex partition into memory and compute one partition at a time to create locality. To support on-demand processing framework, Seraph maintains index data to record the location of each edge list, just like our revamped GridGraph-ODP. Moreover, §3.2 presents a new *hybrid format* to split the edge data into both row and grid formats (e.g., for edge list of u, row format stores v and w, and grid format keeps a and f). It reduces I/O during graph computation by combining the advantages of both formats.

With these data structures, Seraph's execution flow is briefly illustrated as follows. It runs graph algorithms in iterations, and in each iteration, it first handles row format and then grid format. No matter which format is handled, Seraph computes one partition at a time. Specifically, graph computation involves identifying the active vertices, reading their edge lists, and updating the corresponding vertex attributes. However, when computing row format, the to-be-updated vertex attribute could not be inside memory (details in §3.3.1). To resolve this issue, §3.3 proposes *vertex passing* to delay the vertex updating for creating locality. On the other hand, §3.4 presents *selective pre-computation*. It explores the feature of asynchronous processing for further I/O reduction by re-using loaded data to compute the active vertices of future iterations in advance. Please note that the detailed execution flow involving the proposed designs will be elaborated in §3.5.

3.2 Hybrid Format

3.2.1 Observation

The motivation of hybrid format comes from the inefficiency of applying on-demand processing to the traditional grid format. Although it is common to utilize streaming-based processing with grid format, using on-demand processing with grid format is a double-edged sword. On one hand, grid format creates good locality of vertex access by confining the access range to two partitions only. On the other hand, it increases the overhead in reading edge lists and index data.

We first discuss how grid format negatively impacts the performance in reading edge lists. Compared with the edge list stored in row format, the grid format breaks an edge list into multiple segments and stores them in different grids, making on-demand processing issue a greater number of I/O blocks to read all the edge lists of the same source vertex in different grids. Under the example of Twitter graph[2] [9], the average number of 4KB pages to read the entire edge list (i.e., all neighbors) of a vertex is 3.88 in grid format, whereas row format only requires 1.03 pages, demonstrating a signifiant 73.5% improvement.

Next, reading the index data in grid format is inefficient due to the high ratio of redundant indexes. We call an index redundant if it points to an empty edge list, as such an index provides no information about the graph. Specifically, because the edge distribution of real-world graphs is typically skewed [12, 13], there is a high likelihood that a vertex has no edge list in a grid, leading to a high ratio of redundant indexes. In the same example of Twitter graph, over half (53.6%) of

[2]We assume the graph is divided into eight partitions.

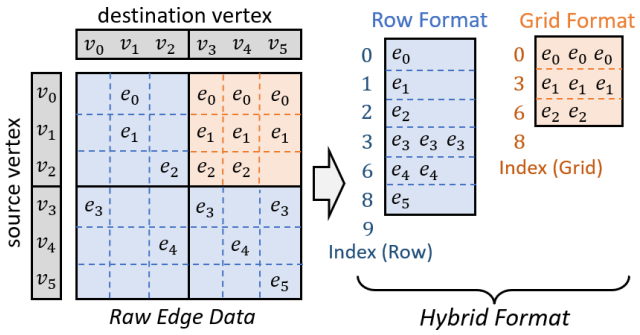


Figure 6: Example of hybrid format.

the indexes in grid format are redundant and provide no graph information. Further, as each grid maintains its own index data to enable on-demand processing, the total size of index data in grid format is considerably larger than that in row format. Thus, reading the index data in grid format becomes a non-negligible overhead to the overall performance.

Although row format seems better than grid format based on the above discussion, using row format in a fully-external system causes the issue of *accessing vertex attributes randomly*; the system may update arbitrary vertices in row format, rather than just a partition of vertices as in grid format. This leads to a serious issue that many vertex attributes that need to be updated may be on storage (i.e., not in memory). To update those attributes on storage, the system will extensively swap the pages between memory and storage, severely degrading the performance. Thus, §3.2.2 introduces *hybrid format* to resolve the dilemma between row and grid format.

3.2.2 Hybrid Format Construction

This section presents a new format, called *hybrid format*, to store the edge data while improving the performance of graph computation. The goal of hybrid format is to exploit the advantages of both row and grid formats. As shown in Fig. 6, it first stores the graph into grid format. Following, several grids are converted to row format to prevent the inefficiency in reading edge and index data, while retaining the rest in grid format to preserve good locality of vertex access.

To store the raw edge data into hybrid format, we first divide the edges into multiple chunks where every edge in the same chunk shares the same source and destination vertex partition(s). If a chunk contains many edges, computing the edges in the chunk will easily generate many vertex updates; storing the chunk in grid format is beneficial because we can create a good locality of vertex access during graph computation. On the contrary, for the chunks containing few edges, it is preferable to store them in row format for two reasons: (i) Because the edge lists in these chunks are small in general, storing them in row format can append all of the small edge lists together for reading them with the I/O block efficiently. (ii) Since the chunks of few edges tend to contain many empty edge lists, storing those chunks in row format prevents high ratio of redundant index data.

Hybrid format uses 8 bytes to store each index that points to the beginning location of each edge list; the edge list of a vertex v_s is recorded by s^{th} index and $(s+1)^{th}$ index. As shown in Fig. 6, row format contains $V+1$ indexes. Grid format requires $V/P+1$ indexes for each grid, so the total number of indexes in grid format is $(V/P+1)\times G$, where P is the number of partitions and G is the number of grids created (e.g., example in Fig. 6 is one grid in total).

To construct a graph into hybrid format, Seraph first sequentially reads the raw edge data[3], while deciding each edge chunk to be stored in grid or row format (details are discussed in §3.3.2). Next, based on the decision, it reads and re-distributes the raw edge data into grid or row format while creating the index data. Thus, the I/O complexity of constructing hybrid format is $O(5E+(V/P+1)\times G+V)$, whereas GridGraph takes $O(4E)$ to create grid format as it reads the raw edges and re-distributes them into different grids. Table 1 shows the construction times of GridGraph and Seraph (settings are described in §4). We can observe that the construction times of Seraph are roughly 30% slower than those of GridGraph. However, since graph construction is a one-time procedure per graph and can be performed offline, these costs are lightweight for both systems in terms of runtime performance, as shown in §4.1.

Table 1: Construction times of Seraph and GridGraph.

Time(sec)	Twitter	Gsh2015	Eu2015	RMAT
GridGraph	15.6	382.9	979.9	2285.5
Seraph	20.6	475.5	1294.5	2906.2

However, although the chunks that strongly require good locality of vertex access will be stored in grid format based on the above construction method, computing the edge lists in row format still results in several random accesses to vertex attributes. Therefore, §3.3 proposes a simple yet effective technique, called *vertex passing*, to tackle this issue. Please note that the further details such as the criterion for storing a chunk in the row or grid format will be discussed after introducing the concept of vertex passing.

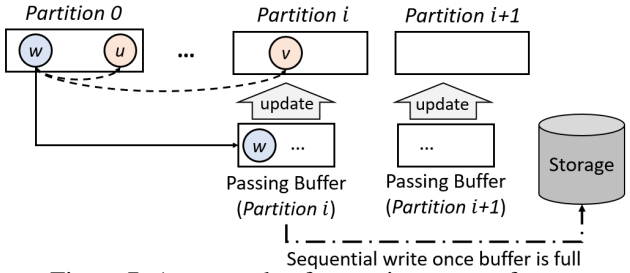


Figure 7: An example of execution on row format.

3.3 Vertex Passing

3.3.1 Design Concept

Vertex passing is proposed to resolve the issue of accessing vertex attributes randomly, as mentioned in §3.2. That is, it is

[3]Raw edge data and GridGraph present an edge with (`src`, `dst`). Hybrid format presents an edge with `dst` with the help of index data.

used when the to-be-updated vertex attributes are on storage. To enable vertex passing, each vertex partition is associated with an in-memory *passing buffer*. Suppose a vertex u is updated by its source vertex yet u is on storage, vertex passing will delay the update and transfer the information of the source vertex (i.e., active vertex) to the passing buffer of u's partition. Afterwards, once u is loaded into memory, vertex passing will merge the update in the passing buffer back to the attribute. Because a vertex partition shall be held in memory when Seraph currently computes it, every vertex can be held in memory by turns for update merging.

Fig. 7 shows an example about how vertex passing works. Suppose Seraph is computing P_0 (so the vertex attributes of P_0 shall be in memory), w is an active vertex, u and v are the neighbors of w. At this moment, the system can directly update vertex u but not vertex v since u is in memory and v is on storage; compelling the direct update to on-storage vertices will severely degrade the system's performance, as it involves extensive page swapping between memory and storage. Hence, vertex passing addresses this problem by delaying the on-storage update and transferring the update information of w to the passing buffer of P_i. Later, when Seraph is computing P_i, the vertex v can be loaded into memory and thus updated by the information stored in the P_i's passing buffer. In general, Seraph transfers the information of neighbor vertex ID and update value to the passing buffer, but the transferred information is configurable by the user based on the requirement of graph algorithms.

To mitigate the impact on memory usage in Seraph, each passing buffer is set to be negligibly small (e.g., 1 MB) compared to the vertex data size. Thus, each passing buffer is associated with a logging file to keep the vertex updates once the total size exceeds the in-memory passing buffer size. Seraph issues sequential I/O to write the information from the passing buffer to the corresponding file once the buffer is full. Similarly, sequential read is used for merging the information in the file back to vertex attributes. Please note that we use logging files to manage vertex updates instead of KV store because manipulating vertex updates is relatively simple; each update is only valid for one iteration, and they are bulky deleted after being used. Thus, we choose the straightforward implementation of logging files.

Although vertex passing requires I/O to transfer the updates, the overhead is generally small for three reasons. First, since vertex passing only transfers the update information of active vertices, it is effective for many graph algorithms activating a few vertices in most of the iterations. Second, as discussed in §3.2.2, a chunk stored in row format contains few edges; thus, even if all vertices are active and produce updates to their neighbors, the maximum overhead is bounded by the few edges inside each chunk. Third, because all transferred information is consecutively kept together in passing buffers and files, accessing them is efficient by using sequential I/O. Thus, vertex passing can efficiently tackle the issue of random vertex update caused by computation on row format.

The core concept of vertex passing (VP) is to delay operations as logs, which is a useful technique to create locality for different scenarios [21, 22]. For example, [22] exploits VP to optimize PageRank under shared-memory premise. It buffers all logs in DRAM efficiently and aims to improve VP's efficiency with more designs (e.g., lock-free layout). In contrast, our VP is naturally slower since the logs are recorded on storage via I/O. Thus, over-using our VP in fully-external environment will waste much I/O and eventually hurt Seraph's performance. In this regard, hybrid format and VP are proposed as a combination which complements each other with the aim of minimizing storage I/O.

3.3.2 Details in Hybrid Format Construction with Consideration of Vertex Passing

Seraph decides whether a chunk should be stored in row or grid formats by comparing their respective overheads. In particular, since this work targets fully-external environments (storage I/O is typically slower than CPU computation), each chunk is decided between two formats by considering their upper-bounded "I/O overheads" (more specifically, "I/O amounts"). That is, for a chunk C in grid format, the upper-bounded I/O amount includes reading all grid-related data (e.g., the chunk's vertex attributes, index data, and edge lists). For C in row format, the upper-bounded I/O amount is to log the updates generated by all edges in C. Therefore, the time complexity of gathering the above-mentioned information is linear to chunk size, while making decision is in constant time by simply comparing the two I/O-amounts.

On the other hand, a fully-external graph system typically reserves enough memory for holding two vertex partitions, one for the source partition and the other for the destination partition. In other words, when Seraph is computing a partition P in row format, it loads P into memory as source partition; there is an empty space for another partition to be destination partition. Thus, for every row of chunks (i.e., the chunks share the same source vertex partition), we can store one chunk in row format regardless of its number of edges because we can hold the chunk's destination partition in memory to absorb the vertex updates during graph computation. In fact, due to the natural locality of real-world graphs [49], many edges reside on the diagonal chunks (share the same source and destination partition). Thus, Seraph stores the diagonal chunks in row format by default.

Because different graph algorithms incur various access patterns, it is hard to tailor hybrid format for a specific access pattern. Thus, we construct hybrid format by heuristically assuming that all vertices are active for the following two reasons: 1) because the major bottleneck in graph computation is the dense access pattern under on-demand processing; optimizing dense access pattern is more beneficial than the sparse pattern in general, and 2) since the real-world graphs are typically skewed [12, 13], many chunks will be stored in

row format to enjoy the benefits even under the assumption of all active vertices. For example, we divides Twitter graph [9] into 64 chunks, and only 14 chunks are stored in grid format. For Eu2015 graph [7], all chunks are stored in row format as the vast majority of edges reside on the diagonal chunks.

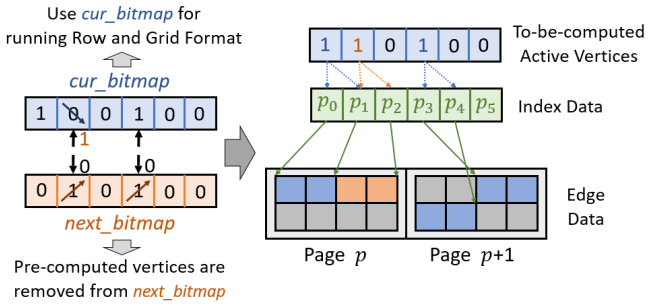


Figure 8: Example of pre-computation.

3.4 Selective Pre-Computation

Asynchronously computing the data of future iteration in advance is a well-known technique [1, 42]. As streaming-based processing often loads excessive-but-irrelevant data, existing work [1] exploits this concept to increase the utilization of the loaded grid by re-computing it many times. On the other hand, although Seraph leverages on-demand processing, there is still a granularity mismatch between a common I/O block size (e.g., 4096 bytes) and the typical size of an edge list (e.g., 132 bytes on average in Twitter). This mismatch provides the opportunity that the current and future active vertices[4] could reside on the same I/O block. Thus, *selective pre-computation* is introduced into Seraph to opportunistically re-use the loaded data for current active vertices to pre-compute the future ones asynchronously, reducing the total number of issued I/O.

However, under the framework of on-demand processing, recognizing whether the loaded data contain the information of future active vertices requires high implementation cost by traversing three different data structures (i.e., vertex attributes, indexes, and edges). To prevent this cost, we use vertex IDs for estimation due to the following three reasons. First, since all data structures are stored sequentially based on vertex ID, there is a high likelihood that two vertices reside on the same I/O blocks if their ID gap is small. Second, the implementation cost of tracking vertex IDs is low. Third, even for the worst case that we might spend a few extra I/O to load future active vertex, pre-computation merely beforehand performs computation that was supposed to happen in the next iteration, making it only require little cost yet offer the opportunity to re-use I/O. Based on our investigation, it is challenging to set an optimal value of ID gap for all scenarios because different algorithms/graphs have different features. Thus, we set the ID gap to be 32 by default as it is a reasonable value (by considering 4KB page and the average edge list size) and generally performs well for all graphs. In other words, Seraph will selectively pre-compute those future active vertices which are within the same gap based on the current active vertices to opportunistically re-use I/O.

[4]Current active vertices means the active vertices of current iteration. Future active vertices means the active vertices of next iteration.

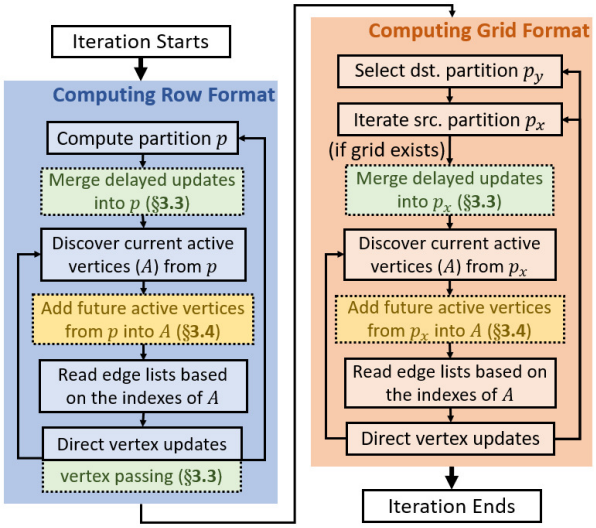


Figure 9: Execution flow in Seraph.

Fig. 8 shows the mechanism of selective pre-computation. In particular, there are two bitmaps originally maintained in Seraph that record the activeness of each vertex: `cur_bitmap` and `next_bitmap` respectively indicate the active vertices of current and next iteration. In normal computation (i.e., without pre-computation), Seraph identifies and computes the active vertices based on the set-bits in `cur_bitmap`. In pre-computation, Seraph first determines the pre-computed vertices based on the method described above and move the set-bits from `next_bitmap` to `cur_bitmap`. In other words, all active vertices (including the current and future ones) are marked in `cur_bitmap`, and the pre-computed ones are removed from `next_bitmap` so that we will not compute them again in the next iteration. Finally, Seraph can simply perform graph computation based on `cur_bitmap`.

As Seraph stores graphs in hybrid format, it is essential to maintain consistency of the pre-computed vertices between row and grid formats to ensure the correctness of algorithms. Even when computing grid format, it is possible for a vertex to be pre-computed in one grid but not in another, as each grid is computed independently. To ensure consistency, Seraph first computes row format while modifying `cur_bitmap`. Next, Seraph uses the modified `cur_bitmap` to compute all grids to maintain consistency. However, pre-computation can only work when the graph algorithm generates future active vertices on the fly during graph computation. If this is not feasible for the algorithm, Seraph offers an option to disable pre-computation and run the algorithm normally.

3.5 Execution Flow

Based on the the available amount of memory specified by user, Seraph divides a graph into the number of partitions

that can hold two partitions of vertex data (i.e., source and destination partitions), one partition of index data, and other small data structures (e.g., edge buffer and active bitmap) in memory. With the information of partitions, Seraph constructs a graph into hybrid format based on the method described in §3.3.2. Next, Seraph performs graph computation based on the execution flow shown in Fig. 9. As mentioned in §3.4, Seraph computes row format first, and then computes the grid format, for each iteration. When computing a partition p, Seraph first merges all the delayed updates back to p to keep all attributes with the latest values. Next, each thread in Seraph will parallelly identify and record the set of active vertices in partition p; if pre-computation is enabled, future active vertices of the same partition will also be handled as mentioned in §3.4. Seraph converts the indexes of active vertices into page-aligned offsets. Later, Seraph exploits the kernel-level Linux Asynchronous IO (AIO) to group multiple required pages into one I/O request, and issue the request with direct I/O to read the edge data into local edge buffer. Finally, Seraph performs graph computation, and vertex passing described in §3.3 will be enabled when computing row format. Please note that, since Seraph uses memory mapping mechanism to reference the vertex and index data backed in files, Seraph accesses the needed vertices and indexes like accessing normal arrays, and does not declare extra user-space memory buffer for holding the vertex and index data.

Seraph computes row format by simply processing all partitions by turns, and grid format is computed by column-based execution order. In other words, when computing grid format, Seraph selects a destination partition p_y and then iteratively computes the source partition p_x if there is a grid between p_x and p_y. As p_y is fixed, all generated vertex updates can directly be absorbed in memory based on this column-based execution order. The computation in grid format ends when all the grids are computed by Seraph.

4 Evaluation

We implement Seraph in C++ and compare it against different state-of-the-art graph computation systems: fully-external (GridGraph [50], V-Part [11], CLIP [1], Lumos [42]), semi-external (Graphene [25]), and shared-memory (Ligra+ [37]) systems. As V-Part and CLIP are not open source, we implement their systems ourselves based on their papers.

We use breath-first search (BFS) [30], weakly connected component (WCC) [15], K-core (Kcore) [29, 35], all-pair shortest-path (APSP) [24, 41], and pagerank (PR) [14, 33] for evaluation. BFS is a typical algorithm for graph traversal by exploring the neighbors until all connected vertices are visited. WCC discovers the number of connected components of a graph; we implement WCC by the method of label propagation [48]. Kcore iteratively removes the vertices of degree less than k, and finally returns a subgraph where each vertex has the degree of at least k. APSP calculates the shortest paths from all vertices. Due to the high complexity of computing all the shortest paths, this evaluation leverages an approximate approach by randomly sampling 32 source vertices, and performs multi-source traversal from the sampled vertices [24, 41]. Finally, PR calculates the popularity of a vertex based on its neighbors' rank values. We run PR for four iterations and activate all vertices in each iteration. Please note that, except for PR constantly activating all vertices, the other evaluated algorithms activate a (dense or sparse) set of vertices in each iteration and represent various access patterns.

Table 2 lists the graphs used for evaluation in this work. The first three are billion-scale, real-world graphs from SNAP [23] and webgraph [3, 4]. In particular, Twitter [9] is social graph, while Gsh2015 [8] and Eu2015 [7] are web crawler graphs. Since the open-sourced graph datasets are all quite small, we use [18] to generate a large-scale graph (called RMAT) for testing the scalability. RMAT contains 8.6 billions of vertices and 112 billions of edges. Notably, because 4 bytes (`unsigned int`) is not enough to represent all the vertex IDs of RMAT graph, we use 8 bytes (`long`) to store the vertex ID for this graph in all systems instead.

Table 2: Evaluated graph datasets.

Graph Name	Num Vertices	Num Edges	Graph Size[5]
Twitter	42 M	1.4 B	11.2 GB
Gsh2015	988 M	33.88 B	271 GB
Eu2015	1.1 B	91.8 B	734 GB
RMAT	8.6 B	112 B	1.7 TB

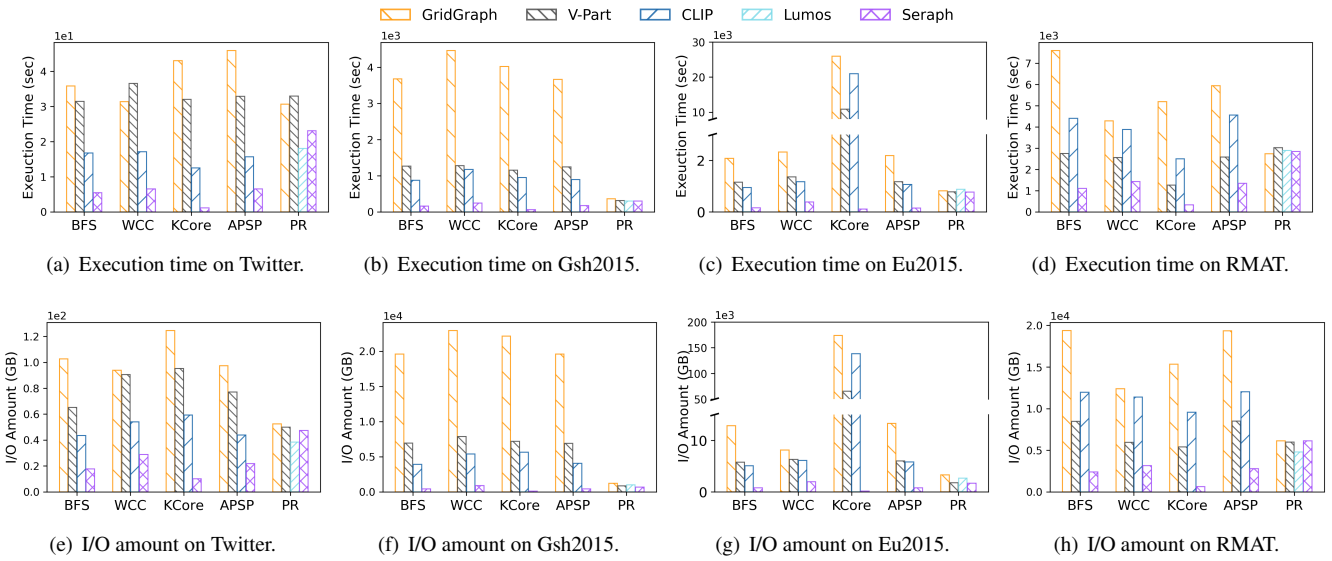
Table 3: Fully-external memory usages for §4.1 and §4.2.

Graph Name	Twitter	Gsh2015	Eu2015	RMAT
Memory Usage	130 MB	2.4 GB	2.7 GB	18 GB

To investigate fully-external graph systems, §4.1 and §4.2 conduct detailed comparisons and study the proposed design choices by offering each system with fixed amounts of memory that are adjusted based on the graph sizes. Specifically, we aim to compute the largest evaluated graph, RMAT, with a reasonable resource available for most people nowadays. Thus, each system is provided with 18 GB to compute RMAT, while the memory amounts for computing other evaluated graphs are also based on a similar ratio of each graph size. The exact memory usage for each evaluated graph are reported in Table 3. §4.3 further enhances the evaluation by examining each fully-external system with different memory amounts. Moreover, we also evaluate semi-external and shared-memory systems in §4.3 to have a comprehensive study about different types of single-machine graph systems.

To compare all types of graph systems on the same platform, all experiments are conducted on the same server: HPE ProLiant DL560 Gen10 server with Intel Xeon Platinum 8160 CPU and 32 x 32GB Dual Rank DDR4-2666 memory (1TB in total) on Debian GNU/Linux 9, and two 1TB Samsung NVMe SSD drives [40] with 6.0 GB/s sequential read bandwidth in total. We use `cgroup` to limit the available memory

[5]The size is measured by storing the graph in the format that each edge is represented by (source vertex ID, destination vertex ID).



(a) Execution time on Twitter. (b) Execution time on Gsh2015. (c) Execution time on Eu2015. (d) Execution time on RMAT.

(e) I/O amount on Twitter. (f) I/O amount on Gsh2015. (g) I/O amount on Eu2015. (h) I/O amount on RMAT.

Figure 10: Overall comparison among Seraph and other fully-external systems.

and `taskset` to confine the used cores. The number of threads for all systems is set to 16 because it is reasonable for commodity PCs nowadays.

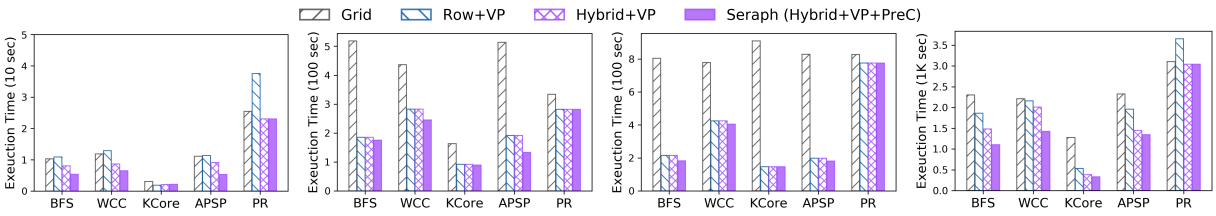
4.1 Fully-external Systems Comparison

This section compares Seraph against four state-of-the-art fully-external graph systems, which are GridGraph [50], V-Part [11], CLIP [1], and Lumos [42]. Fig. 10 illustrates the execution times (seconds) and I/O amounts (GBs) of running the chosen algorithms on different graphs.

We first discuss BFS, WCC, Kcore, and APSP; they are asynchronous graph algorithms, which offers room for optimization via exploiting the algorithmic feature, as discussed in Section 2.2. Overall speaking, GridGraph performs the worst among all systems since it naïvely adopts streaming-based processing (SBP). V-Part issues on-demand I/O to read the edge data, so it averagely improves GridGraph by 37.3% in execution time and 45.1% in I/O amount. However, V-Part's performance is impacted by 1) loading the source vertex table in a partition-based granularity and 2) requiring the overhead of mirror update to handle vertex updates. Thus, compared to CLIP which is an advanced version of GridGraph for asynchronous algorithms, V-Part is slightly slower than CLIP by 5.7% in execution time on average. Last but not least, Seraph performs the best among all systems. Compared to V-Part, Seraph on-demand accesses both edge and vertex data. Compared to CLIP, Seraph not just relies on the effectiveness of on-demand processing but it also leverages pre-computation to optimize asynchronous algorithms. In summary, since Seraph can effectively reduce I/O, it achieves decent performance correspondingly. For execution time (resp., I/O amount), Seraph outperforms GridGraph, V-Part, and CLIP, by 8.9x, 4.9x, and 4.0x (resp., 8.5x, 5.0x, and 4.5x).

Moreover, we can observe that Seraph is especially efficient in computing Gsh2015 graph. Taking BFS as an example; Seraph improves CLIP by 5.0x on Gsh2015 and 3.1x on Twitter. The reason is that running BFS on Gsh2015 takes lots of iterations to traverse the entire graph, and only a few vertices are activated in most iterations. This feature damages the systems adopting SBP as they typically read plenty of data in each iteration. By contrast, since Seraph on-demand accesses the necessary data, the main overhead is the number of issued I/Os, not the number of iterations. Thus, Seraph works better on Gsh2015 than other systems. A similar observation can be found on Kcore which takes lots of iterations to complete but only activates a few vertices in most iterations, providing Seraph more advantages in saving I/O than other systems.

Following, we discuss synchronous algorithm (i.e., PR). Compared to asynchronous ones, PR has to obey the strict synchronous semantics, as discussed in Section 2.2. Moreover, because PR is computation-intensive and constantly activates all vertices, on-demand processing does not show advantages, and all systems perform similarly. Nevertheless, Seraph is still slightly better in I/O as the graph in hybrid format is more lightweight than the graph structures in other systems. On the other hand, Lumos, to save I/O for synchronous algorithm via future computation, is specialized by imposing several constraints during graph computation. However, due to the feature of computation-intensiveness, loading less I/O only brings minor benefit. Please note that we replace the bar of CLIP with Lumos for PR in Fig. 10. Compared to GridGraph, V-Part, and Lumos, Seraph saves the I/O amount by 22.7%, 8.5%, and 7.1%, and improves the time by 11.1%, 10.4%, and 0.6%. Besides, although Lumos optimizes PR via I/O reduction based on GridGraph, the improvement is minor because (1) PR is computation-intensive in our testing, and (2) several designs in Lumos fail to run in fully-external environment. Thus, Lumos improves GridGraph by 39.8% on Twitter, but barely any improvement on the other evaluated graphs.



(a) Execution time on Twitter. (b) Execution time on Gsh2015. (c) Execution time on Eu2015. (d) Execution time on RMAT.

Figure 11: Performance studies of different major designs in Seraph.

4.2 Design Choices

This section demonstrates the performance impact of major designs in Seraph. Specifically, §4.2.1 reveals that hybrid format with vertex passing performs the best than the other two formats. Next, based on the hybrid format with vertex passing, §4.2.2 further studies the performance of selective pre-computation. The configuration of the system and environment in this section is the same as that in §4.1.

4.2.1 Hybrid Format and Vertex Passing

Before showing the results, we discuss the necessity of combining hybrid/row format with vertex passing. Since vertex passing creates the imperative locality for vertex access, computing hybrid/row format without vertex passing will cause the serious problem of memory thrashing. Based on our investigation, it severely degrades the execution time by at least two orders of magnitude, making the system intolerably slow.

Thus, in the following experiments, we compare the performance of grid format (denoted as *Grid*), row format with vertex passing (denoted as *Row+VP*), and hybrid format with vertex passing (denoted as *Hybrid+VP*), respectively. To clearly observe the effects of different formats, we disable selective pre-computation, which will be discussed in §4.2.2. Please note that, the hybrid format stores all edges of Gsh2015 and Eu2015 into row format. Thus, the results of *Row+VP* and *Hybrid+VP* are identical for Gsh2015 and Eu2015.

As revealed in Fig. 11, *Hybrid+VP* generally performs the best. Particularly, *Hybrid+VP* outperforms *Grid* by 37.4% on average, while *Hybrid+VP* averagely improves *Row+VP* (for Twitter and RMAT graphs) by 18.1%. Such improvement is because hybrid format strikes a good balance between utilizing row and gird formats. Compared to *Row+VP*, *Hybrid+VP* alleviates the overhead of vertex passing by storing several dense edge blocks in grid format. Compared to the traditional *Grid*, *Hybrid+VP* improves the efficiency of reading indexes and edge data. Moreover, the comparison between *Hybrid+VP* and *Grid* further implies that naïvely applying on-demand processing into the traditional fully-external designs will lead to limited improvements; proposing techniques suitable for on-demand processing (e.g., hybrid format and vertex passing) is essential for bringing improvement. Finally, *Row+VP* is not necessarily better than *Grid* because, for certain graphs and algorithms, over-using VP will degrade the overall performance instead by wasting too much I/O in transferring vertex updates.

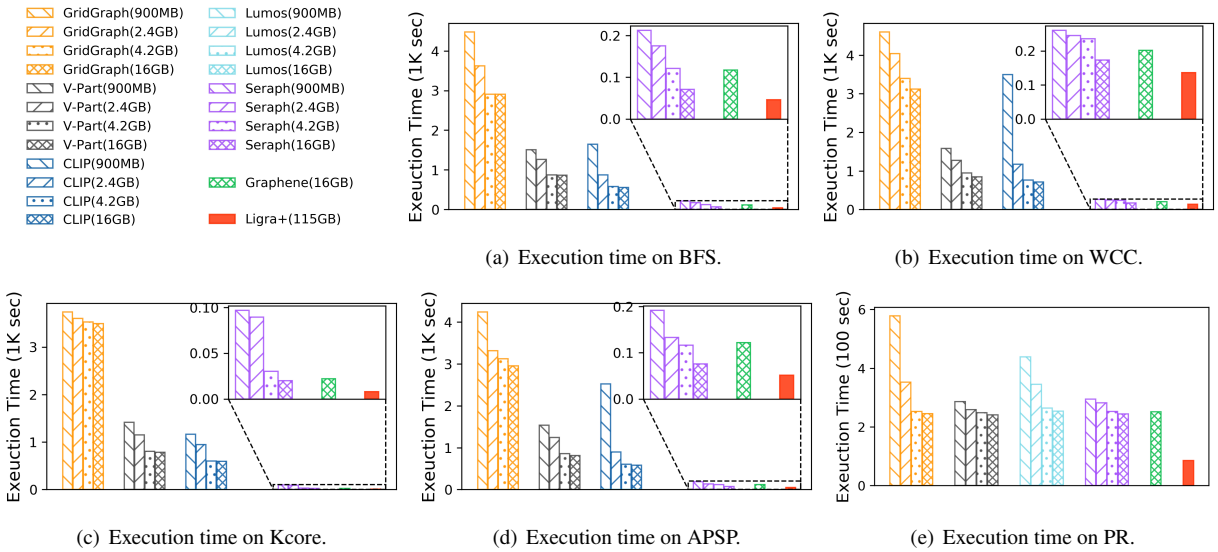
4.2.2 Selective Pre-computation

This section examines selective pre-computation based on the proposed hybrid format with vertex passing. The combination of designs is referred to as *Hybrid+VP+PreC*, and the results are presented in Fig. 11. Please note that, because the selective pre-computation is not feasible for PR, the result of *Hybrid+VP+PreC* is identical to *Hybrid+VP* in PR. We mainly discuss of the other four algorithms in the following. As a whole, selective pre-computation averagely improves *Hybrid+VP* by 16.1% in execution time. This improvement is achieved by re-using I/O opportunistically. Take the largest RMAT graph as example, pre-computation helps to improve execution time (resp., I/O amounts) by 25.6%, 29.1%, 14.5%, and 8.2% (resp., 25.8%, 25.1%, 15.2%, and 12.1%) in terms of BFS, WCC, KCore, and APSP. The results exhibit a similar trend between two metrics. Moreover, different graphs may lead to different amounts of improvement. Given that Twitter and RMAT have shorter diameter than Gsh2015 and Eu2015, the algorithms running on Twitter and RMAT typically incur denser access pattern than Gsh2015 and Eu2015. Thus, pre-computation has better improvement on Twitter and RMAT (21.5%) than Gsh2015 and Eu2015 (10.7%) because it can re-use more I/O under dense access pattern. A similar trends happens on KCore which usually has sparse access patterns. This makes pre-computation have a smaller impact on improving KCore (3.7%) compared to other algorithms (20.2%). However, pre-computation remains a reasonable design choice as it provides harmless benefit.

4.3 Evaluation with Different Amounts of Memory

This section evaluates with different memory amounts. Besides fully-external systems, we also include the state-of-the-art semi-external system (i.e., Graphene [25]) and shared-memory system (i.e., Ligra+ [37]). Particularly, Ligra+ leverages compression schemes to reduce runtime memory footprint. However, the compression program implemented in Ligra+ requires multiple in-memory edge-scale arrays, which significantly limits the graph scale that Ligra+ can handle. In fact, among all the evaluated graph datasets, the largest graph that Ligra+ can handle with our 1TB memory server is the Gsh2015 graph [8]. Hence, this section presents the results based on Gsh2015 graph.

For Gsh2015 graph, 16 GB is enough to hold the entire vertex and index data in memory (i.e., semi-external mode). Thus,



(a) Execution time on BFS. (b) Execution time on WCC.

(c) Execution time on Kcore. (d) Execution time on APSP. (e) Execution time on PR.

Figure 12: Evaluation with different amounts of memory, which is adjusted based on the scale of Gsh2015 graph [8].

for fully-external systems, we vary the provided amounts of memory from 900 MB, 2.4 GB, 4.8 GB, to 16 GB. On the other hand, because Graphene has to run in semi-external mode and Ligra+ must hold the entire compressed graph in memory, we offer them 16 GB and 115 GB of memory to meet their requirements, respectively. It is worth nothing that the *memory requirements of both Graphene and Ligra+ proportionally increase with larger graph scales*. Taking RMAT as an example, Graphene and Ligra+ respectively require 132 GB and 1.7 TB (reported based on raw graph size), while Seraph can compute it with only 18 GB as shown in §4.1.

Fig. 12 shows that, for fully-external systems, their performances improve along with the increasing memory, and Seraph outperforms the other systems regardless of the provided amounts of memory. Moreover, Seraph shows a greater cost-effectiveness than other systems. Compared to fully-external ones, Seraph can use much less memory while achieving better performance. Compared to Graphene, Seraph(4.2GB) almost catches up the performance of Graphene(16GB), with only a minor 10.6% degradation on average. In semi-external mode, Seraph(16GB) averagely improves Graphene(16GB) by 1.31x due to the help of pre-computation. Finally, Ligra+ spends 7.2x more memory than Seraph to achieve an average 1.83x speedup, yet for certain algorithms like WCC, the speedup is only 1.27x. Ligra+ performs the best on PR (improves Seraph by 2.87x). This is because Ligra+ expensively keeps two versions of edge data (out-edges and in-edges) in memory and switches the access between them to resolve the computation-intensive issue of PR. Conversely, Seraph shows greater scalability by computing much larger graphs (e.g., Eu2015 and RMAT) that Ligra+ cannot handle.

5 Related Work

Besides fully-external graph systems, distributed graph systems [5, 13, 27, 49] also show high scalability in graph computation by splitting a graph across multiple machines. For example, Pregel [27] is the first distributed system proposing vertex-centric programming model. Based on vertex-centric model, PowerGraph [13] proposes to optimize the graph computation on natural graphs. Gemini [49] adopts many optimizations to greatly improve the efficiency. Although distributed systems also demonstrate the capability of large-scale graph computation, these approaches are high-cost as user need to build the environment of many machines for large-scale graphs. In contrast, Seraph, as a fully-external system, exhibits optimal scalability by decoupling the capability of large-scale graph computation from the single machine's memory capacity. Thus, Seraph is low-cost for computing any large-scale graph with a constant memory amount.

6 Conclusion

This work develops a new fully-external graph computation system, Seraph, based on the principle of on-demand processing to save I/O. To purse a higher performance improvement, three practical designs are proposed based on the framework of on-demand processing. Specifically, hybrid format is introduced to store the graph while optimizing graph computation, vertex passing is presented for handling vertex updates efficiently, and selective pre-computation explores the possibility of re-using I/O. Seraph, as a fully-external system, exhibits optimal scalability and offers decent performance. It significantly outperforms the other state-of-the-art fully-external systems on all evaluated graphs, based on our experiments.

Acknowledgments

We sincerely thank our shepherd, Ashvin Goel, 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] Scott Beamer, Krste Asanovic, and David Patterson. Direction-optimizing breadth-first search. In *SC '12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 1–10, 2012.

[3] 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.

[4] 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.

[5] Rong Chen, Jiaxin Shi, Yanzhe Chen, and Haibo Chen. Powerlyra: Differentiated graph computation and partitioning on skewed graphs. In *Proceedings of the Tenth European Conference on Computer Systems*, EuroSys '15, New York, NY, USA, 2015. Association for Computing Machinery.

[6] 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.

[7] Eu2015 dataset from WebGraph. https://law.di.unimi.it/webdata/eu-2015/ , 2015.

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

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

[10] 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.

[11] 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.

[12] Michalis Faloutsos, Petros Faloutsos, and Christos Faloutsos. On power-law relationships of the internet topology. *SIGCOMM Comput. Commun. Rev.*, 29(4):251–262, aug 1999.

[13] 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.

[14] 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.

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

[16] 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.

[17] 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.

[18] Farzad Khorasani, Rajiv Gupta, and Laxmi N. Bhuyan. Scalable simd-efficient graph processing on gpus. In *Proceedings of the 24th International Conference on Parallel Architectures and Compilation Techniques*, PACT '15, pages 39–50, 2015.

[19] 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.

[20] 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.

[21] Kartik Lakhotia, Rajgopal Kannan, Sourav Pati, and Viktor Prasanna. Gpop: A scalable cache- and memory-efficient framework for graph processing over parts. *ACM Trans. Parallel Comput.*, 7(1), mar 2020.

[22] Kartik Lakhotia, Rajgopal Kannan, and Viktor Prasanna. Accelerating PageRank using Partition-Centric processing. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 427–440, Boston, MA, July 2018. USENIX Association.

[23] Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection. http://snap.stanford.edu/data, June 2014.

[24] Hang Liu, H. Huang, and Yang Hu. ibfs: Concurrent breadth-first search on gpus. pages 403–416, 06 2016.

[25] 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.

[26] 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.

[27] 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.

[28] Kiran Kumar Matam, Gunjae Koo, Haipeng Zha, Hung-Wei Tseng, and Murali Annavaram. Graphssd: Graph semantics aware ssd. In *Proceedings of the 46th International Symposium on Computer Architecture*, ISCA '19, page 116–128, New York, NY, USA, 2019. Association for Computing Machinery.

[29] Alberto Montresor, Francesco De Pellegrini, and Daniele Miorandi. Distributed k-core decomposition. *IEEE Transactions on Parallel and Distributed Systems*, 24(2):288–300, 2013.

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

[31] Lifeng Nai, Yinglong Xia, Ilie G. Tanase, Hyesoon Kim, and Ching-Yung Lin. Graphbig: understanding graph computing in the context of industrial solutions. In *SC '15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–12, 2015.

[32] 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.

[33] 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.

[34] 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.

[35] 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.

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

[37] Julian Shun, Laxman Dhulipala, and Guy E. Blelloch. Smaller and faster: Parallel processing of compressed graphs with ligra+. In *2015 Data Compression Conference*, pages 403–412, 2015.

[38] Intel Optane 905P SSD. https://www.intel.com/content/www/us/en/products/details/memory-storage/consumer-ssds/optane-ssd-9-series.html.

[39] Samsung 860 EVO SSD. https://www.samsung.com/semiconductor/minisite/ssd/product/consumer/860evo/.

[40] Samsung 970 PRO SSD. https://www.samsung.com/semiconductor/minisite/ssd/product/consumer/970pro/.

[41] Manuel Then, Moritz Kaufmann, Fernando Chirigati, Tuan-Anh Hoang-Vu, Kien Pham, Alfons Kemper, Thomas Neumann, and Huy T. Vo. The more the merrier: Efficient multi-source graph traversal. *Proc. VLDB Endow.*, 8(4):449–460, dec 2014.

[42] 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.

[43] Keval Vora, Rajiv Gupta, and Guoqing Xu. Kickstarter: Fast and accurate computations on streaming graphs via trimmed approximations. In *Proceedings of the twenty-second international conference on architectural support for programming languages and operating systems*, pages 237–251, 2017.

[44] Keval Vora, Guoqing Xu, and Rajiv Gupta. Load the edges you need: A generic i/o optimization for disk-based graph processing. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, pages 507–522, Denver, CO, June 2016. USENIX Association.

[45] 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.

[46] Mingxing Zhang, Yongwei Wu, Youwei Zhuo, Xuehai Qian, Chengying Huan, and Kang Chen. Wonderland: A novel abstraction-based out-of-core graph processing system. *SIGPLAN Not.*, 53(2):608–621, mar 2018.

[47] 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.

[48] Xiaojin Zhu and Zoubin Ghahramani. Learning from labeled and unlabeled data with label propagation. Technical report, 2002.

[49] Xiaowei Zhu, Wenguang Chen, Weimin Zheng, and Xiaosong Ma. Gemini: A Computation-Centric distributed graph processing system. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 301–316, Savannah, GA, November 2016. USENIX Association.

[50] 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.