Published: 05 December 2025

RESEARCH-ARTICLE

ACGraph: An Efficient Asynchronous Out-of-Core Graph Processing Framework

DECHUANG CHEN, Chinese University of Hong Kong, Hong Kong, Hong Kong

SIBO WANG, Chinese University of Hong Kong, Hong Kong, Hong Kong

QINTIAN GUO, Hong Kong University of Science and Technology, Hong Kong, Hong Kong

Open Access Support provided by:

Hong Kong University of Science and Technology

Chinese University of Hong Kong

ACGraph: An Efficient Asynchronous Out-of-Core Graph Processing Framework

DECHUANG CHEN, The Chinese University of Hong Kong, Hong Kong, China
SIBO WANG*, The Chinese University of Hong Kong, Hong Kong, China
QINTIAN GUO*, The Hong Kong University of Science and Technology, Hong Kong, China

Graphs are a ubiquitous data structure in diverse domains such as machine learning, social networks, and data mining. As real-world graphs continue to grow beyond the memory capacity of single machines, out-of-core graph processing systems have emerged as a viable solution. Yet, existing systems that rely on strictly synchronous, iteration-by-iteration execution incur significant overheads. In particular, their scheduling mechanisms lead to I/O inefficiencies, stemming from read and work amplification, and induce costly synchronization stalls hindering sustained disk utilization. To overcome these limitations, we present *ACGraph*, a novel asynchronous graph processing system optimized for SSD-based environments with constrained memory resources. ACGraph employs a dynamic, block-centric priority scheduler that adjusts in real time based on workload, along with an online asynchronous worklist that minimizes redundant disk accesses by efficiently reusing active blocks in memory. Moreover, ACGraph unifies asynchronous I/O with computation in a pipelined execution model that maintains sustained I/O activation, and leverages a highly optimized hybrid storage format to expedite access to low-degree vertices. We implement popular graph algorithms, such as Breadth-First Search (BFS), Weakly Connected Components (WCC), personalized PageRank (PPR), PageRank (PR), and k-core on ACGraph and demonstrate that ACGraph substantially outperforms state-of-the-art out-of-core graph processing systems in both runtime and I/O efficiency.

CCS Concepts: • **Information systems → Data management systems**; **Graph-based database models**.

Additional Key Words and Phrases: Asynchronous graph processing system, Out-of-core computing

ACM Reference Format:
Dechuang Chen, Sibo Wang, and Qintian Guo. 2025. ACGraph: An Efficient Asynchronous Out-of-Core Graph Processing Framework. *Proc. ACM Manag. Data* 3, 6 (SIGMOD), Article 290 (December 2025), 26 pages. https://doi.org/10.1145/3769755

1 Introduction

Graphs are a powerful data structure widely used in machine learning [22, 30, 52], social networks [42, 57], data mining [35, 59], among other domains. The need for fast processing of large-scale graphs has become a critical requirement, giving rise to a series of parallel graph processing systems (GPS) [34, 41, 43, 49, 67]. Yet, as real-world graphs, such as social networks, web graphs, e-commerce networks, and recommendation graphs, continue to grow in size and complexity, they often exceed the memory capacity of a single machine. This challenge has led to the development of both distributed graph computing frameworks [19, 20, 69] and out-of-core GPSs [3, 28, 33, 45, 47, 65, 70]. Although distributed frameworks offer strong scalability, the substantial overhead from fault

*Sibo Wang and Qintian Guo are the corresponding authors.

Authors' Contact Information: Dechuang Chen, The Chinese University of Hong Kong, Hong Kong, China, dcchen@se.cuhk.edu.hk; Sibo Wang, The Chinese University of Hong Kong, Hong Kong, China, swang@se.cuhk.edu.hk; Qintian Guo, The Hong Kong University of Science and Technology, Hong Kong, China, qtguo@ust.hk.

ACM 2836-6573/2025/12-ART290
https://doi.org/10.1145/3769755

tolerance, network communication, and load balancing limits their overall performance potential. By contrast, single-machine out-of-core GPSs achieve a favorable trade-off between scalability and efficiency. Prior work has demonstrated that these systems deliver higher performance per unit cost, making them an attractive alternative for many applications [33, 36, 40, 65, 70].

Early out-of-core GPSs, such as GraphChi [33], X-Stream [47], and GridGraph [70], were specifically designed for hard disk drives (HDDs). Due to the poor random access performance and limited bandwidth of HDDs, optimizing access locality of edges and vertices becomes critical for system performance. To achieve this, they often sacrifice computational efficiency and incur redundant I/O overhead, prioritizing sequential disk access. For instance, GridGraph partitions graphs into 2-D grids and processes each grid entirely in memory to maximize sequential disk access, but at the cost of significant additional disk reads.

Nowadays, advancements in storage technology and declining hardware costs have enabled the adoption of SSDs for large-scale graph storage and processing. Modern SSDs provide orders-of-magnitude faster read/write speeds and near-uniform random/se-quential access performance. Unfortunately, these HDD-era frameworks have proven unable to directly leverage the advantages of these faster storage devices, without substantial redesign [40]. Recent SDD-optimized out-of-core GPSs focus optimization efforts on minimizing overall I/O volume while fully leveraging the available SSD bandwidth. For instance, CAVE [45] leverages intra-/inter-subgraph parallelization approaches to fully exploit the capabilities of underlying storage devices, whereas Blaze [28] employs a pipelined architecture with a new scatter-gather technique called online binning to overlap computation with I/O operations, ensuring uninterrupted data access within an iteration. Experimental results demonstrate that these SDD-era designs consistently outperform those HDD-era designs on high-performance SSDs [40, 45, 65].

However, these GPSs rely on strict synchronization semantics that enforce iteration-by-iteration execution, leading to two key limitations. The first is *I/O inflation*, where the actual amount of data retrieved via I/O operations far exceeds what the task initially requires. This phenomenon primarily stems from two sources: *read inflation* and *work inflation*. Read inflation occurs when accessing a small amount of data (e.g., a few neighboring vertices) requires transferring an entire SSD block. More critically, synchronization semantics prevent accesses to the same disk block across different iterations from being merged into a single I/O operation. Furthermore, these accesses typically have long reuse intervals, as they are interleaved with numerous accesses to other blocks. This long-interval access pattern reduces temporal locality and wastes fetched data. The work inflation arises when a synchronous algorithm incurs higher workload than its sequential counterpart, often because the sequential one leverages prioritized vertex processing to complete the task more efficiently. We empirically validate both sources of overhead in synchronous GPSs in Sec. 3.1.

The second limitation lies in synchronization stalls imposed by strict barrier requirements. In these systems, a new iteration cannot begin until the entire active set of vertices (i.e., the processing frontiers) has been fully processed. The system must pause between iterations to wait for all worker threads to complete their tasks, aggregate the newly activated vertices for the next iteration, and then redistribute them to balance the load. Due to memory constraints, I/O threads remain idle until worker threads have processed loaded data blocks and freed up memory. This bottleneck impedes sustained disk utilization. As will be shown in Sec. 3.2, synchronous systems like Blaze [28] and CAVE [45] show intermittent low disk activity between iterations of BFS, while our proposed ACGraph achieves persistent disk saturation after a short initialization phase.

Contribution. Prior theoretical analyses of graph algorithms have shown that many graph algorithms can be executed in both synchronous and asynchronous modes [8, 14, 58]. This motivates us to consider designing efficient out-of-core asynchronous GPSs. However, directly porting existing in-memory asynchronous paradigm to out-of-core environments exposes fundamental challenges. Out-of-core execution makes data movement a first-order cost, so existing asynchronous schedulers, designed for memory-resident data, fails: coordinating fine-grained task execution with preloading and eviction requires rethinking scheduling. Moreover, ignoring access locality leads to severe SSD read inflation and under-utilization of fetched blocks, as vertices sharing blocks become scattered across priority queues. At the same time, in-memory asynchronous optimizations (e.g., fine-grained parallelism) conflict with out-of-core efficiency needs (e.g., batched I/O), and higher disk latency demands co-design of task scheduling and data movement to overlap computation with I/O. Motivated by these insights, we propose **ACGraph** (Asynchronous Out-of-Core Graph Processing System), a novel asynchronous GPS optimized for SSDs with constrained memory. ACGraph uses the *semi-external* model, where vertex data fits in memory while the full edge set does not and resides on disk, which has been widely adopted in prior work [28, 36, 40, 45, 54, 62, 65, 68].

Unlike conventional frameworks that rely on fixed or coarse-grained scheduling, ACGraph introduces a dynamic, block-centric priority scheduler that adjusts block priorities in real time based on workload dynamics. ACGraph distinguishes itself through an online asynchronous worklist that continuously monitors and reuses loaded blocks immediately upon reactivation, thereby reducing redundant disk accesses and adapting dynamically to changing graph workloads. Further, by unifying asynchronous I/O with computation into a seamless pipelined execution model, our ACGraph achieves continuous execution and high I/O throughput. Finally, a highly optimized hybrid graph storage format is dedicated to ACGraph that streamlines access to low-degree vertices without additional memory or computational overhead. Collectively, these innovations enable ACGraph to deliver significant runtime and I/O improvement compared to state-of-the-art systems under the memory constraints of modern SSD-based platforms.

We implement multiple popular algorithms on ACGraph, including Breadth-First Search (BFS), Weakly Connected Component (WCC), personalized PageRank (PPR), PageRank (PR)[1], and k-core, and compare it against two recent out-of-core GPSs, CAVE [45] and Blaze [28]. Experimental results show that ACGraph significantly outperforms both. Our contributions are as follows:

- We systematically identify that current out-of-core GPS struggles to achieve high I/O efficiency due to I/O inflation and synchronization stalls caused by the synchronization semantics.
- We propose ACGraph, a novel asynchronous out-of-core GPS with block-based priority scheduling, which eliminates synchronization overhead. A novel worklist is designed to dynamically reuse loaded blocks, reducing redundant disk I/O while adapting to evolving graph workloads.
- We propose a highly optimized hybrid storage architecture that enhances access efficiency for low-degree vertices, while simultaneously reducing the graph storage cost, complemented by targeted optimizations to ensure computational efficiency and reduced memory overhead.
- We evaluate ACGraph against state-of-the-art out-of-core GPSs on various workloads including BFS, WCC, k-core, PPR, and PR, and show that ACGraph significantly improves over competitors. Our ACGraph achieves up to 12× speedup over CAVE and 15× speedup over Blaze under similar or even lower memory costs.

2 Preliminary

Consider a directed graph $G = (V, E)$, where V is the vertex set and E is the edge set. An edge from vertex u to vertex v is denoted as (u, v). Without loss of generality, we focus on directed graphs, as undirected graphs can be transformed into a directed one by replacing each edge with two directed ones, an approach commonly used in graph processing [39, 48]. Let $N(v)$ denote the set of out-neighbors of vertex v, and $deg(v) = |N(v)|$ its out-degree.

[1]Note that PR is a special case of PPR with a uniform initial distribution and can be derived using PPR algorithms [44, 55, 56].

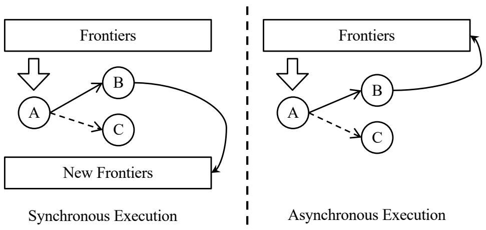


Fig. 1. The vertex-centric synchronous/asynchronous execution model, where vertex A is being processed and passes message to its neighbors B and C, while only B gets activated.

2.1 Graph Processing System (GPS)

Most GPSs are built on the consensus that many graph algorithms follow an iterative message passing pattern: Each iteration starts with a set of active vertices (the *frontier*) that explore their out-neighbors. Neighboring vertex states are updated, and those meeting activation criteria form the new frontier. This process repeats until no new vertices are activated, and the algorithm *converges*.

As an example, we apply the Label Propagation algorithm to compute WCC. Each vertex maintains a WCC label, initially set to its own ID. All vertices start in the initial frontier. In each iteration, a frontier vertex v propagates its label to out-neighbors, who compare it with their own. If the received label is smaller, the out-neighbor updates its label, becomes *activated*, and joins the next frontier. The process continues until there are no new frontiers.

The above WCC example naturally motivates the *vertex-centric* execution model, where vertices serve as the fundamental scheduling units. As illustrated in Fig. 1, a worker thread is processing vertex A that belongs to the current frontier set. After processing, vertex B is activated (indicated by a solid line) and is immediately inserted into the frontier set for the next iteration, while vertex C remains inactive (as shown by a dashed line). In the rest of this paper, the terms *active vertex* and *frontier* are used interchangeably to denote vertices requiring processing.

Synchronous/Asynchronous GPSs. The execution models of GPS can be categorized as either synchronous or asynchronous, as illustrated in Fig. 1. In a synchronous execution model, explicit barriers are inserted between iterations to ensure that all frontiers are completely processed before the next iteration begins. For example, as shown in the left side of Fig. 1, the new frontier set are generated based on the outcome of the fully processed frontiers from the preceding iteration. In contrast, asynchronous execution model removes these strict barriers and often incorporates priority scheduling to improve efficiency for certain algorithms. For instance, as shown in the right side of Fig. 1, once vertex B is activated, it is immediately inserted into the frontier set, typically maintained as a list. Note that vertex B may be scheduled before vertices from prior iterations, if it has high priority. To illustrate further, when executing the WCC algorithm, prioritizing vertices with the smallest label usually lead to faster convergence.

Most existing out-of-core GPSs adopt a synchronous model to enable pre-identification of required disk blocks and coordinated I/O-computation scheduling. However, this convenience comes at the cost of significant I/O inefficiency and synchronization overhead, as detailed in Sec. 3.

Asynchronous I/O. Modern NAND-based SSDs use multi-layer architectures and parallel access across multiple flash chips to boost data transfer bandwidth [2, 26, 27]. They typically use 4KB as the minimum I/O unit and offer comparable performance for sequential and 4KB random reads [21]. Leveraging this, our ACGraph adopts 4KB blocks as the basic partitioning unit for graph data.

Synchronous I/O incurs performance bottlenecks, as threads block on I/O, causing frequent context switches and underutilized CPU resources. Fortunately, asynchronous I/O, such as `io_uring`

[5], addresses this by using dual ring buffers shared between user and kernel space, enabling zero-copy transfer and reducing syscall overhead. Our ACGraph integrates `io_uring` asynchronous mechanism, successfully overlapping computational tasks and I/O operations to establish an efficient pipeline execution architecture.

2.2 Existing Out-of-Core GPSs

Out-of-core GPSs have been widely studied [3, 28, 33, 40, 45, 47, 53, 61, 62, 65, 70, 71]. Earlier systems [33, 47, 70], like GridGraph [70], target HDDs and exploit cheap sequential scans, often at the cost of extra I/O, to avoid random accesses. Later work focuses on reducing I/O costs. Some systems accelerate convergence by asynchronous semantics [3, 13, 63]. For example, CLIP [3] maximizes I/O efficiency by reusing loaded data across multiple processing steps. In contrast, Grafu [62] reduces I/O under synchronous semantics, achieving up to 50% lower runtime and I/O costs both theoretically and in practice compared to GridGraph. However, all of these methods rely on GridGraph-style partitioning, using large grid partitions as the basic scheduling unit, which leads to additional I/O overhead. Besides, MiniGraph [71] adopts a hybrid vertex-/graph-centric model with pipelined I/O and computation to boost parallelism. Yet, like GPSs mentioned above, it partitions the graph into memory-resident fragments as scheduling units and issues a few bulk-load I/Os to reduce random-access, still incurring unnecessary I/Os. As noted in Sec. 1, such design require substantial redesign for SSDs.

The superior bandwidth and random access performance of modern SSDs have shifted the focus to maximizing device utilization and minimizing I/O operations [28, 36, 40, 45, 61, 65]. Graphene [36] introduces four techniques: an I/O-centric programming model, bitmap-based asynchronous I/O, direct hugepage support, and workload balancing, to boost I/O performance. FlashGraph [65] overlaps computation with I/O and optimizes bandwidth through selective access, compact external-memory data structures, SSD access rescheduling to improve page cache hit rates, and conservative I/O merging to reduce CPU overhead. Mosaic [40] is a new locality-optimizing, space-efficient graph representation called Hilbert-ordered tiles, and a hybrid execution model that enables vertex-centric operations in fast host processors and edge-centric operations in massively parallel coprocessors (e.g. Xeon Phi). These frameworks adopt the semi-external setting, consistent with the setting of ACGraph. In contrast, Seraph [61] targets on-demand processing in a *fully-external* setting where even vertex data does not fit in memory, unlike our semi-external approach. Blaze [28] and CAVE [45] are recent out-of-core GPSs under the semi-external model, specifically optimized for SSDs. As shown in their evaluations, both systems deliver superior performance compared to alternative solutions. Thus we take them as our main competitors.

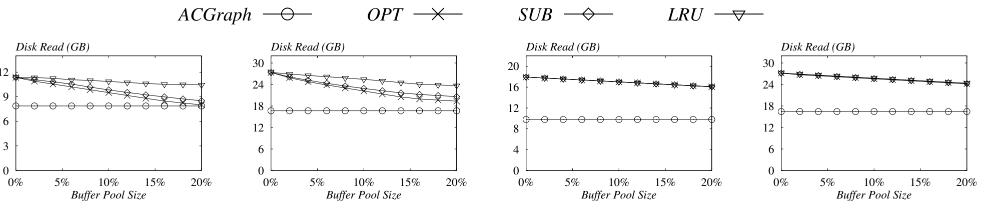
3 Key Observations

As discussed in Sec. 1, we identify two primary limitations that hinder the development of out-of-core processing systems under synchronous semantic models: I/O inflation and synchronization stalls. This section elaborates on these challenges in detail, which serves as the key motivation for our ACGraph framework.

3.1 I/O Inflation

We use the term **I/O inflation** to describe the situation where the actual amount of data retrieved through I/O operations is significantly greater than the amount of data originally requested by the application or task. The I/O inflation under synchronous execution primarily stems from two sources: **read inflation** and **work inflation**, which we describe in detail shortly.

To empirically investigate these two factors and evaluate why conventional buffer pool technique fail to mitigate them, we design the following experiment: we run BFS and WCC algorithms



(a) Twitter (BFS) (b) Friendster (BFS) (c) Twitter (WCC) (d) Friendster (WCC)

Fig. 2. Disk read volume vs. buffer size (% of graph size).

under synchronous execution modes and record disk block accesses at each iteration. Having full knowledge of the block access patterns, we implement various cache replacement strategies to reduce the total number of I/O accesses, as follows:

(1) Belady's optimal algorithm (*OPT*) [7, 50]: A theoretically optimal policy that evicts the item with the longest time until its next access, establishing a minimal disk access baseline.
(2) A suboptimal policy (*SUB*): A heuristic that evicts blocks unused in the next iteration (when identifiable) or randomly selects victims otherwise.
(3) Least Recently Used (*LRU*): A widely adopted policy that prioritizes eviction of the least recently accessed items.

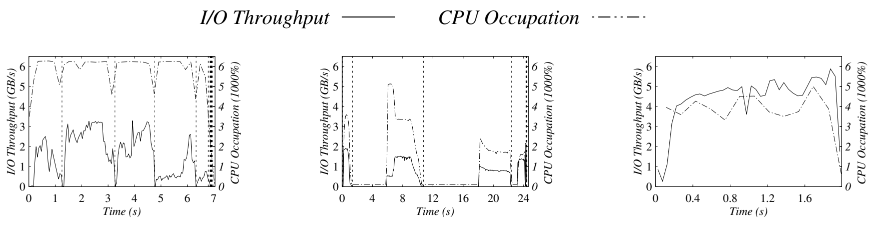
As the *OPT* policy assumes full future access knowledge (albeit unrealistic), it serves as a theoretical lower bound for the disk read volume under synchronous execution. Fig. 2 shows the total disk read volume under the three cache replacement strategies, as buffer pool size varies from 4 MB to 20% of the graph size (with minimum size 4 MB denoted as 0%). For comparison, we also include the disk read volume of our ACGraph, using a fixed 32 MB buffer (less than 1% of the graph), shown as horizontal lines in Fig. 2.

Read Inflation. Now consider the first source of I/O inefficiency in synchronous execution: *read inflation.* Although BFS accesses each edge exactly once, SSDs operate at block granularity—so accessing a single edge requires loading its entire block. Many of these edges may be unused, resulting in unnecessary reads. Worse, the limited cache size may evict a block before all its edges are accessed, leading to redundant reloads, which we call it as *read inflation.* For example, if block B contains edges for vertices v_1 and v_2, and v_1 is activated in Iteration 1 while v_2 is activated in Iteration 3, block B may be evicted and must be reloaded—resulting in duplicated I/O.

A larger cache pool can alleviate the issue of read inflation to some extent by holding more blocks, increasing the chance of cache hits for repeated accesses. In the extreme case where the entire graph fits in memory, BFS incurs no read inflation. However, as noted in the introduction, real-world graphs are rapidly growing and often exceed a single machine's memory capacity.

As shown in Fig. 2(a)–(b), all cache replacement strategies reduce disk read volume as the buffer pool increases from 4 MB to 20% of the graph size. For instance, the OPT strategy reduces reads by 29% on Twitter and 25% on Friendster. OPT consistently outperforms the others due to its idealized knowledge of future accesses. However, even with a buffer size equal to 20% of the graph size, OPT still performs worse than our asynchronous ACGraph on Friendster using only a 32 MB buffer, highlighting the limitations of synchronous execution under read inflation.

Work Inflation. The WCC experiments reveal the second source of I/O inefficiency in synchronous graph processing systems: work inflation. In parallel WCC computation, the common approach is to use the Label Propagation (LP) algorithm, which initializes each vertex with a unique label (typically its vertex ID) and iteratively updates each vertex's label to the minimum label among its neighbors until global convergence is achieved. In a sequential LP algorithm, the propagation can be focused



(a) Blaze (b) CAVE (c) ACGraph (ours)

Fig. 3. Disk throughput & CPU occupation over time on BFS queries; Vertical dashed lines mark the starting point of each iteration.

by starting from the vertex with the minimum label and selectively propagating to its neighbors. In contrast, the synchronous setting propagates updates across all vertices in each iteration to accelerate convergence, a strategy that often leads to unnecessary computations compared to the more targeted sequential approach, thereby resulting in work inflation. This work inflation further causes unnecessary I/Os. For example, on both the Twitter and Friendster graphs, over 99% of I/O costs stem from the first 2–3 iterations, during which nearly all vertices remain active. This forces almost the full graph to be reloaded into memory each iteration, limiting the effectiveness of cache policies. As shown in Figs. 2 (c)-(d), disk read reduction scales near-linearly with cache pool size. Even with idealized policies (*OPT*, *SUB*) or practical ones (*LRU*), the performance gap between strategies is minimal (under 0.7% on both graphs), and achieve only 10% I/O reduction with 20% buffer pool size. Consequently, relying solely on the buffer pool is insufficient to further mitigate the I/O costs arising from work inflation in synchronous GPSs. Actually, within a connected component, only updates from vertices with the smallest label to their higher-labeled neighbors are effective. All other label updates will finally be overwritten and thus redundant. By prioritizing blocks containing active vertices with the minimum label, ACGraph enables the algorithm to execute in a better order, thereby reducing redundant edge accesses and computations, leading to reduced disk I/O.

3.2 Synchronization Stalls

Fig. 3 presents the I/O throughput and CPU occupation over time across graph processing frameworks [28, 45] during BFS execution on the Twitter graph using 64 threads. We have omitted the system initialization and graph index loading phases, with time zero representing the actual start of the algorithm. The IO throughput curves are obtained through disk sampling using the `blk_trace` [6] tool on Linux, with a 50 ms sampling window, where throughput is calculated based on the number of complete actions of read operations within each window. Both Blaze and CAVE exhibit periodic throughput and CPU drops around each iteration, which we refer to as **synchronization stalls**. Notably, CAVE even experiences prolonged zero I/O at the beginning of each iteration, as it employs a single thread for result collection, task generation, and distribution. A more critical challenge arises from "log-tail" iterations: In large-diameter graphs (e.g. Uk-Union), algorithms like BFS may run for hundreds of iterations, with over 80% of them activating fewer than 1,000 vertices. This lead to low average throughput despite high peak performance. Hence, even though both systems utilize work-stealing to reduce load imbalance, Blaze's average throughput is only 2.4 GB/s (compared to a 4.1 GB/s peak), and CAVE's is merely 0.27 GB/s (compared to a 2.3 GB/s peak).

The inefficiencies inherent in synchronous out-of-core GPSs as mentioned above motivate us to introduce ACGraph, an asynchronous graph processing system. In the following section, we detail our design of the ACGraph framework.

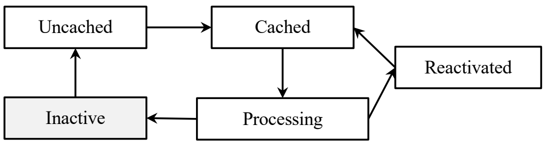


Fig. 4. Block state machine.

4 The ACGraph Framework

In this section, we present the details of the ACGraph framework. In ACGraph, edges are partitioned into 4KB blocks and stored on SSD, while vertex data (e.g., vertex states, degrees, and edge offsets) are maintained in memory. ACGraph ensures that the edges of a vertex span only one block if they can fit into 4KB. If the edges of a vertex are partitioned into a block, we logically assign the vertex to that block, hereafter referred to as the vertex's *assigned block* or *associated block*. Graph partitioning strategies and corresponding storage optimizations are discussed in Sec. 5.

4.1 Block-Centric Async. Execution Model

The challenges outlined in Sec. 3 present significant resolution difficulties under synchronous semantics or stem from inherent limitations of the synchronous paradigm. This motivates our exploration of an asynchronous execution model. Without the synchronization constraints, when a block is loaded, processing can continue as long as it contains active vertices, thereby improving resource utilization and reducing read inflation. While not every vertex within a scheduled block may contribute optimal progress, this overhead is negligible against the substantial reduction in I/O operations. In addition, block execution order can be priority-scheduled to enhance work efficiency for certain algorithms (e.g., WCC). Furthermore, the asynchronous model only requires a single global synchronization at the algorithm's conclusion, eliminating per-iteration synchronization overhead and associated load imbalance issues.

Building on these principles, we propose a novel priority-enabled asynchronous execution model designed for modern SSDs: **the block-centric execution model**. The block-centric execution model, as what the name suggests, explicitly treats blocks as first-class entities rather than mere disk-stored data, as seen in traditional out-of-core systems, while vertices are associated with their assigned blocks, instead of being treated as standalone entities. To clarify terminology, a *block* in ACGraph refers to a logical scheduling unit containing a disjoint partition of vertices. A block corresponds to an SSD region termed a *disk block*, which becomes *block data* when loaded into memory. Each block maintains metadata including status, priority, active vertices in it, and a pointer referencing its block data. Scheduling decisions operate at the block level, where block priorities are derived by aggregating the priorities of their frontiers (e.g., using the maximum value for high-priority-first or minimum value for low-priority-first). Individual vertex priorities, in contrast, are application-defined—for instance, using vertex distance as the priority metric in BFS. Every active vertex is maintained by the local frontier set of its assigned block, instead of a global one. When the priority of a vertex v gets updated, the priority of its associated block is also updated.

Each block in this execution model works as a state machine as shown in Fig. 4. Except *Inactive*, the remaining four states indicate that a block contains active vertices at different execution phases:

- *Uncached*: Block data has not been loaded into memory.
- *Cached*: Block data is loaded in memory but not yet processed.
- *Processing*: The block is being processed by an executor.
- *Reactivated*: Some vertices in the block are newly activated during processing.

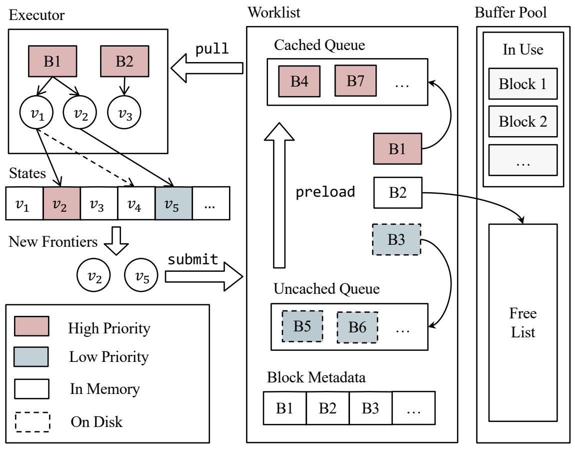


Fig. 5. The architecture of ACGraph (Bi indicates block i).

The state of each block dynamically evolves. All blocks initialize in an inactive state. When a vertex is activated, the associated block transits to an uncached state. When memory is available, the system prioritizes and loads the uncached block with the highest priority into memory, changing its state to cached for execution readiness. A cached block enters the processing state when being executed. If new vertex activations occur during processing, the block is marked as reactivated. After the processing of the block, reactivated blocks immediately return to the cached state for prioritized re-execution, while blocks without reactivation revert to the inactive state. Under this design, an in-memory subgraph is dynamically maintained by active blocks. Vertices within the in-memory subgraph are processed preferentially once activated.

4.2 The Architecture of ACGraph

Fig. 5 shows the architecture of ACGraph, comprising three core components: *executors*, *worklist*, and *buffer pool*. Each executor runs on a dedicated thread and processes tasks as described in Alg. 1. Executors continuously interact with the worklist—the system's scheduler—to pull and submit tasks. The worklist dictates execution order and interleaves task scheduling with I/O using `io_uring` to avoid I/Os blocking executors. It also maintains block metadata and manages state transitions. The buffer pool maintains and recycles a fixed set of memory buffers to improve access efficiency.

Executor. The visualization in Fig. 5 employs color (red for high-priority, blue for low-priority blocks) and border style (solid for memory-resident, dashed for disk-resident) to represent block states. Alg. 1 outlines the executor's workflow. The executor continuously pulls a batch of tasks from the worklist until it is exhausted (Line 1), where a task corresponds to a block and includes a pointer to preloaded block data and a set of active vertices (Line 4). For example, the executor in Fig. 5 acquires two block tasks, where block 1 contains active vertices v_1 and v_2, while block 2 has only one frontier v_3. The executor iteratively processes the active vertices of each task (Lines 2-5). The adjacency lists of the vertices in a block task are stored in separate locations within the block data, which can be located via the vertex information maintained in memory (Line 6). The executor processes these vertices by first calling a user-defined `apply` function to generate messages for their neighbors (Line 7). Then it updates their neighbors' states using a user-defined `propagation` function (Line 13). Newly activated vertices (v_2 and v_5) are temporarily stored in a local buffer (Lines 14-15), which is submitted to the worklist when necessary (Line 9). These vertices are grouped by their assigned blocks to update corresponding metadata in the worklist (v_2 is grouped to block

Algorithm 1: Workflow of executors

```
Input: Input graph $G = (V, E)$, worklist $W$, user-defined function apply and process
1  while $tasks \leftarrow W$.pull() do
2      for $task \in tasks$ do
3          init an empty list $buffer$;
4          $tid, frontiers, data \leftarrow task$ ;
5          for $u \leftarrow frontiers$ do
6              $neighbors \leftarrow G$.getNeighbors($u$, $data$) ;
7              $msg \leftarrow$ apply($u$) ;
8              process($msg$, $neighbors$, $buffer$);
9          $W$.submit($buffer$);
10         $W$.finish($tid$) ;
11 Function process($msg, neighbors, buffer$)
12     for $v \leftarrow neighbors$ do
13         $priority \leftarrow$propagation($msg$, $v$);
14         if $priority > 0$ then
15             $buffer$.append($priority$, $v$);
```

1 while v_5 to block 3). Upon completing a block's processing, the executor invokes the `finish` method to notify the worklist and update the block's state (Line 10).

Worklist. The worklist serves as the core data structure of ACGraph and is pivotal for enabling sustained computation and I/O triggering. The worklist maximizes block data reuse through two components: a queue for scheduling loaded blocks and a priority queue for scheduling unloaded blocks. This dual-queue architecture ensures memory-resident blocks always precede disk-resident ones in execution order. This prioritization stems from the `pull` operation exclusively retrieving batched blocks from the cached queue. During task submission, the worklist dynamically adapts to block states: uncached blocks such as block 3 are routed to the uncached queue, while cached blocks enter the cached queue. For blocks in processing or reactivated states (exemplified by block 1), the system updates their frontier metadata and priority values, transitioning their state to reactivated without queue insertion. The `finish` method initiates state-specific transitions: reactivated blocks like block 1 re-enter the cached queue, while inactive blocks release allocated buffer pool resources, as demonstrated by block 2. This state-aware design guarantees execution priority through three operational principles: cached queue dominance in task retrieval, state-driven transitions for active blocks, and automated memory reclamation via buffer pool integration.

Buffer Pool. The buffer pool pre-allocates memory space for storing block data and employs a concurrent queue (the free list shown in Fig. 5) to manage available slots. It supports batch allocation and release operations for block data through a single call. Notably, the buffer pool itself does not provide data query functions; whether a specific block data resides in the pool is ultimately determined by external metadata queries tied to each block.

4.3 Synchronous Execution

Asynchronous execution suits algorithms that converge rapidly under arbitrary ordering, such as k-core. Another class of problems requires specific execution orders for optimal time complexity yet converges correctly under relaxed orders (e.g., WCC). ACGraph supports such cases, delivering correct results while leveraging priority scheduling to enhance practical performance.

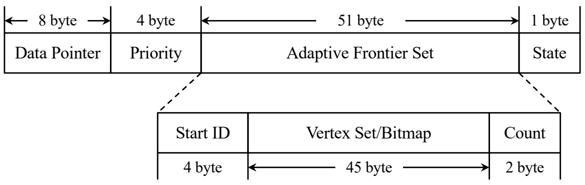


Fig. 6. Block metadata.

However, some algorithms fundamentally require global synchronization for correctness. Consider maximal independent set (MIS) problem: an efficient parallel algorithm (i.e. Blelloch's Algorithm 2 [9]) randomly assigns each vertex a fixed unique label, initially marks all vertices as live and create an empty set s. Each iteration adds the live vertices with no lower-labeled live neighbors to set s, then marks them and their neighbors as dead. This repeats until all vertices become dead, and set s forms a valid MIS.

Fortunately, synchronous execution is a special case of asynchronous execution. ACGraph supports synchronous mode with minimal modification: new frontiers are inserted into a fresh worklist instead of the current one. This creates global barriers, ensuring that newly activated vertices are only processed after all current frontiers have completed. With this flexibility, users can select the execution mode best suited to their algorithm's requirements.

Note that ACGraph does not automatically select execution modes. Instead, it exposes distinct APIs for users to match specific requirements (see Sec. 4.6). While adaptive execution mode selection based on runtime conditions represents an active research area [17, 58], this capability remains beyond our current scope. We therefore designate automated execution mode switching on out-of-core graph processing as promising future work.

4.4 Consistency and Correctness

ACGraph permits concurrent vertex/edge access by multiple threads. To ensure consistency, users must implement appropriate concurrent control (e.g., atomic variables or locks) in user-defined functions (see Sec. 4.7), as in prior frameworks [43, 45, 49]. This offers two benefits: first, it avoids system-level locking overhead or additional scheduling constraints that could hinder parallelism (e.g., preventing simultaneous execution of vertices with shared neighbors); second, it enables users to balance performance and consistency via selecting concurrency control primitives. For example, in scenarios permitting precision loss, one may trade consistency and correctness for higher performance by employing lightweight atomic operations (e.g. replacing `compare_and_swap` with a simple `store`) and relaxed memory ordering.

With proper user-level implementations, such as atomic and data-race-free `apply` and `propagation` functions, ACGraph achieves *sequential consistency*, meaning its results correspond to some valid sequential order. Thus, if the algorithm is correct under all sequential executions, it remains correct on ACGraph. This matches the consistency level of existing in-memory asynchronous GPSs, such as GraphLab [37]. In asynchronous mode, ACGraph ensures that a vertex is processed only after it is activated, and the number of invocations never exceeds its activation count. These properties help verify algorithm correctness (see Sec. 4.7 for example).

4.5 Block Management

Block Metadata. The block metadata structure (Fig. 6) uses 64-byte alignment to reduce false sharing in concurrent access. It consists of four parts: an 8-byte atomic data pointer, a 4-byte priority field encoded as a signed integer, a 51-byte adaptive frontier set (AFS), and a 1-byte state flag. Mutual exclusion is implemented using the lowest bit of the data pointer as a spinlock, avoiding dedicated mutexes. The 51-byte AFS is organized into three segments: a 4-byte Start ID v_{start} which marks the smallest vertex ID in the block, a 2-byte counter to track the number of active vertices, and a 45-byte storage space. The AFS implements dual storage strategies for memory and computation efficiency. In sparse mode, active vertices are directly stored in an array with a maximum capacity of $\lfloor 45/4 \rfloor = 11$ vertices (4-byte per vertex). The dense mode uses the 45 bytes (360 bits) as bitmap encoding, to represent vertices' activity within the range $[v_{start}, v_{start} + 360)$. Mode transitions occur dynamically based on vertex count, with v_{start} enabling consistent offset calculations.

A 4KB disk block can hold up to 1,024 edges. Extreme cases with all degree-1 vertices would require 1,024 bits (128 bytes), exceeding metadata limits. The hybrid storage solution (Sec. 5) resolves this by keeping vertices with degree ≤ 2 in memory, and retaining only those with degree ≥ 3 within blocks, without incurring additional memory overhead. This reduces the theoretical maximum vertex capacity to $\lfloor 1024/3 \rfloor < 342$, while the AFS supports 360 vertices in the dense model (using bitmaps), which is sufficient.

Preload. Whenever an executor attempts to pull tasks from the worklist (Algorithm 1 Line 1), it first triggers a *preload* process for other blocks. This mechanism retrieves a batch of highest-priority blocks from the uncached_queue, allocates buffers from the buffer pool if the buffer pool is not full, and generates asynchronous I/O tasks to read disk blocks. The preload process also manages synchronization of completed I/O tasks, updates the state of these blocks to cached, and delegates them to the cached_queue for scheduling. Disk I/O is executed asynchronously via io_uring, enabling threads to collect completed I/O operations (freeing resources) in a non-blocking manner, submit new I/O tasks, and return immediately without waiting. ACGraph avoids segregating I/O and computation threads, as it employs adaptive thread distribution. For example, under a "fast computation, slow I/O" scenario, the executor increases its task-pulling frequency, thereby issuing more asynchronous I/O requests to saturate disk bandwidth and maximize throughput.

Early-stop. A concern regarding the reuse of reactivated blocks is potential computational inefficiency, as this approach may disrupt the priority-based block execution order. For example, in BFS algorithms, blocks retained in memory might form a suboptimal long path, rendering many subsequent updates redundant. To mitigate this, ACGraph employs an early-stop strategy: any block reused consecutively beyond a user-set threshold is forcibly evicted from memory and returned to the uncached queue for scheduling. Notably, our experiments do not reveal significant effect of this issue. Consequently, ACGraph disables this strategy by default.

4.6 APIs

ACGraph abstracts implementation details via two high-level APIs: `foreachVertex` and `asyncRun`, which follow the design philosophy of Ligra [49]: The first supports parallel traversal of vertices (analogous to VERTEXMAP), while the second iterates over the edges of the frontier (similar to EDGEMAP). This conceptual alignment enhances both usability and simplicity. We adapt their semantics to better support asynchronous execution. Specifically, the API `foreachVertex` iterates over all vertices, inserting activated vertices back into the given worklist rather than creating a new one. Meanwhile, the API `asyncRun` repeatedly processes active vertices in the worklist and reinserts newly activated vertices, continuing this cycle until no active vertices remain.

The definition of `foreachVertex` is listed in Eqn. (1), with three parameters: a graph G, an optional worklist W, and a vertex-to-priority function f. When invoked, `foreachVertex` applies f to all vertices of G in parallel. If the return value of f for a vertex is greater than 0, the vertex is submitted to the worklist W (if provided), indicating it is activated. `foreachVertex` is ideal for

Algorithm 2: Breadth-First Search

```
Input: Input graph $G = (V, E)$, user-defined function apply and process, source vertex $s$
1  $dis \leftarrow \{\infty, \ldots, \infty\}$;
2  $dis[s] \leftarrow 0$;
3  Initialize a worklist $W$;
4  $W$.submit($s$);
5  asyncRun($G$, $W$, apply, propagation);
6  function apply($u$)
7      return $dis[u]$.load();
8  function propagation($msg, v$)
9      $d \leftarrow dis[v]$.load();
10     while $d > msg+1$ do
11         if $dis[v]$.CAS($d$, $msg+1$) then
12             return $msg+1$;
13         $d \leftarrow dis[v]$.load();
14     return $0$;
```

initializing vertex states and generating initial frontiers efficiently.

$$\begin{aligned} &\texttt{foreachVertex}(G : \texttt{graph}, W : \texttt{worklist}, \\ &\qquad f : (\texttt{vertex}) \rightarrow \texttt{priority}) \rightarrow \texttt{void} \end{aligned} \tag{1}$$

The asyncRun method, as abstracted in Eqn. (2), takes four parameters: a graph G, a worklist W of initial active vertices, and two functions apply and propagation. During execution, asyncRun processes each active vertex in W by first invoking the apply function to generate messages destined for its neighbors. These messages are then propagated to neighboring vertices via the propagation function. If the return value of propagation for a neighbor exceeds zero, the neighbor is marked as activated and inserted into the worklist W, ensuring iterative updates until convergence.

$$\begin{aligned} &\texttt{asyncRun}(G : \texttt{graph},\ W : \texttt{worklist},\ \texttt{apply} : (\texttt{vertex}) \rightarrow msg\_t, \\ &\qquad \texttt{propagation} : (msg\_t, \texttt{vertex}) \rightarrow \texttt{priority}) \rightarrow \texttt{void} \end{aligned} \tag{2}$$

Additionally, ACGraph provides a synchronous syncRun API. Its interface resembles asyncRun but aggregates newly activated vertices into a distinct worklist , which is returned upon completion.

4.7 Example

We demonstrate the versatility of these APIs through two canonical examples: BFS (Breadth-First Search) and k-core. In our example, atomic variables are employed to enforce thread safety. Two atomic operations are utilized: load() reads the current value of a variable. The CAS(old, new) operation compares the variable's current value with old; if equal, updates it to new and returns true; otherwise, returns false without modification. The fetchSub(x) operation atomically subtracts x from the variable's value and returns its original value. All operations are performed atomically, utilizing std::memory_order_acquire for load and std::memory_order_acq_rel for CAS and fetchSub to establish acquire-release semantics. This guarantees that all memory writes before a given release operation become visible to subsequent acquire operations on the same variable.

Algorithm 3: k-core

```
   Input: Input graph $G = (V, E)$, user-defined function apply and process, threshold $k$
1  $deg \leftarrow \{0, \dots, 0\}$ ;
2  Initialize a worklist $W$;
3  foreachVertex($G$, $W$, init);
4  asyncRun($G$, $W$, apply, propagation);
5  function init($u$)
6      $deg[u] \leftarrow G$.getDegree($u$);
7      if $deg[u] < k$ then
8          return $1$;
9      return $0$;
10 function apply($u$)
11     return $0$;
12 function propagation($msg, v$)
13     $d \leftarrow deg[v]$.fetchSub($1$) ;
14     if $d = k$ then
15         return $1$;
16     return $0$;
```

BFS. Alg. 2 implements BFS using the offered APIs. Lines 1-2 initialize a distance array dis of size $|V|$ with all values set to ∞ representing unvisited vertices, except the source vertex s's distance to 0. Lines 3-4 create an empty worklist W and activate the source vertex. The ACGraph framework then autonomously executes message passing via its `asyncRun` method until convergence (when the worklist processed by the executor becomes empty). The `apply` function (Lines 6-7) directly returns the vertex's current distance as messages. The `propagation` function (Lines 8-14) compares each neighbor v's distance with the candidate value $msg + 1$. If $msg + 1$ is smaller, it attempts to atomically update $dis[v]$ using a CAS operation. Successful updates activate neighbor v with priority $msg + 1$ via insertion into W, propagating the BFS frontier iteratively.

The correctness of Alg. 2 can be proved by mathematical induction. Assume that all vertices with $dis[\cdot] = d$ (for any $d \geq 0$) have correctly converged to their distances. Now consider a vertex v with distance $d + 1$. By definition, v must be a neighbor of some vertex u' with distance d. The `propagation` function ensures that when $dis[u']$ is updated to d, u' is activated. ACGraph assures that once activated, u' will be processed, triggering `propagation` and allowing $dis[v]$ to be updated based on $dis[u']$. As a result, $dis[v]$ converges to $d + 1$. Since source s is correctly initialized with distance 0 and activated, this inductive process ensures that all vertices reachable from s will finally converge to their correct distances.

***k*-core.** The k-core algorithm iteratively prunes vertices with degree less than k and their incident edges until no such vertices remain. We model vertex states using their degrees, constructing a global array deg to store these values. Unlike BFS, Line 3 leverages the `foreachVertex` method to parallelize initialization: the `init` function (Lines 5-9) sets each vertex's state to its initial degree, while vertices with $deg[v] < k$ are activated and added to the worklist (Lines 7-8). During the propagation phase, the algorithm decrements the target vertex's degree by 1 to simulate edge removal (Line 13). Following this update, if the vertex's degree equals k exactly before the update,

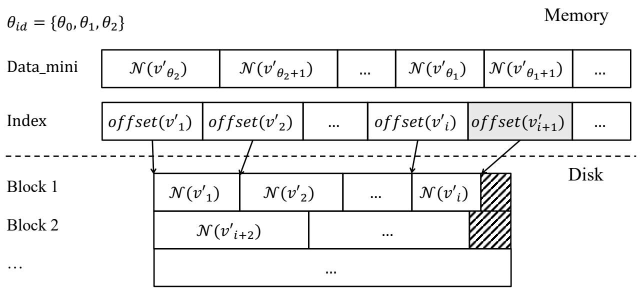


Fig. 7. The hybrid graph storage in ACGraph, with $\delta_{deg} = 2$, where v'_i is the i-th vertex after vertex reordering and the gray cell represents virtual vertices.

it is activated to propagate further pruning (Lines 14-15). The equality constraint ensures that each vertex is activated at most once, guaranteeing algorithmic correctness. This process continues until convergence, i.e., when the worklist processed by the executor becomes empty.

The correctness of Alg. 3 can be established by mathematical induction also. Consider a valid vertex removal sequence $v_1, v_2, \ldots, v_t$ produced by the serial k-core algorithm. Suppose vertex v_1 through v_i have already been correctly activated in Alg. 3. For v_{i+1}, if its initial degree is less than k, it is activated in `init`. Otherwise, there exists a in-neighbor $u \in \{v_1, \ldots, v_i\}$ of v_{i+1} whose activation triggers a `propagation` operation that reduces $\deg[v_{i+1}]$ to $k-1$ (guaranteed by the atomicity of `fetchSub`), leading to the correct activation of v_{i+1}. Since v_1 always has initial degree less than k, it is correctly activated in `init`. Besides, ACGraph ensures that each activated vertex executes exactly once in this context, preventing erroneous activation of k-core vertices. Therefore, a vertex is activated in Alg. 3 if and only if it lies outside the k-core.

5 Hybrid Storage Architecture

In in-memory GPSs, *Compressed Sparse Row (CSR)* is widely used for graph storage. The CSR structure has two core arrays: the *index* array storing offsets to adjacency lists in the *data* array, and the *data* array containing edges, which stores only destination vertices. As the adjacency list is closely arranged in *data*, vertex degrees can derive directly from offset differences via $\texttt{deg}(v) = \texttt{offset}(v+1) - \texttt{offset}(v)$, avoiding extra degree storage.

Existing semi-external GPSs present several techniques to optimize storage. Blaze [28] groups sixteen 4-byte degrees per cache line and stores only cache-line locations in the offsets region. While this saves memory by removing offset fields for vertices, it introduces additional prefix-sum computation overhead for indirect vertex offset access. CAVE [45] uses 4-byte offsets to reduce memory, but this limits its scalability for large graphs. ChunkGraph [54] embeds mini edge lists into offsets, but still requires a separate degree array to identify vertex types. In contrast, we propose a hybrid storage architecture that eliminates the degree field and allows mini adjacency lists (for small-degree vertices) to reside directly in memory without extra metadata, while supporting block-centric execution (see Fig. 7). The in-memory *index* array and vertex states (not shown in Fig. 7) leverage fast random RAM access, which now can hold billions of vertices on standard DRAM. To support block-centric execution, ACGraph enforces that adjacency lists smaller than 4KB fit within a single block, while larger ones span consecutive blocks.

To achieve this, ACGraph employs a locality-preserving last-fit partition policy strategy (Sec. 5.1), while preserving inherent locality [43]. Yet, these constraints disrupts CSR's degree-offset property, i.e., $\texttt{deg}(v) = \texttt{offset}(v+1) - \texttt{offset}(v)$. This disruption occurs for two reasons: *(i)* Non-monotonic offsets, as now it is possible for $\texttt{offset}(v) > \texttt{offset}(u)$ even when $v < u$; *(ii)*

Internal block fragmentation, as not all blocks are fully utilized. Hence, each vertex must store a 12-byte index (8-byte for the offset and 4-byte for the degree), which may expand to 16-byte due to memory alignment when stored in a `struct`, inflating memory costs. We eliminate the overhead with two simple yet effective techniques: *vertex reordering* and *virtual vertex insertion.* These allow us to store mini adjacency lists directly in memory without additional overhead. Together, these optimizations (see Sec. 5.2) form the cornerstone of ACGraph's hybrid storage.

5.1 Graph Partition Strategy

Partitioning graphs into 4KB blocks under the specified constraints introduces a trade-off between preserving graph locality and minimizing intra-block fragmentation. For example, a best-fit approach that minimizes fragmentation by distributing vertices to blocks with an exact fit may scatter low-degree vertices across multiple blocks. This scattering undermines inherent locality and is particularly disadvantageous for algorithms, such as k-core, that primarily access small vertices. On the other hand, partitioning vertices without reordering may introduce large fragmentation across many blocks, resulting in severe read inflation issues.

To balance these factors, ACGraph employs a heuristic locality-preserving last-fit partitioning strategy. Specifically, only blocks within a sliding window are considered as candidate placements for the current vertex. When no block in the window can accommodate the vertex's edge list, new blocks are created and the window shifts accordingly. Adjusting the window size allows for a tunable trade-off between locality preservation and fragmentation control.

This strategy can be implemented via a segment tree or direct sliding window traversal (efficient for small window sizes). Both approaches iterate over vertices while tracking allocated blocks. In the segment tree approach, each tree node stores the maximum remaining space across a range of blocks. For each vertex, the system queries the rightmost block that can accommodate the vertex's edge list. If no suitable block exists within the window, a new block is created. The vertex is then inserted into the selected/created block, with subsequent updates to both the sliding window and segment tree. For small sliding windows, a direct traversal identifies the rightmost available block that meets the space requirements. We use a default window size of 8 and the direct window traversal method in our system to prioritize spatial locality.

We further enhance efficiency through multi-threading, assigning each thread a subset of vertices to process. Although this parallelism may result in a slight increase in intra-block fragmentation, it significantly reduces preprocessing time. Notably, because only vertex degrees are required for this strategy, it remains practical in memory-constrained environments.

5.2 Space Optimization

We implement vertex reordering in two key optimizations: degree field elimination and mini edge list optimization. The degree field elimination technique specifically targets vertices whose degrees exceed a configurable threshold δ_{deg} (classified as large vertices), while the mini edge list optimization handles only those with degrees at or below the threshold (designated as mini vertices). We use v'_i to indicate the i-th vertex after reordering, and id_v to represent the id of vertex v. An SSD-stored `v2id` table is hence generated to record the corresponding relation. This table does not need to reside in memory and is only useful during program initialization and termination. ACGraph operates on the reordered graph.

Degree Field Elimination. ACGraph removes the 4-byte degree overhead through virtual vertex insertion and vertex reordering. First, virtual vertices are created with their offsets pointing to fragmentation boundaries in blocks. Then, those virtual vertices and large vertices are sorted together by offset and reassigned a new id based on their sequence. For example, we generate a virtual vertex in Fig. 7 (filled with gray color), and set its offset to indicate the end of the last vertex in block 1. This virtual vertex gets a *id* of $i+1$ after reordering. The virtual vertices are distinguished by setting their offset's highest bit, and ACGraph provides `is_virtual(v)` method to filter them during traversal. In this way, original large vertices retain the $\mathtt{deg}(v'_i) = \mathtt{offset}(v'_{i+1}) - \mathtt{offset}(v'_i)$ invariant, while virtual vertices keep unreached to preserve algorithm correctness, as they are not visited. Because the number of blocks is typically much smaller than the number of vertices, the additional space introduced is far lower than that of the degree field.

Mini Edge List Optimization. It can be noticed that edge lists no more than 8-byte can be embedded directly within the index field of corresponding vertices, bringing no extra memory overhead, as what [54] does. However, this approach cannot be directly adapted to ACGraph, as the absence of the degree field (which is needed in [54]) prevents distinguishing whether the offset field encodes a true offset or an edge list. We address this challenge and extend the optimization through the just-mentioned vertex reordering technique. To be specific, these mini vertices are sorted in descending degree order and assigned consecutive *id*s that directly follow those of large vertices. Their edge lists are compactly arranged in a dedicated `mini_data` array, mirroring the pattern of the CSR format.

Then, to trace the degree and the offset of mini vertices, we construct an θ_{id} array of size $\delta_{deg}+1$:

$$\theta_{id}[deg] = \min_i\{i \mid \mathtt{deg}(v'_i) \le deg\}. \tag{3}$$

The degree and edge list offset of a mini vertex v'_i can be computed rather than explicit storage. The degree calculation employs:

$$\mathtt{deg}(v'_i) = \max_{0\le deg\le\delta_{deg}} \{deg \mid \theta_{id}[deg] \le i\},$$

While the offset in `data_mini` is computed via:

$$\mathtt{offset}(v'_i) = (i - \theta_{id}[\mathtt{deg}(v'_i)]) \cdot \mathtt{deg}(v'_i) + \sum_{i=deg(v'_i)+1}^{\delta_{deg}} (\theta_{id}[i-1] - \theta_{id}[i]) \cdot i,$$

Next, we include an example to illustrate how it works.

Example 5.1. Assume $\delta_{deg} = 3$ and there are 10 large vertices, 500 vertices of degree 3, 1000 of degree 2, and 2000 of degree 1. Then, $\theta_{id}[3] = 10, \theta_{id}[2] = 510, \theta_{id}[1] = 1510, \theta_{id}[0] = 3510$. Consider a vertex v'_{1200}, i.e., the reordered ID is 1200. After checking the $\theta_{id}[2]$, it is the maximum degree such that $\theta_{id}[2] \le 1200$. Hence, the degree of v'_{1200} is 2. Then, to derive the offset in the `data_mini` array, it is computed as $(510-10)\times 3 + (1200-510)\times 2$.

Normally, the degree threshold δ_{deg} is a constant value of 2 or 3. This architecture eliminates per-node degree storage overhead while maintaining $O(1)$ access complexity through algebraic computation. The δ_{deg} enables explicit trade-off between memory consumption and I/O overhead.

6 Experiments

In this section, we compare our solution ACGraph against two state-of-the-art (SOTA) out-of-core graph processing systems, Blaze [28] and CAVE [45]. We further compare against Galois [43], a popular in-memory asynchronous system, which highlights the performance-scalability trade-offs across in-memory and out-of-core approaches. We implement our ACGraph in C++ [1]. For a fair

Table 1. Summary of the tested datasets.

Dataset	Abbr.	$\lvert V\rvert$	$\lvert E\rvert$	Size	Type	T_p
Twitter [32]	TW	42M	1.5B	5.8GB	D	6.2
Friendster [60]	FR	66M	3.6B	15GB	U	12.4
UK-Union [11]	UK	133M	5.5B	22GB	D	21.4
ClueWeb12 [10]	CL	978M	43B	166GB	D	166.2

comparison, we use the open-source implementations of Blaze[2], CAVE[3] and Galois[4]. All code is compiled by g++ 11.5.0 with the -O3 flag. Additionally, the `O_DIRECT` flag is enabled to bypass the OS page cache, ensuring data is read directly from disk to userspace.

6.1 Experimental Settings

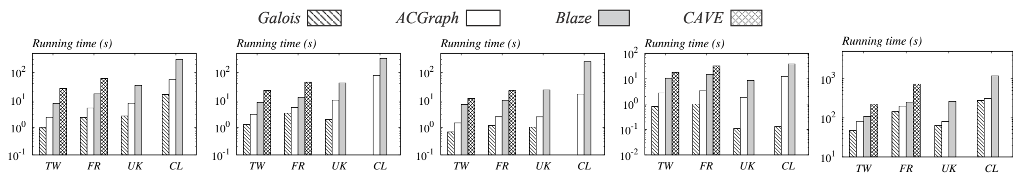
Graph Algorithms. We evaluate our system using five popular graph algorithms that admit asynchronous execution: breadth first search (BFS), weakly connected component (WCC), k-core, single source personalized PageRank (SSPPR), and PageRank (PR). For WCC, we adopt *label propagation (LP)* [51] algorithm, which is a common algorithm used by graph processing systems [28]. For SSPPR, we use the forward push (FP) algorithm [4, 23, 24], with probability $\alpha = 0.15$ and threshold $r_{\max} = 10^{-9}$. As a special case of personalized PageRank, PageRank set the initial distribution to $\{\frac{1}{n}, \frac{1}{n}, \cdots, \frac{1}{n}\}$, and then employs the same algorithm as SSPPR with $r_{\max} = 10^{-10}$. The parameter k of k-core algorithm is set to 10. To ensure a fair comparison, all systems use the same source vertices when running BFS and SSPPR queries. The cache pool size for ACGraph, CAVE, and Blaze is set to 256 MB by default. However, we observe that ACGraph becomes insensitive to cache pool size once it exceeds a modest threshold (e.g., 40 MB). For fairness, we maintain the same cache size across all methods in our comparisons.

Hardwares. All experiments are conducted on a Linux server running Ubuntu 20.04.6 LTS with kernel version 5.8.0. The system is equipped with 377GB of main memory and two Intel(R) Xeon(R) Gold 5218 CPUs @ 2.30GHz, each featuring 16 physical cores (32 threads with hyper-threading). The datasets are stored on a 1TB PCIe SSD device. To ensure consistency, we run each experiment three times, and report the average of the results [45].

Datasets. We evaluate our solution using four real-world graphs: Twitter, Friendster, UK-Union and ClueWeb12. These datasets are widely adopted in benchmarking graph processing frameworks, as seen in prior works [3, 18, 28, 46, 49, 67, 70]. Twitter and Friendster are social networks, while the others are web graphs. Their key characteristics are summarized in Table 1. The graph sizes are reported based on the *CSR (compressed sparse row)* format, where the offset array uses 8-byte unsigned integers and the edges array uses 4-byte unsigned integers. In the type column, 'U' denotes an undirected graph, and 'D' denotes a directed graph. To meet the requirement of algorithms that require undirected graph inputs (e.g., WCC and k-core), we preprocess the graphs accordingly by generating symmetrized edges. CAVE cannot process UK-Union and ClueWeb12 due to its use of 4-byte integers for offsets in the open-source implementation, so results on these datasets are omitted. In Table 1, T_p denotes the partitioning and reordering time of ACGraph for preprocessing, measured in *seconds*. On the largest dataset, ClueWeb12 (43B edges), it takes only 166 seconds.

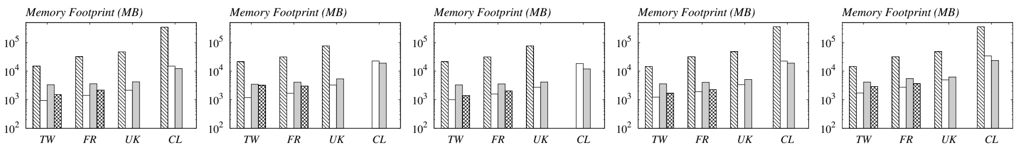
Metrics. We use a variety of metrics to measure the performance of the framework, including:

[2]https://github.com/NVSL/blaze
[3]https://github.com/BU-DiSC/CAVE
[4]https://github.com/IntelligentSoftwareSystems/Galois



(a) BFS (b) WCC (c) k-core (d) SSPPR (e) PageRank

Fig. 8. Running time of each graph processing system (lower is better).



(a) BFS (b) WCC (c) k-core (d) SSPPR (e) PageRank

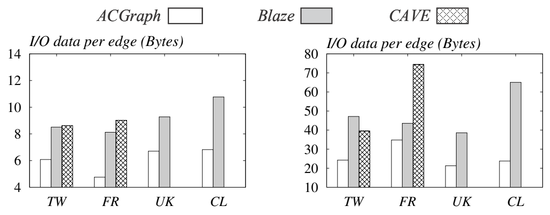
Fig. 9. Memory footprint of each graph processing systems.

- *End-to-End Runtime.* Total execution time is measured from program launch to algorithm completion, including system initialization, graph loading and building, vertex state initialization, initial frontier construction, and algorithm execution phases. For Galois, we report only the algorithm execution time, as it is an in-memory system and places less emphasis on graph loading.
- *Memory Footprint.* Peak memory consumption is measured by the Maximum Resident Set Size (max RSS) metric, which is obtained from the `/bin/time -v -p` command.
- *Disk I/O.* Disk read operations are quantified as $\Delta_{\text{I/O}} = \texttt{iostat}_{\text{end}} - \texttt{iostat}_{\text{start}}$, where $\texttt{iostat}_{\text{end}}$ and $\texttt{iostat}_{\text{start}}$ represent the pre-execution and post-execution measurements of cumulative disk I/O operations, respectively recorded via the `iostat` utility.

6.2 Overall Performance

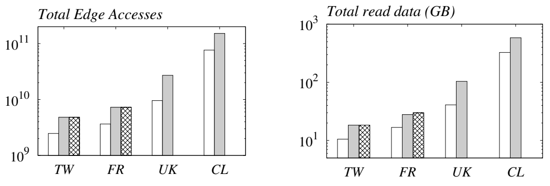
First, we test five graph algorithms (i.e., BFS, WCC, k-core, SSPPR and PR) that support asynchronous execution, to evaluate the performance of each GPS in terms of running time, as shown in Fig. 8. The reported running time of out-of-core GPSs includes initialization time (e.g., index loading), while the in-memory GPS does not include this time. As shown in Fig. 8, ACGraph consistently outperforms Blaze and CAVE across all tested datasets and algorithms. In particular, ACGraph achieves notable speedups over the second-fastest out-of-core system, Blaze: up to 5.50× (BFS), 4.26× (WCC), 15.21× (k-core), 4.67× (SSPPR) and 3.80× (PR). Compared to CAVE, ACGraph yields up to 12.34×, 10.75×, 8.80×, 9.62× and 3.68× improvement , respectively. Moreover, across the four algorithms excluding locally-focused SSPPR, ACGraph achieves higher speedup ratios on web graphs than on social networks. This stems from web graphs' larger diameters, which require more iterations to coverage and incur greater synchronization overhead. These significant improvements can be attributed to our asynchronous design, which effectively addresses the limitations discussed in Sec. 3. Galois consistently outperforms other frameworks: When executing BFS, WCC, k-core, SSPPR and PR, Galois achieves speedups of 2.20–3.51×, 1.42–2.45×, 2.10–2.38×, 3.41-95.72× and 1.13–1.72× over ACGraph respectively, benefiting from no I/O overhead. However, Galois is unable to run WCC and k-core on ClueWeb12, as these algorithms require undirected graph processing, which exceeds the machine's memory limits. The limitation highlights the low scalability of in-memory systems.

We also report the memory footprint of our ACGraph against its three competitors, as illustrated in Fig. 9. We observe that ACGraph consistently requires lower memory usage than others across all evaluated graph algorithms on the Twitter, Friendster, and UK-Union datasets. To explain, Blaze suffers from higher inherent memory overheads due to its implementation on top of the Galois



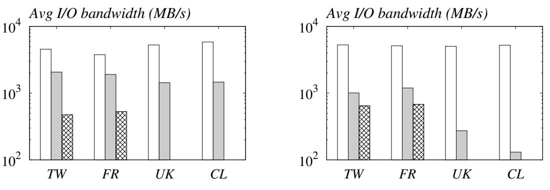
(a) BFS (b) SSPPR

Fig. 10. Read inflation (lower is better).



(a) Edge accesses (b) I/O read volume

Fig. 11. Work inflation (lower is better).



(a) BFS (b) k-core

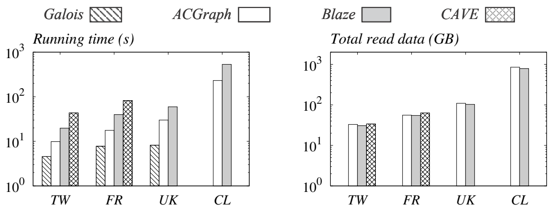
Fig. 12. I/O Throughput (higher is better).

framework [43], while CAVE maintains the entire frontier set in array format without bitmap optimization. On ClueWeb12, ACGraph consumes slightly more memory than Blaze, mainly because it maintains a 64-byte metadata for each block and allocates an 8-byte offset field for every vertex. Since ClueWeb12 contains over 978 million vertices, this leads to additional memory overhead, as observed. Considering its superior performance of achieving up to 15× speedup over Blaze, it demonstrates a favorable trade-off between computational efficiency and memory usage. Compared to Galois, ACGraph achieves a remarkable trade-off between performance and scalability. While incurring a performance penalty of 1–2× in most scenarios, it reduces memory cost by 7.49–27.95×.

6.3 I/O Efficiency

Read Inflation. We further examine how ACGraph addresses read inflation issues during BFS and SSPPR execution compared to the two out-of-core GPSs: Blaze and CAVE. To quantify the extent of read inflation, we measure the average I/O volume per edge access. Note that each edge is represented using 4 bytes. Since each edge is accessed exactly once in BFS, the theoretical minimum I/O volume per edge is 4 bytes. The experimental results in Fig. 10 show that ACGraph achieves superior I/O efficiency, requiring fewer than 7 bytes per edge on average. On Friendster, this improves to just 4.8 bytes per edge, approaching the theoretical minimum. In contrast, Blaze incurs at least 8.1 bytes per edge across all four datasets, while CAVE requires a minimum of 8.6 bytes per edge. For SSPPR, all frameworks experience increased read inflation. Nevertheless, ACGraph maintains a clear advantage, using only 37% of Blaze's I/O volume per edge on ClueWeb12.

Work Inflation. Fig. 11 illustrates the work inflation issue in WCC algorithms. As discussed in Sec. 3, the synchronous semantics of Blaze and CAVE lead to near-full graph loading in early iterations.



(a) Running Time (b) I/O Read Volume

Fig. 13. Synchronous execution: MIS.

In contrast, ACGraph converges faster with significantly fewer edge accesses by leveraging priority-based scheduling. On Twitter and Friendster, CAVE and Blaze process 1.95× and 1.99× more edges, respectively. For UK-Union and ClueWeb12, Blaze incurs 2.84x and 1.98x more edge accesses than ACGraph. Hence, ACGraph reduces I/O access volume by up to 60% compared to Blaze and 44% compared to CAVE, confirming its effectiveness in mitigating work inflation.

I/O Throughput. Fig. 12 shows the average bandwidth during k-core and BFS execution. ACGraph achieves a bandwidth of 4.5 GB/s and 3.7 GB/s for BFS on Twitter and Friendster respectively and over 5.0 GB/s in all other cases. These bandwidths approach the maximum 6.0 GB/s sequential access speed of our SSD hardware, confirming that ACGraph effectively saturates the storage bandwidth. In comparison, CAVE is limited to no more than 800 MB/s across all test cases due to inefficient single-threaded task collection and distribution. While Blaze achieves over 2.0 GB/s for BFS and over 1.0 GB/s for k-core on Twitter and Friendster, its performance degrades significantly on UK-Union and ClueWeb12, due to longer convergence on these two datasets. In particular, Blaze requires 12 – 24 iterations for BFS and k-core on Twitter and Friendster, while it takes 290 – 670 iterations on UK-Union and ClueWeb12.

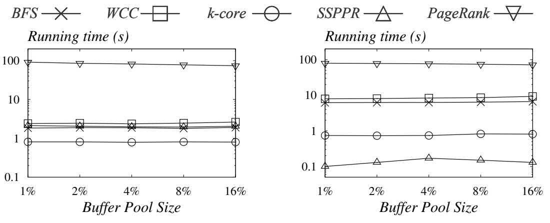
6.4 Synchronous Execution

As discussed in Sec. 4.3, some algorithms inherently require synchronous execution. To evaluate ACGraph's performance in this context, we use the Maximal Independent Set (MIS) problem as a case study. Specifically, we adopt Blelloch's Alg. 2 [9], ensuring determinism and comparability by fixing random seeds across all implementations. For this experiment, ACGraph is configured to use its synchronous execution interface. Note that Galois fails to execute the algorithm on the ClueWeb12 dataset due to the memory requirements exceeding our machine's capacity.

As shown in Fig. 13, across all graphs, the out-of-core systems show similar I/O volumes (the difference is within 16%), as they access the same number of edges. Thanks to its asynchronous I/O pipeline, which boosts edge loading and processing throughput, ACGraph achieved 2.0–2.3× speedup over Blaze. However, in synchronous mode, ACGraph is also subject to synchronization stalls. As a result, the runtime ratios among the out-of-core systems remain similar across all four datasets. This contrasts with the results in Sec. 6.2, where ACGraph achieves larger speedups on large-diameter web graphs, highlighting the benefits of asynchronous execution.

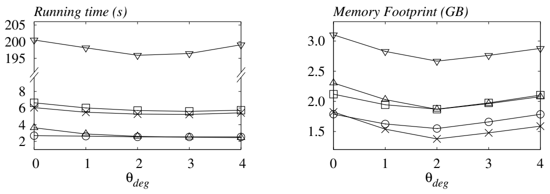
6.5 Tuning Parameters

Buffer Pool Size. We further examine the impact of the buffer pool size on the performance of ACGraph, by varying the pool size from 1% to 16% of the input graph size, as shown in Fig. 14. Our results demonstrate that ACGraph delivers stable and efficient performance across all configurations. This robustness stems from its effective graph locality preservation and efficient block reuse mechanism, which enable excellent performance even with minimal buffer allocation.



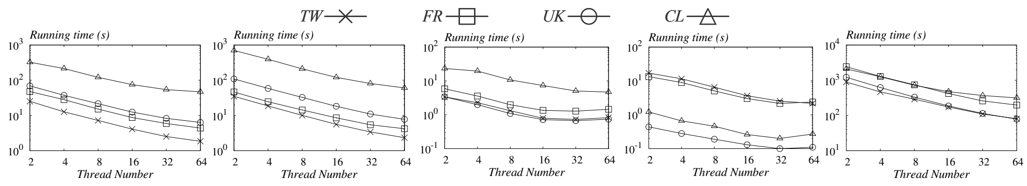
(a) Twitter (b) UK-Union

Fig. 14. Performance sensitivity to buffer pool size.



(a) Running Time (b) Memory Footprint

Fig. 15. Perf. sensitivity to degree threshold on FR.



(a) BFS (b) WCC (c) k-core (d) SSPPR (e) PageRank

Fig. 16. Algorithm execution time under varying numbers of threads.

Besides, we observe a marginal runtime increase at larger pool sizes, which is due to the initialization and maintenance overhead associated with expanded pool size.

Configurable Degree Threshold. Fig. 15 shows the performance of ACGraph in terms of runtime and memory cost on dataset Friendster, as a function of the degree threshold δ_{deg}. The default 64-byte metadata layout only supports the settings of $\delta_{deg} \geq 2$. For $\delta_{deg} = 1$, we increase the metadata size to 128 bytes, and for $\delta_{deg} = 0$, further to 196 bytes, ensuring cacheline alignment. Consequently, the space usage is minimized at $\delta_{deg} = 2$. This is because for $\delta_{deg} < 2$, excessive space is occupied by metadata, and for $\delta_{deg} > 2$, the mini edge list dominates the space overhead. Furthermore, Fig. 15 shows that while the execution time generally decreases with larger δ_{deg}, the reduction becomes marginal when δ_{deg} exceeds 2. Therefore, balancing time and space efficiency, we set $\delta_{deg} = 2$ by default.

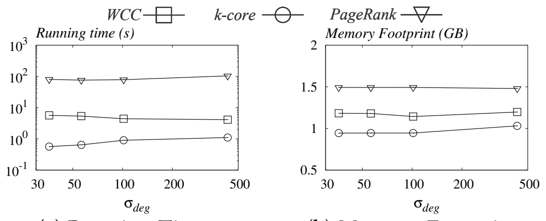
Thread Scaling. Fig. 16 presents the execution time of our ACGraph on four graph algorithms, with the number of worker threads varying from 2 to 64. Note that the time spent in the initialization phase, which is executed in a single thread, is excluded from the reported measurements. As shown, when the disk is not saturated, ACGraph demonstrates a near-linear performance improvement as the number of threads increases, achieving up to a 14.9×s speedup.

6.6 Ablation Study on the Partitioner

We conduct experiments to explain why we adopt the locality-preserving last-fit (LPLF) partitioner as the default. While partitioning is not the main focus of our framework, we seek a lightweight and robust solution. We show that LPLF is more I/O- and time-efficient than simpler alternatives

Table 2. Perf. ratio (BF/LPLF) based on I/O read volume and running time, with 1 indicating equal performance.

	BFS	**WCC**	***k*-core**	**SSPPR**	**PageRank**
UK-Union I/O	1.16	1.26	0.59	2.22	1.04
UK-Union Time	1.14	1.26	0.62	1.75	1.03
ClueWeb12 I/O	1.31	1.17	0.59	1.79	1.21
ClueWeb12 Time	1.28	1.32	0.73	1.75	1.02



(a) Running Time (b) Memory Footprint

Fig. 17. Perf. sensitivity to degree skewness, where σ_{deg} represents standard deviation of graph degrees.

such as the degree-sorted best-fit (BF) packing method, where vertices are processed in descending degree order and assigned to the tightest available block, creating new blocks when necessary. Table 2 reports the ratio of disk reads and runtime for BF relative to LPLF on the two large datasets (i.e., UK-Union and ClueWeb12). In the table, values greater than 1 indicate that BF performs worse. LPLF outperforms BF in 4 out of 5 algorithms, showing better robustness. BF performs better only for k-core due to better alignment with the algorithm's access pattern. For other algorithms, the loss of locality degrades performance. As general-purpose systems demand consistent performance across workloads, we adopt LPLF by default while allowing users to implement custom partitioners for specific use cases.

We further evaluate the impact of degree skewness on ACGraph using four synthetic graphs (all with 64M vertices, 1.5B edges) generated via R-MAT [12], with parameters adjusted to vary skewness. As shown in Fig. 17, the standard deviation of vertex degrees ranges from 30 to 500, resulting in skewness scores of 2.12, 3.68, 8.97, and 79.77. We report query times only for global asynchronous algorithms, as PPR and BFS are highly sensitive to source vertices and thus less generalizable. ACGraph shows stable performance across all graphs, underscoring the robustness of its preprocessing and overall design. Finally, ACGraph exhibits consistent preprocessing times ($\approx$10 seconds) across all graphs due to their identical sizes.

7 Related Work

Beyond the out-of-core GPSs discussed in Sec. 2.2, prior work falls into two main categories: in-memory GPSs and distributed GPSs.

In-memory GPSs. Ligra [49] introduces a lightweight vertex-centric API for multicore machines, switching automatically between push and pull traversals to balance work and minimize overhead. GPOP [34] adopts a partition-centric model that partitions the graph into fine-grained blocks, reducing synchronization and improving cache utilization. Corograph [67] employs a hybrid execution model to achieve both cache efficiency and work efficiency for batched query processing, which differs from our setting. ForkGraph [39] partitions the graph into cache-sized blocks and processes query batches concurrently, boosting cache hit rates and overall throughput via carefully designed scheduling strategies. GraphLab [37] pioneers an asynchronous, shared-memory, graph-parallel framework for iterative machine learning algorithms. Galois [43] is a popular in-memory asynchronous GPS that provides an optimistic execution engine with fine-grained parallelism, priority scheduling and work stealing, allowing developers to express algorithms at a high level. The design philosophies of these in-memory asynchronous GPSs, such as the vertex-centric paradigm, have influenced subsequent GPSs. However, as noted in Sec. 1, they require substantial redesign to perform well in out-of-core environments.

Distributed GPSs. Distributed GraphLab [38] scales the GraphLab paradigm to clusters for efficient large graph processing on commodity clouds. Pregel [41] popularized the Bulk Synchronous Parallel (BSP) vertex-centric model via message passing and global barriers to process large graphs. PowerGraph [19] extends it with the Gather–Apply–Scatter (GAS) abstraction and vertex-cut partitioning to handle power-law graphs. GraphX [20] builds on Spark RDDs to unify graph and dataflow operations with fault tolerance and query optimization. Gemini [69] introduces multiple computation-centric optimizations to improve scalability. TurboGraph++ [31] employs three-level parallelism and overlapping to maximize the CPU, disk, and network utilization in clusters.

Additionally, extensive work has explored graph processing on emerging hardware such as GPUs, with frameworks like BTS [29], Lux [25], Scaph [66], CGraph [16], TGraph [64], and Nezha [15]. Some out-of-core GPSs (e.g., [28, 45, 54]) introduce custom storage formats to reduce space usage, but as discussed in Sec. 5, these approaches have limitations, motivating our hybrid design.

8 Conclusions

Motivated by the I/O inefficiencies in existing out-of-core GPSs, we propose ACGraph, a novel asynchronous GPS for SSDs built on our proposed block-centric execution model. It features a hybrid storage architecture optimized for space and computation. Extensive evaluation on real graphs with tens of billions of edges shows that ACGraph significantly outperforms existing SSD-based GPSs.

Acknowledgments

This work was supported by the Hong Kong RGC GRF grant (No. 14217322) and the 1+1+1 CUHK-CUHK(SZ)-GDSTC Joint Collaboration Fund (No. 2025A0505000045).

References

[1] 2025. https://github.com/CUHK-DBGroup/ACGraph.

[2] Nitin Agrawal, Vijayan Prabhakaran, Ted Wobber, John Davis, Mark Manasse, and Rina Panigrahy. 2008. Design Tradeoffs for SSD Performance. In *ATC*. 57–70.

[3] Zhiyuan Ai, Mingxing Zhang, Yongwei Wu, Xuehai Qian, Kang Chen, and Weimin Zheng. 2017. Squeezing out All the Value of Loaded Data: An Out-of-core Graph Processing System with Reduced Disk I/O. In *ATC*. 125–137.

[4] Reid Andersen, Christian Borgs, Jennifer Chayes, John Hopcraft, Vahab S Mirrokni, and Shang-Hua Teng. 2007. Local computation of pagerank contributions. In *WAW*. 150–165.

[5] Jens Axboe. 2019. Efficient IO with io_uring.

[6] Jens Axboe and Alan D Brunelle. 2007. Blktrace User Guide.

[7] Laszlo A. Belady. 1966. A Study of Replacement Algorithms for Virtual-Storage Computer. *IBM Syst. J.* (1966), 78–101.

[8] Dimitri P. Bertsekas and John N. Tsitsiklis. 1997. *Parallel and Distributed Computation: Numerical Methods.*

[9] Guy E. Blelloch, Jeremy T. Fineman, and Julian Shun. 2012. Greedy sequential maximal independent set and matching are parallel on average. In *SPAA*. 308–317.

[10] Paolo Boldi, Andrea Marino, Massimo Santini, and Sebastiano Vigna. 2018. BUbiNG: Massive Crawling for the Masses. *ACM Trans. Web* 12, 2 (2018), 12:1–12:26.

[11] Paolo Boldi, Massimo Santini, and Sebastiano Vigna. 2008. A Large Time-Aware Graph. *SIGIR Forum* 42, 2 (2008), 33–38.

[12] Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. 2004. R-MAT: A Recursive Model for Graph Mining. In *SIAM*. 442–446.

[13] Shu-han Cheng, Guangyan Zhang, Jiwu Shu, and Weimin Zheng. 2016. AsyncStripe: I/O efficient asynchronous graph computing on a single server. In *CODES*. 32:1–32:10.

[14] Thomas H. Cormen and Michael T. Goodrich. 1996. A Bridging Model for Parallel Computation, Communication, and I/O. *ACM Comput. Surv.* 28, 4es (1996), 208.

[15] Pengjie Cui, Haotian Liu, Dong Jiang, Bo Tang, and Ye Yuan. 2025. Nezha: An Efficient Distributed Graph Processing System on Heterogeneous Hardware. *Proc. ACM Manag. Data* (2025), 57:1–57:27.

[16] Pengjie Cui, Haotian Liu, Bo Tang, and Ye Yuan. 2024. CGgraph: An Ultra-fast Graph Processing System on Modern Commodity CPU-GPU Co-processor. *Proc. VLDB Endow.* (2024), 1405–1417.

[17] Wenfei Fan, Ping Lu, Xiaojian Luo, Jingbo Xu, Qiang Yin, Wenyuan Yu, and Ruiqi Xu. 2018. Adaptive Asynchronous Parallelization of Graph Algorithms. In *SIGMOD*. 1141–1156.

[18] Gurbinder Gill, Roshan Dathathri, Loc Hoang, Ramesh Peri, and Keshav Pingali. 2020. Single Machine Graph Analytics on Massive Datasets Using Intel Optane DC Persistent Memory. *Proc. VLDB Endow.* (2020), 1304–1318.

[19] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. 2012. PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs. In *OSDI*. 17–30.

[20] Joseph E. Gonzalez, Reynold S. Xin, Ankur Dave, Daniel Crankshaw, Michael J. Franklin, and Ion Stoica. 2014. GraphX: Graph Processing in a Distributed Dataflow Framework. In *OSDI*. 599–613.

[21] Gabriel Haas and Viktor Leis. 2023. What Modern NVMe Storage Can Do, And How To Exploit It: High-Performance I/O for High-Performance Storage Engines. *Proc. VLDB Endow.* (2023), 2090–2102.

[22] William Hamilton, Rex Ying, and Jure Leskovec. 2017. Inductive Representation Learning on Large Graphs. In *NIPS*.

[23] Guanhao Hou, Xingguang Chen, Sibo Wang, and Zhewei Wei. 2021. Massively Parallel Algorithms for Personalized PageRank. *Proc. VLDB Endow.* 14, 9 (2021), 1668–1680.

[24] Guanhao Hou, Qintian Guo, Fangyuan Zhang, Sibo Wang, and Zhewei Wei. 2023. Personalized PageRank on Evolving Graphs with an Incremental Index-Update Scheme. *Proc. ACM Manag. Data* 1, 1 (2023), 25:1–25:26.

[25] Zhihao Jia, Yongkee Kwon, Galen M. Shipman, Patrick S. McCormick, Mattan Erez, and Alex Aiken. 2017. A Distributed Multi-GPU System for Fast Graph Processing. *Proc. VLDB Endow.* (2017), 297–310.

[26] Yuhun Jun, Shin-Hyun Park, Jeong-Uk Kang, Sang-Hoon Kim, and Euiseong Seo. 2024. We Ain't Afraid of No File Fragmentation: Causes and Prevention of Its Performance Impact on Modern Flash SSDs. In *FAST*. 193–208.

[27] Jeong-Uk Kang, Jinsoo Kim, Chanik Park, Hyoungjun Park, and Joonwon Lee. 2007. A multi-channel architecture for high-performance NAND flash-based storage system. *J. Syst. Archit.* (2007), 644–658.

[28] Juno Kim and Steven Swanson. 2022. Blaze: Fast Graph Processing on Fast SSDs. In *SC*. 44:1–44:15.

[29] Min-Soo Kim, Kyuhyeon An, Himchan Park, Hyunseok Seo, and Jinwook Kim. 2016. GTS: A Fast and Scalable Graph Processing Method based on Streaming Topology to GPUs. In *SIGMOD*. 447–461.

[30] Thomas N. Kipf and Max Welling. 2017. Semi-Supervised Classification with Graph Convolutional Networks. In *ICLR*.

[31] Seongyun Ko and Wook-Shin Han. 2018. TurboGraph++: A Scalable and Fast Graph Analytics System. In *SIGMOD*. 395–410.

[32] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. 2010. What is Twitter, a social network or a news media?. In *WWW*. 591–600.

[33] Aapo Kyrola, Guy E. Blelloch, and Carlos Guestrin. 2012. GraphChi: Large-Scale Graph Computation on Just a PC. In *OSDI*. 31–46.

[34] Kartik Lakhotia, Rajgopal Kannan, Sourav Pati, and Viktor K. Prasanna. 2020. GPOP: A Scalable Cache- and Memory-efficient Framework for Graph Processing over Parts. *ACM Trans. Parallel Comput.* (2020), 7:1–7:24.

[35] Jure Leskovec, Jon Kleinberg, and Christos Faloutsos. 2007. Graph evolution: Densification and shrinking diameters. *TKDD* 1, 1 (2007), 2.

[36] Hang Liu and H. Howie Huang. 2017. Graphene: Fine-Grained IO Management for Graph Computing. In *FAST*. 285–300.

[37] Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph M. Hellerstein. 2010. GraphLab: A New Framework For Parallel Machine Learning. In *UAI*. 340–349.

[38] Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph M. Hellerstein. 2012. Distributed GraphLab: A Framework for Machine Learning in the Cloud. *Proc. VLDB Endow.* (2012), 716–727.

[39] Shengliang Lu, Shixuan Sun, Johns Paul, Yuchen Li, and Bingsheng He. 2021. Cache-Efficient Fork-Processing Patterns on Large Graphs. In *SIGMOD*. 1208–1221.

[40] Steffen Maass, Changwoo Min, Sanidhya Kashyap, Woon-Hak Kang, Mohan Kumar, and Taesoo Kim. 2017. Mosaic: Processing a Trillion-Edge Graph on a Single Machine. In *EuroSys*. 527–543.

[41] Grzegorz Malewicz, Matthew H. Austern, Aart J. C. Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: a system for large-scale graph processing. In *SIGMOD*. 135–146.

[42] M. E. J. Newman. 2003. The structure and function of complex networks. *SIAM Rev.* 45, 2 (2003), 167–256.

[43] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. 2013. A lightweight infrastructure for graph analytics. In *SOSP*. 456–471.

[44] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. 1999. *The PageRank citation ranking: Bringing order to the web.* Technical Report. Stanford infolab.

[45] Tarikul Islam Papon, Taishan Chen, Shuo Zhang, and Manos Athanassoulis. 2024. CAVE: Concurrency-Aware Graph Processing on SSDs. *Proc. ACM Manag. Data* (2024), 125.

[46] Ha-Myung Park, Sung-Hyon Myaeng, and U Kang. 2016. PTE: Enumerating Trillion Triangles On Distributed Systems. In *SIGKDD*. 1115–1124.

[47] Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. 2013. X-Stream: edge-centric graph processing using streaming partitions. In *SOSP*. 472–488.

[48] Xuanhua Shi, Zhigao Zheng, Yongluan Zhou, Hai Jin, Ligang He, Bo Liu, and Qiang-Sheng Hua. 2018. Graph Processing on GPUs: A Survey. *ACM Comput. Surv.* (2018), 81:1–81:35.

[49] Julian Shun and Guy E. Blelloch. 2013. Ligra: a lightweight graph processing framework for shared memory. In *PPoPP*. 135–146.

[50] Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne. 2018. *Operating System Concepts, 10th Edition*. Wiley.

[51] Stergios Stergiou, Dipen Rughwani, and Kostas Tsioutsiouliklis. 2018. Shortcutting Label Propagation for Distributed Connected Components. In *WSDM*. 540–546.

[52] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, and Yoshua Bengio. 2018. Graph Attention Networks. In *ICLR*.

[53] Keval Vora. 2019. LUMOS: Dependency-Driven Disk-based Graph Processing. In *ATC*. 429–442.

[54] Rui Wang, Weixu Zong, Shuibing He, Xinyu Chen, Zhenxin Li, and Zheng Dang. 2024. Efficient Large Graph Processing with Chunk-Based Graph Representation Model. In *ATC*. 1239–1255.

[55] Sibo Wang, Renchi Yang, Runhui Wang, Xiaokui Xiao, Zhewei Wei, Wenqing Lin, Yin Yang, and Nan Tang. 2019. Efficient Algorithms for Approximate Single-Source Personalized PageRank Queries. *ACM Trans. Database Syst.* 44, 4 (2019), 18:1–18:37.

[56] Sibo Wang, Renchi Yang, Xiaokui Xiao, Zhewei Wei, and Yin Yang. 2017. FORA: Simple and Effective Approximate Single-Source Personalized PageRank. In *SIGKDD*. 505–514.

[57] Duncan J Watts and Steven H Strogatz. 1998. Collective dynamics of 'small-world' networks. *Nature* 393, 6684 (1998), 440–442.

[58] Chenning Xie, Rong Chen, Haibing Guan, Binyu Zang, and Haibo Chen. 2015. SYNC or ASYNC: time to fuse for distributed graph-parallel computation. In *PPoPP*. 194–204.

[59] Xifeng Yan and Jiawei Han. 2002. gSpan: Graph-Based Substructure Pattern Mining. In *ICDM*. 721–724.

[60] Jaewon Yang and Jure Leskovec. 2015. Defining and evaluating network communities based on ground-truth. *Knowl. Inf. Syst.* (2015), 181–213.

[61] Tsun-Yu Yang, Yizou Chen, Yuhong Liang, and Ming-Chang Yang. 2025. Leveraging On-demand Processing to Co-optimize Scalability and Efficiency for Fully-external Graph Computation. *ACM Trans. Storage* (2025), 11:1–11:31.

[62] Tsun-Yu Yang, Cale England, Yi Li, Bingzhe Li, and Ming-Chang Yang. 2024. Grafu: Unleashing the Full Potential of Future Value Computation for Out-of-core Synchronous Graph Processing. In *ASPLOS*. 467–481.

[63] Mingxing Zhang, Yongwei Wu, Youwei Zhuo, Xuehai Qian, Chengying Huan, and Kang Chen. 2018. Wonderland: A Novel Abstraction-Based Out-Of-Core Graph Processing System. In *ASPLOS*. 608–621.

[64] Yongliang Zhang, Yuanyuan Zhu, Hao Zhang, Congli Gao, Yuyang Wang, Guojing Li, Tianyang Xu, Ming Zhong, Jiawei Jiang, Tieyun Qian, Chenyi Zhang, and Jeffrey Xu Yu. 2025. TGraph: A Tensor-centric Graph Processing Framework. *Proc. ACM Manag. Data* (2025), 81:1–81:27.

[65] Da Zheng, Disa Mhembere, Randal C. Burns, Joshua T. Vogelstein, Carey E. Priebe, and Alexander S. Szalay. 2015. FlashGraph: Processing Billion-Node Graphs on an Array of Commodity SSDs. In *FAST*. 45–58.

[66] Long Zheng, Xianliang Li, Yaohui Zheng, Yu Huang, Xiaofei Liao, Hai Jin, Jingling Xue, Zhiyuan Shao, and Qiang-Sheng Hua. 2020. Scaph: Scalable GPU-Accelerated Graph Processing with Value-Driven Differential Scheduling. In *ATC*. 573–588.

[67] Xiangyu Zhi, Xiao Yan, Bo Tang, Ziyao Yin, Yanchao Zhu, and Minqi Zhou. 2023. CoroGraph: Bridging Cache Efficiency and Work Efficiency for Graph Algorithm Execution. *Proc. VLDB Endow.* (2023), 891–903.

[68] Andy Diwen Zhu, Xiaokui Xiao, Sibo Wang, and Wenqing Lin. 2013. Efficient single-source shortest path and distance queries on large graphs. In *SIGKDD*. 998–1006.

[69] Xiaowei Zhu, Wenguang Chen, Weimin Zheng, and Xiaosong Ma. 2016. Gemini: A Computation-Centric Distributed Graph Processing System. In *OSDI*. 301–316.

[70] Xiaowei Zhu, Wentao Han, and Wenguang Chen. 2015. GridGraph: Large-Scale Graph Processing on a Single Machine Using 2-Level Hierarchical Partitioning. In *ATC*. 375–386.

[71] Xiaoke Zhu, Yang Liu, Shuhao Liu, and Wenfei Fan. 2023. MiniGraph: Querying Big Graphs with a Single Machine. *Proc. VLDB Endow.* (2023), 2172–2185.

Received April 2025; revised July 2025; accepted August 2025