CAVE: Concurrency-Aware Graph Processing on SSDs

TARIKUL ISLAM PAPON*, Boston University, USA
TAISHAN CHEN*, Boston University, USA
SHUO ZHANG, Columbia University, USA
MANOS ATHANASSOULIS, Boston University, USA

Large-scale graph analytics has become increasingly common in areas like social networks, physical sciences, transportation networks, and recommendation systems. Since many such practical graphs *do not fit in main memory*, graph analytics performance depends on efficiently utilizing underlying storage devices. These *out-of-core* graph processing systems employ sharding and sub-graph partitioning to optimize for storage while relying on efficient sequential access of traditional hard disks. However, today's storage is increasingly based on solid-state drives (SSDs) that exhibit *high internal parallelism* and *efficient random accesses*. Yet, state-of-the-art graph processing systems do not *explicitly exploit* those properties, resulting in subpar performance.

In this paper, we develop CAVE, the first graph processing engine that optimally exploits underlying SSD-based storage by harnessing the available storage device parallelism via carefully selecting graph I/Os that can be issued concurrently. Thus, CAVE traverses multiple paths and processes multiple nodes and edges concurrently, achieving parallelization at a granular level. We identify two key ways to parallelize graph traversal algorithms based on the graph structure and algorithm: intra and inter-subgraph parallelization. The first identifies subgraphs that contain vertices that can be accessed in parallel, while the latter identifies subgraphs that can be processed in their entirety in parallel. To showcase the benefit of our approach, we build within CAVE parallelized versions of five popular graph algorithms (Breadth-First Search, Depth-First Search, Weakly Connected Components, PageRank, Random Walk) that exploit the full bandwidth of the underlying device. CAVE uses a blocked file format based on adjacency lists and employs a concurrent cache pool that is essential to the parallelization of graph algorithms. By experimenting with different types of graphs on three SSD devices, we demonstrate that CAVE utilizes the available parallelism, and scales to diverse real-world graph datasets. CAVE achieves up to one order of magnitude speedup compared to the popular out-of-core systems Mosaic and GridGraph, and up to three orders of magnitude speedup in runtime compared to GraphChi.

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

ACM Reference Format:
Tarikul Islam Papon, Taishan Chen, Shuo Zhang, and Manos Athanassoulis. 2024. CAVE: Concurrency-Aware Graph Processing on SSDs. *Proc. ACM Manag. Data* 2, 3 (SIGMOD), Article 125 (June 2024), 26 pages. https://doi.org/10.1145/3654928

1 INTRODUCTION

The Rise of Large Graphs. Graphs are *natural encoders* of interconnected relations that can be leveraged to analyze many real-world applications. With the unprecedented growth of such

*Both authors contributed equally to this research.

Authors' addresses: Tarikul Islam Papon, papon@bu.edu, Boston University, Boston, MA, USA; Taishan Chen, utallow@bu.edu, Boston University, Boston, MA, USA; Shuo Zhang, sz3177@columbia.edu, Columbia University, New York City, NY, USA; Manos Athanassoulis, mathan@bu.edu, Boston University, Boston, MA, USA.

ACM 2836-6573/2024/6-ART125
https://doi.org/10.1145/3654928

interconnected data stemming from various applications like machine learning [29], recommendation systems [54], physical sciences [56], and social networks [59], analytics over large graphs is becoming increasingly popular in both academia and industry [2, 19, 27, 31, 42]. Real-world graphs often exhibit a vast scale, frequently encompassing millions, or even billions, of nodes interconnected by several billion edges. The sheer size of these graphs often exceeds the capacity of main memory, posing a significant challenge for efficient processing. Consequently, specialized techniques have emerged to address the need for scalable solutions to handle these massive graphs.

State-of-the-art Graph Management Systems. Many scalable systems have been recently proposed that handle large graphs by *distributed processing* [14, 21, 29, 31, 44, 61], which come with unique challenges such as partitioning, load balancing, cluster management, network overhead, and fault tolerance. On the other hand, *single-node* systems process large graphs in-memory [50, 51, 55, 58] and achieve scalability through increasing memory size and adding more CPUs. This work is orthogonal to the aforementioned approaches, however, it can benefit any system that spills data into storage. For example, our techniques can be applied at the local shard level in distributed graph management systems to enhance performance. *Single-node out-of-core* systems (which we focus on) primarily rely on (i) optimizing data partitioning techniques, (ii) improving memory and disk locality, and (iii) reducing random I/O to utilize fast sequential I/Os [16, 24, 30, 45, 62]. These techniques mainly address *slow random disk access*, which is particularly relevant for traditional hard disk drives (HDDs). However, the storage layer of data-intensive systems today employs solid-state disks (SSDs) and non-volatile memory (NVM) devices that have quite different characteristics than HDDs, which require a careful system redesign to be effectively exploited [36–38].

Modern Storage Devices. SSDs dominate as secondary storage devices, while classical HDDs are nowadays primarily used for archival storage [47]. SSDs offer fast data access, high chip density, and low energy consumption by utilizing NAND flash memory as their storage medium [3, 20, 40], thus eliminating the mechanical overheads of HDDs. Further, SSD internals follows a hierarchical structure (discussed in §2.1) that creates high *internal parallelism*, which can be leveraged to enhance performance [8, 9, 32, 37, 39, 48]. That is, an SSD can perform multiple *concurrent I/Os* until its bandwidth is saturated. Following the Parametric I/O model [37], we call this property **concurrency**, k, which is the number of I/Os the device can perform concurrently without hurting latency per request. The level of concurrency supported by a device depends on the request type (read/write), access granularity and on the device internals.

SSD Parallelism for Graph Processing. Graph traversal operations can utilize SSD concurrency by parallelizing node and edge accesses, effectively distributing the workload across SSD's parallel architecture [5]. This idea takes advantage of the availability of multiple paths that can be explored during graph traversal. However, most out-of-core graph processing systems simply attempt to better utilize underlying storage devices by reducing random (in favor of sequential) I/O. They do not aim to aggressively exploit opportunities for concurrent accesses, thus failing to use the full potential of SSDs. Our goal is to parallelize graph traversal algorithms without changing their core properties in order to fully utilize the underlying SSD concurrency. We identify two fundamental approaches to achieve this goal, each tailored to specific scenarios.

- **Intra-Subgraph Parallelization:** This approach focuses on parallelizing operations within a single subgraph. This approach is effective when the nodes of a subgraph can be processed independently. For example, a parallel version of Breadth-First Search (BFS) can follow this approach since multiple nodes of the same level can be processed independently. The core integrity of the algorithm can be maintained via communication among the processing units, result aggregation and synchronization. This approach harnesses the inherent parallelism present in subgraphs and utilizes modern storage concurrency for faster and more efficient graph traversal.

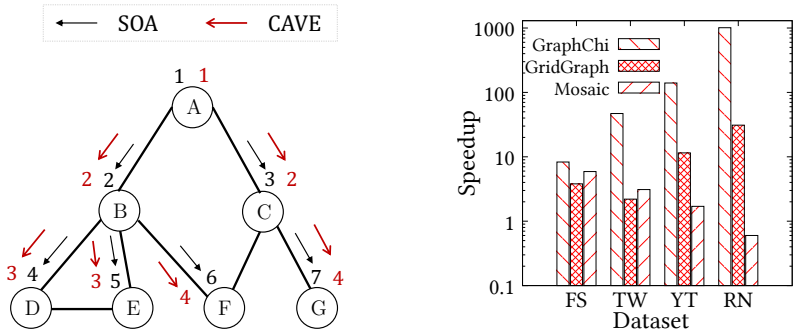


Fig. 1. (A) Parallellized version of BFS in CAVE takes fewer iterations to converge. (B) CAVE is upto three orders of magnitude faster than GraphChi and up to one order of magnitude faster than GridGraph and Mosaic.

- **Inter-Subgraph Parallelization:** In contrast to the previous approach, inter-subgraph parallelization involves processing multiple subgraphs concurrently. This method is particularly useful when we can identify that multiple subgraphs can be processed independently. For example, in the pseudo Depth-First Search algorithm [1], the stack used for traversal can be split into smaller stacks and processed in parallel by different threads. Multiple threads can then work on different parts of the graph concurrently, thus traversing multiple branches simultaneously.

In both approaches, the key objective is to maximize the utilization of SSD concurrency, ensuring that multiple operations can be performed in parallel. We integrate both approaches into a prototype graph processing system as discussed next.

Our Approach. We build an SSD-aware graph processing system, named CAVE[1] that is able to harness the *concurrency* of the underlying storage devices via *intra/inter-subgraph parallelization*. Specifically, CAVE provides the necessary infrastructure to parallelize graph traversal algorithms when several independent vertex accesses can be performed in parallel. A prime example is our Parallel Breadth-First Search (PBFS) implementation that uses *intra-subgraph* parallelization, which is outlined in Figure 1(A). The algorithm accesses the next *wave* of nodes (as we move on a level-by-level fashion) in parallel since we have already identified the nodes of the next wave while processing the current one. Figure 1(A) is a high-level overview where we consider a device with read concurrency 2. Hence, while vertex D, E, F and G are at the same level, only two of them can be processed in parallel. This leads to a faster response time of the BFS search simply by carefully exploiting the underlying storage concurrency, resulting in faster convergence within fewer iterations. CAVE uses a block-based file format based on adjacency lists, ensuring that graph metadata, vertex information, and edge information are stored in aligned blocks while enabling efficient support for graph traversal and analytical operations by ensuring optimized data retrieval. Furthermore, CAVE employs a concurrent cache pool mechanism that enhances locality and ensures thread safety. Overall, CAVE identifies storage accesses that are independent (thus can be parallelized) based on the task at hand and performs them concurrently based on the device's *optimal concurrency* [37], i.e., the number of I/O requests the device can handle without compromising latency.

To our best knowledge, CAVE is the first graph processing system that is capable of fully exploiting the available parallelism of the underlying flash-based storage leading to significant performance improvements. State-of-the-art graph processing systems focus on the design of graph processing/traversal algorithms and the distribution of the work (e.g., partitioning), but not on the specific characteristics of the underlying hardware and especially storage devices. By building a better

[1] **CAVE**: **C**oncurrency-**A**ware Graph (**V**, **E**) system

understanding of how to efficiently use SSDs, we build a faster (and/or cheaper in the cloud) graph processing system. Further, one of the key benefits of this approach is that it is applicable in any graph system that spills data on disk, so it can benefit a wide variety of systems. CAVE's architecture is designed to pave the way for developing new parallel graph algorithms that leverage the inherent concurrency of SSDs for both intra/inter-subgraph parallelization. As our first step, we develop in CAVE the parallelized versions of five popular graph algorithms. In addition to Breadth-First Search (BFS), CAVE offers parallelized, SSD-aware versions of Depth-First Search (DFS), Weakly Connected Components (WCC), PageRank (PR), and Random Walk (RW). We compare the performance of CAVE with three popular out-of-core processing systems, GraphChi [24], GridGraph [62] and Mosaic [30], as they are widely recognized for their efficiency in handling large-scale graphs in a single machine. Figure 1(B) shows the speedup of CAVE's BFS compared to these systems for four datasets (Friendster, Twitter, YouTube, and RoadNet) running on top of our PCIe SSD (details in §6). We observe that CAVE can be up to three orders of magnitude faster than GraphChi and up to one order of magnitude faster than GridGraph and Mosaic.

Contributions. Our contributions are as follows:

- We identify the importance of *SSD concurrency* with respect to graph processing.
- We identify two fundamental ways to parallelize graph traversal operations: *intra-subgraph* and *inter-subgraph parallelization*.
- We propose CAVE, the first SSD-aware graph engine that *fully exploits the parallelism of the underlying SSD storage* via concurrent I/O, its novel file structure, and a concurrent cache pool.
- We develop on CAVE the parallelized version of five popular graph algorithms (BFS, DFS, WCC, PageRank, Random Walk) to showcase that CAVE is flexible enough to implement diverse graph traversal algorithms.
- We evaluate CAVE against GraphChi, GridGraph and Mosaic where CAVE achieves up to 984× speedup vs. GraphChi, up to 22× speedup vs. GridGraph and up to 15× speedup vs. Mosaic.

2 BACKGROUND

In this section, we provide the necessary background for SSD *concurrency* and an overview of the algorithms we parallelize.

2.1 SSD Concurrency

Flash-based SSDs exhibit inherent internal parallelism due to their architectural design [3, 8, 37]. This parallelism stems from several factors, including the presence of multiple flash memory chips within the SSD, each capable of performing read and write operations independently. Further, each memory chip includes multiple dies, each die has multiple planes, and each plane consists of blocks where pages reside. Figure 2 shows the hierarchical architecture of a flash SSD. When multiple I/Os are issued in parallel, the flash controller tries to distribute them across different segments of the device [32, 41, 48], effectively increasing throughput without hurting latency (up to a point). The degree of *observed concurrency* varies across devices, and it also depends on the access type (read/write) [37]. The *optimal concurrency* of the device is the number of I/Os needed to saturate the device bandwidth without

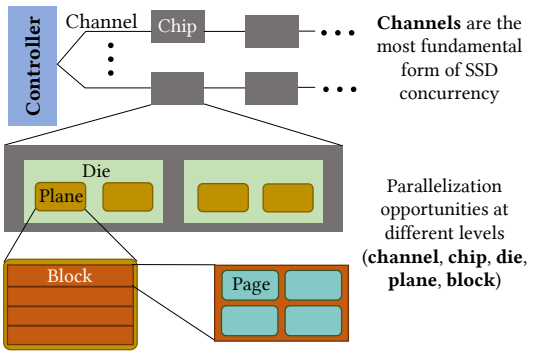


Fig. 2. Internal architecture of an SSD.

hurting latency. Additionally, modern SSD controllers exploit and manage internal parallelism for wear-leveling and garbage collection [18, 34]. To sum up, to better exploit SSDs, we need to issue concurrent I/Os while respecting the device's characteristics.

2.2 Graph Traversal Algorithms

We now introduce the necessary background for the graph traversal algorithms we use and discuss the opportunities for parallelization.

Breadth-First Search (BFS). BFS is a graph traversal algorithm that starts from a designated starting vertex and then explores all neighboring vertices in a level-by-level manner [11]. It begins by visiting all the immediate neighbors of the starting vertex and then moves on to their neighbors in subsequent levels. By traversing the graph in a level-wise manner, BFS uncovers the shortest paths and analyzes the structural properties of the graph.

Since BFS processes nodes in a level-by-level manner, nodes of the same level can be processed independently (hence concurrently), thus providing an opportunity for parallelizing and, in turn, harnessing the SSD's concurrency.

Depth-First Search (DFS). DFS is a widely-used graph traversal algorithm that starts from a specified vertex and systematically explores as deep as possible along each branch before backtracking [13]. This approach involves visiting a vertex and then recursively visiting its unvisited neighbors until there are no more unvisited vertices. DFS is particularly useful for identifying cycles, determining connected components, and finding paths between vertices.

While the classical DFS is tricky to parallelize, the pseudo-DFS [1] algorithm offers the opportunity to parallelize by running multiple parallel mini-DFSs. A parallel version of pseudo-DFS can dynamically split and distribute the vertex stack among multiple threads, allowing concurrent exploration of different branches of the graph.

Weakly Connected Components (WCC). In an undirected graph, a connected component refers to a subgraph where every vertex is connected to every other vertex through pathways within the graph. WCC aims to identify and group together nodes that are weakly connected [22], meaning they can be reached from each other by traversing the edges regardless of their direction. This algorithm typically involves traversing the graph using techniques like BFS or DFS to identify the connected components.

The previous approaches used to exploit SSD concurrency can be used to parallelize WCC. For example, while using BFS to discover WCCs, each subgraph's connected components can be computed concurrently, and the results from different subgraphs can be merged to determine the weakly connected components.

PageRank (PR). PR is a well-known algorithm to estimate the importance of vertices in graphs, used by Google to rank webpages on the Internet [7]. It works by evaluating the importance of a web page based on the number and quality of links pointing to it. The algorithm assigns a numerical value, known as PR score, to each web page on the Internet and measures the importance of a web page based on its backlinks and the quality of those links. PR employs an iterative process. Initially, all pages are assigned an equal PR score. In each iteration, the scores are updated based on the scores of linking pages. This process continues until PR scores converge or after a certain number of iterations.

Due to this iterative traversal nature, this algorithm can be parallelized, similar to BFS. Within each subgraph, PR calculations can be performed concurrently by assigning individual nodes to threads. They can independently compute PR values for nodes within their respective subgraphs, leading to efficient parallel execution while preserving the algorithm's core structure. Finally, each subgraph's results should be combined to obtain the overall PR scores.

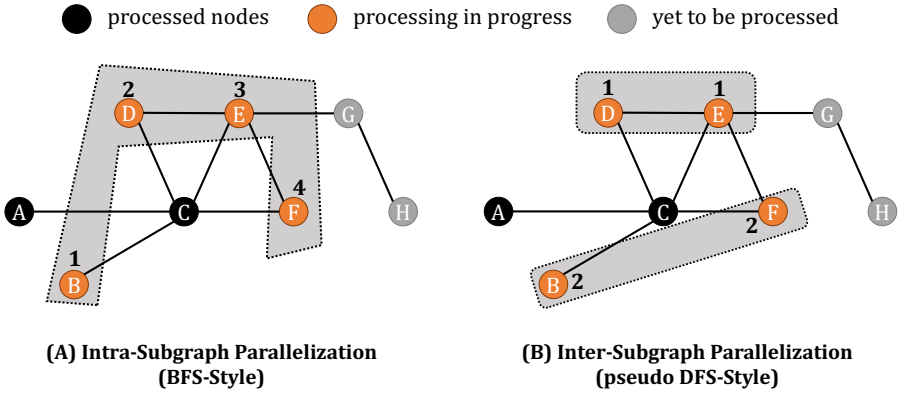


Fig. 3. Example of Intra/Inter-Subgraph Parallelization. (A) { B, D, E, F } are at the same level of BFS and are processed concurrently by 4 threads. (B) As pseudo-DFS progresses, the stack is split into two subgraphs ({ D, E } and { B, F }), which are processed in parallel by 2 threads.

Random Walk (RW). RW is a probabilistic algorithm in which a walker moves through a network (graph), taking steps based on random choices [28]. It is used to analyze the network structure and understand properties such as connectivity and reachability. RW can be viewed as a Markov Chain, where the probability of transitioning to the next state depends only on the current state.

To accelerate RW, we can divide the graph into manageable subgraphs and simultaneously explore multiple nodes within these subgraphs. This approach accelerates the exploration and allows for parallelization of transition probability calculations, making it suitable for estimating node importance through RWs on vast networks. Further, different subgraphs can be processed in parallel while accounting for crossing into a different subgraph.

3 PARALLELIZING GRAPH TRAVERSAL

Our main objective is to efficiently parallelize graph traversal operations with out-of-core systems while maintaining the core properties of the graph algorithms. In this section, we discuss how to achieve this with **intra-subgraph** and **inter-subgraph** parallelization. We present these two techniques with examples and discuss how they can be seamlessly integrated and leveraged alongside SSD parallelization.

3.1 Intra-Subgraph Parallelization

For this approach, we identify subgraphs, the nodes of which can be processed independently so that we can access them in parallel. This means that the processing of one node does not depend on the result or state of other nodes outside the subgraph. Thus, multiple nodes within the subgraph can be processed concurrently by different computing units (threads), allowing for concurrent I/Os, leading to better device utilization. After processing their respective nodes, the results obtained by each thread are aggregated to produce the final result of the algorithm. This ensures efficient exploitation of the underlying device which can speed up the execution of graph traversal operations by processing multiple graph blocks (vertex and edge) in parallel, resulting in faster convergence.

Example. A prime example of this type of parallelization is a *parallel BFS*. BFS explores the graph level by level, where each level represents a set of equidistant vertices from the source vertex. Since vertices of the same level can be accessed independently of each other, all vertices within the same level can be processed concurrently, and thus accessed in parallel using multiple threads. A queue maintains the nodes to be visited next, which are ordered on a per-level basis. Each thread

dequeues nodes from the shared queue and processes them independently. The edges of each node are accessed from the underlying SSD concurrently. Figure 3(A) illustrates the application of this technique for parallelizing the BFS algorithm. Once nodes A and C have been traversed, nodes B, D, E, and F are all at the same level (a subgraph where nodes are independent), enabling them to be processed concurrently. Other BFS-based algorithms (e.g., PageRank, WCC) can also be parallelized with this approach as a building block.

3.2 Inter-Subgraph Parallelization

The subtle difference between Inter-Subgraph and Intra-Subgraph Parallelization is that it identifies subgraphs that can be independently accessed (like two different branches of DFS) and processes them in parallel. That way, multiple subgraphs (or paths) can be traversed concurrently, thus covering the entire graph faster and allowing for faster convergence. The algorithmic correctness and other properties (like the order of accessing nodes) can be ensured by communication and synchronization between the threads processing independent subgraphs. This approach is particularly useful for large-scale graphs that cannot fit entirely in memory or when distributing the computation across multiple threads.

Example. We now use the *pseudo-DFS* [1] as an example. In the classical DFS algorithm, a stack keeps track of the nodes to be explored and maintains the visiting order. In the pseudo-DFS algorithm, a stack can be split into smaller stacks when its size exceeds a predefined threshold, and the smaller stacks are processed in parallel. This allows for multiple threads to work on different subgraphs (paths) concurrently. Figure 3(B) shows an example of this approach. In this example, after traversing nodes A and C, the stack size grows to four and (assuming this is the threshold) is split in two. The first stack contains nodes D and E, while the second contains B and F. These smaller stacks are processed in parallel, leading to two independent graph traversals with the additional need for communication to avoid crossing from one subgraph (path) to another. Inter-subgraph parallelization also benefits finding Strongly Connected Components (SCCs) or groups of nodes within a graph where each node is accessible from every other node in the same group.

3.3 Discussion

Which approach, which data structure? The selection between intra-subgraph and inter-subgraph parallelization, as well as the choice of data structure depends on the algorithm being parallelized. For example, in cases where the algorithm involves BFS-like exploration, intra-subgraph parallelization is the best fit. On the other hand, for algorithms resembling pseudo-DFS or those focused on connectivity exploration, inter-subgraph parallelization can be more effective since it allows different subgraphs to be processed concurrently, facilitating quicker convergence. In both cases, graph traversal is accelerated by overlapping the standard accesses of the original algorithm with several other accesses that would normally be scheduled for later. Thus, a larger subgraph is traversed than the original algorithm without altering its key properties.

Parallelizing Essentials. When parallelizing graph traversal algorithms, we need to guarantee the correctness and the efficiency of the parallel execution. To achieve this, we use *result aggregation*, *synchronization*, and *communication* mechanisms. In algorithms like PageRank, where the goal is to calculate rankings, the individual results obtained from different subgraphs or processing units must be aggregated to calculate the final rankings. Algorithms like DFS require synchronization to prevent race conditions and maintain the same vertex visiting order and, thus, the core guarantees of the algorithm. Further, many algorithms need some form of communication between the threads working on subgraphs (signaling or message passing) to indicate convergence. Minimizing such communication and synchronization overhead is a key challenge to avoid bottlenecks.

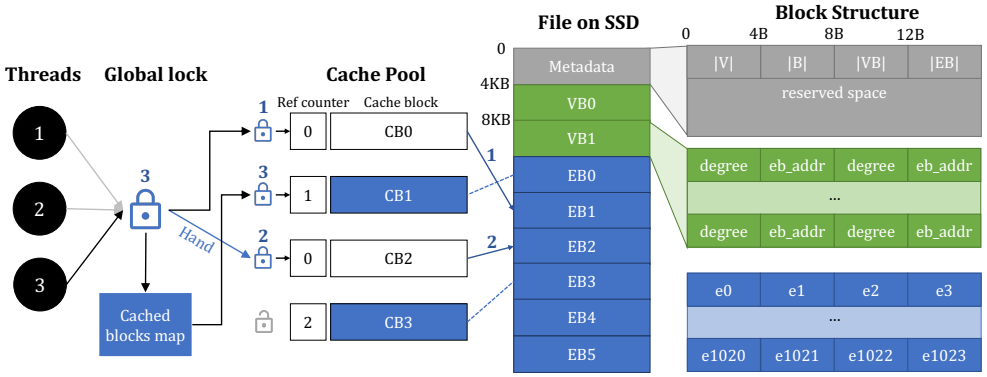


Fig. 4. Architecture of CAVE comprises block-based file structure (right) and a concurrent cache pool (left)

4 CONCURRENT GRAPH ALGORITHMS

The core idea of our approach is to implement parallel graph algorithms that take advantage of concurrency at the storage level. Our system, CAVE, identifies and parallelizes independent I/Os, similar to how out-of-order processors parallelize load and store commands that are not dependent on each other. This enables parallel graph data processing, allowing multiple nodes to be accessed simultaneously, thus significantly reducing the number of iterations required. We carefully tune CAVE to employ the *optimal concurrency* [37] for the underlying storage devices to guarantee maximum benefit. To do this, we issue in parallel as many *independent I/O operations* as the storage device supports without hurting latency. As a result, graph algorithms in CAVE have faster convergence and more efficient data accesses. Overall, the work presented in this paper contributes to the system-level understanding of how to build efficient graph processing systems that maximize the utilization of the underlying SSD. To demonstrate the benefits of our approach, we parallelize five of the most common graph traversal algorithms: BFS, WCC, PageRank, Random Walk, and DFS. In this section, we first provide a quick overview of the physical data layout CAVE and then discuss the details of the parallel versions of these algorithms.

4.1 CAVE Physical Data Layout

CAVE uses a memory-mapped binary file format, with three main parts: the metadata block, the vertex block, and the edge block – right part of Figure 4. They are stored using 4KB aligned blocks to support direct reading and writing from/to the SSDs. All blocks are cached in memory by a cache pool described in detail in Section 5.

Metadata Block. The metadata block serves as a repository for essential graph information such as the number of vertices, the total number of blocks, edge blocks, and vertex blocks, each of which is stored as a 32-bit integer. The remaining space is reserved for future utilization, allowing for additional usage-specific information to be incorporated when necessary.

Vertex Block. Each vertex block, sized at 4KB, stores information about up to 512 vertices. Within each vertex, 8 bytes are allocated, encompassing two 32-bit unsigned integers: $degree$, eb_addr (edge block index and offset). The low 10-bit of eb_addr represents the offset eb_offset inside of an edge block, which just fits its capacity of 1024. The high 22-bit states the index of the edge block eb_idx. The reading process can start by calculating the appropriate address $((eb\_idx \cdot 4\text{KB}) + eb\_offset)$.

Edge Block. To optimize storage and retrieval, we utilize a compact representation of edges. Each edge is represented by a 4-byte integer denoting the index of the ending vertex. Hence, each edge block can store up to 1024 edges (adding up to 4KB). The edges of vertices with a degree less

than 1024 are contained within a single edge block (note that in many datasets, most nodes have indeed a degree of less than 1024). This ensures efficient single read I/O access, while the starting index inside the block (eb_offset) can vary. However, vertices with a degree over 1024 will occupy multiple edge blocks. In this case, the first block always has an $eb\_offset = 0$ to simplify the packing and subsequent reading process. The number of edge blocks per vertex is given by its degree divided by 1024.

Bin-Packing Edges. Previous practices often stored all edges in a single extensive list, resulting in inefficient I/O operations when accessing edges of small vertices spanning multiple blocks. To mitigate this issue and optimize I/O and cache utilization, we approached it as a *bin-packing problem*. Edges of small vertices are stored within a single container (block), while larger vertices span multiple consecutive blocks. We employ an offline first-fit strategy, determining the appropriate block to insert new vertex neighbors and ensuring efficient packing and retrieval of edge data.

Handling Updates. While CAVE currently does not support graph updates, we consider this as part of our future work. Currently, we have a very compact representation of vertex and edges. To handle updates on vertex/edge values (e.g., edge weights, vertex payloads), we need to modify our file architecture to accommodate these values. For instance, for each edge, we store a vertex ID using 4 bytes, which would need to be increased to account for additional edge information like weights. To exploit the *write concurrency* of the underlying device, updates can be batched in a memory buffer and applied to the corresponding blocks concurrently (using the appropriate degree of concurrency when writing). Further, this style of updating (and deleting) can use storage-resident update components similar to the log-structured merge (LSM) design [35, 46]. In this case, updates (and deletes) will be buffered in memory and later organized on disk before being eventually merged with the base data [6]. That way, the update mechanism can exploit both the good sequential performance and, when merging with the based data, the device concurrency.

4.2 Building Blocks for Parallelizing

***ProcessQueue* function.** In the context of BFS, WCC, PR, and RW algorithms, the parallelization process is structured as an iterative procedure. Each iteration involves processing a list of vertices (known as the $frontier$), accessing the neighbors of each vertex, updating vertex values, and determining which vertices should be visited in the next iteration, which are stored in the *next* queue. This iterative process can be naturally parallelized by having multiple threads working on individual vertices of the $frontier$ (intra-subgraph parallelization). We achieve this using a *ProcessQueue* function, which takes the $frontier$, a user-defined *process* function and the device's read concurrency k_r as parameters. The *process* function specifies the actions the algorithm should perform for each vertex and its neighbors. The *ProcessQueue* function parallelizes at the vertex level based on the k_r value where each thread is responsible for processing a vertex and executes a *getEdge* operation to retrieve the edge block from the cache pool. Since each edge block stores neighbors of multiple vertices, it is possible that an edge block swapped out from the cache will need to be read again from the disk, especially when the cache size is limited.

***ProcessQueueBlock* function.** To avoid multiple accesses of the same edge blocks, we provide a new variation that processes data at the granularity of edge blocks to benefit from caching. Initially, all edge blocks associated with vertices in the $frontier$ are found. Next, each thread is assigned to work on one of the edge blocks. That block, in turn, may contain (i) the edges of a single vertex where the execution will be the same as before, or (ii) the edges of multiple vertices where the processing of those vertices will now be completed with a single I/O. By simultaneously processing all vertices connected to a specific block, the approach ensures that each edge block is only read once in each iteration. While this strategy involves some overhead in terms of preprocessing the

Algorithm 1: Parallelization Building Blocks

```
1:  function ProcessQueue(frontier, Func Process, k_r)
2:      next ← ∅
3:      // Process vertices in parallel with max k_r threads
4:      for v_1 in frontier do
5:          // Read neighbors from the cache pool
6:          neighbors ← GetEdges(v_1)
7:          // Process v_1 with its neighbors, get next_v queue
8:          next_v ← Process(v_1, neighbors)
9:          // Merge next_v to next
10:         mtx.lock()
11:         next.insert(next_v)
12:         mtx.unlock()
13:     end for
14:     return next
15: end function
16:
17: function ProcessQueueBlock(frontier, Func Process, k_r)
18:     block_set ← HashSet()
19:     for v_1 in frontier do
20:         block_idx ← GetBlockIdx(v_1)
21:         block_set.Insert(block_idx)
22:         block[block_idx].Insert(v_1)
23:     end for
24:     // next queue of whole frontier
25:     next ← ∅
26:     // Process blocks in parallel with max k_r threads
27:     for block_idx in block_set do
28:         // next queue of this block
29:         next_b ← ∅
30:         block_data ← GetBlock(block_idx)
31:         // For each v_1 associated with this edge block
32:         for v_1 in block[block_idx] do
33:             // Get v_1 neighbors from this block locally
34:             neighbors ← ReadFromBlock(block_data, v_1)
35:             next_v ← Process(v_1, neighbors)
36:             // Merge next_v in next_b
37:             next_b.insert(next_v)
38:         end for
39:         mtx.lock()
40:         next.Insert(next_b)
41:         mtx.unlock()
42:     end for
43:     return next
44: end function
```

$frontier$, it offers the advantage of being minimally impacted by the size of the cache. Further, the edge block retrieval is performed concurrently, which contributes to its superior runtime performance. The two building-block algorithms are outlined in Algorithm 1.

Algorithm 2: Parallel Breadth-first Search

```
1:  function BFSprocess(v_1, neighbors)
2:      next_v ← ∅
3:      for v_2 in neighbors do
4:          if visited[v_2].CAS(False, True) then
5:              // Add v_2 to the next queue of v_1
6:              next_v.INSERT(v_2)
7:          end if
8:      end for
9:      return next_v
10: end function
11:
12: function PBFS(v_s, k_r)
13:     frontier ← {v_s}
14:     vertices_count ← 0
15:     while frontier.SIZE > 0 do
16:         next ← ProcessQueue(frontier, BFSprocess, k_r)
17:         // Or call ProcessQueueBlock()
18:         vertices_count ← vertices_count + frontier.SIZE
19:         frontier ← next
20:     end while
21:     return vertices_count
22: end function
```

4.3 Parallel Breadth-First Search

We develop a parallel BFS (PBFS for short) algorithm using two queues: the $frontier$ queue, which contains the indices of vertices in the current level, and the *next* queue, which stores the indices of the neighbors of vertices in the $frontier$ queue, which correspond to the vertices in the next level. To leverage parallelism, each vertex in the $frontier$ queue is assigned to a separate thread so that multiple I/Os can be issued in parallel as shown in Figure 1(A). The complete algorithm is listed in Alg. 2. For each vertex in $frontier$, as *BFSprocess* defines, *ProcessQueue* will assign threads to vertices. Each thread accesses the assigned vertex, retrieves the indices of its neighbors, checks and flags the index of every neighbor as *visited*, inserts it to $next_v$ queue of this vertex, and merges $next_v$ of all vertices to the final *next* protected by a global lock *mtx* to prevent data races and ensure thread safety. The PBFS level of concurrency is controlled by the number of threads, which we tune according to the *optimal concurrency* of the SSD. Once all the vertices in the $frontier$ queue have been processed, the contents of the *next* queue are copied back to the $frontier$ queue, and the *next* queue is cleared. This process is repeated until the $frontier$ queue becomes empty, signifying the completion of the BFS traversal. We also developed a *blocked* variant of the $frontier$ processing that uses the *ProcessQueueBlock* function. As discussed in §4.2, this approach discovers the edge blocks of the $frontier$ vertices and allocates threads to edge blocks, parallelizing at the edge block level while ensuring that each edge block is read only once during an iteration. This results in two

Algorithm 3: Parallel Weakly Connected Component

```
1:  function PWCC(k_r)
2:      wcc_count ← 0
3:      for i ← 0 to N − 1 do
4:          // If not flagged
5:          if visited[i] = False then
6:              // Call BFS to flag all vertices in this WCC
7:              PBFS(i, k_r)
8:              wcc_count ← wcc_count + 1
9:          end if
10:     end for
11:     return wcc_count
12: end function
```

benefits: (i) overall runtime improvement since edge blocks are not read multiple times, and (ii) performance does not depend on cache pool size.

Algorithm 4: Parallel PageRank

```
1:  function PRprocess(v_1, neighbors)
2:      pr_next[v_1] ← 0
3:      for v_2 in neighbors do
4:          // Sum up last pr value of neighbors
5:          pr_next[v_1] ← pr_next[v_1] + pr[v_2]
6:      end for
7:      // Add damping factor and divide by its degree
8:      pr_next[v_1] ← ((1−d)+d·pr_next[v_1]) / GetDegree(v_1)
9:      return ∅
10: end function
11:
12: function ParallelPageRank(iterations, k_r)
13:     frontier ← {0, 1, ..., N − 1}
14:     for i ← 0 to N − 1 do
15:         pr[i] ← 1 / GetDegree(i)
16:         pr_next[i] ← pr[i]
17:     end for
18:     while iterations > 0 do
19:         ProcessQueue(frontier, PRprocess, k_r)
20:         pr ← pr_next
21:         iterations ← iterations − 1
22:     end while
23:     // Prepare return value
24:     for i ← 0 to N − 1 do
25:         pr[i] ← pr[i] · GetDegree(i)
26:     end for
27:     return pr
28: end function
```

4.4 Parallel Weakly Connected Components

Computing WCC entails repeatedly searching from each vertex in the graph. Since we utilize the adjacency list format, the most efficient approach to compute WCC involves repeatedly applying the search algorithm starting from each vertex. During the search process, a visited vertex is marked as *true* and subsequently avoided in subsequent iterations. We parallelize WCC by performing multiple concurrent searches using PBFS due to its low overhead and well-established efficiency. Algorithm 3 lists the algorithm for PWCC.

4.5 Parallel PageRank

We consider the topology approach for PR, which involves updating the PR values (pr) of all vertices based on the values of their neighbors from the previous iteration (Algorithm 4). Since all vertices need to be processed in each iteration, the $frontier$ queue always contains the entire list of vertices, and there is no need for a *next* queue. Initially, the $frontier$ queue includes all vertices, from vertex 0 to vertex $N-1$. In every iteration, the *ProcessQueue* is called with the desired concurrency to parallelize each step of the algorithm. For the blocked implementation, the *ProcessQueueBlock* function is called. The initial PageRank values, $pr[i]$ and $pr_{next}[i]$, are assigned as the inverses of the degrees of their respective vertices v_i. It is worth noting that in the original PageRank algorithm, the initial PageRank value for each vertex is set to 1, and its neighbors are assigned values of $\frac{pr[i]}{deg[i]}$. To optimize the computation, we perform this division in advance so it does not need to be repeatedly calculated by the neighbors in each iteration.

Algorithm 5: Parallel Random Walk

```
1:  function RWProcess(v_1, neighbors)
2:      // Randomly selects a neighbor
3:      v_2 ← RandomSelect(neighbors)
4:      return {v_2}
5:  end function
6:
7:  function ParallelRandomWalk(K, iterations, k_r)
8:      frontier ← ∅
9:      visited_count ← 0
10:     for i ← 0 to K − 1 do
11:         // Initialize starting vertices randomly
12:         frontier[i] ← Random(0, N − 1)
13:     end for
14:     while iterations > 0 do
15:         next ← ProcessQueue(frontier, RWprocess, k_r)
16:         frontier ← next
17:         visited_count ← visited_count + K
18:         iterations ← iterations − 1
19:     end while
20:     return visited_count
21: end function
```

Algorithm 6: Parallel Pseudo Depth-first Search

```
1:  function DFSTask(stack, k_r)
2:      max_stack_count ← k_r
3:      while stack.size() > 0 do
4:          // Get and pop vertex at the stack top
5:          v_1 ← stack.top()
6:          stack.pop()
7:          visited_count ← visited_count + 1
8:          neighbors ← GetEdges(v_1)
9:          // Push all unvisited neighbors on stack
10:         for v_2 in neighbors do
11:             if visited[v_2].CAS(False, True) then
12:                 stack.push(v_2)
13:             end if
14:         end for
15:         // Check if the stack size is larger than threshold
16:         while stack.size() > max_stack_size do
17:             if stack_count < max_stack_count then
18:                 stack_count ← stack_count + 1
19:                 // Split the stack and generate new task
20:                 new_stack, stack ← stack.split()
21:                 ThreadPool.push(DFStask, new_stack)
22:             end if
23:         end while
24:     end while
25:     stack_count ← stack_count − 1
26: end function
27:
28: function ParallelPseudoDFS(v_s, k_r)
29:     init_stack ← {v_s}
30:     visited[v_s] ← True
31:     stack_count ← 1
32:     visited_count ← 0
33:     // Push the initial task in the thread pool
34:     ThreadPool.push(DFStask(init_stack, k_r))
35:     // Wait for all tasks to be finished
36:     ThreadPool.WaitAll()
37:     return visited_count
38: end function
```

4.6 Parallel Random Walk

A single random walk is inherently a serial process and does not significantly benefit from data concurrency. However, an effective strategy is to run multiple random walks concurrently, which not only improves the precision of the results but also reduces the overall running time. Initially, k vertices are randomly chosen from the whole vertex set and put in the $frontier$ queue. In each

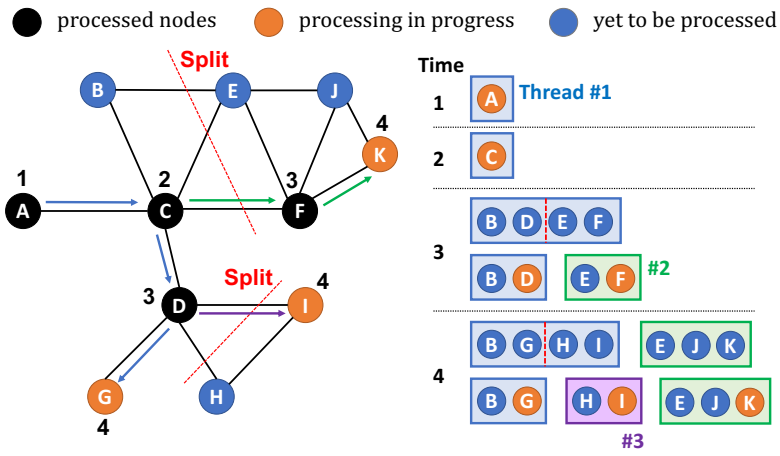


Fig. 5. Example of the Parallel Pseudo DFS algorithm, demonstrating the progression of the stack over time.

iteration, the *RWprocess* function randomly selects one of the neighbors for each vertex in $frontier$ as the successor in the next iteration. Algorithm 5 outlines the complete algorithm.

4.7 Parallel Pseudo Depth-First Search

While DFS is inherently a serialized algorithm, it is possible to enhance its performance by introducing parallelism through a technique known as *unordered* or *pseudo-DFS* [1]. We take inspiration from this idea, and we incorporate a mechanism to monitor the size of the vertex stack for each thread in our implementation (Algorithm 6). In the beginning, only one stack is active with the starting vertex v_s. We create a new *DFStask* with this stack in the thread pool. The *DFStask* continuously pops the stack, reads its neighbors, and pushes them into the stack as a normal DFS does. After visiting the neighbors of a vertex, we check if the size of the stack exceeds a predefined threshold. If it does, the stack is evenly divided into two smaller stacks, and one of these stacks is assigned to a new thread for further exploration. Figure 5 illustrates the algorithm, with the right side of the figure depicting the timeline status of the stack and its splitting. The graph is a snapshot after *time 4*, where three threads are working in parallel to process the nodes. This approach allows each thread to independently perform DFS on its allocated stack and split it when necessary. By dynamically splitting the stacks in this manner, we achieve increased concurrency during the DFS traversal. The choice of the threshold value determines the trade-off between concurrency and thread creation overhead. Setting a smaller threshold allows for higher concurrency but may result in a larger number of threads being created. On the other hand, a larger threshold reduces the number of thread creations but may limit the degree of parallelism. The selection of an appropriate threshold is crucial to strike a balance between concurrency and overhead.

5 IMPLEMENTATION

In this section, we present implementation details of CAVE.

Concurrent Cache Pool. To prevent redundant disk reads, CAVE has a cache pool that stores recently used edge blocks in main memory and employs a clock eviction policy. This caching mechanism becomes crucial because an edge block can contain information for multiple small vertices. It is designed to support concurrent access from multiple threads and enables concurrent I/O operations. As shown in the left-hand-side of Figure 4, it comprises three key components: a global lock, a list of slots to store cached blocks, and a cached block map that tracks the mapping

of cached block IDs to their positions in the list. Each cached slot within the pool has its lock and a reference counter, while the global lock ensures that only one thread can manipulate the clock hand and modify the map of cached blocks at a given time, preventing potential conflicts.

When a thread requires a specific block, it first checks the cached block map to determine if it is already cached. If the block is found, the thread attempts to acquire the associated lock. Upon successful acquisition, the thread retrieves the content from the block, releases both the global lock and the block lock, and proceeds with the required operations. However, when the desired block is not found in the cache, the thread searches for an available or evicted cached block by moving the clock hand and decrementing the reference counter. Once a suitable cached block is identified, the thread acquires the lock associated with the cache block, releases the global lock to allow other threads to enter the cache pool, and initiates the process of loading the data block from the SSD. With the global lock released, multiple threads can access the cache pool concurrently and initiate their own I/O operations. After reading the block into the cache slot, the lock is released, making the block available for subsequent use by other threads.

Note that the global lock is kept for a small duration: either until the cached page is accessed in memory, or until a block for eviction/loading is identified and locked. After this, the global lock is released and more threads can enter the cache pool. Our experiments show that in I/O-bound scenarios this short-lived critical section does not create a bottleneck. Figure 6 shows the percentage of time that is spent due to the global lock when running parallel BFS in the Friendster dataset (details in §6). The total lock waiting time remains low even with a high number of concurrent I/Os (e.g., around 1.3% of running time for 64 concurrent I/Os), which shows that the global lock does not create a bottleneck. With its concurrent access support and efficient management of cached blocks, this caching mechanism significantly improves overall performance by minimizing unnecessary storage accesses.

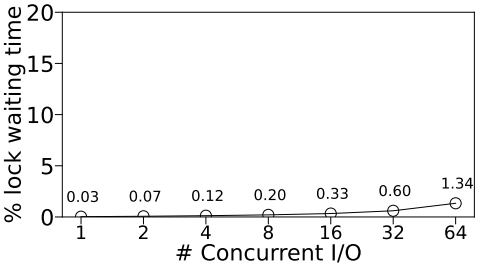


Fig. 6. Lock waiting time remains low as we increase the number of concurrent I/Os.

Codebase. We develop CAVE using C++17, and we leverage its native support for concurrent execution through the `std::thread` functionality. We incorporate the lightweight `BS::thread_pool` library [49] that enhances portability and minimizes overhead. We also use the library parallel hashmap [43] for the cache pool and ensure high accuracy in our runtime measurements with `chrono:: high_resolution_clock`.

I/O Interface. To have full control over the device, we perform direct I/O using the `O_DIRECT` flag so that data is transferred directly from the storage device to main memory, bypassing the system cache. Our blocked file structure ensures that each access is aligned. This alignment is further guaranteed by using the `aligned_alloc()` function whenever new blocks are allocated. For concurrent I/O, we use `pread` and `pwrite` in conjunction with the `BS::thread_pool` library, and we are compatible with both Linux and Windows (leveraging *Overlapped I/O* for the latter).

Data Files. We developed a custom parser to convert common graph data into our binary file structure. It accepts standard adjacent list and edge list files in plain text format as input, parses them, and converts them to our binary file structure.

6 EVALUATION

We now present the experimental evaluation of CAVE for the five algorithms and compare it with three storage-optimized graph processing systems GraphChi [24], Mosaic [30] and GridGraph [62] for multiple datasets and devices.

Experimental Setup. Our experimental server has two Intel Xeon Gold 6230 CPUs, each with 20 cores with virtualization, and with 384GB of main memory. We experiment with three storage devices: (i) an *Optane SSD* (375GB P4800X), (ii) an *PCIe SSD* (1TB PCIe P4510), and (iii) a *SATA SSD* (240GB SATA S4610). For all three devices, we quantify the read concurrency (k_r) through careful benchmarking (6 for Optane SSD, 60 for PCIe SSD, and 25 for SATA SSD). Unless otherwise mentioned, we match the number of concurrent I/Os to k_r of the corresponding device for optimal device utilization [37]. All devices were pre-conditioned by sequentially writing on the entire device three times before running the experiments to ensure stable performance [12]. All experimental results are averaged over three iterations, and the standard deviation was less than 1%.

Dataset. We use five datasets of different sizes and types from the Stanford Large Network Dataset Collection [25] and LDBC Graph Analytics Benchmark [17]: Friendster Social Network (FS), Twitter Social Network (TW), RoadNet Network of PA (RN), LiveJournal Social Network (LJ) and YouTube Social Network (YT). FS is the largest dataset among these, with 65M nodes and 32GB size. We also experiment with a synthetic dataset (SD) which is generated following the Barabási–Albert model [4]. We configured the graph with 50 million vertices, each connected to 25 neighbors, resulting in a total of 1.25 billion edges. Note that the RN graph is very sparse while the SD graph is extremely dense. The key properties of the datasets are presented in Table 1.

Table 1. Dataset Description

Dataset	Description	#Nodes	#Edges	Diameter	Size
FS	Friendster Social Network	65M	1.8B	32	32 GB
TW	Twitter Social Network	53M	2B	18	28 GB
RN	RoadNet Network of PA	1M	1.5M	786	47 MB
LJ	LiveJournal Social Network	5M	69M	16	1 GB
YT	YouTube Social Network	1.1M	3M	20	39 MB
SD	Synthetic data	50M	1.25B	6	20 GB

Preprocessing time and space requirement. Table 2 presents the preprocessing time and space requirement of all systems for the FS and TW dataset. CAVE exhibits the lowest preprocessing time and a reduced space requirement compared to GridGraph and Mosaic. For example, Mosaic's preprocessing time and space requirement is 9× and 2× that of CAVE for the FS dataset, respectively. While GridGraph has a similar preprocessing time to CAVE, its space requirement is 6× that of CAVE. This efficiency stems from CAVE's compact file architecture and simple design, contrasting with the more demanding preprocessing requirements of systems like GridGraph and Mosaic.

Table 2. Preprocessing Time and Space Comparison

System	Preprocessing Time (s)		Data File Size (GB)	
	Dataset: FS	Dataset: TW	Dataset: FS	Dataset: TW
GraphChi	819	784	8.3	8.4
GridGraph	55	86	84	75
Mosaic	469	370	27	17
CAVE	52	49	14	13

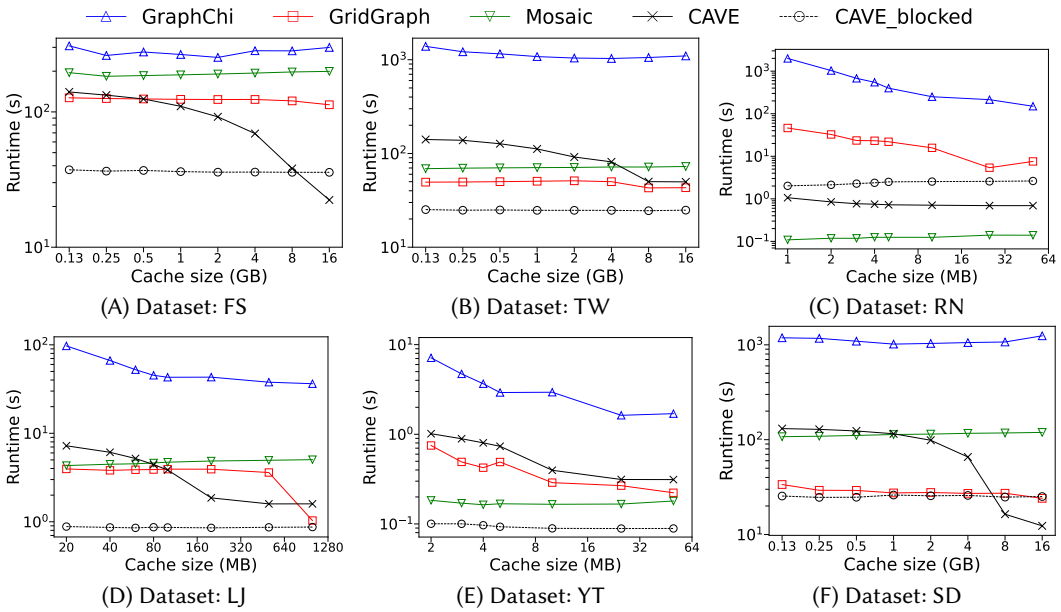


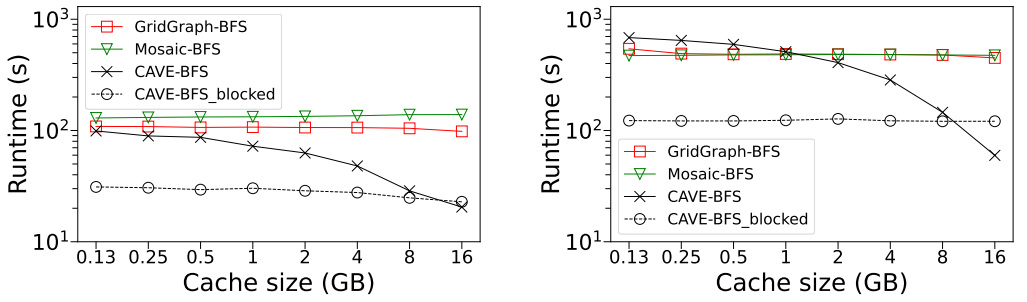
Fig. 7. (A) - (F) Performance graph for BFS on our PCIe SSD. In general, CAVE outperforms the baselines GraphChi, GridGraph and Mosaic for all six datasets. Mosaic performs well for the sparse RN dataset.

6.1 Parallel BFS

CAVE Outperforms all baselines. In our first set of experiments, we evaluate the performance of CAVE, GraphChi, GridGraph and Mosaic as we vary the cache size for all six datasets. Figures 7(A) - (F) show the performance of the four systems for BFS when the underlying device is the PCIe SSD. We compare using both the blocked and non-blocked variants of the frontier processing to see their effect on different graphs. Since the datasets have different sizes, the cache value is set accordingly. The results show that CAVE (both blocked and non-blocked) significantly outperforms GraphChi for any cache ratio and any dataset. Notably, when the cache ratio is low, CAVE outperforms GraphChi with a significantly higher speedup due to its better utilization of SSD concurrency.

For example, Figure 7(A) presents a performance comparison of the four systems for the Friendster dataset (65M nodes, 32GB size). The figure shows that the non-blocked implementation benefits from a higher cache size while the blocked implementation remains unaffected by the cache size. This is due to the design techniques of the blocked implementation, which ensure that all edge blocks are read only once during an iteration. We observe that GridGraph and Mosaic are faster than GraphChi, but both CAVE implementations outperform them. Specifically, the blocked version provides up to 3.4× and 5.6× speedup across various cache sizes against GridGraph and Mosaic, respectively. The non-blocked variant delivers comparable performance to GridGraph for smaller cache sizes and up to 5.1× speedup for larger cache sizes while always outperforming Mosaic by a higher margin. CAVE's blocked implementation outperforms the other three systems for the TW, LJ and YT datasets as shown in Figures 7(B), (D) and (E).

CAVE is Well-suited for Sparse Graphs. In Figure 7(C), we see that in a sparse graph with an unusually high diameter (RN dataset), CAVE performs better than both GraphChi and GridGraph, but falls short of Mosaic. The non-blocked implementation of CAVE works well for this graph because, in sparse graphs with lower average degrees where edge blocks can be associated with multiple vertices, all required edge blocks for an iteration can fit in the cache pool without swapping. With a sufficiently large cache, the non-blocked variant benefits from multiple threads working



(A) Optane SSD (B) SATA SSD

Fig. 8. (A, B) Performance graph for PBFS on Optane SSD and SATA SSD for the FS dataset. In both devices, CAVE outperforms the baselines.

on cache data. The figure shows that the blocked implementation (black dashed line) outperforms both GraphChi and GridGraph while the non-blocked implementation (black solid line) can be up to 3.8× faster than the blocked one. While CAVE consistently outperforms Mosaic for other datasets, Mosaic outperforms CAVE for this dataset. This is because Mosaic uses *Hilbert-ordered tiles* as its graph representation, which allows to skip lots of empty/unneeded tiles for sparse graphs.

CAVE Provides Good Performance for Dense Graphs. Our synthetic SD dataset is an unusually dense graph (the diameter is only 6 with 50M nodes and 1.25B edges). In Figure 7(F), we observe that CAVE outperforms GraphChi and Mosaic. However, CAVE-blocked provides marginal benefit compared to GridGraph for the SD dataset. The reason behind this is that GridGraph is designed for dense graphs because its data structures and algorithms are optimized to efficiently handle the high connectivity and dense nature of such graphs. GridGraph achieves this by utilizing a grid structure to partition and manage the graph's data. This approach allows it to optimize memory usage and reduce access times for dense graph structures. However, CAVE still provides up to 25.4% faster runtime compared to GridGraph for dense graphs.

Similar Performance Benefit Across All Devices. We now compare CAVE against Mosaic and GridGraph in the Optane SSD and SATA SSD as we vary the cache size for BFS on the FS dataset. Figures 8(A) and (B) show that the performance trend of these systems remains similar to the PCIe SSD (Figure 7(A)). CAVE consistently outperforms both GridGraph and Mosaic. We also observe that all systems running on the Optane SSD have an overall lower runtime than the PCIe SSD because of Optane SSD's faster read performance. In contrast, all systems running on the SATA SSD have a higher runtime than the PCIe SSD due to the slowness of the SATA SSD.

CAVE Utilizes Concurrent I/Os. To analyze how the concurrent I/O affects the performance when using various devices and datasets for the algorithms, we now vary the number of concurrent I/Os in the blocked PBFS implementation. Since GraphChi, GridGraph or Mosaic do not have the support of varying concurrent I/Os, we do not include them in this experiment. Figure 9 shows CAVE's performance graph for the FS dataset for all three of our devices. The figure shows that as we increase the number of concurrent I/Os, PBFS's runtime decreases until the device becomes saturated. For example, the SATA SSD gets saturated when using 16-32 concurrent I/Os (red line), which is consistent with the device's optimal concurrency value (25). However, if we issue more concurrent I/Os, performance starts to degrade because of the thread management overhead while the device is already saturated. The PCIe SSD

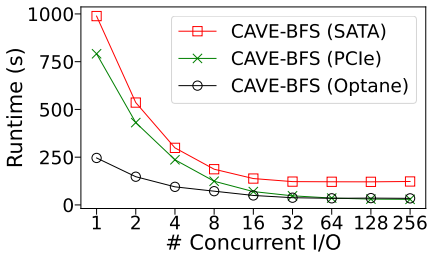
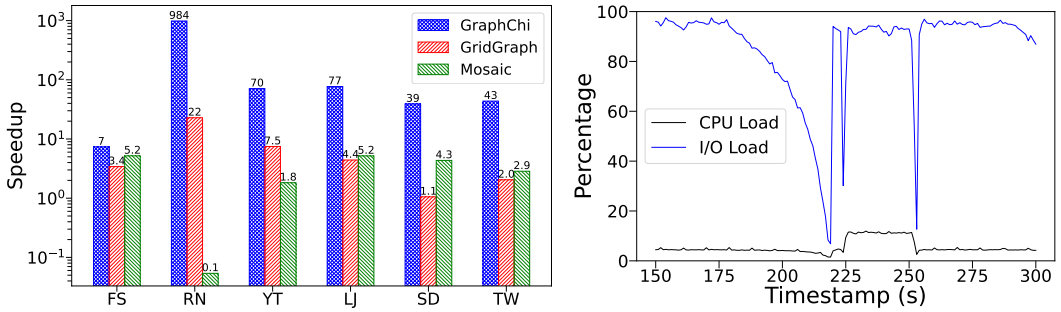


Fig. 9. As we increase the number of concurrent I/Os, the benefit of PBFS increases until the device gets saturated.



(A) CAVE's speedup for PBFS (cache size: 3% of dataset) (B) CAVE's CPU and I/O load for PBFS on PCIe SSD

Fig. 10. (A) CAVE performs well across all datasets for PBFS. Only Mosaic has a better performance for the sparse RN dataset. (B) CPU usage of CAVE remains low showing that its benefit comes from better SSD utilization. In other words, CAVE is I/O-bound, not CPU-bound.

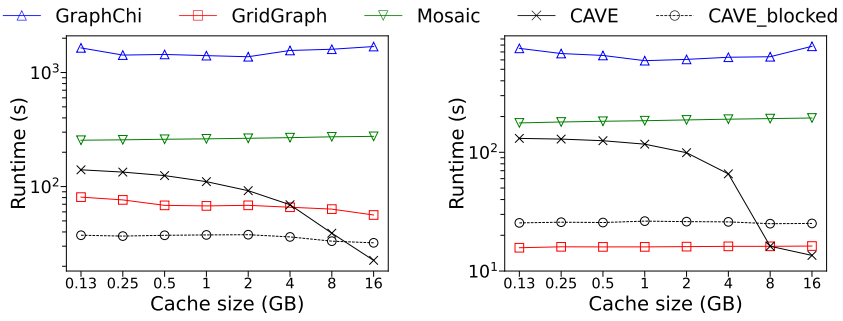
curve is moved towards higher concurrency, as expected, and Optane SSD has a flatter curve, which is consistent with prior work [37, 39].

CAVE Excels Across Different Datasets. Figure 10(A) shows CAVE's speedup vs. GraphChi, GridGraph, and Mosaic for all five datasets with cache size 3% of the dataset when running on the PCIe SSD. The speedup of CAVE compared to GraphChi, GridGraph, and Mosaic ranges from $7 - 984\times$, $1.1 - 22\times$, and $0.1 - 5.2\times$, respectively. The unusually high speedup compared to GraphChi for the RN dataset is attributed to the high diameter of the graph, where GraphChi needs an exceptionally large number of iterations to achieve convergence. GridGraph also takes a high number of iterations to converge which shows that CAVE can handle graphs with high diameters better than both GraphChi and GridGraph. Although Mosaic performs the best for this sparse dataset, CAVE outperforms Mosaic for the other datasets. For dense graphs like SD, GridGraph performs well because of its grid structure to partition and manage dense graphs, however, CAVE still outperforms GridGraph.

CAVE's Benefits Come From Better Storage Utilization. Our profiling shows that CAVE is I/O-bound (rather than CPU-bound), thus, the performance benefits come from better utilization of the underlying storage devices. We capture a snapshot of the CPU load and disk bandwidth for PBFS on the FS dataset in Figure 10(B). The figure shows that CAVE consistently maintains low CPU utilization, generally below 5%, while the disk bandwidth remains around 100%. To calculate bandwidth utilization, we divide the observed bandwidth by the maximum device bandwidth achieved when using the same number of concurrent threads to issue I/Os.

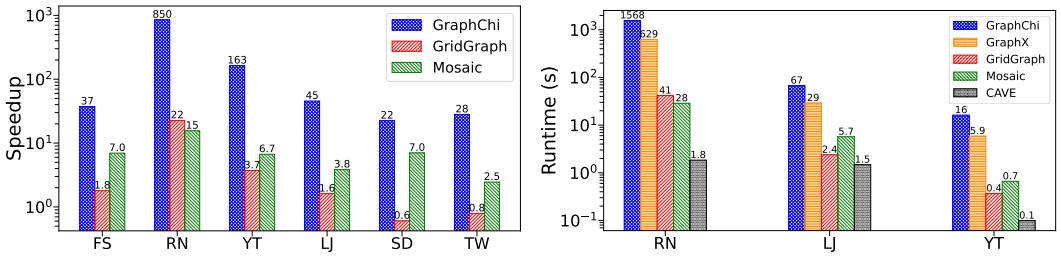
6.2 Parallel WCC, PR & RW

6.2.1 Parallel Weakly Connected Components. Figures 11(A) and (B) present the performance graph of two CAVE implementations, GraphChi, GridGraph, and Mosaic, for weekly connected components as we vary cache size running on the PCIe SSD device for FS and SD datasets. Similarly to the PBFS experiments, the runtime of the blocked implementation does not depend on the cache size. Figure 11(A) shows that CAVE's blocked implementation achieves up to 44×, 2.2×, and 8.6× speedup compared to GraphChi, GridGraph and Mosaic for the FS dataset. However, for the SD dataset in Figure 11(B), GridGraph achieves 1.6× better runtime than CAVE. As mentioned earlier, this is because the SD dataset is seriously dense which works in favor of GridGraph. For both datasets, the non-blocked implementation has a higher runtime when the cache size is small. However, the performance improves as the cache size increases, providing up to 2.5× and 1.2× speedup compared to GridGraph for the FS and SD datasets for higher cache sizes.



(A) Dataset: FS (B) Dataset: SD

Fig. 11. (A) Blocked CAVE implementation outperforms other systems for WCC. (B) For dense graphs, GridGraph works well for finding WCC.



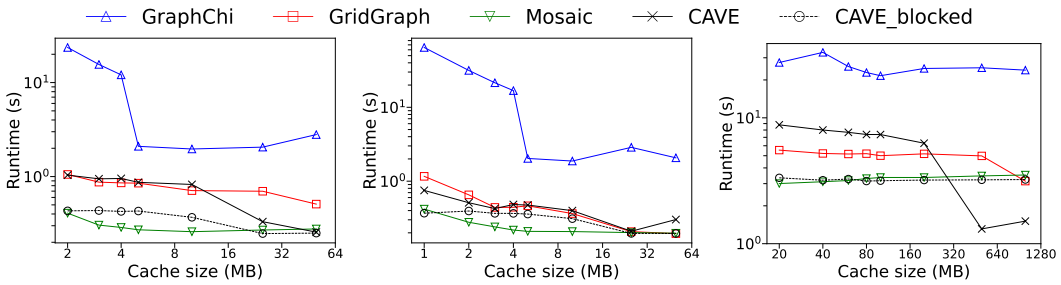
(A) CAVE's speedup for PWCC (B) Experiment with GraphX and other systems

Fig. 12. (A) CAVE performs well across all datasets for parallel weekly connected component (PWCC). (B) CAVE outperforms single-node GraphX deployment.

Figure 12(A) presents a summary result of WCC for all five datasets on the PCIe SSD (3% cache size for each dataset) which shows CAVE can achieve $22-850\times$, $0.6-22\times$ and $2.5-15\times$ speedup compared to GraphChi, GridGraph and Mosaic respectively.

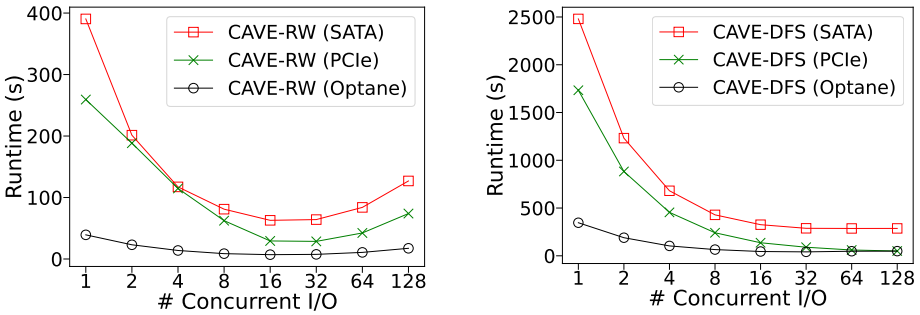
Comparison against GraphX. For completeness, we experiment with GraphX [15] that implements graph-parallel computation by distributing the computation in multiple compute nodes using a different abstraction while requiring all data in memory. Since CAVE is a single-node out-of-core system, a direct comparison against a distributed system should be taken with a grain of salt. We experiment with a single-node deployment of GraphX on our server. Figure 12(B) presents the runtime of GraphChi, GraphX, GridGraph, Mosaic, and CAVE for three datasets when running WCC. We observe that CAVE can be up to two orders of magnitude faster than GraphX. Specifically, CAVE achieves $350\times$, $19\times$ and $59\times$ speedup compared to GraphX for RN, LJ and YT datasets. It is worth highlighting that CAVE achieves this superior performance despite GraphX having all data in memory whereas CAVE only uses 3% memory of the dataset size. We recognize that GraphX is designed for distributed environments, however, the potential costs associated with distributed computing, both monetary expenses and the overheads of managing a distributed computing infrastructure can be significant.

6.2.2 Parallel PageRank. Figures 13(A) - (C) illustrates the performance graph for YT, RN and LJ datasets in the PCIe SSD. The blocked implementation of CAVE achieves lower runtime than both GraphChi and GridGraph while achieving comparable performance to Mosaic. The figure shows that CAVE achieves up to $98\times$ and $3.3\times$ speedup compared to GraphChi and GridGraph, respectively, for the YT dataset. For the RN dataset shown in Figure 13(B), CAVE's speedup is up to $170\times$ ($3.0\times$),



(A) Dataset: YT (B) Dataset: RN (C) Dataset: LJ

Fig. 13. CAVE achieves lower runtime for PageRank, outperforming GraphChi and GridGraph.



(A) Dataset: LJ, Algorithm: Random Walk (B) Dataset: FS, Algorithm: DFS

Fig. 14. (A) Concurrent I/Os improve performance for CAVE's Random Walk till the device is saturated. (B) CAVE's PDFS can attain the maximum benefit of the device by exploiting its *optimal concurrency*.

while for the LJ dataset shown in Figure 13(C), CAVE achieves 8.2× (1.6×) speedup compared to GraphChi (GridGraph). Similarly to our previous experiments, the performance of the non-blocked implementation of CAVE improves as the cache size increases.

6.2.3 Parallel Random Walk. Figure 14(A) shows the representative performance graph of CAVE as we vary the number of concurrent I/O for the LJ dataset across our three SSDs. The figure shows that as we increase the number of concurrent I/Os the runtime decreases across all devices. As expected, we also observe that the Optane SSD is faster than the PCIe SSD and the PCIe SSD is faster than the SATA SSD. As the number of concurrent I/Os reaches the *optimal concurrency* of each device, its runtime is minimized. However, beyond this point, the runtime starts to increase as the device is saturated, and additional parallelism increases the thread management overhead.

6.3 Parallel pseudo DFS

We now focus on the impact of I/O concurrency on the PDFS algorithm. We generate a list of target keys at random and run the algorithm to search each key multiple times in a depth-first manner. Note that GraphChi, GridGraph, Mosaic, and most graph processing systems do not support DFS.

CAVE's PDFS Exploits Device Concurrency. Figure 14(B) shows the performance of CAVE's PDFS as we vary the number of concurrent I/Os for the FS dataset across three devices. We observe that for all devices, as we increase the number of concurrent I/Os, we have improved runtime until the device performance plateaus. The figure shows that CAVE's PDFS achieves 7.7×, 12.6×, 7.6× speedup on the Optane SSD, SATA SSD, and PCIe SSD. We can also reason about each device's concurrency values from the graph as the runtime flattens when the device bandwidth is saturated. The figures

also show that the fastest device (Optane SSD) has a much lower runtime than the slowest device (SATA SSD). We observe a similar trend for other datasets. Overall, these experiments show that CAVE can perform parallel pseudo-DFS while leveraging the underlying SSD concurrency.

7 RELATED WORK

Many scalable graph processing systems like PowerLyra [10], PowerGraph [14], GraphX [15], GBase [21], TurboGraph++ [23], Chaos [44], GraphLab [29], Pregel [31], Gemini [61], VC-Tune [63] can process large graphs in a distributed manner which requires finding optimal partitioning, load-balancing, fault tolerance, and managing the communication overhead. There are some single-node shared-memory systems like Ligra [50], GraphMat [51], GRACE [55], Polymer [57], CGraph [58] that process graphs in memory, and as expected, these systems are highly CPU bound. There are several popular out-of-core processing systems including GraphChi [24], TurboGraph [16], Graphene [26], Mosaic [30], X-Stream [45], GridGraph [62], Graspan [52], RStream [53], and FlashGraph [60]. These systems attempt to minimize random disk access while relying on sequential I/O, extensive preprocessing, and optimal data placement. GraphSSD [33] is a graph-aware SSD framework where the SSD controller is made aware of the graph data structures stored on the SSD. In contrast to these approaches, our goal is to develop a general approach for parallelizing graph traversal algorithms and develop the necessary infrastructure to exploit the underlying *SSD concurrency*.

8 CONCLUSION

Modern storage devices are characterized by their access *concurrency*, which needs to be carefully harnessed to attain maximum benefit from the device. Graph processing systems are a natural candidate for exploiting this property explicitly, yet, most systems do not consider it, resulting in device underutilization. We propose CAVE, a concurrency-aware graph processing system designed to leverage the underlying SSD concurrency. CAVE parallelizes independent I/Os through its concurrent cache pool design, supported by its file structure, enabling the implementation of storage-aware parallel graph algorithms. We develop the parallelized versions of five popular graph algorithms in CAVE and compare their performance with three out-of-core systems, GraphChi, GridGraph, and Mosaic, for multiple datasets and devices. Our evaluation reveals that CAVE achieves up to three orders of magnitude higher speedup than GraphChi and up to one order of magnitude higher speedup than GridGraph and Mosaic.

ACKNOWLEDGMENTS

We sincerely thank the reviewers for their constructive feedback, as well as Vasia Kalavri and Renato Mancuso for their feedback during the revision phase, which substantially helped this project. We further thank the members of the DiSC lab for their useful remarks. Special thanks to Andy Huynh for suggesting the system name. This material is based upon work supported by the National Science Foundation under Grant No. IIS-2144547, a Facebook Faculty Research Award, and a Meta Gift.

REFERENCES

[1] Umut A Acar, Arthur Charguéraud, and Mike Rainey. 2015. A work-efficient algorithm for parallel unordered depth-first search. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. 67:1–67:12. https://doi.org/10.1145/2807591.2807651

[2] T M Tariq Adnan, Md. Saiful Islam, Tarikul Islam Papon, Shourav Nath, and Muhammad Abdullah Adnan. 2022. UACD: A Local Approach for Identifying the Most Influential Spreaders in Twitter in a Distributed Environment. *Soc. Netw. Anal. Min.* 12, 1 (2022), 37. https://doi.org/10.1007/S13278-022-00862-3

[3] Nitin Agrawal, Vijayan Prabhakaran, Ted Wobber, John D. Davis, Mark Manasse, and Rina Panigrahy. 2008. Design Tradeoffs for SSD Performance. In *Proceedings of the USENIX Annual Technical Conference (ATC)*. 57–70. http://dl.acm.org/citation.cfm?id=1404019

[4] Réka Albert and Albert-László Barabási. 2001. Statistical mechanics of complex networks. *CoRR* cond-mat/0 (2001). http://arxiv.org/abs/cond-mat/0106096

[5] Manos Athanassoulis, Bishwaranjan Bhattacharjee, Mustafa Canim, and Kenneth A. Ross. 2012. Path Processing using Solid State Storage. In *Proceedings of the International Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures (ADMS)*. 23–32. http://www.adms-conf.org/athanassoulis_adms12.pdf

[6] Manos Athanassoulis, Subhadeep Sarkar, Tarikul Islam Papon, Zichen Zhu, and Dimitris Staratzis. 2022. Building Deletion-Compliant Data Systems. In *IEEE Data Engineering Bulletin*. 21–36.

[7] Sergey Brin and Lawrence Page. 1998. The Anatomy of a Large-Scale Hypertextual Web Search Engine. *Comput. Networks* 30, 1-7 (1998), 107–117. https://doi.org/10.1016/S0169-7552(98)00110-X

[8] Feng Chen, Binbing Hou, and Rubao Lee. 2016. Internal Parallelism of Flash Memory-Based Solid-State Drives. *ACM Transactions on Storage (TOS)* 12, 3 (2016), 1–13. https://doi.org/10.1145/2818376

[9] Feng Chen, Rubao Lee, and Xiaodong Zhang. 2011. Essential roles of exploiting internal parallelism of flash memory based solid state drives in high-speed data processing. In *Proceedings of the IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 266–277. https://doi.org/10.1109/HPCA.2011.5749735

[10] Rong Chen, Jiaxin Shi, Yanzhe Chen, Binyu Zang, Haibing Guan, and Haibo Chen. 2018. PowerLyra: Differentiated Graph Computation and Partitioning on Skewed Graphs. *ACM Trans. Parallel Comput.* 5, 3 (2018), 13:1–13:39. https://doi.org/10.1145/3298989

[11] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. 2009. *Introduction to Algorithms, 3rd Edition*. MIT Press. http://mitpress.mit.edu/books/introduction-algorithms

[12] Diego Didona, Nikolas Ioannou, Radu Stoica, and Kornilios Kourtis. 2020. Toward a Better Understanding and Evaluation of Tree Structures on Flash SSDs. *Proceedings of the VLDB Endowment* 14, 3 (2020), 364–377. http://www.vldb.org/pvldb/vol14/p364-didona.pdf

[13] Shimon Even and Guy Even. 2012. *Graph Algorithms (2nd ed.)*. Cambridge University Press. http://www.cambridge.org/us/academic/subjects/computer-science/algorithmics-complexity-computer-algebra-and-computational-g/graph-algorithms-2nd-edition

[14] Joseph E Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. 2012. PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 17–30. https://www.usenix.org/conference/osdi12/technical-sessions/presentation/gonzalez

[15] 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 *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 599–613. https://www.usenix.org/conference/osdi14/technical-sessions/presentation/gonzalez

[16] Wook-Shin Han, Sangyeon Lee, Kyungyeol Park, Jeong-Hoon Lee, Min-Soo Kim, Jinha Kim, and Hwanjo Yu. 2013. TurboGraph: a fast parallel graph engine handling billion-scale graphs in a single PC. In *The 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD 2013, Chicago, IL, USA, August 11-14, 2013*. 77–85. https://doi.org/10.1145/2487575.2487581

[17] Alexandru Iosup, Tim Hegeman, Wing Lung Ngai, Stijn Heldens, Arnau Prat-Pérez, Thomas Manhardt, Hassan Chafi, Mihai Capota, Narayanan Sundaram, Michael J Anderson, Ilie Gabriel Tanase, Yinglong Xia, Lifeng Nai, and Peter A Boncz. 2016. LDBC Graphalytics: A Benchmark for Large-Scale Graph Analysis on Parallel and Distributed Platforms. *Proceedings of the VLDB Endowment* 9, 13 (2016), 1317–1328. https://doi.org/10.14778/3007263.3007270

[18] Ziyang Jiao and Bryan S Kim. 2022. Generating realistic wear distributions for SSDs. In *HotStorage '22: 14th ACM Workshop on Hot Topics in Storage and File Systems, Virtual Event, June 27 - 28, 2022*. 65–71. https://doi.org/10.1145/3538643.3539757

[19] Guodong Jin, Xiyang Feng, Ziyi Chen, Chang Liu, and Semih Salihoglu. 2023. KÙZU Graph Database Management System. In *Proceedings of the Conference on Innovative Data Systems Research (CIDR)*. https://www.cidrdb.org/cidr2023/papers/p48-jin.pdf

[20] Jeong-Uk Kang, Heeseung Jo, Jinsoo Kim, and Joonwon Lee. 2006. A superblock-based flash translation layer for NAND flash memory. In *Proceedings of the 6th ACM & IEEE International conference on Embedded software, EMSOFT 2006, October 22-25, 2006, Seoul, Korea*. 161–170. https://doi.org/10.1145/1176887.1176911

[21] U Kang, Hanghang Tong, Jimeng Sun, Ching-Yung Lin, and Christos Faloutsos. 2011. GBASE: a scalable and general graph management system. In *Proceedings of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, San Diego, CA, USA, August 21-24, 2011*. 1091–1099. https://doi.org/10.1145/2020408.2020580

[22] Donald E. Knuth. 2022. *"Weak components", The Art of Computer Programming, Volume IV, Pre-Fascicle 12A: Components and Traversal*. 11–14 pages. https://cs.stanford.edu/~knuth/fasc12a+.pdf

[23] Seongyun Ko and Wook-Shin Han. 2018. TurboGraph++: A Scalable and Fast Graph Analytics System. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 395–410. https://doi.org/10.1145/3183713.3196915
[24] Aapo Kyrola, Guy E Blelloch, and Carlos Guestrin. 2012. GraphChi: Large-Scale Graph Computation on Just a PC. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 31–46. https://www.usenix.org/conference/osdi12/technical-sessions/presentation/kyrola
[25] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data.
[26] Hang Liu and H Howie Huang. 2017. Graphene: Fine-Grained IO Management for Graph Computing. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*. 285–300. https://www.usenix.org/conference/fast17/technical-sessions/presentation/liu
[27] Ning Liu, Dong-sheng Li, Yiming Zhang, and Xiong-lve Li. 2020. Large-scale graph processing systems: a survey. *Frontiers Inf. Technol. Electron. Eng.* 21, 3 (2020), 384–404. https://doi.org/10.1631/FITEE.1900127
[28] László Lovász. 1993. Random walks on graphs. *Combinatorics, Paul erdos is eighty* 2, 1-46 (1993), 4.
[29] 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. *Proceedings of the VLDB Endowment* 5, 8 (2012), 716–727. https://doi.org/10.14778/2212351.2212354
[30] 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 *Proceedings of the Twelfth European Conference on Computer Systems, EuroSys 2017, Belgrade, Serbia, April 23-26, 2017*. 527–543. https://doi.org/10.1145/3064176.3064191
[31] 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 *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 135–146. https://doi.org/10.1145/1807167.1807184
[32] Bo Mao, Suzhen Wu, and Lide Duan. 2018. Improving the SSD Performance by Exploiting Request Characteristics and Internal Parallelism. *IEEE Trans. on CAD of Integrated Circuits and Systems* 37, 2 (2018), 472–484. https://doi.org/10.1109/TCAD.2017.2697961
[33] Kiran Kumar Matam, Gunjae Koo, Haipeng Zha, Hung-Wei Tseng, and Murali Annavaram. 2019. GraphSSD: graph semantics aware SSD. In *Proceedings of the 46th International Symposium on Computer Architecture, ISCA 2019, Phoenix, AZ, USA, June 22-26, 2019*. 116–128. https://doi.org/10.1145/3307650.3322275
[34] Junoh Moon, Mincheol Kang, Wonyoung Lee, and Soontae Kim. 2022. Salvaging Runtime Bad Blocks by Skipping Bad Pages for Improving SSD Performance. In *Proceedings of the Design, Automation & Test in Europe Conference & Exhibition (DATE)*. 576–579. https://doi.org/10.23919/DATE54114.2022.9774677
[35] Patrick E. O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth J. O'Neil. 1996. The log-structured merge-tree (LSM-tree). *Acta Informatica* 33, 4 (1996), 351–385. https://doi.org/10.1007/s002360050048
[36] Tarikul Islam Papon. 2024. Enhancing Data Systems Performance by Exploiting SSD Concurrency & Asymmetry. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE) PhD Symposium*.
[37] Tarikul Islam Papon and Manos Athanassoulis. 2021. A Parametric I/O Model for Modern Storage Devices. In *Proceedings of the International Workshop on Data Management on New Hardware (DAMON)*. https://doi.org/10.1145/3465998.3466003
[38] Tarikul Islam Papon and Manos Athanassoulis. 2021. The Need for a New I/O Model. In *Proceedings of the Biennial Conference on Innovative Data Systems Research (CIDR)*.
[39] Tarikul Islam Papon and Manos Athanassoulis. 2023. ACEing the Bufferpool Management Paradigm for Modern Storage Devices. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*.
[40] Stan Park and Kai Shen. 2009. A performance evaluation of scientific I/O workloads on Flash-based SSDs. In *Proceedings of the IEEE International Conference on Cluster Computing (CLUSTER)*. 1–5. https://doi.org/10.1109/CLUSTR.2009.5289148
[41] Stan Park and Kai Shen. 2012. FIOS: a fair, efficient flash I/O scheduler. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*. 13.
[42] Roger A Pearce, Maya B Gokhale, and Nancy M Amato. 2010. Multithreaded Asynchronous Graph Traversal for In-Memory and Semi-External Memory. In *Conference on High Performance Computing Networking, Storage and Analysis, SC 2010, New Orleans, LA, USA, November 13-19, 2010*. 1–11. https://doi.org/10.1109/SC.2010.34
[43] Gregory Popovitch. 2020. The Parallel Hashmap. *https://github.com/greg7mdp/parallel-hashmap* (2020).
[44] Amitabha Roy, Laurent Bindschaedler, Jasmina Malicevic, and Willy Zwaenepoel. 2015. Chaos: scale-out graph processing from secondary storage. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*. 410–424. https://doi.org/10.1145/2815400.2815408
[45] Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. 2013. X-Stream: edge-centric graph processing using streaming partitions. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*. 472–488. https://doi.org/10.1145/2517349.2522740

[46] Subhadeep Sarkar, Tarikul Islam Papon, Dimitris Staratzis, and Manos Athanassoulis. 2020. Lethe: A Tunable Delete-Aware LSM Engine. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 893–908. https://doi.org/10.1145/3318464.3389757

[47] Arman Shehabi, Sarah Smith, Dale Sartor, Richard Brown, Magnus Herrlin, Jonathan Koomey, Eric Masanet, Nathaniel Horner, Inês Azevedo, and William Lintner. 2016. United States Data Center Energy Usage Report. *Ernest Orlando Lawrence Berkeley National Laboratory* LBNL-10057 (2016). https://eta.lbl.gov/publications/united-states-data-center-energy

[48] Kai Shen and Stan Park. 2013. FlashFQ: A Fair Queueing I/O Scheduler for Flash-Based SSDs. In *Proceedings of the USENIX Annual Technical Conference (ATC)*. 67–78.

[49] Barak Shoshany. 2021. BS::thread_pool: a fast, lightweight, and easy-to-use C++17 thread pool library. *https://github.com/bshoshany/thread-pool* (2021).

[50] Julian Shun and Guy E Blelloch. 2013. Ligra: a lightweight graph processing framework for shared memory. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '13, Shenzhen, China, February 23-27, 2013*. 135–146. https://doi.org/10.1145/2442516.2442530

[51] Narayanan Sundaram, Nadathur Satish, Md. Mostofa Ali Patwary, Subramanya Dulloor, Michael J Anderson, Satya Gautam Vadlamudi, Dipankar Das, and Pradeep Dubey. 2015. GraphMat: High performance graph analytics made productive. *Proceedings of the VLDB Endowment* 8, 11 (2015), 1214–1225. https://doi.org/10.14778/2809974.2809983

[52] Kai Wang, Aftab Hussain, Zhiqiang Zuo, Guoqing Xu, and Ardalan Amiri Sani. 2017. Graspan: A Single-machine Disk-based Graph System for Interprocedural Static Analyses of Large-scale Systems Code. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 389–404. https://doi.org/10.1145/3037697.3037744

[53] Kai Wang, Zhiqiang Zuo, John Thorpe, Tien Quang Nguyen, and Guoqing Harry Xu. 2018. RStream: Marrying Relational Algebra with Streaming for Efficient Graph Mining on A Single Machine. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 763–782. https://www.usenix.org/conference/osdi18/presentation/wang

[54] Shoujin Wang, Liang Hu, Yan Wang, Xiangnan He, Quan Z Sheng, Mehmet A Orgun, Longbing Cao, Francesco Ricci, and Philip S Yu. 2021. Graph Learning based Recommender Systems: A Review. In *Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence, IJCAI 2021, Virtual Event / Montreal, Canada, 19-27 August 2021*. 4644–4652. https://doi.org/10.24963/ijcai.2021/630

[55] Wenlei Xie, Guozhang Wang, David Bindel, Alan J Demers, and Johannes Gehrke. 2013. Fast Iterative Graph Computation with Block Updates. *Proceedings of the VLDB Endowment* 6, 14 (2013), 2014–2025. https://doi.org/10.14778/2556549.2556581

[56] Xifeng Yan, Philip S Yu, and Jiawei Han. 2004. Graph Indexing: A Frequent Structure-based Approach. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 335–346. https://doi.org/10.1145/1007568.1007607

[57] Kaiyuan Zhang, Rong Chen, and Haibo Chen. 2015. NUMA-aware graph-structured analytics. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2015, San Francisco, CA, USA, February 7-11, 2015*. 183–193. https://doi.org/10.1145/2688500.2688507

[58] Yu Zhang, Xiaofei Liao, Hai Jin, Lin Gu, Ligang He, Bingsheng He, and Haikun Liu. 2018. CGraph: A Correlations-aware Approach for Efficient Concurrent Iterative Graph Processing. In *Proceedings of the USENIX Annual Technical Conference (ATC)*. 441–452. https://www.usenix.org/conference/atc18/presentation/zhang-yu

[59] Peixiang Zhao and Jiawei Han. 2010. On Graph Query Optimization in Large Networks. *Proceedings of the VLDB Endowment* 3, 1 (2010), 340–351. https://doi.org/10.14778/1920841.1920887

[60] 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 *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*. 45–58. https://www.usenix.org/conference/fast15/technical-sessions/presentation/zheng

[61] Xiaowei Zhu, Wenguang Chen, Weimin Zheng, and Xiaosong Ma. 2016. Gemini: A Computation-Centric Distributed Graph Processing System. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 301–316. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/zhu

[62] Xiaowei Zhu, Wentao Han, and Wenguang Chen. 2015. GridGraph: Large-Scale Graph Processing on a Single Machine Using 2-Level Hierarchical Partitioning. In *Proceedings of the USENIX Annual Technical Conference (ATC)*. 375–386. https://www.usenix.org/conference/atc15/technical-session/presentation/zhu

[63] Zichen Zhu, Siqiang Luo, Xiaokui Xiao, Yin Yang, Dingheng Mo, and Yufei Han. 2022. VC-Tune: Tuning and Exploring Distributed Vertex-Centric Graph Systems. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*. 3142–3145. https://doi.org/10.1109/ICDE53745.2022.00283

Received October 2023; revised January 2024; accepted February 2024