CAVE: Concurrency-Aware Graph Processing System for SSD

Tarikul Islam Papon, Manos Athanassoulis (Boston University, USA)

The Rise of Large Graphs. Graphs are *natural encoders* of interconnected relations which can be leveraged to analyze many real-world applications. With the unprecedented growth of such interconnected data stemming from various applications, analytics over large graphs is becoming increasingly popular. Real-world graphs often exhibit a vast scale, 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*, 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 and achieve scalability through increasing memory size and adding more CPUs. This work [4] is orthogonal to these approaches, however, it can benefit any system that spills data to 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. 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 [1, 2, 5].

Modern Storage Devices. SSDs dominate as secondary storage devices, while classical HDDs are nowadays primarily used for archival storage. SSDs offer fast data access, high chip density, and low energy consumption by utilizing NAND flash memory as their storage medium, thus eliminating the mechanical overheads of HDDs. Further, SSD internals follows a hierarchical structure that creates high *internal parallelism*, which can be leveraged to enhance performance [1, 3]. That is, an SSD can perform multiple *concurrent I/Os* until its bandwidth is saturated. Following the Parametric I/O model [2], 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.

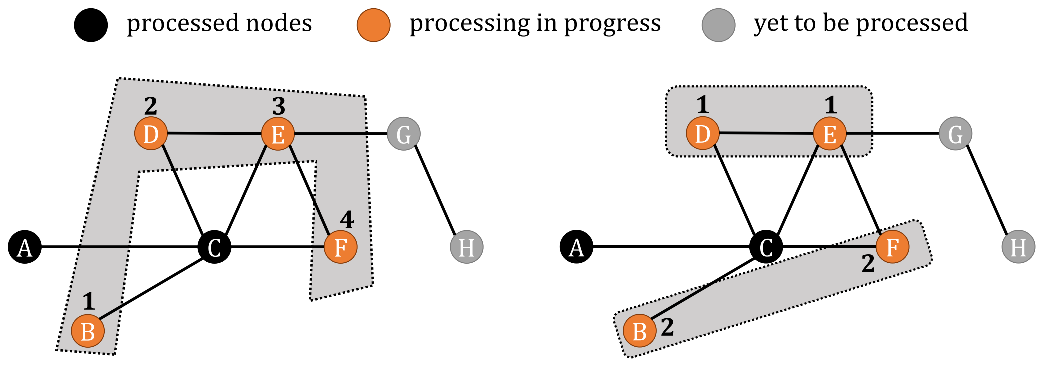
SSD Parallelism for Graph Processing. Graph traversal operations can utilize SSD concurrency by parallelizing node and edge accesses, effectively distributing the workload across SSD's parallel architecture. This idea takes advantage of the availability of multiple paths that can be explored during graph 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.

- **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.
- **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 [**?**], 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. Figure 1 illustrates these two techniques for a simplified graph. We integrate both approaches into our prototype graph processing system.

Our Approach. We build an SSD-aware graph processing system, named CAVE[1] [5] 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

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



(A) Intra-Subgraph Parallelization (BFS-Style)

(B) Inter-Subgraph Parallelization (pseudo DFS-Style)

Figure 1: 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.

in parallel. A prime example is our Parallel Breadth-First Search (PBFS) implementation that uses *intra-subgraph* parallelization as shown in Figure 1(A). The algorithm accesses the next *wave* of nodes {B,D,E,F} in parallel since we have already identified the nodes of the next wave while processing the current one (node C). 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. Since classical Depth-First-Search (DFS) is tricky to parallelize, we use a pseudo-DFS algorithm where the stack is split into smaller stacks when its size goes beyond a predefined threshold, and the smaller stacks are processed in parallel. This allows for multiple threads to work on different subgraphs (paths) concurrently, illustrated in Figure 1(B). 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* [2], 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 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 BFS and DFS, CAVE offers parallelized, SSD-aware versions of 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, GridGraph and Mosaic, as they are widely recognized for their efficiency in handling large-scale graphs in a single machine. We also compare with a single-node deployment of a distributed system named GraphX. By experimenting with six different types of graphs on three SSD devices, we demonstrate that CAVE utilizes the available parallelism, and scales to diverse real-world graph datasets. 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.

References

[1] T. I. Papon, and M. Athanassoulis. "The Need for a New I/O Model," *CIDR*, 2021.
[2] T. I. Papon, and M. Athanassoulis, "A Parametric I/O Model for Modern Storage Devices," *DAMON*, 2021.
[3] T. I. Papon, and M. Athanassoulis, "ACEing the Bufferpool Management Paradigm for Modern Storage Devices," *ICDE*, 2023.
[4] T. I. Papon, T. Chen, S. Zhang, and M. Athanassoulis, "CAVE: Concurrency-Aware Graph Processing on SSDs," *SIGMOD*, 2024.
[5] T. I. Papon, "Enhancing Data Systems Performance by Exploiting SSD Concurrency & Asymmetry," *ICDE PhD Symposium*, 2024.